COEN 175

Lecture 9: C++ Coding

C++ Guidelines

- Use a `typedef` to shorten a long type declaration and make it easier to understand.
  - `typedef std::vector<std::string> strings;`
- Use `nullptr`, and not `NULL`, when referring to a pointer type.
  - `NULL` is just zero and can sometimes be confused with an integer: `void f(int x)` vs. `void f(int *x)`.
- Pick a naming convention and stick to it!
  - Class names in Pascal Case, Camel Case, etc.
  - Member variables begin with an underscore.

C++ Classes

- Class declarations are placed in the header file.
- Public variables are usually a bad idea.
  - There's usually always a reason to encapsulate (i.e., hide behind an interface) the state of a class.
- Member function definitions should be placed in the source file.
  - If the function is small then it can be written inside the class declaration if you want.
- Make life easy on yourself and overload the output stream operator. It'll be useful for debugging.

C++ Header Files

- Always `#include` any headers that define any types that your class needs.
  - Otherwise, you rely on the client to do it for you. Bad idea!
- Protect your header file against multiple inclusions with `#ifndef` and `#endif`.
- Do not open the standard namespace.
  - Otherwise, the client might end up with name conflicts.
- Be consistent in the order of declarations.
  - Private, then protected, then public.
  - Constructors, then accessors, then other member functions.

C++ Source Files

- It’s okay to open the standard namespace here.
- Define your member functions in the same order as you declared them in the header file.
- It’s okay to write non-member functions as well, especially utility functions.
    - Place them before the member functions.
    - Declare them static so they aren’t visible outside the file.
- Use the C++ initialization syntax in constructors.
    - It’s required in some cases and always a good idea.

Simple C Types

- Last time we identified the requirements for our `Type` class to represent Simple C types.

- Every type has a specifier, indirection, and kind.
  - An array type also has its length.
  - A function type also has its parameters.

- We will overload the `==` and `!=` operators so that checking for identical types would be easy.

- Note that types will be **immutable** as we will only provide accessors and not mutators.

Symbols

- Once we have developed and tested our `Type` class, we can move on to our `Symbol` class.
- For now, a symbol will just hold a name and a type.
  - In later phases of the project, it will hold more information.
- Like types, symbols are more or less immutable.
  - Once created, the information doesn’t change since neither the name nor type can change.
  - Thus, we will provide accessors but not mutators.

Scopes

- A `scope` is simply a container for symbols.
  - Each scope holds a collection of symbols.
  - Each scope also has a link to its enclosing scope.
- A scope will need to support basic ADT operations:
  - `insert(symbol)`: add the symbol to the scope
  - `remove(name)`: remove the symbol with the given name
  - `find(name)`: return the symbol with the given name
  - `lookup(name)`: return the nearest symbol with the name
- To preserve declaration order, we will simply store the symbols in a vector.

Architectural Design

- We are building our semantic checker as a **layered architecture**.
- The lower layer is an **object-oriented design**.
  - We have our three classes: `Type`, `Symbol`, and `Scope`.
  - Very little at this layer deals with the language semantics.
- The upper layer is a **functional design**.
  - We will have functions to open and close scopes.
  - We will have functions to manage symbols and report errors according to our language semantics.

The Semantic Checker

- The semantic checking logic belongs in its own module, `checker.cpp`.

- In this module, we will have functions to manage scopes and symbols.
  - `openScope()`, `closeScope()`
  - `declareFunction()`, `defineFunction()`
  - `declareVariable()`, `checkIdentifier()`, `checkFunction()`

- These functions will be called by the parser to implement the required semantic checks.