COEN 175

Phase 3 - Week 5

TAs

Stephen Tambussi

Email: stambussi@scu.edu

Office Hours: Tuesday 1:00 - 2:00PM / Wednesday 4:00 - 5:00PM (Heafey Atrium)

Jordan Randleman

Email: jrandleman@scu.edu

Office Hours: Tuesday 9:10-10:10AM / Thursday 9:10-10:10AM (Heafey Atrium)

Extra Help / Tutoring

Tau Beta Pi Tutoring

- Thursday 2-3pm Heafey Atrium or its alcove prior the conference room
  - Location depends on table availability

Reminder: the New Lab Policy

- From Dr. Atkinson: You can only attend the lab you registered!
  - No more staying past 5 to the next lab
  - No more going to other labs to ask questions till you understand.
    - Dr. A is really emphasizing the need for y'all to learn how to write/debug programs on your own.
- If a TA figures out you already went to another lab, you will be kicked out.
  - Anything else risks the TAs directly going against a directive straight from Dr. Atkinson, and could literally get us fired (believe it or not, this does happen 🙃).
- If you want to go another lab ***instead*** of (not as a supplement to) your registered class, ***make sure you get permission from the TA before lab***.

Review of what the TAs ***are*** and ***are not*** here for!

There are 3 kinds of questions you can ask:

1. I don't understand how to build a compiler
    a. **That's ok!** So long as you've checked the lab slides, PDF, and lecture slides–and you're still confused as to how to structure your compiler–that's what we're here for! Ask away :)
2. I don't understand C++
    a. **These are problems you have to figure out on your own.** You can Google each and every single one of these questions, C++ is a massive language with an even more massive amount of documentation on the web. You will get fired if you ask your employer how to use a string, where to find a function, or which method you should call on an object.
3. I don't understand how to program
    a. **This is a call for introspection if you truly should even be taking compilers.** Everybody makes silly mistakes now and again, but consistently neglecting to run the code in your head, or not visualizing how the compiler is navigating the data as you write your program, is a recipe for a disastrous failure in this course.

Main Objectives for ALL of Phase 3

- You’re given a working solution for phase 2
  - Download these solutions to your machine, then immediately rename the directory to be “phase3” to avoid issues with moving files around
- Make the Symbol, Scope & Type classes
- Modify your parser
- Write a checker

High-Level Overview: Phase 3 Week 2

Goal: Create a symbol table

Note: parser.cpp is now complete! *You should no longer be editing it for the rest of phase 3!*

Week 5 Objectives (more on these in the following slides)

1. Write the Symbol class
2. Write the Scope class
3. Implement the rest of *checker.cpp*
    a. Global variables to track things
    b. openScope() and closeScope()
    c. defineFunction() and declareFunction()
    d. declareVariable() and checkIdentifier()

Submission

- Submit a tarball of your cpps and makefiles in a folder called *phase3*
- Worth 20% of your project grade
- Due Sunday, February 12th by 11:59 PM (*Super Bowl*)

Objective 1: Symbol.cpp/.h Class

- Don’t overthink it
- Literally just contains a string name and a type
- If you can’t do this please just drop the class

Objective 2: Scope.cpp/.h Class

- **High level overview:**
  - Linked list of scopes pointing to each other, with the global scope as the “tail”
  - Each scope has a vector of symbol ***POINTERs*** that were defined in that scope
  - The “next” pointer for each scope points to the enclosing scope
- **Member variables:**
  - Pointer to enclosing scope
  - Vector of symbols
- **Methods:**
  - *void insert(Symbol *symbol)*
    - insert a symbol
  - *void remove(const string &name)*
    - remove a symbol
  - *Symbol *find(const string &name) const*
    - find and return a symbol with a given name **ONLY** in the current scope
  - *Symbol *lookup(const string &name) const*
    - find and return nearest symbol with a given name **starting** the search in the given scope and **moving into** the enclosing scopes **ONLY if you didn’t find it yet!**

Objective 3a: *checker.cpp* - Global Variables

- Define global variable to track the head of your scope linked list
  - Head is the “current scope” being operated upon by the compiler
    - Conceptual check: ***why is this the case?*** If you don’t understand why then implementing checker is going to be impossible.
    - *Think about how you can access the global scope via the head pointer!*
    - “Current scope” is sometimes referred to as “toplevel scope”–use whatever naming convention you prefer.
- Define global E1-E5 error messages as const string variables
  - These messages will be ***reported*** (*use the report() function!*) from within your checker functions
    - “lexer.h” has the “report” function prototype
  - E1-E5 have ***SPECIFIC MESSAGES*** that you ***MUST RETURN EXACTLY AS WRITTEN***
    - More details on their contents and use cases are in the lab PDF
    - CHECKSUB.sh will check for these error messages word-for-word, so ***do not mistype them!***

Objective 3b: *checker.cpp* - Opening/Closing Scopes

- *openScope()*
  - Creates a new scope and passes it the enclosing scope as the new scope's next pointer
    - Just adding a node to the front of a linked list
- *closeScope()*
  - Pops off the current scope
    - Just removing a node from the front of a linked list

Objective 3c: *checker.cpp* - Checking Functions

- *declareFunction()*
  - Functions are always declared globally
  - Check for conflicting declarations (types must be identical)
- *defineFunction()*
  - Check for redefinition (type's parameters is not a *nullptr*)
  - Check for conflicting declarations (types must be identical)
  - Always insert newest definition
    - The newest function definition always replaces any previous definition or declaration

Objective 3d: *checker.cpp* - Checking Variables

- *declareVariable()*
  - Variables are always declared in the current scope
  - Check the type of the variable is a valid type (see assignment for rules regarding void)!
  - Check for redeclared variables
    - e.g. the variable is both declared more than once in the current scope ***and*** the current ***is not*** the global scope.
  - Check for conflicting types in redeclared variables within the global scope
- *checkIdentifier()*
  - lookup() identifier to see if given name has been declared, *if not issue [E4]*

TIPS

- **A lot of code is/will be provided in class**
- ***Include new files in the Makefile after you create them!***
  - *Affects Scope and Symbol*
- **READ THE SCOPE SEMANTIC RULES CAREFULLY**
- **MAKE SURE TO REPORT THE ERROR MESSAGES *VERBATIM* FROM CHECKER FUNCTIONS!**
  - *These messages are what will be checked by CHECKSUB.sh!*
- For those of you who can't Google, **this is a reference for C++'s vector class.**