CSCI 104
Staff
Schedule
Homework
Labs
Exercises
Syllabus
Sections
Wiki
Resources

CSCI 104

Homework 1

- Due: See assignments page
- Directory name in your github repository for this homework (case sensitive): `hw1`
    - Once you have cloned your `hw-username` repo, create this `hw1` folder underneath it (i.e. `hw-username/hw1`)
    - If your `hw-username` repo has not been created yet, please do your work in a separate folder and you can copy over relevant files before submitting

Written Portion

A Few Notes on Repositories

1. Never clone one repo into another. If you have a folder `cs104` on your VM, Docker, or laptop (wherever you created your Github keys from Lab 1) and you clone your personal repo `hw-username` under it (i.e. `cs104/hw-username`) then whenever you want to clone some other repo, you need to do it back up in the `cs104` folder or other location, NOT in the `hw-username` folder.
2. Your repos may not be ready immediately but be sure to create your GitHub account described in Lab 0 on the Labs Page. If you've followed those steps and still cannot access your repository, you can then make a private post on the class Q&A to let your instructors know that your repository needs to be created. Be sure to include your USC username and github username for reference.

Skeleton Code

On many occasions we will want to distribute skeleton code, tests, and other pertinent files. To do this we have made a separate repository, `resources`, under our class GitHub site. You should clone this repository to your laptop (**but only if you have not already done this as part of lab**) and do a `git pull` regularly to check for updates.

```
$ git clone git@github.com:usc-csci104-spring2022/resources
```

Again, be sure you don't clone this repo into your `hw-username` repo but at some higher up point like in a `cs104` folder on your laptop. You can then manually copy (in your OS's GUI or at the command line) the skeleton files from `resources/hw1` to `hw-username/hw1`.

For example if you are in the folder containing both the `resources` and `hw-username` folders/repos, you could enter the following command at the terminal:

```
$ cp -rf resources/hw1 hw-username/
```

Again be sure to replace `hw-username` with your USC username (e.g. `hw-ttrojan`)

Problem 1 - Course Policies (12%)

Carefully study the information on the course web site, then answer the following questions about course policies:

Place your answers to this question in a file name `hw1.txt` in your `hw-username/hw1` folder.

Part (a):

Which of the following are acceptable behaviors in solving homeworks/projects (list all that apply)?

1. Looking up relevant C++ reference information online.
2. Looking up or asking for sample solutions online from sites like Chegg, Github, etc.
3. Talking to my classmates about general approaches about the problems (but no specific coding statements or description of your own code or someone else's code).
4. Copying code from my classmates or an online source, and then editing it significantly.
5. Asking the course staff for help.
6. Sitting next to my classmate and coding together as a team or with significant conversation about approach.
7. Sharing my code with a classmate, even if he/she just wants to read over it and learn from it

Part (b):

To dispute a grade on a HW assignment you should (list all that apply):

1. Email the professor immediately.
2. Complete the regrade request form within 1 week of receiving the grade and wait for an issue to be posted to your Github repo.
3. Visit the designated regrade TA within 1 week of your score posting.

Part(c):

What is the late submission policy (list all that apply)?

1. One hour late submission is acceptable for each assignment.
2. Each assignment can be submitted up to two days late for 50% credit.
3. Each student has 5 late days of which only 2 can be used per HW
4. Students need to get an approval before submitting an assignment late.

Part(d):

After pushing your code to Gihub you should… (list all that apply)

1. Do nothing. Once you push your code you are done.
2. Clone your repo to a temporary folder to ensure all the files you desire are pushed and that your code compiles.
3. Complete the online submission page using your FULL **(30 or more digit)** SHA.

Part (e):

If you have NO grace days left and it is after the submission deadline, we will accept your assignment under what circumstances (list all that apply).

1. None. We will not accept your submission.
2. There is an hour grace period after the deadline.
3. If there is a technical difficulty (such as wireless trouble, or github commit/push issues, etc.) with submission.
4. If you email us your code as attachments.

Part (f):

General submission policies (indicate True/False).

1. *True/False*: Before submitting your HW you should reclone your repo to a separate folder and ensure all files are present and your code compiles.
2. *True/False*: If you forget to submit a file via GITHUB you can still apply for a regrade after the deadline and submit the missing file.
3. *True/False*: You only have 7 days to submit a regrade for homeworks, unless otherwise stated, and after that you are not eligible for a regrade for ANY reason.

Problem 2 - Git (6%)

Carefully review and implement the steps discussed in Lab1. Then, answer the following questions:

Continue your answers to this question in the file name `hw1.txt` in your `hw-username/hw1` folder.

Part (a):

When cloning a Git repo, which of the following should you avoid:

1. Cloning into a folder that itself is a git repo
2. Cloning into a sync'ed folder like Dropbox or Google Drive
3. Cloning into the `Desktop` folder of your VM

Part (b):

Provide the appropriate git command to perform the following operations:

1. Stage an untracked file to be committed. The file is called 'hw1q2b.cpp'.
2. Display the details of the last three commits in the repository.

Part (c)

What is the full command to re-clone your private CSCI104 repo to your VM? Assume you are in an appropriate folder.

Problem 3 - Runtime Analysis (24%)

In Big-Θ notation, analyze the running time of the following three pieces of code/pseudo-code. Describe it as a function of the input (here, `n`). Submit your answers as a PDF (using some kind of illustration software or scanned handwritten notes where you use your phone to convert to PDF) showing your work and derivations supporting your final answer. **You must name the file `q3_answers.pdf`**. As usual, answers without supporting work will receive 0 credit.

Part (a)

```
void f1(int n)
{
  int t = sqrt(n);
  for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
      // do something O(1)
    }
    n -= t;
  }
}
```

Part (b)

```
Assume A is an array of size n+1.

void f2(int* A, int n)
{
  for(int i=1; i <= n; i++){
    for(int k=1; k <= n; k++){
      if( A[k] == i){
        for(int m=1; m <= n; m=m+m){
          // do something that takes O(1) time
          // Assume the contents of the A[] array are not changed
        }
      }
    }
  }
}
```

Part (c)

```
void f3(int* A, int n)
{
  if(n <= 1) return;
  else {
    f3(A, n-2);
    // do something that takes O(1) time
    f3(A, n-2);
  }
}
```

Part (d)

Notice that this code is very similar to what will happen if you keep inserting into an ArrayList (e.g. `vector`). Notice that this is **NOT** an example of amortized analysis because you are only analyzing *1* call to the function `f()`. If you have discussed amortized analysis, realize that does NOT apply here since amortized analysis applies to *multiple* calls to a function. But you may use similar ideas/approaches as amortized analysis to analyze this runtime. If you have NOT discussed amortized analysis, simply ignore it’s mention.

```
int f (int n)
{
  int *a = new int [10];
  int size = 10;
  for (int i = 0; i < n; i ++)
     {
        if (i == size)
          {
             int newsize = 4*size;
             int *b = new int [newsize];
             for (int j = 0; j < size; j ++) b[j] = a[j];
             delete [] a;
             a = b;
             size = newsize;
          }
        a[i] = i*i;
    }
}
```

Problem 4 - NOT GRADED (PRACTICE ONLY) - Constructors and Destructors (0%)

Place your answers to the questions below in the same file as your answers to problem 1 and 2 (i.e. `hw1.txt`). You do not need to test and compile the code below. You are just writing out your answers in the `hw1.txt` file.

Suppose you were given the following class to model an *entry* in your contacts list which uses a custom `Str` class that models a string and replaces `std::string`.

```
#ifndef ENTRY_H
#define ENTRY_H
#include "str.h"

class Entry {
 public:
  Entry(const Str& name, const Str& phone);
  ~Entry();
  const Str& name() const;
  const Str& phone() const;

 private:
  Str name_;
  Str phone_;
};
#endif
```

Further assume that print statements are added to all constructor and destructors that print:

- `Str` in the default and initializing constructor(s)
- `~Str` in the destructor
- `Copy Str` in the copy constructor
- Nothing is printed in the assignment operator(s)

Question a: If the `Entry` constructor is as shown below, what will be printed when a new Entery object is constructed.

```
Entry::Entry(const Str& name, const Str& phone)
{
  cout << "Entry" << endl;
  name_ = name;
  phone_ = phone;
}
```

Question b: If the `Entry` constructor is as shown below, what will be printed when a new Entry object is constructed.

```
Entry::Entry(const Str& name, const Str& phone)
  : name_(name), phone_(phone)
{
  cout << "Entry" << endl;
}
```

Now suppose a new `Wrapper` class is written that uses an `Entry` as a data member as shown below.

```
#ifndef WRAPPER_H
#define WRAPPER_H
#include "entry.h"
```

```
class Wrapper
{
 public:
  Wrapper(const Str& name, const Str& phone);
  void print() const;
 private:
  Entry e_;
};
#endif
```

Question c: Show how to complete the constructor such that the data member `e_` is initalized with the arguments `name` and `phone`. Be sure to avoid any compile errors or runtime errors. You can always try your code out using a compiler. Show the entire constructor in your answer (i.e. start by copying the code below and then add to it).

```
// initialize e_ with name and phone
Wrapper::Wrapper(const Str& name, const Str& phone)
{

}
```

Programming Portion

Problem 5 - LabelList (Classes, Pointers, LinkedLists) (58%)

LabelList (Multi-level Doubly-Linked List)

How To Approach This Assignment

We use this homework not just as a coding exercise but to teach aspects of C++, object-oriented design, and rationale for certain data structure choices. Several of these aspects are discussed in the Motivation and Background Information sections. Links to other examples or reference material are often provided. Take time to consider these, read more on your own, and ask questions. We recommend you read through the entire assignment once or twice before even considering writing code (though you can look at the skeleton files, especially the header file for reference).

In addition, we suggest you an **incremental** coding/test approach. Avoid the temptation to write the all the code at once. Instead, write one portion (feature), compile, and test it before moving on to the next. In this way, dealing with compile and debugging errors becomes MUCH more manageable and by committing/pushing your code you can have snapshots of working features.

Introduction

This exercise will build your linked list skills and confidence and ability in using pointers. You will implement a multi-level, doubly-linked list of *labelled messages* that models something akin to an email mailbox and allows messages to be members of mulitple, user-defined **labels**. All messages are part of the default `all` label, while new labels may be defined at any time by the user. Each label is represented by a separate doubly-linked containing only those messages that are members of that label. Messages can be part of any number of labels with a next/previous pointer for each label list, as shown in the diagram below.

drawing

Note: the `1` or `0` in the message nodes is a Boolean indicate membership in a given label's list.

New messages are always added to the end of the `all` list. Once added, they can be labelled or unlabelled, as desired. Messages can also be deleted (removed) permanently. New labels need not be created explicitly by the user, but the first time a label is applied to a message, a new list for that label should be created and the message will added to that new list. For example, suppose a new label `fam` (short for `family`) is applied to message 1. `fam` should be added to the list of labels and a new head pointer should be added to the `heads_` list. The index (i.e. `4` in the diagram below) of this new label and head pointer should now be used as the index of the **next** and **prev** pointer for this label in each member message's `next_` and `prev_` lists. This will require resizing those vectors (of next and prev pointers) in the message node. (Note that only member messages need to have their next/previous vectors resized, while messages that are not a member of the new label need not be modified).

Effects of adding the label `fam` to message 1:

drawing

When a message is unlabeled, it is simply removed from the corresponding list.

Effects of unlabeling `usc` *from message 0:*

drawing

Removing a message should delete the entire message node, first unlinking it from all lists of which it is a member.

Effects of removing message 0:

drawing

Approach and Motivation

Many possible data structures can be implemented to support these operations. For example, one could simply store all messages in a singly-linked list or vector and then use a separate vector per label that stores pointers to the messages that are members of that label. This would lead to a vector of vectors iemplementation.

However, such an approach lacks two key features we would like to support.

1. First, having obtained a pointer or reference to a message, we would like to be able to quickly (roughly constant time) find neighboring (next/previous) messages of a different label. If we had a separate vector per label, we'd need to perform a new search within the new label's vector for the current message to then find its neighbors.
2. Second, erasing a message should be a constant time operation. Again, we do not want to have to search for the message pointer/reference in each label's vector.

drawing

drawing

Thus, we have chosen to use a multi-level, doubly-linked list approach. In this implementation, we will have a vector of labels and a vector of head pointers (1 head pointer per label) stored in the `LabelList` object. The index of a label stored in the label vector should correspond to the index of that label's head pointer. Each message node we add will have a vector of previous and next pointers as well as a vector of booleans indicating whether the message is a member of the label that corresponds to that level/index.

A further benefit of this approach is that if the client obtains a pointer to a message and then other messages are erased, the original message pointer will still be valid. Unlike vectors where removing an item may shift other elements up to new indices/addresses and thus invalidate pointers held by third parties, the linked list approach allows an item to remain at the same memory address throughout its lifetime and keeps any pointers/references valid, even as other elements are added or removed.

Basic Implementation

LabelList class

The primary `LabelList` class requires only two vectors as data members: a vector of labels (`labels_`) and a vector of head pointers (`heads_`). These two vectors should always be the same size. As new labels are added, new lists (via new head_ pointers) are added to their respective vectors.

All the operations to add, remove, label, unlabel, find, and print messages (that belong to particular labels) will be implemented in the `LabelList` class.

MsgNode

To model a message and its next/previous pointers, a nested struct `MsgNode` has been created to store the actual `string` that contains the message, vectors for next and previous pointers, and a vector of `bool`s to indicate membership of the given label. This struct is similar to the typical `Item` struct that is often defined for a linked list. As a design point, you may want to consider if the `bool`s are needed or if simply setting both next and previous pointers to `nullptr` would suffice to indicate that a node is NOT a member of a given label.

MsgToken

One goal of object-oriented design is **encapsulation** which usually means giving only the minimum access necessary to third parties. We want/need to provide the client some way of referencing a particular message and performing operations on it (accessing the message, navigating to neighboring messages for a given label, etc.). However, if we simply returned a pointer to a `MsgNode` the client would have access to the public members of the `MsgNode` struct. Instead, to limit access to only the operations we want to expose to clients, we define a `MsgToken` class that internally (privately), stores a pointer to a paritcular `MsgNode` as well as the overall `LabelList` that the `MsgNode` belongs to and provides public functions to access the message or navigate to the next or previous messages of a given label.

A look ahead: You should shortly learn about `iterator`s in the C++ STL container library. This `MsgToken` object is very similar and should help the concept of iterators make more intuitive sense in the future.

drawing

C++ and Background Information

There are several features of C++ that are used in the given `LabelList` implementation, but may be unfamiliar. Read below to understand some of these features. It would help to open the provided `labellist.h` and reference the provided code as each feature is described.

Forward declarations

Suppose two classes each contain pointers to each other. Which would you define first? One could not be fully defined without defining the other. And `#include` statements wouldn't work because each would need to `#include` the other.

```
#include "TypeB.h"
class TypeA {
  TypeB* b;
};
```

```
#include "TypeA.h"
class TypeB {
  TypeA* a;
};
```

To break the dependence, we can use a forward *declaration*. A forward *declaration* effectively states a class/struct *will exist and be defined* without actually providing its definition (i.e. data and function members). Given a forward declaration **NO objects of that type can be declared**, BUT **pointers and references to that type CAN BE declared**.

Thus, the example above could be made to work using these forward declarations:

```
// forward declaration
class TypeB;

class TypeA {
  TypeB* b;  // pointer to TypeB can be used
};
```

```
// forward declaration
class TypeA;

class TypeB {
  TypeA* a;  // pointer to TypeA can be used
};
```

We use that approach in our `LabelList` to `typedef` a vector of `MsgNode*` before we actually define `MsgNode`, since it will contain those vectors as members.

Friend classes

Recall the motivation for the `MsgToken` class is to provide an access mechanism to 3rd-party clients without exposing the underlying structure of (and access to) our `MsgNode`. However, we may want the `LabelList` methods to access private members or construct `MsgTokens` differently than the public interface. Thus, we use C++'s `friend` declaration. Classes can indicate that a particular non-member function **OR another class** is a `friend` and thus can access private members of the class. Thus the `MsgToken` class specifies the `LabelList` as a `friend` class. Note: `friend` declarations are one way (just because `A` declares `B` as a friend class, `A` is not automatically a friend of `B`). However, because we are using nested types and `MsgToken` is a member of `LabelList`, then members have access to other members. So as a member of `LabelList`, `MsgTokens` have access to other `LabelList` members, such as `findLabelIndex`, etc.

Notice we also have two constructors for `MsgToken`. The default constructor `MsgToken()` allows 3rd party clients to declare `MsgToken` objects but prevents initialization of its members to anything but an invalid state. This forces clients to then assign the token with the return values produced by `LabelList` members (like `add()` or `find()`). The initializing constructor `MsgToken(MsgNode* node, LabelList* list)` is private, so that only the friend `LabelList` class can use it to create valid tokens. This again requires clients to use the `LabelList` interface to generate valid tokens.

Static members

Often, certain data members or constants may be the same for **ALL** instances of the object and need not require a separate data member (storage) in each object. Instead, we can define the data member once and have it be shared by all objects. `LabelList` defines two static members: the constant `size_t INVALID_LABEL` and the `MsgToken end_`. `INVALID_LABEL` is used as the return value to the helper function `findLabelIndex` to indicate a non-existent label. `end_` is returned by any `LabelList` member functions such as `find()` if the desired message cannot be found, or when attempting to advance to the next or previous message when none exists.

Once declared in the class with the `static` keyword, static members should generally be instantiated and initialized in the `.cpp` (though for integral values, newer versions of C++ allow them to be defined inline in the header file). We intialize `INVALID_LABEL` to `(size_t)-1` since `-1` is all 1s in binary and will be the largest unsigned value when **cast to an unsigned type**. Note: An example of a similar use of static members is the constant `npos` defined in `std::string` (really `std::basic_string`). Info about its use is here

We construct `end_` in the `labellist.cpp` implementation. You can see that it looks like a global variable declaration, but is preceded with the `static` keyword and is **scoped** to indicate it is a member of the `LabelList` class.

You may need to use `INVALID_LABEL` as you implement certain functions. Similarly, `end_` can be returned by functions that need to return a `MsgToken` when no valid token is possible.

Exceptions

Errors happen in programming. We receive unexpected or illegal inputs or arguments or we reach a state that we cannot handle. In those cases we can take some action like returning a specific value, but in some cases, based on the function signature, that may be impossible. An alternate approach is exceptions.

You should learn about exceptions by watching the provided lecture video and reading online or in the textbook.

- In this homework, you do not need to catch any exceptions.
- However, we will ask you to throw certain exceptions in the following problem. You can just use an appropriate `throw` statement. Look in the skeleton code documentation for where/when to throw an exception.

Implementation Tasks

To start your implementation, read over the provided `labellist.h` and the documentation provided for each member function. **This documentation acts as a set of requirements by which you will be graded for partial credit** (in addition to some automated tests). Ensure you meet the specified requirements (including runtime). **You may declare additional data members and private helper functions but should never change the given public interface without express consent from the instructor** (ask on our Q&A site).

In `labellist.cpp` note the functions that have been specified as `Complete` or `To be completed`. Only the latter are functions you need to write.

For the `MsgToken` class you need to complete the following functions, among others:

- Comparison operators (`==` and `!=`). Since a token refers to a message, what do you think you should actually compare when the user compares the entire `MsgToken` objects.
- The `operator bool()` allows `MsgTokens` to be used in the context of a condition (i.e. `if(token)`) and should return `true` for valid tokens.
- The `msg()` accessors for both `const` and non-`const` contexts should return the underlying `std::string` message
- The `next` and `prev` routines to update the token to refer to the appropriate message.
- The `ostream` operator (`operator<<`) to output the underlying message. Do NOT output a newline at the end.
- If you find yourself duplicating work or wanting to break a complex task into simpler tasks, consider adding **private** member functions to help.

For the `LabelList` class, you will need to implement the following:

- Constructor and destructor (ensuring no memory leaks, etc.)
- `clear` to reset the list to an empty state, removing and deleting all messages in the list.
- `empty` should return true if no messages exist in the entire list.
- `add/remove` should add a new message to the `all` list while remove should delete the message node, removing it from all labels (lists) of which it is a member.
- `label/unlabel` should cause the referenced message to be added or removed from membership of the given label.
- `find` should return a desired message based on its string content or position (0-based index) in the given label's list.
- the `findLabelIndex` helper function to return the index/level of a given label (i.e. what index in the `heads_` and `next_/prev_` vectors corresponds to the specified label). **Note:** `findLabelIndex` returns `INVALID_LABEL` when a message cannot be found while `getLabelIndex` will throw an exception (i.e. `getLabelIndex` should be used when a correspond label **SHOULD** be defined while `findLabelIndex` should be used if a label may appropriately not exist yet)
- If you find yourself duplicating work or wanting to break a complex task into simpler tasks, consider adding **private** member functions to help.

Coding and Testing

Coding Approach

As mentioned earlier, it would be wise to code and test incrementally and not try to write all the code at once. A possible approach would be:

- Implement just the base necessities of the `MsgNode`, `MsgToken`, and `LabelList` class to **add** and **remove** nodes from the `all` list/label. Test your code in `labellist-test.cpp` by creating a `Labellist`, adding a few nodes, saving the tokens, and using them to access the message and/or remove the messages. Use `LabelList::print()` to verify the contents visually.
- Add the label method and more tests to use additional labels, again using `LabelList::print()` to verify the contents of each list visually. Note: `print()` only traverses a list in the forward direction. If you have any errors setting previous pointers, they may remain hidden. Consider using the `MsgTokens` to traverse a label list in the backward direction.
- Add the unlabel method and test it
- Add any remaining functionality.

Unit Tests

Within several days of the assignment posting, we will provide suite of unit tests that will also be utilized for automated grading. Information on how to run those tests will either be posted here or on our Q&A site.

Using Valgrind

Remember, it is **ALWAYS** a good idea to run your tests through `valgrind`. You can do so by executing the following command line.

```
valgrind --tool=memcheck --leak-check=yes ./labellist_tests
```

Scroll through the output and look for invalid reads, writes, and the heap usage summary at the end. However, please note, that just as a doctor can only diagnose you based on the symptoms or the info you provide, **valgrind can only check for errors based on what the test code exercises**. If the test code never triggers a function and there are memory leaks or invalid accesses in that function, `valgrind` will say no errors occurred. You are only as good as what your tests exercise, so it helps to write tests that will trigger each line of code in your class (this is often referred to as *code coverage*).

Requirements

- You **MUST** meet all runtime, exception, and other requirements listed in the documentation/comment of each member function given in the skeleton code header file.
- You may not alter the public interface of `LabelList` provided but can add data members and private helper functions.
- You may not alter the public interface of the `MsgToken` or `MsgNode` classes but may add to the public interface as well as adding private data or helper functions.
- We want you to practice with linked lists and pointers. Thus, no use of any other container is allowed (i.e. no additional `vectors` or any other C++ data structures beyond the already given data members can be used). **Failure to follow this guideline will result in a 0 since it likely alleviates the need to practice the desired pointer/linked list skills.**

Related Videos

- A video tutorial on debugging is available and demonstrates techniques that can be used to debug errors in your labellist.

Checkpoint

For checkpoint credit, commit and push your `hw-username` repo with a `hw1` subfolder that contains:

- `hw1.txt` with your answers to question 1 and 2
- `q3_answers.pdf` or `hw1.pdf` that AT least contains your answer and justification to runtime question **3a**
- a version of `labellist.h/cpp` that can pass the `hw1-checkpt` tests in `hw1-checkpt.cpp`. To attempt to compile and run the `hw1-checkpt` tests, type `make check` at the command line which will both compile AND run the tests (if the compilation succeeded). To pass the checkpoint tests you must have implementation of the
  - `LabelList` constructor, `add()`, and `label()`
  - `LabelList::MsgToken` member functions `msg()` (both const and non-const version) and operators `==`, `!=` and `bool`
- **THEN** you must submit your SHA on our Submit page linked from the Homework Page

Submission Files

Ensure you add/commit/push all your source code files, `Makefile`, and written problem files. Do **NOT** commit/push any test suite folder/files that we provide from any folder other than the `resources/hw1` repo.

WAIT You aren't done yet. Complete the last section below to ensure you've committed all your code.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your hw1 directory to your `hw-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. In your terminal, `cd` to the folder that has your `resources` and `hw-username`
2. Create a `verify-hw1` directory: `$ mkdir verify-hw1`
3. Go into that directory: `$ cd verify-hw1`
4. Clone your hw_username repo: `$ git clone git@github.com:usc-csci104-spring2022/hw-username.git`
5. Go into your hw1 folder `$ cd hw-username/hw1`
6. Switch over to a docker shell, navigate to the same `verify-hw1/hw-username/hw1` folder.
7. Recompile and rerun your programs and tests to ensure that what you submitted works. You may need to copy over a test-suite folder from the `resources` repo, if one was provided.