Computer Science II — CSci 1200
Lecture 15
Sets; Program Design; Life

Test 2 Statistics

- Average: 74.1
- Distribution:

Range	Count
90-100	28
80's	44
70's	55
60's	25
50'	13
< 50	18

- There is a slight upwards curve, and your curved score should be written on your tests.
- Solutions are posted on the course website.
- For anything other than an error in totaling your score, any regrade requests will cause a regrade of your entire exam.

Overview

- Standard library sets
  - You may use sets in HW 7.
- Steps in solving programming problems
- Conway's Game of Life

Standard Library Sets

- Ordered containers storing unique "keys".
- An ordering relation on the keys, which defaults to `operator<`, is necessary.
- Because of this ordering requirement, standard library sets are not truly mathematical sets.
- Sets are like maps except they have only keys — there are no associated values.
  - In particular, this means you can't change a key while it is in the set. You must remove it, change it, and then reinsert it.
- Access to items in sets is extremely fast!

We will skim through the remaining material on sets fairly quickly during lecture.

Set: Definition

```
template <class Key, class Compare = less<Key> >
class set {
...
```

- The set stores **constant** Keys at each node.
- Sets have the usual constructors as well as the `size` member function.
- The second component of the template, the Compare function, is usually not used. Instead, we generally assume `operator<` is defined for the Key. For example,

```
set<string> words;
```

Set iterators

- Set iterators, similar to map iterators, are bidirectional: they allow you to step forward (`++`) and backward (`--`) through the set.
- Set iterators refer to const keys (as opposed to the pairs referred to by map iterators).
- Sets provide `begin()` and `end()` iterators to delimit the bounds of the set.
- For example, the following code outputs all strings in the set `words`:

```
for ( set<string>::iterator p = words.begin(); p!= words.end(); ++p )
  cout << *p << endl;
```

Set insert

- There are two different versions of the `insert` member function:
- Version 1:

```
iterator set<Key>::insert( iterator pos, const Key& entry );
```

  This inserts the key if it is not already there. The iterator `pos` is a "hint" as to where to put it. This makes the insert faster if the hint is good.

- Version 2:

```
pair<iterator,bool> set<Key>::insert( const Key& entry );
```

  Insert the entry into the set. The function returns a pair. The first component of the pair refers to the location in the set containing the entry. The second component is true if the entry was not already in the set and therefore was inserted. It is false otherwise.

Set erase

- Three `erase` functions:

```
size_type set<Key>::erase( const Key& x );
void set<Key>::erase( iterator p );
void set<Key>::erase( iterator first, iterator last );
```

  where `size_type` is generally equivalent to an `unsigned int`.

- The first `erase` returns the number of entries removed, either 0 or 1.
- Note that the `erase` functions do not return iterators. This differs from the `vector` and `list` erase functions.
- The second and third erase functions are just like the corresponding erase functions for maps.

Set find

- The `find` member function:

```
const_iterator set<Key>::find( const Key& x) const;
```

  This function returns the `end` iterator if the key is not in the set.

- The `lower_bound` member function:

```
iterator set<Key>::lower_bound( const Key& x);
```

  Returns an iterator referring to the first entry in the set whose key is not less (at least as large as) the search key.

- The `upper_bound` function:

```
iterator set<Key>::upper_bound( const Key& x);
```

  Returns an iterator referring to the first entry in the map whose key is greater than the search key.

Exercises

1. Write a function to count the number of times a given first name appears in a set of names. Assume the `Name` class from Lecture 14, which has `first` and `last` member functions and a `operator<`. Here is the prototype:

```
int count_first( const string& fname, const set<Name>& names );
```

2. The problem of association between words from Test 2 can be solved with a map of sets:

```
map< string, set<string> > assoc;
```

Write code to output all pairs of words `s1` and `s2` such that `s2` is in the set associated with `s1` and `s1` is in the set associated with `s2`.

Problem Solving Strategies

Here are some of the major steps I use in solving programming problems.

1. Before getting started: study the requirements, carefully!
2. Get started:
   (a) What major operations are needed and how do they relate to each other as the program flows?
   (b) What important data / information must be represented? How should it be represented? Consider and analyze several alternatives, thinking about the most important operations as you do so.
   (c) Develop a rough sketch of the solution, and write it down.
3. Review: reread the requirements and examine your design. Are there major pitfalls in your design? Does everything make sense? Revise as needed.
4. Details, level 1:
   (a) What major classes are needed to represent the data / information? What standard library classes can be used entirely or in part? Evaluate these based on efficiency, flexibility and ease of programming.
   (b) Draft the main program, defining variables and writing function prototypes as needed.
   (c) Draft the class interfaces — the member function prototypes.

   These last two steps can be interchanged, depending on whether you feel the classes or the main program flow is the more crucial consideration.
5. Review: reread the requirements and examine your design. Does everything make sense? Revise as needed.
6. Details, level 2:
   (a) Write the details of the classes, including member functions.
   (b) Write the functions called by the main program. Revise the main program as needed.
7. Review: reread the requirements and examine your design. Does everything make sense? Revise as needed.
8. Testing:
   (a) Test your classes and member functions. Do this separately from the rest of your program, if practical. Try to test member functions as you write them.
   (b) Test your major program functions. Write separate "driver programs" for the functions if possible. Use the debugger and well-placed output statements and output functions (to print entire classes or data structures, for example).
   (c) Be sure to test on small examples and boundary conditions.

   The goal of testing is to incrementally figure out what works — line-by-line, class-by-class, function-by-function. When you have incrementally tested everything (and fixed mistakes), the program will work.

Notes

- For larger programs and programs requiring sophisticated classes / functions, these steps may need to be repeated several times over.
- Depending on the problem, some of these steps may be more important than others.
  - For some problems, the data / information representation may be complicated and require you to write several different classes. Once the construction of these classes is working properly, accessing information in the classes may be (relatively) trivial.
  - For other problems, the data / information representation may be straightforward, but what's computed using them may be fairly complicated.
  - Many problems require combinations of both.

Conway's Game of Life

As a design example we will consider Conway's Game of Life. We will focus on the main data structures of needed to solve the problem.

Here is an overview of the Game:

- Two-dimensional grid of cells, which can grow arbitrarily large in any direction.
- Simulation of life / death of cells on the grid through a sequence of generations.
- In each generation, each cell is either alive or dead.
- At the start of a generation, a cell that was dead in the previous generation becomes alive if it had exactly 3 live cells among its 8 possible neighbors in the previous generation.
- At the start of a generation, a cell that was alive in the previous generation remains alive if and only if it had either 2 or 3 live cells among its 8 possible neighbors in the previous generation.
  - With fewer than 2 neighbors, it dies of "loneliness".
  - With more than 3 neighbors, it dies of "overcrowding".
- Important note: all births / deaths occur simultaneously in all cells at the start of a generation.
- Other birth / death rules are possible, but these have proven by far the most interesting.
- The website

  `http://www.math.com/students/wonders/life/life.html`

  has an excellent discussion of the Game of Life, including simulations and many interesting patterns.

Our Problem

We will think about how to write a simulation of the Game of Life, focusing on the representation of the grid and on the actual birth and death processes.

Game of Life Implementation

- Representing the live cells in each generation is by far the most critical issue.
- The rest of the representation depends on what additional features are needed.

Step 1 of Program Design: Understanding the Requirements

We have already been working toward understanding the requirements. This effort includes playing with small examples by hand to understand the nature of the game, and a preliminary outline of the major issues.

Step 2 of Program Design: Main Program Flow and Representations

- Major operations: We will list the major operations and then organize them to form one possible flow for the main program.
- Major representation: Clearly the major problem is representing the life grid, which may be arbitrarily large. We will brainstorm several possibilities, outline how they might work, and then critique them.

We will discuss any remaining steps as time permits.