

# CSCI-1200 Computer Science II — Spring 2006

## Lecture 15 — Problem Solving Part II: Program Design

### Review from Lecture 14

- Classes can be used as map keys if a well-behaved `operator<` is available. Maps can store more complicated values, such as vectors or classes. The *syntax* of using maps in this way can become a bit complicated, you'll get more comfortable with practice.
- Maps to solve the-word-to-line-number indexing problem: using a vector or a list would be significantly more difficult and the solution is less natural.
- Maps should be used when there is an association — natural or easily created — between two classes. As a larger example we looked at the MP3 database.

### Overview

- Standard library sets
- Steps in solving programming problems
- Conway's Game of Life

### 15.1 Standard Library Sets

- *Ordered* containers storing unique “keys”. An ordering relation on the keys, which defaults to `operator<`, is necessary. Because STL sets are ordered, they are technically not traditional mathematical sets.
- Sets are like maps except they have only keys, there are no associated values. Like maps, the keys are **constant**. 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!

### 15.2 Set definition

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

- Like other containers, sets have the usual constructors as well as the `size` member function.
- The second component of the `set` template, the compare function, is optional if `operator<` is defined for the key. For example: `set<string> words;`

### 15.3 Set iterators

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

### 15.4 Set insert

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

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

Inserts the entry into the set and 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 wasn't already in the set and therefore was inserted. It is false otherwise.

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

This also 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.

## 15.5 Set erase

- There are three versions of `erase`:

```
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). The second and third `erase` functions are just like the corresponding `erase` functions for maps. Note that the `erase` functions do not return iterators. This is different from the `vector` and `list` `erase` functions.

## 15.6 Set find, lower\_bound, and upper\_bound

- The `find` function returns the end iterator if the key is not in the set:

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

- This function returns an iterator referring to the first entry in the set whose key is not less (at least as large as) the search key:

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

- And this function returns an iterator referring to the first entry in the map whose key is greater than the search key:

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

## 15.7 Exercise

Write a function to count the number of times a given first name appears in a set of names. Here is the prototype:

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

Here is the `Name` class from Lecture 14:

```
class Name {  
public:  
    Name(const string& first, const string& last) : first_(first), last_(last) {}  
    const string& first() const { return first_; }  
    const string& last() const { return last_; }  
private:  
    string first_;  
    string last_;  
};  
  
bool operator< (const Name& left, const Name& right) {  
    return left.last() < right.last() || (left.last() == right.last() && left.first() < right.first());  
}  
  
bool operator== (const Name& left, const Name& right) {  
    return left.last() == right.last() && left.first() == right.first();  
}
```

## 15.8 Problem Solving Strategies

Here is an outline of the major steps to 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. There are advantages to working on paper first. Don't start hacking right away!
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.

## 15.9 Design Example: Conway's Game of Life

Let's design a program to simulate Conway's Game of Life. Initially, due to time constraints, we will focus on the main data structures of needed to solve the problem.

Here is an overview of the Game:

- We have an infinite two-dimensional grid of cells, which can grow arbitrarily large in any direction.
- We will simulate the 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 to be a very interesting balance.
- Many online resources are available with simulation applets, patterns, and history. For example:

```
http://www.math.com/students/wonders/life/life.html
http://www.radicaleye.com/lifepage/patterns/contents.html
http://www.bitstorm.org/gameoflife/
```

### Applying the Problem Solving Strategies

In class we will brainstorm 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.

### 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.

### Getting Started

- What are the important operations?
- How do we organize the operations to form the flow of control for the main program?
- What data/information do we need to represent?
- What will be the main challenges for this implementation?

### Details

- New Classes? Which STL classes will be useful?

### Testing

- Test Cases?