

# CSCI-1200 Computer Science II — Fall 2006

## Lab 10 — Sudoku Sets

In this lab we will write a simple Sudoku puzzle solver using the STL `set` container class. Sudoku puzzles are typically played on a 9x9 grid in which some of the cells have been initialized and many of the cells left blank. The instructions are simply to: “Fill in the grid so that every row, every column, and every 3x3 box contains the digits 1 through 9”. If you are unfamiliar with Sudoku puzzles you may read about them here:

<http://en.wikipedia.org/wiki/Sudoku>

Please download the following files which contain a partial implementation of a Sudoku puzzle solver, and then turn off your internet connection.

[http://www.cs.rpi.edu/academics/courses/fall06/cs2/labs/10\\_sets/sudoku.h](http://www.cs.rpi.edu/academics/courses/fall06/cs2/labs/10_sets/sudoku.h)

[http://www.cs.rpi.edu/academics/courses/fall06/cs2/labs/10\\_sets/sudoku.cpp](http://www.cs.rpi.edu/academics/courses/fall06/cs2/labs/10_sets/sudoku.cpp)

[http://www.cs.rpi.edu/academics/courses/fall06/cs2/labs/10\\_sets/puzzles.txt](http://www.cs.rpi.edu/academics/courses/fall06/cs2/labs/10_sets/puzzles.txt)

### Overview of Standard Library Sets

- Sets are *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:  $O(\log n)$ !
- The second component of the `set` template (below), the compare function, is optional if `operator<` is defined for the key. For example: `set<string> words;`

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

### Set Operations

- Like other containers, sets have the usual constructors as well as the `size` member function.
- 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;
```

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

```
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. The second:

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

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.

- 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.

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

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

## Checkpoint 1

First familiarize yourself with the code and how we will be representing partially solved Sudoku puzzles. The 2D grid is stored using nested vectors. The elements of the inner vector are sets of integers, which represent the possible values for that cell. We know we’re done when each cell has exactly one possible value. If in the course of solving the puzzle we end up with no values in a particular cell, we know the puzzle is impossible to solve. The program handles 4x4 puzzles with 2x2 blocks and 9x9 puzzles with 3x3 blocks (and even larger puzzles too!)

Finish the implementation of the `Sudoku` class constructor and the `Sudoku::IsSolved` member function. At this point you should be able to compile and test your program on the input file (using file re-direction). The provided code for this assignment will print out the contents of each cell in the puzzle. At this point, every number (1-4 on the small puzzles and 1-9 on the large puzzles) will be an possible value for each unspecified cell in the puzzle.

**To complete this checkpoint:** Show a TA your program and how it behaves on the sample puzzles.

## Checkpoint 2

In the next step you will propagate information from the values in the grid which are known. For example if “7” is the only remaining choice for the cell  $(i, j)$  then we can remove “7” from all the other cells on the  $i$ th row and the  $j$ th column and the corresponding block. Follow the structure in the code provided and complete the `Sudoku::Propagate` member function. Test your program on the provided input.

**To complete this checkpoint:** Show a TA your completed program. Be prepared to discuss why it can only solve some of the puzzles. Think about how you would extend the program to solve some of the more difficult puzzles.