

Computer Science II — CSci 1200 — Sections 6-10
Week 10, Thursday Lecture — March 28, 2002
Program Design

Review from Monday and Lab

- `Vec<T>` — templated class that mimics the behavior of vectors.
- Dynamically allocated array bounds delimited by three pointers.
- Requires use of constructor, destructor and assignment operator

Today's Class

- Deep vs. shallow copy
- Memory management in the `Vec` class
- The Game of Life
- Program design
- Our implementation of the Game of Life
- Standard library sets

Deep Copy vs. Shallow Copy

- A *shallow copy* of an object is an assignment of each member variable in the old copy to the corresponding member variable in the new object.
 - This is the default when you don't write your own copying functions (assignment operator and copy constructor).
 - It is safe when the member variables are not dynamically allocated or when they have their own copy constructors / assignment operators.
 - All standard library containers implement copy constructors, assignment operators, and destructors.
- A *deep copy* of an object is code that explicitly copies all of the details of the old object in completely separate memory of the new object.

- Thus far you have done deep copying for `str` and `Vec<T>`.
- Deep copying becomes complicated for linked data structures such as lists and trees.

Memory Management in the Vec Class

- We used `new` and `delete`, the text uses `allocators` from the standard library.
- The line

```
data_start = new T[n];
```

creates an array of `n` objects of type `T`.

- The line

```
delete [] data_start;
```

calls the destructor on each of the objects of type `T`.

- This can be expensive and in general isn't necessary.
- Some classes outright prevent the use of a default constructor, so this could be illegal!
- Standard library `allocators` have functions to accomplish the same things as `new` and `delete`, without unnecessary calls to the default constructor and destructor on type `T`.
 - These are used in the text's version of the `Vec<T>` class.
- We studied the `Vec<T>` class without using `allocators` to emphasize intuition/simplicity over correctness.

Conway's Game of Life

- Two-dimensional grid of cells, which is potentially infinite 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 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.
- We will look at the URL

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

that has a discussion of the Game of Life and a simulation. The course web page has a link to this URL.

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.

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. Getting started:
 - (a) What major operations are needed and how do they relate to each other as the program flows?
 - (b) What important data / information needs to be represented? How should it be represented? Consider and analyze several alternatives.
 - (c) Develop a rough sketch of the solution, 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.
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.

Game of Life Implementation

- Representing the live cells in each generation is by far the most critical issue. It may take working through the details of the program to help.
- The rest of the representation depends on what additional features are needed. For example, my implementation detects generations that are repetitions of a previous generation.
- The remaining important considerations are the display of the board and the input of the initial configuration and user requests.

We will focus on these issues during lecture. The rest of these notes outline a standard library container that we haven't discussed yet, but you may find useful!

Standard Library Sets

- Ordered containers storing unique “keys”.
- An ordering relation on the keys, which defaults to the less than operator, is necessary.
- Because of this ordering requirement, STL sets are not truly mathematical sets.
- Sets are like maps except they have only keys — there are no associated values.
- Access to items in sets is extremely fast!

Important Note: This information is just here to help in problem solving. You will not be tested on standard library sets!

Set: Definition

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

```

- The set stores **constant** Keys at each node. You can use an iterator to change a key.
- Sets have the usual constructors as well as the **size** member function.

Set iterators

- Bidirectional
- Indicate const keys (as opposed to the pairs indicated by map iterators).

Set insert

- 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 indicates 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.

Set erase

- Two main `erase` functions:

```
void set<Key>::erase( iterator position );  
size_type set<Key>::erase( const Key& x );
```

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

- Note that the first `erase` does not return an iterator. This is different from the `vector` and `list` `erase` functions.
- The result of the second `erase` returns the number of entries removed, either 0 or 1.

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 pointing 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 pointing to the first entry in the map whose key is greater than the search key.