CS371p: Object-Oriented Programming

Fall 2013: 54010-11

Home

Life

Due: Thu, 21 Nov 2013, 10pm
70 pts, 7% of total grade.

Specification

Write a program to play the game of **Life**. **Life** is a simple simulation of cell automata.

Life contains a two-dimensional grid of cells. A cell can only be in one of **two** states: alive or dead. There are **two** kinds of cells: ConwayCells and FredkinCells.

Once the grid is manually populated with live and/or dead cells, the grid represents the **0th** generation of **Life**. After that, everything is automatic, and **Life** evolves from the **1st** to the **Nth** generation. A generation is simply the state of the grid (i.e. the layout of the live and dead cells).

Live ConwayCells are denoted with an asterisk, "*", and dead cells are denoted with a period, ".". A ConwayCell has **8** neighbors, if it's an interior cell, **5** neighbors, if it's an edge cell, and **3** neighbors, if it's a corner cell. The example below is of **1** ConwayCell that is alive surrounded by **8** ConwayCells that are dead:

```
...
.*.
...
```

ConwayCells do **not** have the notion of **age**, FredkinCells do. A FredkinCell's age is initially zero and **only** increments by one if the cell is alive and stays alive. Its age **never** goes down.

Live FredkinCells are denoted with their age, if their age is less than **10**, otherwise denoted with a plus, "+", and dead cells are denoted with a minus, "-". A FredkinCell has **4** neighbors, if it's an interior cell, **3** neighbors, if it's an edge cell, and **2** neighbors, if it's a corner cell. The example below is of **1** FredkinCell that is alive and of age **5** surrounded by **4** FredkinCells that are dead:

```
  -
- 5 -
  -
```

The rules for going from **one** generation to the next for ConwayCells are:

- a dead cell becomes a live cell, if exactly **3** neighbors are alive
- a live cell becomes a dead cell, if less than **2** or more than **3** neighbors are alive

The rules for going from **one** generation to the next for FredkinCells are:

- a dead cell becomes a live cell, if **1** or **3** neighbors are alive
- a live cell becomes a dead cell, if **0**, **2**, or **4** neighbors are alive

You will define the following classes:

- **AbstractCell**, an **abstract** class that is the base class of class **ConwayCell** and class **FredkinCell**
- **Cell**, a **handle** class that manages derived class objects of class **AbstractCell**
- **ConwayCell**, a **concrete** class
- **FredkinCell**, a **concrete** class
- **Life<T>**, a **concrete** class

Life will be instantiated with either ConwayCell, FredkinCell, or Cell.

If Life is instantiated with Cell, then when a FredkinCell's age is to become **2**, and only then, it becomes a **live** ConwayCell instead.

Create a good **OO** design by writing well-defined classes that are responsible for a specific and modular part of the solution. Avoid **getters**, **setters**, and **friends**, which are often signs of a bad design:

Getters and Setters

Create a **UML** diagram to represent the design. Use any **UML** editor that you like. The diagram **only** needs to show the associations and multiplicity between the classes.

For **all** projects, the **minimum** requirement for getting a **non-zero** grade is to write **standard-compliant C++ (-std=c++0x)**, to satisfy **all** of the **requirements** in the **table** below, including the precise **naming** of all the **files**, and to fill out the **Google Form**.

For this project, yet another additional **minimum** requirement for getting a **non-zero** grade is that your code **must** successfully pass **five** other students' acceptance tests.

You can earn **5 bonus pts**, if you work with a **partner** using **pair programming** and vouch for the fact that you worked on the project **together** for more than **75%** of the time.

Only **one** solution **must** be turned in for the **pair**. If **two** solutions are turned in, there will be a **10%** penalty, and the **later** one will be graded.

You **may not** use **malloc()**, **free()**, or **allocator**. You **will** only use **new** when calling **Cell's** constructor and in **clone()** and **mutate**. You **will** only use **delete** in **Handle**. You **may** use the **STL**.

Analysis

These are additional descriptions:

- Conway Life Wiki
- Paul Callahan's Life Page
- Rudy Rucker's CelLab
- Armchair Universe: An Exploration of Computer Worlds
- Glory Season
- Wheels, Life, and Other Mathematical Amusements
- Winning Ways: For Your Mathematical Plays, Volume 2
- Wikipedia

Tools

- Doxygen
- Git
- GitHub
- Gliffy
- Google Test (1.6.0)
- Valgrind
- yUML

Guides

- Git Cheat Sheet
- Git Guide
- Git Immersion
- Git Reference
- Google C++ Style Guide
- Try GitHub

Requirements

	Points	Description	Files	Submission
1	5 pts	**Git Repository** Set up a **private Git repository** at **GitHub**, named **cs371p-life**. Invite the grader to your repository. Commit **at least 5 times**. Commit once for each **bug** or **feature**. If you cannot describe your changes in a sentence, you are not committing often enough. Write meaningful commit messages and identify the corresponding **issue** in the **issue tracker** (below). Create a **tag** for important milestones (e.g. without a cache, with a lazy cache, etc.). Create a **log** of the commits. Push frequently. It is **your** responsibility to protect your code from the rest of the students in the class. If your code gets out, **you** are as guilty as the recipient of **academic dishonesty**.	Life.log	GitHub Turnin
2	5 pts	**Issue Tracker** The **GitHub** repository comes with an **issue tracker**. Create an issue for each of the **requirements** in this table. Create an issue for each **bug** or **feature**, both open and closed. Describe and label each issue adequately. Create **at least 10** more issues in addition to the **requirements** in this table.		GitHub
3	15 pts	**Unit Tests** The grader's **GitHub** account will have a public **Git repository** for **unit tests** and **acceptance tests**. It is **critical** that you clone the grader's public repo into a **different** directory than the one you're using for your private repo. Write unit tests **before** you write the code. When you encounter a bug, write a unit test that **fails**, fix the bug, and confirm that the unit test passes. Write **at least an average of 3** unit tests for **each** function. Tests corner cases and failure cases. Name tests logically. Push and pull the unit tests to and from the grader's repository. Prepend **<cs-username>**- to the file names at **GitHub** (i.e. **foo-TestLife.c++** and **foo-TestLife.out**). Reach consensus on the unit tests. You **must** use **Valgrind**.	TestLife.c++ TestLife.out	GitHub Turnin
4	15 pts	**Acceptance Tests** The grader's **GitHub** account will have a public **Git repository** for **unit tests** and **acceptance tests**. It is **critical** that you clone the grader's public repo into a **different** directory than the one you're using for your private repo. Write acceptance tests **before** your write the code. When you encounter a bug, write an acceptance test that **fails**, fix the bug, and confirm that the acceptance test passes. Create **at least 200 lines** of acceptance tests. Tests corner cases and failure cases. In your acceptance tests include **five** other students' acceptance tests (another **200 lines** for all **five**, at least). Push and pull **only** your acceptance tests to and from the grader's repository. Prepend **<cs-username>**- to the file names at **GitHub** (i.e. **foo-RunLifeConway.in**, **foo-RunLifeFredkin.in**, **foo-RunLifeCell.in**, and **foo-RunLife.out**). Reach consensus on the acceptance tests. You **must** use **Valgrind**.	RunLife.c++ RunLifeConway.in RunLifeFredkin.in RunLifeCell.in RunLife.out	GitHub Turnin
5	20 pts	**Implementation** Use **assert** to check **pre-conditions**, **post-conditions**, **argument validity**, **return-value validity**, and **invariants**. Worry about this **last**, but your program should run as **fast** as possible and use as **little** memory as possible.	AbstractCell.h AbstractCell.c++ Handle.h Cell.c++ Cell.h ConwayCell.h ConwayCell.c++ FredkinCell.h FredkinCell.c++ Life.h	GitHub Turnin
6		**Documentation** Use **Doxygen** to document the **interfaces**. The above documentation only needs to be generated for **AbstractCell.h**, **ConwayCell.h**,		

	5 pts	**FredkinCell.h**, **Cell.h**, and **Life.h**. Comment each function meaningfully. Use comments **only** if you need to explain the **why** of a particular implementation. Choose a coding convention and be consistent. Use good variable names. Write readable code with good indentation, blank lines, and blank spaces.	html/*	Turnin
7	5 pts	**Design** The **UML** diagram.	Life.pdf	GitHub Turnin
8		**Submission** Rename "**makefile.c++**" to "**makefile**". Fill out the **Google Form** and submit the **ZIP file** to **Turnin**.	makefile.c++ Life.zip	Google Turnin

Grader

Name	GitHub ID	GitHub Test Repository	Turnin ID	Turnin Project Folder	Google Form
Reza Mahjourian	rezama	cs371p-life-tests	reza	cs371ppj5	Google Form

Submission

Submit a single **ZIP** file, named **Life.zip**, to the grader's **Turnin** account, with the following files:

- html/*
- makefile
- AbstractCell.c++
- AbstractCell.h
- Cell.c++
- Cell.h
- ConwayCell.c++
- ConwayCell.h
- FredkinCell.c++
- FredkinCell.h
- Handle.h
- Life.h
- Life.log
- Life.pdf
- RunLife.c++
- RunLife.out
- RunLifeCell.in
- RunLifeConway.in
- RunLifeFredkin.in
- TestLife.c++
- TestLife.out

1760 reads