Assignment 2: ADT Client Applications

作业 2：ADT 客户端应用程序

Inspiration credit goes out to Mike Cleron from Google (random sentence generator) and Owen Astrachan from Duke University (word ladder). Jerry developed the maze generator. Assignment handout by Julie Zelenski and Jerry Cain.

灵感来源于谷歌的 Mike Cleron（随机句子生成器）和杜克大学的 Owen Astrachan（单词阶梯）。Jerry 开发了迷宫生成器。作业分发由 Julie Zelenski 和 Jerry Cain 提供。

Now that you've taken in the CS106 container classes, it's time to put these objects to use. In your role as client, the low-level implementation details have been dealt with and locked away as top secret so you can direct your attention on solving more interesting problems. Having a library of well-designed classes significantly extends the range of tasks you can tackle. This next assignment has you write three short client programs that leverage the container classes to do great things. The tasks may sound a little daunting at first, but given the power tools in your arsenal, each requires less than a hundred lines of code. Let's hear it for abstraction!

现在你已经学习了 CS106 容器类，是时候将这些对象付诸实践了。作为客户端，低级实现细节已经处理并锁定为最高机密，因此你可以将注意力集中在解决更有趣的问题上。拥有一个设计良好的类库显著扩展了你可以处理的任务范围。接下来的作业要求你编写三个短小的客户端程序，利用容器类来做一些伟大的事情。任务一开始可能听起来有些令人生畏，但考虑到你手中的强大工具，每个任务所需的代码行数都不到一百行。让我们为抽象喝彩！

The assignment has several purposes:

这个作业有几个目的：

To more fully explore the idea of using objects.

更全面地探索使用物体的想法。

To stress the notion of abstraction as a mechanism for managing data and providing functionality without revealing internal representations.

强调抽象作为管理数据和提供功能的机制，而不揭示内部表示的概念。

To become more familiar with C++ class templates.

更熟悉 C++类模板。

To gain practice with classic data structures such as the queue, vector, set, lexicon, and map.

为了熟悉经典数据结构，如队列、向量、集合、词典和映射。

Due: Wednesday, October 9^{th} at 5:00 p.m.

截止日期：星期三，十月 9^{th} 下午 5:00

Part I: Word Ladders [prose by Julie Zelenski]

第一部分：单词梯子 [朱莉·泽伦斯基的散文]

Leveraging the vector, queue, and lexicon abstractions, you'll find yourself well equipped to write a program to build word ladders. A word ladder is a connection from one word to another formed by changing one letter at a time with the constraint that at each step the sequence of letters still forms a valid word. For example, here is a word ladder connecting "code" to "data".

利用向量、队列和词汇抽象，您将能够编写一个程序来构建单词阶梯。单词阶梯是通过一次改变一个字母而形成的从一个单词到另一个单词的连接，约束条件是每一步字母序列仍然形成一个有效的单词。例如，这里是一个将"code"连接到"data"的单词阶梯。

```
code -> cade -> cate }->\mathrm{ date }->\mathrm{ data
```

Your program first asks the user to enter start and destination words. Provided the two words are of the same length, your program carries on and finds a word ladder between them if one exists. Using an algorithm known as breadth-first search, your program is guaranteed to find the shortest such sequence. The user can continue to request other word ladders until he or she is done.

您的程序首先要求用户输入起始词和目标词。如果这两个词的长度相同，您的程序将继续进行并查找它们之间的单词阶梯（如果存在的话）。使用一种称为广度优先搜索的算法，您的程序可以保证找到最短的序列。用户可以继续请求其他单词阶梯，直到他或她完成。

Here are some sample outputs of the word ladder program:

以下是单词阶梯程序的一些示例输出：

```
Welcome to the CS106 word ladder application!
Please enter the source word [return to quit]: work
Please enter the destination word [return to quit]: play
Found ladder: work fork form foam flam flay play
Please enter the source word [return to quit]: sleep
Please enter the destination word [return to quit]: awake
Found ladder: sleep sheep sheen shewn shawn sharn share sware aware awake
Please enter the source word [return to quit]: angel
Please enter the destination word [return to quit]: devil
Found ladder: angel anger agger egger eager lager leger lever level devel devil
```

Implementation Overview 实施概述

Finding a word ladder is a specific instance of a shortest path problem, where the challenge is to find the shortest path from source to target. Shortest path problems come up in a variety of situations-packet routing, robot motion planning, social networks, gene mutation, and travel. One approach for finding shortest paths relies on a classic algorithm known as breadth-first search. A breadth-first search stretches outward from the start in a radial fashion until it reaches some goal. For our word ladder problem, this means first examining those ladders that represent "one hop" (i.e. one changed letter) from the start. If any of these reach the destination, then woo! If not, the search now examines all ladders that add one more hop (i.e. two changed letters). By expanding the search at each step, all one-hop ladders are examined before two-hops, and three-

hop ladders are only considered if none of the one-hop or two-hop ladders worked out, and so forth, so that the algorithm is guaranteed to find the shortest one.

寻找单词阶梯是最短路径问题的一个特定实例，挑战在于找到从源到目标的最短路径。最短路径问题出现在各种情况下——数据包路由、机器人运动规划、社交网络、基因突变和旅行。寻找最短路径的一种方法依赖于一种经典算法，称为广度优先搜索。广度优先搜索从起点向外以放射状扩展，直到达到某个目标。对于我们的单词阶梯问题，这意味着首先检查那些代表“一个跳跃”（即一个字母变化）的阶梯。如果其中任何一个到达目的地，那么太好了！如果没有，搜索现在检查所有增加一个跳跃（即两个字母变化）的阶梯。通过在每一步扩展搜索，所有一个跳跃的阶梯在两个跳跃之前被检查，只有在没有一个跳跃或两个跳跃的阶梯有效的情况下，才考虑三个跳跃的阶梯，依此类推，因此算法保证找到最短的路径。

Breadth-first is typically implemented using a queue. The queue is used to store partial ladders that represent the remaining possibilities worth exploring. The ladders are enqueued in order of increasing length. The first enqueued elements are all the one-hop ladders, followed by the two-hop ladders, and so on. Because of the queue's first-in-first-out (forever more abbreviated FIFO) structuring, ladders will be dequeued in increasing (or nondecreasing) length. The algorithm operates by dequeueing the front ladder and determining if it's a solution. If it does, you have a complete ladder, and it is the shortest. If not, you take that partial ladder and extend it to reach words that are one more hop away, and enqueue those extended ladders onto the queue to be examined later. If you exhaust the queue of possibilities without ever finding a word ladder, you can assume no such ladder exists.

广度优先通常使用队列来实现。队列用于存储表示剩余可探索可能性的部分梯子。梯子按长度递增的顺序入队。第一个入队的元素都是一跳梯子，接着是两跳梯子，依此类推。由于队列的先进先出（简称 FIFO）结构，梯子将按长度递增（或非递减）顺序出队。算法通过出队前面的梯子并确定它是否是解决方案来运行。如果是，你就有了一条完整的梯子，并且它是最短的。如果不是，你就取出那条部分梯子并将其扩展到达到距离更远的单词，并将这些扩展的梯子入队以便稍后检查。如果你耗尽了可能性的队列而从未找到单词梯子，你可以假设不存在这样的梯子。

Let's make the algorithm a bit more concrete with some pseudo-code:

让我们用一些伪代码使算法更具体一些：

```
create initial ladder (with just the start word) and enqueue it
while queue is not empty
    dequeue first ladder from queue (this is shortest partial ladder)
    if top word of this ladder is the destination word
        return completed ladder
    else for each word in lexicon that differs by one char from top word
        and has not already been used in some other ladder
            create copy of partial ladder
            extend this ladder by pushing new word on top
            enqueue this ladder at end of queue
```

A few of these tasks deserve a bit more explanation. You will need to find all the words that differ by one letter from a given word. A simple loop can change the first letter to each of the other letters in the alphabet and ask the lexicon if that transformation results in a valid word. Repeat that for each letter position in the given word and you will have discovered all the words that are one letter away.

这些任务中的一些需要更多的解释。您需要找到与给定单词仅相差一个字母的所有单词。一个简单的循环可以将第一个字母更改为字母表中的每个其他字母，并询问词典该转换是否产生有效单词。对给定单词中的每个字母位置重复此操作，您将发现所有相差一个字母的单词。

Another restriction: you should not reuse words that have been included in a previously generated partial ladder. For example, if you have previously tried the ladder ***cat*** → **cot** → **cog** and are now processing **cat** → ***cot*** → **con**, you would find the word **cog** to be one letter away from con, so it looks like a potential candidate to extend this ladder. However, cog has already been reached in an earlier (and thus shorter) ladder, so there's no reason to consider it a second time. The simplest way to enforce this is to keep track of the words that have been used in any ladder and ignore those words when they come up again. This technique is also necessary to avoid getting trapped in an infinite loop by building a circular ladder such as **cat** → **cot** → **cog** → **bog** → **bat** → **cat**.

另一个限制：您不应重复使用已包含在先前生成的部分梯子中的单词。例如，如果您之前尝试过梯子 ***cat*** → **cot** → **cog**，而现在正在处理 **cat** → ***cot*** → **con**，您会发现单词 **cog** 离 con 只有一个字母，因此它看起来是一个潜在的候选者来扩展这个梯子。然而，cog 在之前的（因此更短的）梯子中已经被达到，所以没有理由第二次考虑它。强制执行这一点的最简单方法是跟踪在任何梯子中使用过的单词，并在这些单词再次出现时忽略它们。这种技术也是必要的，以避免通过构建循环梯子（例如 **cat** → **cot** → **cog** → **bog** → **bat** → **cat**）而陷入无限循环。

Since you need linear access to all of the items in a word ladder when time comes to print it, it makes sense to model a word ladder using a vector<string>. And remember that you can make a copy of a vector<string>by just assigning it to be equal to another via traditional assignment (e.g. Vector<string> clone = original).

由于在打印单词阶梯时需要线性访问所有项目，因此使用 vector来建模单词阶梯是有意义的。请记住，您可以通过传统赋值（例如 Vector clone = original）将 vector的副本赋值给另一个。

A few implementation hints

一些实施提示

It's all about leveraging the class libraries. You'll find your job is to coordinate the activities of various objects to do the search.

这完全是关于利用类库。你会发现你的工作是协调各种对象的活动以进行搜索。

The linear, random-access collection managed by a vector is ideal for storing a word ladder.

由向量管理的线性随机访问集合非常适合存储单词阶梯。

A Queue object is a FIFO collection that is just what's needed to track those partial ladders. The ladders are enqueued (and thus dequeued) in order of length so as to find the shortest option first.

队列对象是一个先进先出（FIFO）集合，正是跟踪那些部分梯子所需的。梯子按长度顺序入队（因此也按顺序出队），以便首先找到最短的选项。

As a minor detail, it doesn't matter if the start and destination word are contained in the lexicon or not. The sample application requires that the endpoints be English words, but you can let them be anything you want if you'd rather be more flexible. (The connectors need to be legitimate English words, of course.)

作为一个次要细节，起始词和目标词是否包含在词汇表中并不重要。示例应用程序要求端点为英语单词，但如果您希望更灵活，可以让它们是您想要的任何东西。（连接词当然需要是合法的英语单词。）

Word ladder task breakdown

单词阶梯任务分解

This program requires relatively little code, but it still benefits from a step-by-step development plan to keep things moving along.

该程序所需的代码相对较少，但它仍然受益于逐步开发计划，以保持进展。

Task 1-Try out the demo program. Play with the demo just for fun and to see how it works from a user's perspective.

任务 1 - 尝试演示程序。玩玩这个演示，仅仅是为了好玩，并从用户的角度看看它是如何工作的。

Task 2-Get familiar with our provided classes. You've seen vector and queue in lecture, but Lexicon hasn't gotten as much lecture attention. The reader outlines everything you need to know about the Lexicon, so make sure you read through Chapter 5 and the part about how to use the Lexicon class. (It's really very easy).

任务 2-熟悉我们提供的类。你在讲座中见过向量和队列，但词汇表没有得到太多讲座关注。读者概述了你需要了解的关于词汇表的所有内容，所以确保你阅读第 5 章以及关于如何使用词汇表类的部分。（这真的很简单）。

Task 3-Conceptualize the algorithm and design your data structure. Be sure you understand the breadth-first algorithm and the various data types you will be using. Since the items in the Queue are vectors, you have a nested template here. Deep!

任务 3-构思算法并设计数据结构。确保你理解广度优先算法以及你将使用的各种数据类型。由于队列中的项目是向量，因此你在这里有一个嵌套模板。深奥！

Task 4-Dictionary handling. Set up a Lexicon with the large dictionary read from our data file. Write a function that will iteratively construct strings that are one letter away from a given word and run them by the dictionary to determine which strings are words.

任务 4-字典处理。建立一个词汇表，使用从我们的数据文件中读取的大型字典。编写一个函数，迭代构造与给定单词相差一个字母的字符串，并通过字典检查哪些字符串是单词。

Task 5- Implement breadth-first search. Now you're ready for the meaty part. The code is not long, but it is dense and all those templates will conspire to trip you up. We recommend writing some test code to set up a small dictionary (with just ten or so words) to make it easier for you to test and trace your algorithm while you are in development. Do your stress tests using the large dictionary only after you know it works in the small test environment.

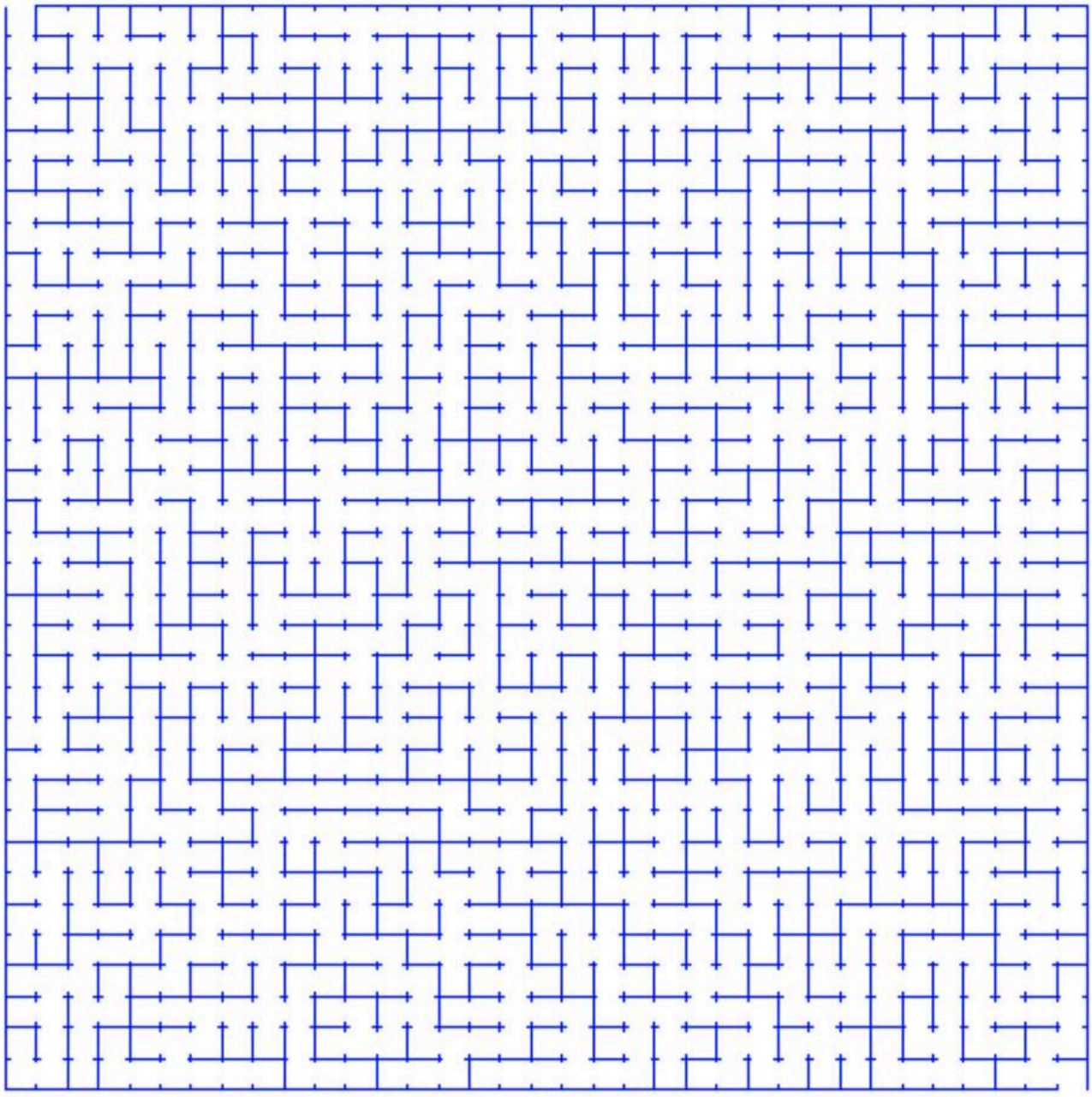
任务 5 - 实现广度优先搜索。现在你准备好进入重要部分了。代码不长，但很密集，所有这些模板会让你绊倒。我们建议编写一些测试代码，设置一个小字典（只有十个左右的单词），以便在开发过程中更容易测试和追踪你的算法。在你确认它在小测试环境中有效后，再使用大字典进行压力测试。

Part II: Generating Mazes [prose by Jerry]

第二部分：生成迷宫 [杰瑞的散文]

Next, you're to write a program that animates the construction of a maze. Here's an example of one your program might produce:

接下来，您需要编写一个程序来动画化迷宫的构建。以下是您的程序可能生成的一个示例：

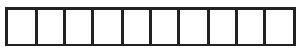


The goal is to create an **n** by **n** maze so there's one unique path from the upper left corner to the lower right one. Our algorithm for generating such mazes is remarkably simple to explain, and given the services of the Set and the Vector, is almost as easy to implement. (By the way, it's not really our algorithm—it's a simplified version of Kruskal's minimal spanning tree algorithm, which we'll more formally discuss later on when we talk about graphs.)

目标是通过 **n** 创建一个 **n** 迷宫，使得从左上角到右下角有一条唯一的路径。我们生成这种迷宫的算法非常简单易懂，并且借助集合和向量的服务，几乎同样容易实现。（顺便说一下，这实际上不是我们的算法——它是克鲁斯克尔最小生成树算法的简化版本，我们稍后在讨论图时会更正式地讨论。）

The basic idea has you start with a "maze" where nearly all walls are present.

基本思想是让你从一个"迷宫"开始，几乎所有的墙壁都存在。

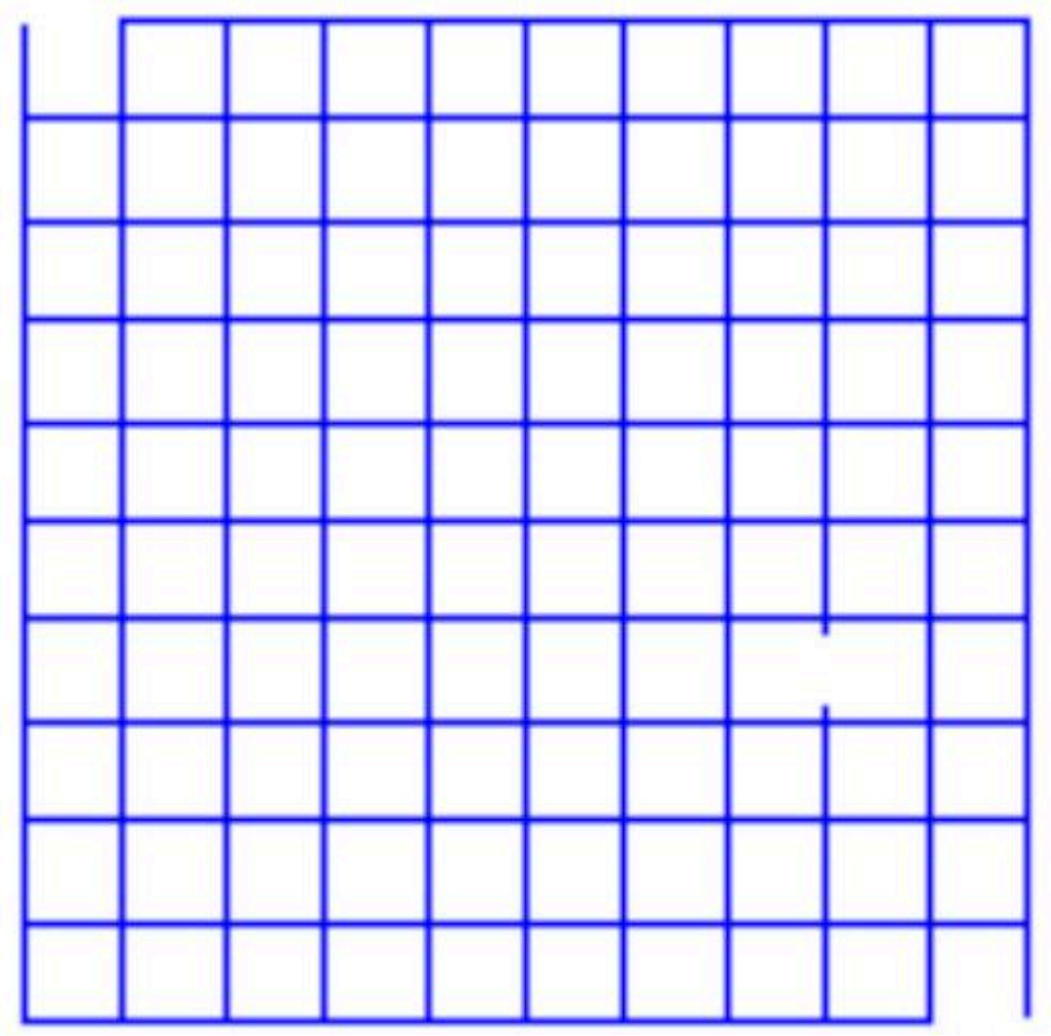


(Our example here happens to involve a 10×10 grid, but it generalizes to any dimension.)

(我们的例子恰好涉及一个 10×10 网格，但它可以推广到任何维度。)

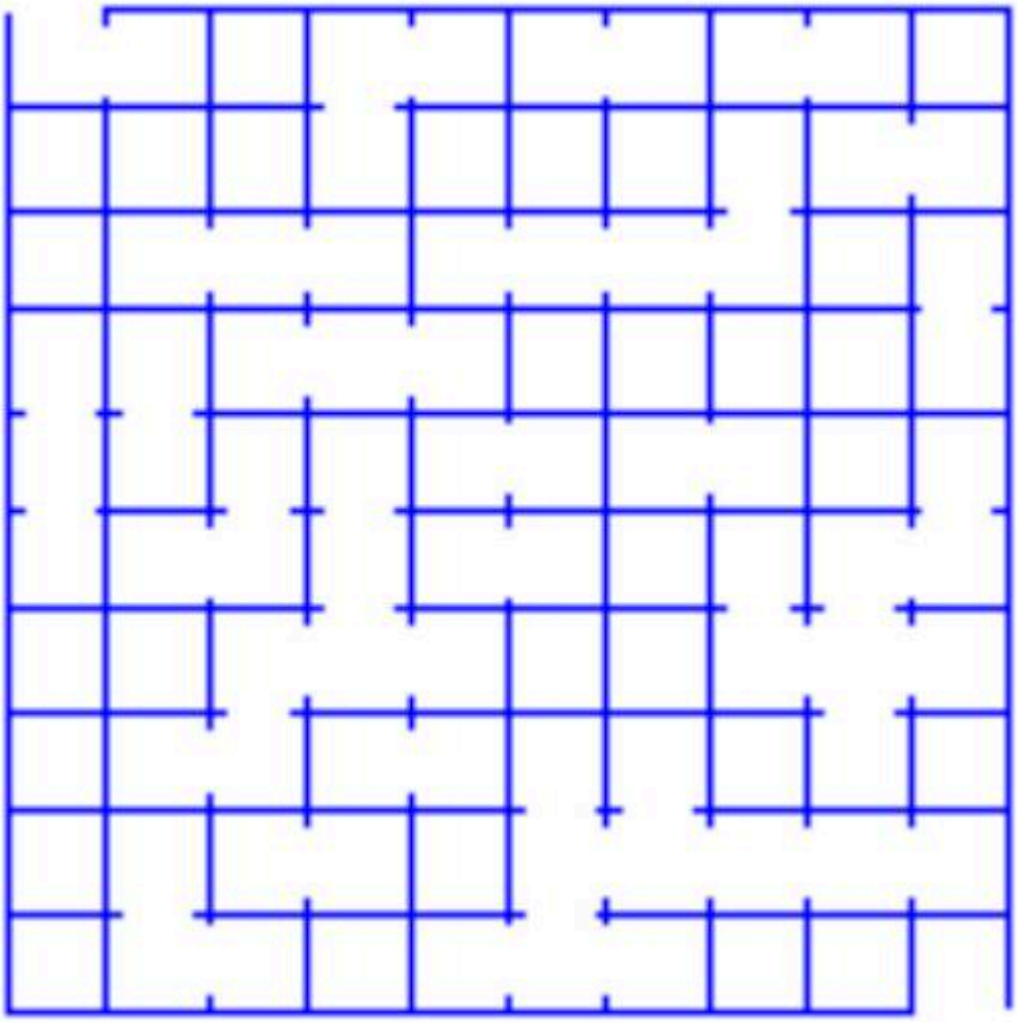
All internal walls are there, and the grid is divided into $10^2 = 100$ chambers. With each iteration, the algorithm considers a randomly selected wall that's not been considered before, and removes that wall if and only if it separates two chambers. The first randomly selected wall can always be removed, because all locations are initially their own chambers. That first wall might be the one that's now missing:

所有内部墙壁都在，网格被分成 $10^2 = 100$ 个房间。每次迭代，算法考虑一个之前未考虑过的随机选择的墙壁，并且仅当它分隔两个房间时才移除该墙壁。第一个随机选择的墙壁总是可以移除，因为所有位置最初都是它们自己的房间。那面第一个墙壁可能就是现在缺失的那面：



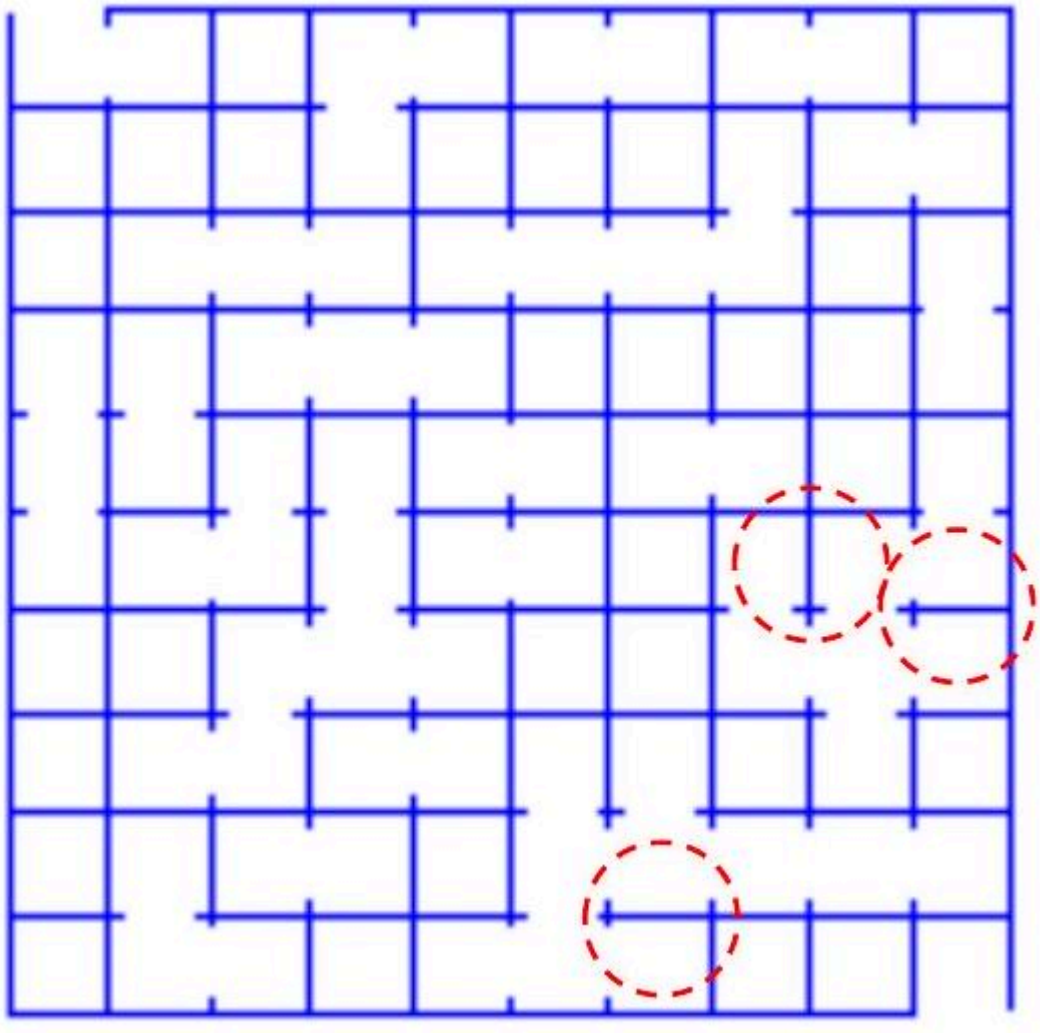
After a several more iterations, you'll see a good number of randomly selected walls have been removed:

经过几次迭代后，您会看到许多随机选择的墙壁已被移除：



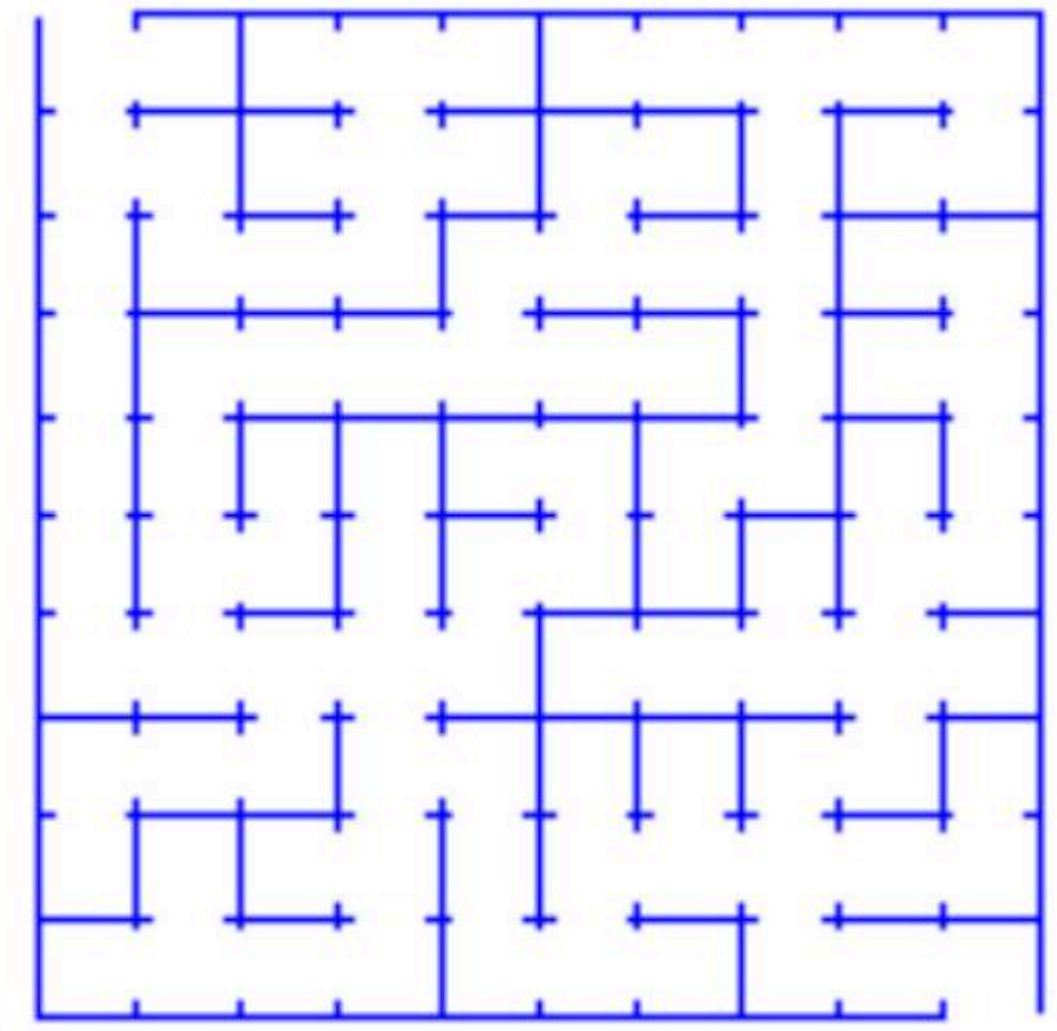
The working maze pictured above includes a few walls that should not be removed, because the cells on either side of each of them are part of the same chamber. I've gone ahead and circled some of them here (save for the circles, it's the same picture you see above):

上面所示的工作迷宫包括一些不应移除的墙壁，因为它们两侧的单元格是同一个房间的一部分。我已经在这里圈出了其中的一些（除了圆圈，和上面看到的图片是一样的）：



All walls are considered exactly once in some random order, and if as you consider each wall you notice that it separates two different chambers, you merge the two into one by removing the wall. Initially, there are 100 chambers, so eventually 99 walls are removed to leave one big, bad maze.

所有墙壁都以某种随机顺序被考虑一次，如果在考虑每面墙时你注意到它将两个不同的房间分开，你就通过移除这面墙将它们合并为一个。最初有 100 个房间，因此最终移除 99 面墙，留下一个巨大的迷宫。



Notice the three walls I circled in the previous snapshot are still present. We can't tell from the photos when (or even if) they were considered, but we shouldn't be surprised they're still there in the final maze.

请注意我在之前快照中圈出的三面墙仍然存在。我们无法从照片中判断它们何时（甚至是否）被考虑过，但我们不应该对它们在最终迷宫中仍然存在感到惊讶。

Maze Generation Pseudo-code

迷宫生成伪代码

Here is a high-level description of my own solution:

这是我自己解决方案的高级描述：

```
choose maze dimensions
generate all walls, draw them, and shuffle them
generate all chambers, where each chamber includes one unique cell
for each wall in randomly-ordered-walls
    if wall separates two chambers
        remove wall
```

We're going to let you tackle this one all by yourself, relying on the pictures and the pseudocode above to get through it. Here are a few other details worth mentioning:

我们将让你独自解决这个问题，依靠上面的图片和伪代码来完成。这里还有一些值得提及的其他细节：

All of the graphics routines are documented in maze-graphics.h. Everything provided is pretty straightforward. If you want to make changes to these files, then go for it.

所有图形例程都在 maze-graphics.h 中有文档说明。提供的一切都很简单明了。如果你想对这些文件进行更改，那就去做吧。

There's a small header file called maze-types. *h*, which defines cell and wall types. You should use these definitions. Note that l've already defined operator< for both cell and wall. Those are needed if you're to store cells and walls in Sets.

有一个名为 maze-types 的小头文件，它定义了单元格和墙的类型。你应该使用这些定义。请注意，我已经为单元格和墙定义了 operator<。如果你要将单元格和墙存储在集合中，这些是必需的。

Our solution uses the Set to help model a chamber and the Vector to maintain an ordered list of walls and chambers. In particular, we didn't use a Grid, so you shouldn't be worried if you don't use one.

我们的解决方案使用集合来帮助建模一个房间，并使用向量来维护墙壁和房间的有序列表。特别是，我们没有使用网格，所以如果你不使用网格也不必担心。

You shouldn't need to modify anything other that maze-generator.cpp. The other cpp files and all interface files should be fine as. If you do change them, then be sure to submit them and detail how you changed them so we know what to look for.

您不需要修改除了 maze-generator.cpp 以外的任何内容。其他 cpp 文件和所有接口文件应该没问题。如果您确实更改了它们，请务必提交并详细说明您是如何更改它们的，以便我们知道要查找什么。

Part III: Random Sentence Generator [prose by Julie Zelenski]

第三部分：随机句子生成器 [朱莉·泽伦斯基的散文]

Over the past four decades, computers have revolutionized student life. In addition to providing entertainment and distraction, computers have also facilitated all sorts of student work. One important area of student labor that has been painfully neglected is the task of filling up space in papers, Ph.D. dissertations, extension requests, etc. with important sounding and somewhat grammatically correct random sequences.

在过去的四十年里，计算机彻底改变了学生生活。除了提供娱乐和分心，计算机还促进了各种学生工作。一个被痛苦忽视的重要学生劳动领域是填充论文、博士论文、延期请求等中的空间，使用听起来重要且在语法上有些正确的随机序列。

Neglected, that is, until now.

被忽视，直到现在。

The Random Sentence Generator is a marvelous piece of technology that creates random sentences from a structure known as a context-free grammar. A grammar is a construct describing the various combinations of words that can be used to form valid sentences. There are profoundly useful grammars available to generate extension requests, Star Trek plots, James Bond manuscripts, "Dear John" letters, and more. You can even

create your own grammar! Let's show you the value of this practical and wonderful tool:

随机句子生成器是一项奇妙的技术，它根据一种称为无上下文文法的结构生成随机句子。文法是描述可以用来形成有效句子的各种单词组合的构造。有许多极其有用的文法可用于生成扩展请求、星际迷航情节、詹姆斯·邦德手稿、“亲爱的约翰”信件等。您甚至可以创建自己的文法！让我们向您展示这个实用而奇妙的工具的价值：

Tactic #1: Wear down the TA's patience.

战术 #1：消耗目标受众的耐心。

I need an extension because I had to go to an alligator wrestling meet, and then, just when my mojo was getting back on its feet, I just didn't feel like working, and, well I'm a little embarrassed about this, but I had to practice for the Winter Olympics, and on top of that my roommate ate my disk, and right about then well, it's all a haze, and then my dorm burned down, and just then I had tons of midterms and tons of papers, and right about then I lost a lot of money on the foursquare semi-finals, oh, and then I had recurring dreams about my notes, and just then I forgot how to write, and right about then my dog ate my dreams, and just then I had to practice for an intramural monster truck meet, oh, and then the bookstore was out of erasers, and on top of that my roommate ate my sense of purpose, and then get this, the programming language was inadequately abstract.

我需要一个延期，因为我不得不去参加鳄鱼摔跤比赛，然后，就在我的状态恢复的时候，我就是不想工作，嗯，我对此有点尴尬，但我不得不为冬季奥运会练习，除此之外，我的室友吃了我的磁盘，就在那时，嗯，一切都模糊了，然后我的宿舍着火了，就在那时我有很多期中考试和很多论文，就在那时我在四方半决赛上损失了很多钱，哦，然后我不断做关于我的笔记的梦，就在那时我忘记了怎么写，就在那时我的狗吃了我的梦想，就在那时我不得不为一场校内怪兽卡车比赛练习，哦，然后书店没有橡皮擦，除此之外，我的室友吃掉了我的目标感，然后你听着，编程语言的抽象程度不够。

Tactic #2: Plead innocence.

策略 #2：请求无罪。

I need an extension because I forgot it would require work and then I didn't know I was in this class.

我需要一个延期，因为我忘了这需要工作，然后我不知道我在这个班级。

Tactic #3: Honesty. 策略 #3：诚实。

I need an extension because I just didn't feel like working.

我需要一个延期，因为我就是不想工作。

What is a grammar? 什么是语法?

A grammar is a set of rules for some language, be it English, Java, C++, or something you just invent for fun. 😀 If you continue to study computer science, you will learn much more about languages and grammars in a formal sense. For now, we will introduce to you a particular kind of grammar called a context-free grammar (CFG).

语法是一组规则，用于某种语言，无论是英语、Java、C++，还是你为了乐趣而发明的东西。😀 如果你继续学习计算机科学，你将以正式的方式了解更多关于语言和语法的知识。现在，我们将向你介绍一种特定类型的语法，称为上下文无关文法（CFG）。

Here is an example of a simple CFG for generating poems:

这是一个生成诗歌的简单 CFG 示例：

```
<start>
1
The <object> <verb> tonight.
<object>
3
waves
big yellow flowers
slugs
<verb>
3
sigh <adverb>
portend like <object>
die <adverb>
<adverb>
2
warily
grumpily
```

According to this grammar, two syntactically valid poems are “The big yellow flowers sigh warily tonight.” and “The slugs portend like waves tonight.” Essentially, the strings in brackets (<>) are variables that expand according to the rules in the grammar.

根据这个语法，两个语法上有效的诗句是“今晚，大黄花小心地叹息。”和“今晚，蛞蝓像波浪一样预示。”本质上，括号中的字符串（<>）是根据语法规则扩展的变量。

More precisely, each string in brackets is known as a nonterminal. A nonterminal is a placeholder that will expand to another sequence of words when generating a poem. In contrast, a terminal is a normal word that is not changed to anything else when expanding the grammar. The word terminal is supposed to conjure up the image that it’s something of a endpoint, and that no further expansion is possible.

更准确地说，括号中的每个字符串被称为非终结符。非终结符是一个占位符，在生成诗歌时会扩展为另一组单词。相比之下，终结符是一个正常的单词，在扩展语法时不会被更改为其他任何东西。终结符这个词应该让人联想到它是某种终点，并且不可能再进行进一步的扩展。

A definition consists of a nonterminal and a list of possible productions (or expansions). There will always be at least one and potentially several productions for each nonterminal. A production is just a text string of words, some of which themselves may be non-terminals. A production can even be the empty string, which makes it possible for a nonterminal to evaporate into nothingness.

定义由一个非终结符和一个可能的产生式（或扩展）列表组成。每个非终结符总会至少有一个，可能还有多个产生式。产生式只是一个单词的文本字符串，其中一些本身可能是非终结符。产生式甚至可以是空字符串，这使得非终结符可以消失得无影无踪。

An entire definition is summarized within a grammar text file as:

整个定义在语法文本文件中总结为：

```
<verb> }\quad\Leftarrow\mathrm{ the first line names the nonterminal and is  i
3}\Leftarrow\mathrm{ the second line is always the number of possible expansions
sigh <adverb> }\Leftarrow\mathrm{ the third line is the first possible expansion
portend like <object> & followed by another expansion if there is a second one
die <adverb> followed by another expansion if there is a third one, etc
\Leftarrowfor readability, there's a blank line after each definition, including
```

You always begin random sentence generation with the single non-terminal <start> as the working string, and iteratively search for the first nonterminal [1] and replace it with any one of its possible expansions (which may and often will include its own nonterminals). Repeat the process over and over until all nonterminals are gone.

您总是以单个非终结符 <start> 作为工作字符串开始随机句子生成，并迭代地搜索第一个非终结符 [1]，并用其可能的扩展之一替换它（这可能并且通常会包括它自己的非终结符）。重复这个过程，直到所有非终结符都消失。

```
<start>
The <object> <verb> tonight. // expand <start>
The big yellow flowers <verb> tonight. // expand <object>
The big yellow flowers sigh <adverb> tonight. // expand <verb>
The big yellow flowers sigh warily tonight. // expand <adverb>
```

Since we choose productions at random, a second generation will almost certainly produce a different sentence.

由于我们随机选择作品，第二代几乎肯定会产生不同的句子。

Your program should repeatedly prompt the user for a grammar file (understood to be in the grammars subdirectory), read in the grammar, and generate three random sentences. Only when the user hits return without actually typing in anything should you end the program.

您的程序应反复提示用户输入一个语法文件（理解为在 grammars 子目录下），读取语法，并生成三句随机句子。只有当用户直接按回车而不输入任何内容时，您才应结束程序。

You may assume all grammar files are well formed, and you needn't worry about word wrap as you print super-long sentences. Using the sample application as a guide, you are to make all design and implementation decisions.

您可以假设所有语法文件都是格式良好的，您无需担心在打印超长句子时的换行。以示例应用程序为指南，您需要做出所有设计和实现决策。

If you have the interest and energy, invent a new grammar or two. If you think others would like them, email them to jerry@cs. stanford. edu and I'll post them to the course website.

如果你有兴趣和精力，可以发明一两种新的语法。如果你认为其他人会喜欢它们，请将它们发送到 jerry@cs.stanford.edu，我会将它们发布到课程网站上。

[1] This problem could also be solved using recursion. We ask that you suppress your desire to use recursion and implement it iteratively.

[1] 这个问题也可以通过递归来解决。我们要求您抑制使用递归的欲望，改为以迭代方式实现。