CSC A48 - 第3单元 - 组织、存储和访问信息 1.- 数据的问题

现在我们对如何在C语言中实现事物有了足够的了解，是时候把注意力转向我们在本课程中要研究的第一个真正有趣的想法了。

我们在这里的目标是了解如何处理一个一般的问题：如何在一个基于计算机的系统中组织、存储和访问信息。下面是一些你可能希望用计算机来存储和组织的东西。

- 文本（文件、文章、书籍等）。
- 音乐（一般来说，声音文件--采访、纪录片、声音剪辑等）。
- 图片（多种格式）
- 视频
- 关于人的信息--取决于你的应用，如：。
    - ACORN中的学生信息
    - 在线商店的客户记录
    - 医院的个人信息和病人病历
    - 浏览偏好（你知道你的浏览器对你有什么了解吗？）等。
- 计算结果
    - 比如说
        - 预测（明天的天气，股票市场，选举结果）
        - 数据库查询（找到所有在CSCA48，Lec0002注册的学生）
        - 从一个地方开车到另一个地方的最短路线（例如，在你最喜欢的地图应用程序中）。
        - 等等。

这些只是非常少的例子，让你了解到我们会发现需要仔细思考的广泛情况。***一般来说，在使用计算机解决问题时，你必须做的第一件事就是仔细思考***。

- 我们要储存什么类型的信息
- 我们需要管理多少东西（处理你手机里的几百张图片和处理谷歌图片上的几亿张图片是不同的问题）。
- 数据将如何被访问和使用，以及由谁访问和使用（效率、存储使用、安全和访问控制、数据备份和冗余）。
- 如何组织数据，使其在程序中易于管理（实际的程序级数据表示和操作）。

在这部分课程中，你将学习的是程序级的数据组织和操作。我们将看到如何使用C语言提供的简单数据类型，以建立更丰富、更有用、更灵活、更容易使用的数据容器，可以用来存储、组织。

你可以访问和管理几乎所有你希望存储在计算机内的东西。

如果你以后想了解如何对信息进行建模，以及信息的建模如何影响为处理信息而编写的软件的设计（这是*软件设计*的主要课题之一，*CSCB07*），这里涉及的概念和技术将是你需要的基础。这也是建立数据库的基础。如今，几乎所有的应用都需要数据库--
从烹饪食谱应用（会有某种形式的可搜索食谱数据库），到各种在线和离线业务的客户信息系统。数据库很吸引人，所以如果你对它们的工作原理感到好奇，别忘了看看我们的*数据库介绍CSCC43*。

旁白：外面有多少数据？

你可能听说过*"大数据"*这个词。这是一个相当缺乏信息的术语，因为它真的没有告诉你到底有多少数据符合*"大"的*标准，或者为什么它应该是这样的。这里有一些事实（注意，有些估计是一两年前的！），让你了解在不同的应用和不同的用途中有多少数据。

1 -
谷歌代码库（2016年）包括大约10亿个文件，有大约3500万次提交的历史，横跨谷歌整个18年的存在。存储库包含[86TBa]的数据，包括900万个独特的源文件中约20亿行的代码。
(来源：https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext)

2 -
*谷歌存储了多少数据？*显然，这个问题的答案是：*没有人（也许包括谷歌？）真正知道*。但这里有一个相当有趣的分析：https://what-if.xkcd.com/63/。所提供的分析指出"让我们假设谷歌的存储容量为15艾字节，或15,000,000,000,000,000字节"。这是一个*不无道理的估计*，而且*很可能是在*几年前谷歌实际可用的存储量的*正确数量级上*，当这个问题出现的时候

3 - 每天上传到facebook的图片数量。根据。
https://www.omnicoreagency.com/facebook-statistics/
'每天有3.5亿张照片被上传，每小时有1458万张照片上传，每分钟有24.3万张照片上传，每秒钟有4000张照片上传。

来自同一网站：https://www.omnicoreagency.com/instagram-statistics/
我们发现。
'到目前为止，已经有超过500亿张照片上传到Instagram。'比萨是全球Instagram上最多的食物，其次是寿司。

4 -
全世界每天产生多少*新*信息？根据*福布斯*：https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind- blowing-stats-everyone-should-read/#6a951bf860ba

以我们目前的速度，每天有2.5万亿字节的数据被创造出来。

'平均而言，谷歌现在每秒钟处理超过40,000次搜索（每天有35亿次搜索）！'。

'我们发送了1600万条短信'（*每分钟！*）'每分钟有10
3,447,520封垃圾邮件发送
哦，你应该看一下这里：http://www.everysecond.io/youtube)

插图1：插图1：这是数据中心内部的样子。在世界的某个地方，一个与此类似的数据中心正在储存这份文件的副本。照片。全球接入点，公共领域

关于资料来源以及你是否应该相信你所读的东西的简要说明。

你可能已经注意到，上面列出的 *"事实"*来自非常不同的来源。1）来自一份研究出版物，2）和3）来自在线博客/网站，4）来自杂志文章的在线版本。

永远记住：在你接受你读到的陈述之前--
*即使是在讲义中！*你应该始终思考*它的来源，*以及*提供什么证据来支持它。*

一篇好的科学期刊或会议文章是值得信赖的。它经历了一个修订和审查的过程，*这些*科学家*与他们正在审查的工作的作者没有关系*。这意味着大的错误或事实性错误通常会在文章发表前被发现并得到纠正。

有信誉的出版物几乎是值得信赖的--他们为保持其作为可信赖的信息来源的声誉投入了大量资金

对博客和网站应持谨慎态度--
你会注意到，我上面列出的网站提供了他们收集事实和数字的来源的参考和链接，*你应该一直寻找这个*。任何网站、博客或帖子，如果陈述了一个事实，但*没有参考有信誉的来源，如科学文章、有信誉的出版物或组织，或公开可用/可验证的调查，*在*你能*独立*验证*信息之前，必须采取健康的怀疑态度。

你们正在训练成为不同领域的专业人士--
你们必须*始终*确保你们所接受并纳入你们对世界如何运作的认识的信息是*准确的、可核实的，而且就你们所能检查的而言是正确的。*

2.- 如何建造一个便当盒

我们知道如何使用C语言中提供的标准数据类型，然而大多数有趣的应用都需要记录比几个整数或浮点数，甚至几个字符串更复杂的数据。目前的问题是如何*设计*和*实现*一种*新的数据类型*，这种类型是C语言中所没有的，而且可以代表更丰富的信息单元。

打个比方--
一顿好的饭菜不是由单一的物品组成的，比如西兰花（顺便说一下，你应该吃西兰花，它对你有好处！），而是由许多不同的成分组成，以一种方式组合起来，成为一顿好的饭菜。单独的成分是你在任何商店都可以找到的成分，但完成的饭菜要有趣得多。如果你去过日本餐厅，那么你可能已经看到了一种饭菜，它是我们现在要应用于数据类型的过程的一个很好的例子。便当盒"。

插图2：便当盒--
饭菜是由各个组成部分组成的，每个组成部分都在自己的容器中，排列起来相互补充，每个组成部分都需要完成饭菜。照片：miheco - Flickr, CC - SA 2.0

我们在这里的任务是弄清楚如何表示一个项目的信息，这些信息比单一的数据类型所能容纳的还要复杂。例如，如果我们要写一个应用程序来存储关于电影的信息，我们可能需要存储。

每部电影需要的信息。

- 标题
- 年
- 理事
- 工作室
- 烂番茄[1]分数
...等等。

有几个单独的组件，每个组件都有自己的数据类型--可能有

字符串、整数、浮点数，等等。我们怎么能在C语言中做到这一点呢？

考虑到我们目前所了解的情况，只使用C语言的标准数据类型，我们需要为构成电影信息的每个不同组成部分创建单独的数组。

- 一个电影标题的字符串数组
- 年的一个整数数组
- 一个导演姓名的字符串数组
- 工作室的一个字符串数组
- 烂番茄评分的一列浮动装置

现在我们可以编写代码，让用户用电影的信息来填充这些数组。

问题。你认为以这种方式存储我们需要的信息有什么优势和劣势？

从概念的角度来看，以及从易于实现与电影等复杂实体的数据相关的代码的角度来看，如果我们能够*将与一部电影*相关的信息*捆绑*在一个*单一的数据项*中，包含我们需要存储的关于电影的所有信息，那就更好了。

在C语言中，我们可以定义我们自己的*复合数据类型*，也被称为*复合数据类型*，它相当于一个*便当盒的*程序。它们是由*一组现有的数据类型*组成的，每一个数据*类型*都提供了关于一个数据项的所需信息，而所有的数据类型都需要用来描述我们的数据项。

电影是相当复杂的数据项（如果你看一下IMDB上关于一部电影的信息，你会发现各种各样的东西）。因此，在本节的例子中，我们将使用一个更简单的例子。假设我们正在写一个小程序来*跟踪餐厅的评论*。假设我们将我们的应用程序称为*Kelp*。

我们需要跟踪的基本信息单位是一个单一的餐厅评论，其中包括。

- 餐厅名称（一个字符串）
- 餐厅地址（一个字符串）
- 评论得分（一个整数，比方说1-5）。

让我们看看我们如何在C语言中建立一个*便当盒*，保存处理一个餐厅评论所需的信息。

```
typedef struct                    Restaurant_Score// 我们正在声明一个新的类型。
{
       char restaurant_name[1024];
       char restaurant_address[1024];
       int score;

} review;                         //我们的新类型将被命名为 "回顾"。
```

上面的小段代码的工作原理如下。

```
Typedef struct Restaurant_Score
```

这告诉编译器我们正在定义一个新的数据类型（*typedef*），该数据类型是复合型的（*struct*），复合型的名字是 "*Restaurant_Score*"，新的数据类型将被称为"*Review*"。这实际上是定义了一个新的*便当盒*，其中包含两个字符串和一个整数。这些*组件*或*部分*中的每一个都被称为一个***字段***。因此，我们新的复合数据类型有***三个字段***，一个*餐厅名称字段，一个餐厅地址字段，*和一个*分数字段。*

此后，我们可以*声明*这个新的复合类型的变量并在我们的程序中使用它们

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define MAX_STRING_LENGTH 1024

typedef struct                    Restaurant_Score// 我们正在声明一
个新的类型。
{
       char restaurant_name[MAX_STRING_LENGTH];

}审查。                           // 我们的新类型将被命名为 "回顾"。

空白的main(void)
{
      审查修订。                  // 声明一个 "回顾 "类型的变量

      // 让我们为'rev'中的信息赋值。
      // 一个复合数据类型的各个组成部分是用以下方式引用的
      //'.'运算符。

      // Score只是一个int，所以我们可以这样做。
      rev.score=4。
```

```
    // strcpy(rev.restaurant_name, "A nice restaurant with good food");
    strcpy(rev.restaurant_address, "Somewhere in Scarborough")。

    printf("This review has: name=%s, address=%s, score=%d\n",\
         rev.restaurant_name,rev.restaurant_address,rev.score);

}
```

上面的程序中发生了很多事情，所以让我们一步步来看看。首先

，注意顶部的<*include*>语句。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
```

头文件*stdio.h*包含了标准输入/输出函数的定义，包括我们将用来从键盘上读取字符串和整数的函数；stdlib.*h*应该在上一节中为你所熟悉，它提供了一些标准函数和常量定义（例如***NULL***在此定义），*string.h*是字符串操作库。

下一句话是我们以前没有见过的。

```
#define MAX_STRING_LENGTH 1024
```

这一行定义了一个叫做 "*MAX_STRING_LENGTH* "的*常量*。在C语言中，通常在程序的顶部使用'*#define*'语句来定义代码中要用到的常量。通常，这些常量的名称都是大写的。此后，只要编译器在你的代码中发现'*MAX_STRING_LENGTH*'，它就会取代'*1024*'并使用它。

下一块是我们的*类型定义*。

```
typedef struct                    Restaurant_Score// 我们正在声明一个新的类型。
{
       char restaurant_name[MAX_STRING_LENGTH];
       char restaurant_address[MAX_STRING_LENGTH];
       int score;

}Review;                                  //我们的新类型将被命名为 "回顾"。
```

这将定义我们的*便当盒*，用一个字符串表示餐厅名称，用一个字符串表示地址，用整数表示评论分数。在*typedef之后的*任何时候，我们都可以像对待普通的C类型一样，声明我们新的数据类型的变量，比如说

审查	修订。
评论	`ten_reviews[10];`

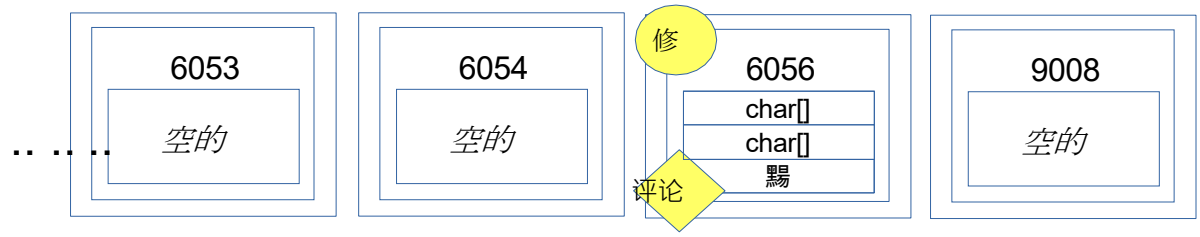
```
审查        *rp;
```

上面的例子声明（按顺序），一个名为'*review*'的变量。一个名为'*ten_reviews*'的10条评论的数组，以及一个指向'*Review*'类型变量的指针。

我们程序的下一部分声明了一个名为 "*one_review* "的 "*Review* "类型的变量。

```
评论 one_review;
```

这就是它在记忆中的样子。



C语言的好处是，一旦一个新的类型被声明，它就会像其他类型一样被对待。因此，"*回顾* "变量 "*rev* "得到一个盒子--大到足以存储它所需要的东西，就像其他变量一样在内存的某个地方。这个盒子被标记为 "*rev*"，它的类型是 "*回顾*"。这个盒子现在包含了一个东西的集合（我们知道里面有两个字符串和一个int），但除此之外它只是多了一个盒子。

当我们编译和运行上面的代码时，我们得到的是。

```
.../a.exe
该评论有： name=一家不错的餐厅，食物不错，地址=士嘉堡的某个地方，分数=4分
```

从上面的例子中需要注意的事情。

- 定义一个新的数据类型并不困难，C语言支持的任何混合类型。我们可以用它来组织我们的程序需要处理的复杂项目（例如，电影、书籍等）的数据。
- 一旦我们定义了一个新的数据类型，我们就可以在我们的代码中像其他类型一样使用它。这包括将我们的新数据类型作为更复杂的*便当盒*的一部分。例如，我们可以定义一个新的数据类型，包含 "*回顾* "类型的项目。一个盒子就是一个盒子。所以对于C语言来说，这是很好的。
- 赋值或访问复合类型变量中的数据是通过'.'操作符完成的。
- 你可能已经注意到，程序中调用*printf()*的那一行被分成了两行，而且在这个调用的第一行末尾有一个'\'字符。在C语言中，你可以通过在一行的末尾使用'\'字符将任何语句

分成多行。它允许你在多行中写长的语句（这使你的代码更容易阅读），而不是一个很

长线。

练习。编写一个复合数据类型的定义，将 *"评论*"扩展到包括提交评论的人的名字，餐厅的电话号码，以及餐厅网页的链接。*仔细思考你将使用的数据类型，以及这些新的信息如何在评论应用程序中使用*（这将对你应该使用的数据类型产生影响）。

3.- 在函数之间传递复合类型

像其他变量一样，你会发现你需要将复合数据类型作为参数传递给函数，和/或从函数中返回这些复合类型。这样做的方式与你在函数之间传递标准C类型的方式是*相同的*。

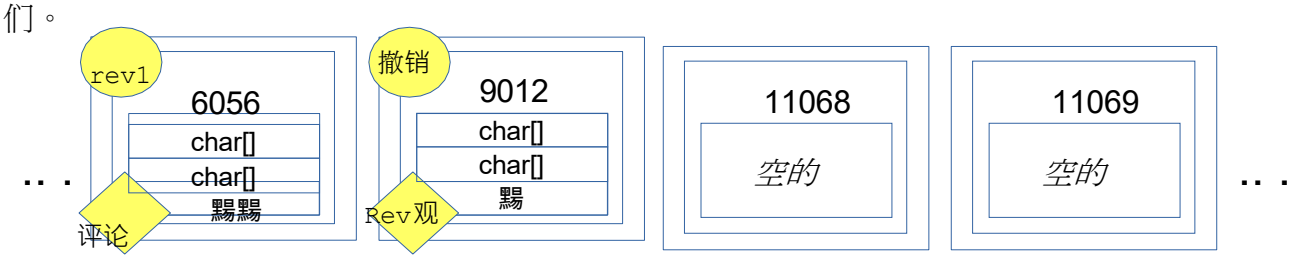
假设我们定义一个函数如下。

```
Reviewchange_score  (Review input, int new_score)
{
      input.score=new_score;
      return input;
}
```

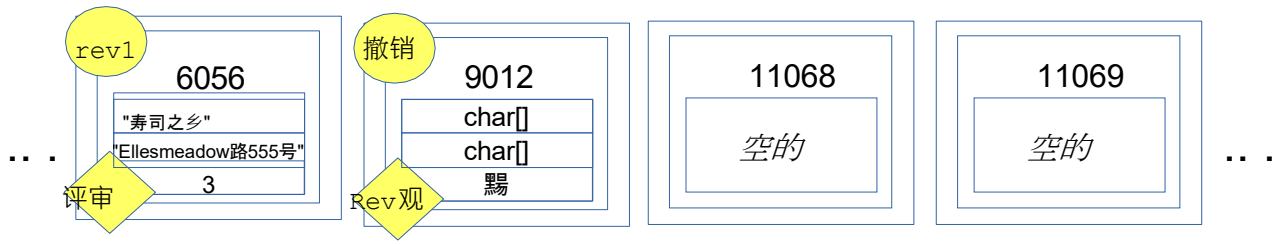
这个函数的输入参数是一个 *"Review* "类型的变量，一个名为 *"new_score "的int*，并返回一个 "*Review* "类型的结果。现在，假设在*main()*中我们做了这样的事情。

```
审查          rev1,    rev2;

strcpy(rev1.restaurant_name, "The Home of Sushi");
strcpy(rev1.restaurant_address, "555 Ellesmeadow Rd.");
rev1.score=3。

rev2=rev1;

rev2=change_score(rev2,4)。
```

让我们看看这在内存中的作用。首先，*main()*声明了两个类型为 *"回顾 "*的变量，名为 *'rev1'*和*'rev2'*。正如预期的那样，这将在内存中保留适当大小的盒子，并以变量的名称来标记它们。



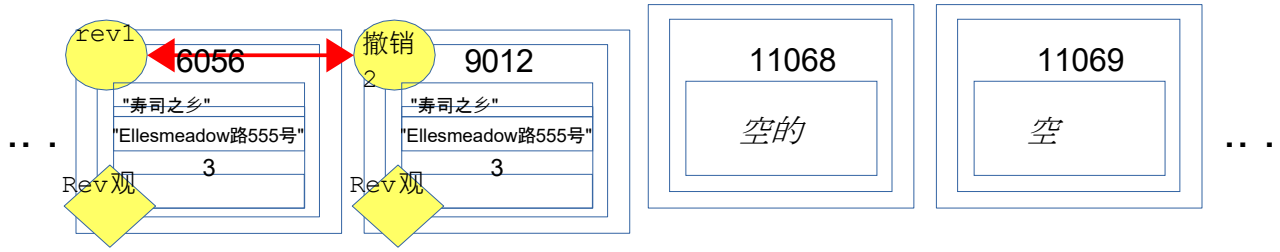
然后我们给*'rev1'*里面的数据赋值。



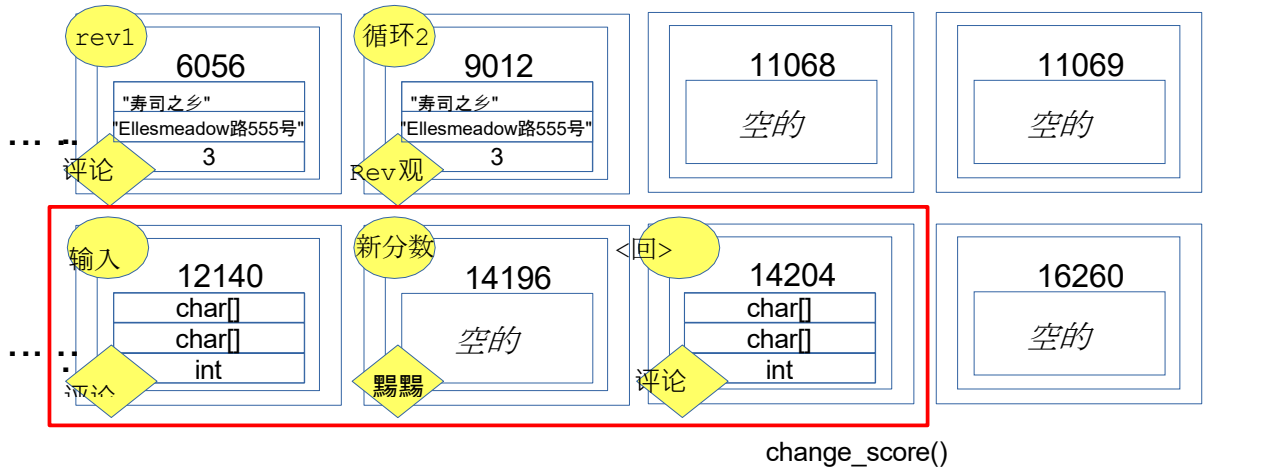
航线

`rev2=rev1`。

读作：*"将标记为'rev1'的盒子的内容复制到标记为'rev2'的盒子里"*。这里要说明的是，***这种类型的赋值是对我们的复合数据类型内的所有内容进行复制。***



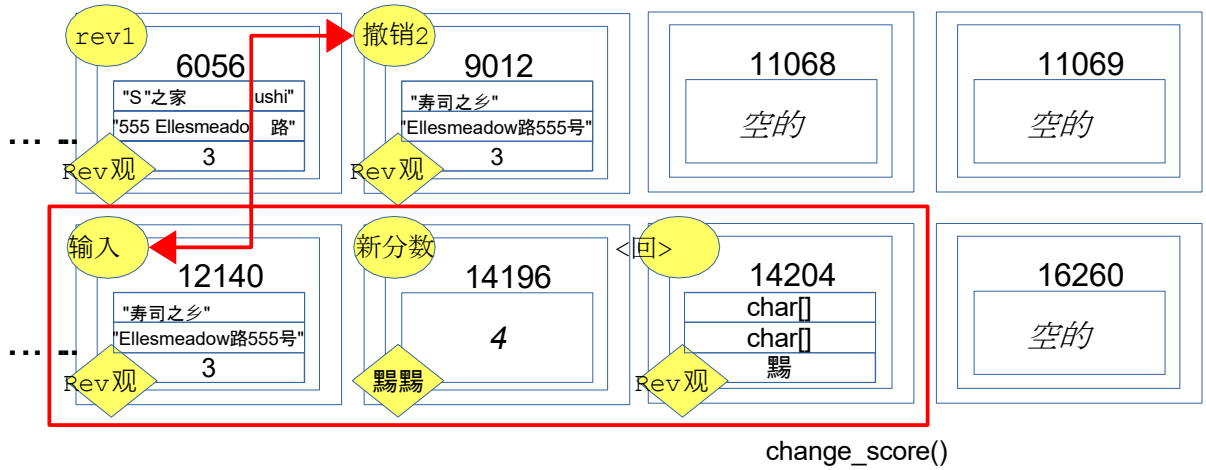
因此，现在我们有两个*相同的*盒子，每个都包含相同的地址、餐厅名称和分数。当调用*change_score()时*，空间被保留给函数的输入参数和返回值。输入参数是一个名为 *"input "*的*"Review "*类型的变量，一个名为 *"new_score "*的*整数*变量，以及 *"Review "*类型的返回值。



*change_score()*为参数和返回类型保留的空间由上面的红框标出。当然，最初它都是未初始化的。这一行

`rev2=change_score(rev2,4)`。

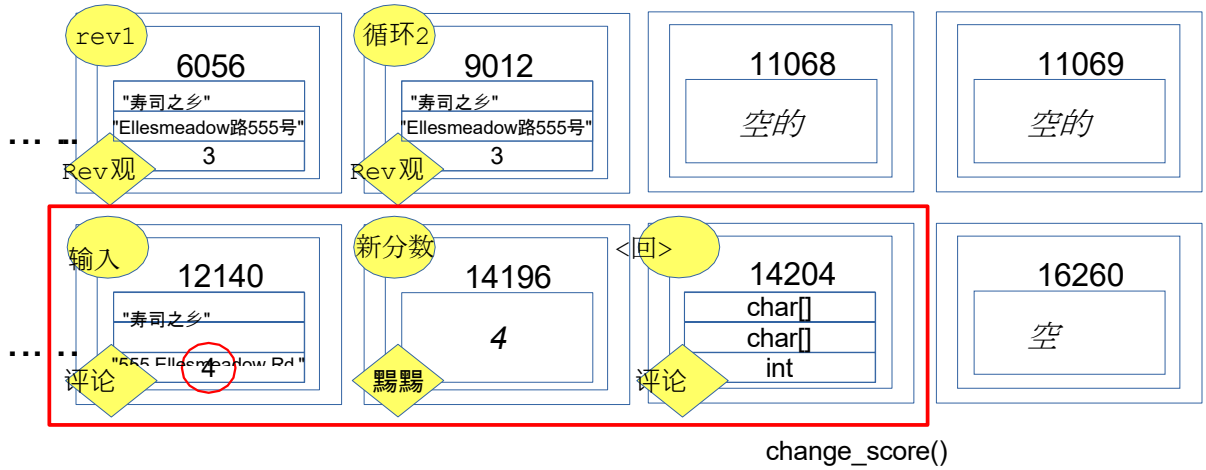
将*'rev2'*作为*'输入'*，将*'4'*的值作为*'new_score'*。因此，*在标记为'input'的盒子里复制了一个标记为'rev2'的内容，*并在标记为*'new_score'*的盒子里存储了一个*'4'*。***请记住：输入参数是本地变量，有自己的保留框。***



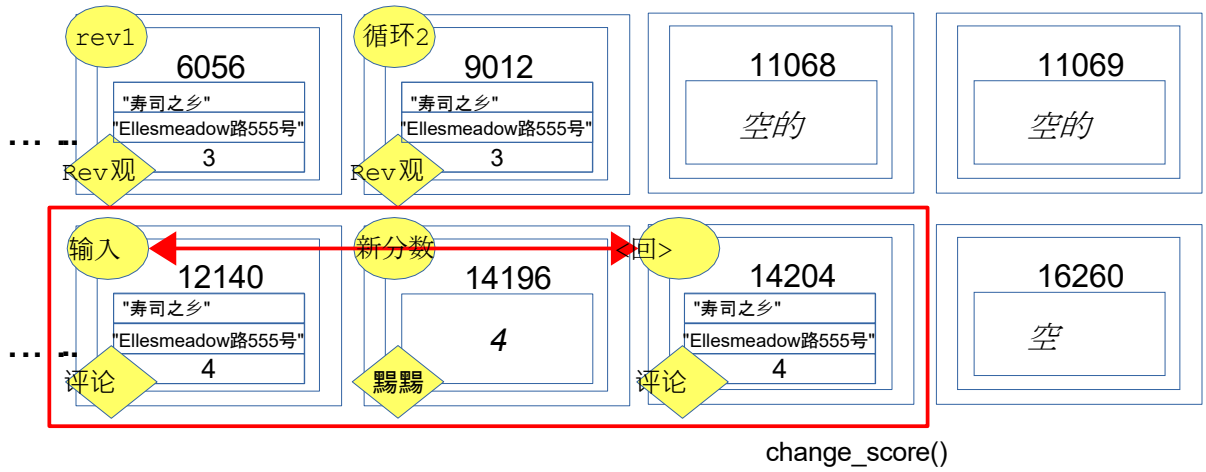
现在我们有*三份*同一评论的副本。每个都在自己的盒子里。功能*change_score()*更新 *"输入 "*的分数。

`input.score=new_score`。

因此，分数在 *"输入 "*变量中被更新。



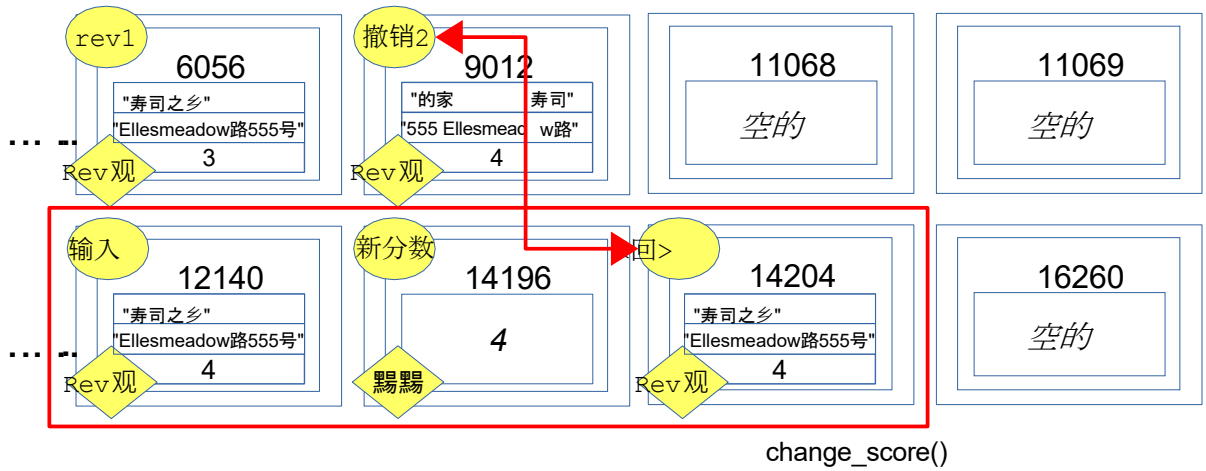
*change_score()*的最后一行返回更新的 *"输入 "*变量--这意味着将其复制到返回值框中。



最后。

`rev2=change_score(rev2,4)`。

从*change_score()*返回的值被复制到*'rev2'*。再一次，这意味着复制标有'<return>'的盒子里的所有东西。



最后，保留给*change_score()*的空间被释放。

你应该从中得到什么。

- 每当复合数据类型的变量被复制时，无论是因为赋值，还是因为它们被作为输入参数传递或从函数中返回；复合数据类型的每个*字段都被复制。在这种情况下，单个复合类型的字段被复制，它总是整个便当盒!*
- 复合数据类型的行为就像其他C语言变量一样。然而，***请注意，你不能在复合数据类型上使用比较运算符。***例如，表达式

```
如果(rev2 < rev1)
```

是无效的。C语言没有办法比较这两种复合类型。如果你需要比较你所声明的数据类型，你必须写一个比较函数，将这种类型的两个变量作为参数，以一种考虑到类型所代表的意义的方式对它们进行比较，并返回一个表示它们应该如何排序的值。

- 通过复制复合数据类型来移动数据会很慢。就像数组一样，我们可以很容易地创建具有相当大内存占用的复合数据类型（即它们包含大量的数据）。将它们复制到函数中或从函数中复制出来是很耗时的，同时也会占用内存。

4.- 使用复合数据类型的指针

正如我们刚刚看到的，移动复合类型涉及大量的信息重复。和数组一样，我们经常需要的是一种方法，让一个函数直接访问并在必要时改变定义在外面的复合类型的内容。就像数组一样，这样做的方法是通过使用指针。让我们看看我们如何使用指针来处理复合数据类型。

```
Review rev;
Review *rp=NULL。
```

```
strcpy(rev.餐厅名称，"The Baking Sleuth")。
```

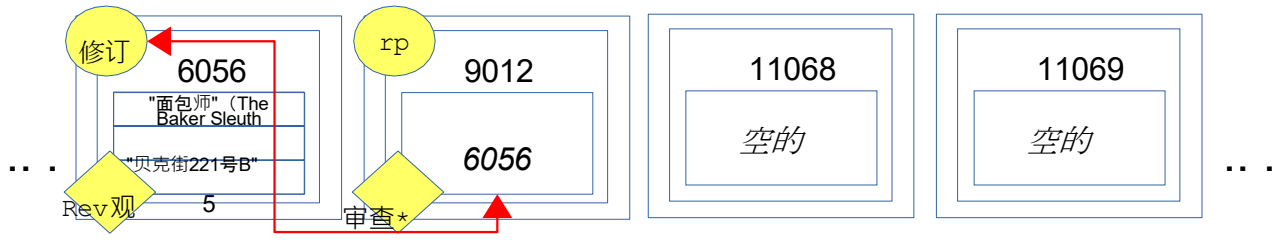
```
strcpy(rev.restaurant_address, "221B Baker Street");
rev.score=5。

rp=&rev;

rp->score=4。
```

上面的代码声明了一个名为 *"review "*的变量，以及一个指向 "Rev *"*类型变量的指针 *"rp"*。然后我们将评论的分数设置为3。

下一行

```
rp=&rev;
```

读作 "*取'rev'的地址并将其存储在指针'rp'中*"。此后，'*rp*'包含了我们的评论的地址，我们可以使用这个指针来访问和修改该评论中包含的信息。在内存中，在上面的行被执行后，我们会期待这样的结果。



请记住，要访问一个复合数据类型的组成部分，我们使用'.'运算符。这只对该类型的*变量*有效。对于指针，我们使用'->'操作符。

```
rp->score=4。
```

上面的内容是使用我们为该变量准备的指针将 *"rev "*中的分数更新为4。同样地，我们可以使用指针来改变地址。

```
strcpy(rp->restaurant_address, "221A Baker Street")。
```

我们将经常使用复合数据类型的指针，所以要练习用这些指针访问和修改数据，直到你能适应这个过程。

练习。编写一个小程序，使*main()*声明一个 *"评论 "*类型的变量，但现在它调用一个函数*fill_out_review()*，该函数接收一个指向评论的*指针*并填写

餐厅的名称、地址和分数。这个函数*没有一个返回值，因为它直接修改了评论*。在函数返回后，让*main()*打印出已完成的评论的内容。

5.- 获得用户输入

我们在***第2***节写的说明复合类型的程序只是为了给你一个例子，说明如何定义一个新的数据类型，如何创建这种类型的变量，以及如何分配和/或访问这些变量中的信息。然而，对于我们新的 "*评论*"类型的任何有趣的使用，我们需要能够获得用户对我们应用程序将存储和管理的评论字段的输入。

现在让我们回顾一下如何从用户那里获得输入到我们的程序中。

数值类型--整数和浮点数

对于数字数据，我们使用*scanf()*函数。这个函数接收一个*格式化字符串*，决定如何将用户的输入转换为可以分配给我们程序中的变量的值。*格式化字符串*使用与*printf()*相同的格式指定器。让我们看一个例子。

```
#include<stdio.h>

int main()
{
      int x,y;
      float pi;

          printf("输入两个整数和一个浮动数在同一个
line\n") 。
      printf("Separated by spaces\n")。

      scanf("%d %d %f",&x,&y,&pi);
      getchar()。

      printf("Read: %d, %d, %f\n",x,y,pi);

     return 0;
```

编译和运行上述代码的结果是：。

```
.../a.exe
在同一行输入两个整数和一个浮点数，用空格隔开
3    7    3.14159265
阅读：3, 7, 3.141593
```

需要注意的事项。

- *main()*的***签名***（标准定义）。请注意，我们声明*main()*返回一个*int*。你可能会问*为什么*

?-

我们让*main()*返回一个值的原因是，我们假设一个用C语言编写的程序可能是一个脚本的一部分，这个脚本会做很多事情，并运行

在一些数据上的多个进程。一个程序的返回值被这样的脚本用来确定程序是否成功完成。惯例是。

* *main()*如果成功完成，返回0（零）。
* 如果在执行过程中出现错误，*main()*会返回一个非零的值

- *scanf()*的*格式化字符串*指定我们要"%d %d %f"，所以，一个*int*、一个*int*和一个*float*。无论用户输入什么，都将被解释为按照格式化字符串指定的顺序分配给这些数据类型的值。
- 在*scanf()*之后有一个对*getchar()*的调用，因为*scanf()*会忽略用户在输入数值后按下的*[enter]*键。如果我们不删除它，它就会扰乱进一步的输入。
- 因为我们想通过调用*scanf()*来读取*多个值*，所以我们不能依靠*scanf()*的返回值来获取信息。相反，*scanf()*接收了指向变量的指针，我们想在这些变量中存储从终端读取的信息。

```
scanf("%d %d %f",&x,&y,&pi)。
```

上面一行写道：*从终端扫描一个int、一个int和一个float，并将它们存储在'x'的地址、'y'的地址和'pi'的地址。*

'x'、*'y'*和*'pi'*的数据类型是*int*、*int*和*float*，这与我们提供给*scanf()*的格式化字符串相符。

一定要确保格式化字符串和用于存储从终端读取的值的变量类型相匹配。如果类型不匹配，scanf()函数不会警告你，你会在你的变量中出现垃圾值。还要记住，scanf()不能保护你不被用户输入垃圾，或提供不符合预期的值。

为了确保你记住这一点，看看当我们运行同一个程序，但用户却不听话时，会发生什么。

```
.../a.exe
在同一行输入两个整数和一个浮点数，用空格隔开
hafs kjas 5
读：32767, 0, 0.000000
```

这显然是没有意义的。在***你的程序中使用用户输入之前，你应该始终检查用户的输入是否合理。***检查输入是否合理被称为*输入消毒*，包括对输入值设置合理的界限。例如，如果我们正在读取一个餐厅评论的*分数*，我们知道分数是在*1*到*5之间*，我们可以检查从终端读取的分数是否有效，如果不是，就要求用户再次输入一个有效的分数。

练习。写一个小程序，声明一个有10个条目的*int数组*，要求用户提供每个条目的值（这些值应该在*0到100之间*）。然后计算并打印出

数组中数值的平均值（实际上，你正在实现大多数电子表格应用程序中的*AVERAGE()*函数！）。

从终端读取字符串

我们不能使用*scanf()*来读取字符串，因为*scanf()*将空格解释为分隔符。输入字符串中的每一个空格都会被认为是为一个单独的变量提供了一个新值。相反，我们将使用一个不同的库函数，叫做*fgets()*（这个名字来自GET String）。

下面是你如何使用*fgets()*从终端读取字符串。

```
#include<stdio.h>

int main()
{
    char my_string[1024];

    printf("Please type one string/n");
    fgets(my_string, 1024, stdin);

    printf("The input string is: %s\n",my_string);
    return 0;
}
```

这里唯一的新东西是对*fgets()*的调用。

`fgets(my_string, 1024, stdin)`。

第一个输入参数是字符串变量的名称，用于存储所读取的内容。第二个参数指定*要读取的最大字符数，必须小于或等于我们的字符串的字符数组的大小*（***注意***：fgets()将比指定的最大字符数少读一个字节，因为它需要在字符串中添加*字符串结束*分隔符*'\0'*，所以在上面的例子中它最多从终端读取1023个字符）。最后一个参数指定了*从哪里读取*。函数*fgets()*可以用来从不同的数据源读取，包括文件，名称*'stdin'*对应于*标准输入*，也就是终端。我们将在后面看到如何使用*fgets()*从其他来源读取字符串。

编译和运行上述程序的结果是：。

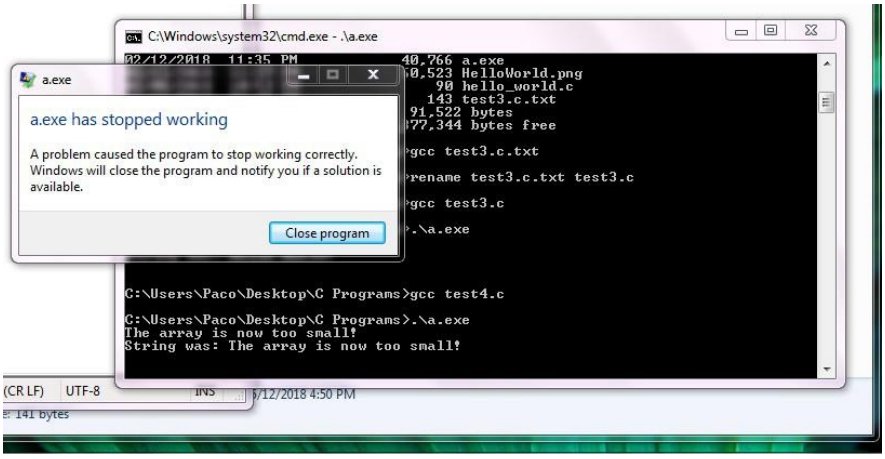
```
.../a.exe
请输入一个字符串Hello
World!
输入的字符串是：Hello World!
```

注意：要注意你的字符串数组要足够大，以包含你所需要的信息，并小心使用*fgets()*。试图在一个小数组中存储一个过长的字符串会使你的程序崩溃。

看看当我们把字符数组的长度改为10，但忘记改变从终端读取的最大字符数时会发生什么。

```
#include<stdio.h>

int main()
{
    char my_string[10];        // 这现在太小了!

    printf("Please type one string\n")。

    fgets(my_string, 1024, stdin); // fgets()认为我们有1024个字符串。
                                   // my_string中的条目

    printf("The input string is: %s\n",my_string);
}
```

让我们编译并运行上述程序。



因此，只要小心你的绳子，一切都会好起来的。

练习。修改我们写的声明和初始化单个餐馆评论的程序，这次它要求用户提供餐馆的名称、地址和分数，然后打印出结果信息。

6.- 处理实际的数据量

在这一点上，我们知道如何创建自定义框来存储信息，现在是时候把我们的注意力转向本课程的一个*基本想法了*。为了理解我们要做的事情，让我们想一想，如果我们想只用我们到目前为止所知道的东西来实现餐厅评论应用程序，会发生什么。

- 我们知道如何实现一个新的数据类型来存储关于评论的信息
- 我们知道如何声明和使用 "*回顾* "类型的变量
- 我们知道如何在我们的程序中的函数之间传递评论
- 我们知道如何从用户那里获得输入以填写评论的数据

问题。我们的程序如何能够以一种使信息易于访问/修改的方式存储多个评论？

假设我们说我们想使用一个数组（到目前为止，这是我们知道的在C语言中存储一个给定类型的多个数据项的唯一方法）。所以我们继续声明。

```
评论        all_reviews[100];
```

这将为100篇评论预留空间，它们将被储存在记忆中的连续盒子里，而且它们将很容易被我们的项目所访问。

*问题。*你能看到这样做可能存在的问题吗？

所以我们为我们的阵列选择了一个非常小的尺寸。我们很可能在我们获得甚至一小部分餐馆的信息之前，就已经用完了存储评论的空间。所以，我们想了一下，决定聪明一点，在我们程序的顶部，我们有。

```
#define    MAX_REVIEWS          100000
```

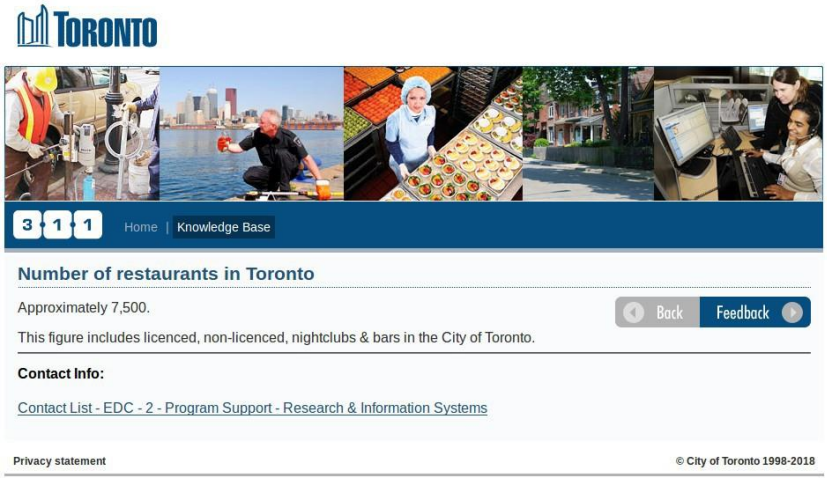
然后在我们声明我们的阵列的地方，我们有。

```
评论        all_reviews[MAX_REVIEWS];
```

现在，这样做的*好处*是，我们不太可能没有空间来为我们的城市储存评论。而且我们可能认为我们已经解决了所有的问题。然而，这个解决方案也有其自身的问题。

*问题。*像我们刚才那样定义阵列有什么缺点？

- 假设多伦多只有7500家餐厅（实际上这是截至2018年12月的准确数据）。



说明3：多伦多的餐馆数量，来自多伦多市的门户网站：https://www.toronto.ca/311/knowledgebase/kb/docs/articles/economic-development-and-culture/program- support/number-of-restaurants-in-toronto.html

现在看来，拥有一个有100,000个条目的数组是个好主意吗？
- 假设我们想扩展我们的应用程序，在评论中添加一个 "城市"字段，并允许来自世界任何地方的评论。我们可以期待我们的数组在某一时刻会发生什么？
- 我们的存储问题解决了吗？

你应该从中得到什么。

- 当你有已知数量的数据需要处理，并且需要一种简单、易于使用的方式来存储和管理这些数据时，数组是非常有用的。它们通常被用于数据处理应用程序，以表示和处理数字数据。
- 它们有以下限制。
    * 它们的大小是固定的。它们不能增长或缩小以适应你的数据需求。
    * 改变数组大小将涉及修改你的代码并重新编译。
    * 如果我们选择一个非常大的数组大小来避免空间耗尽，我们很可能在大部分时间里都有很多未使用的空间（这很糟糕，因为这些空间不能被其他程序使用--或者我们自己的程序）。我们正在浪费宝贵的空间。
- 由于这些限制，它们确实不是信息存储和检索系统的正确工具--你不会用数组来实现数据库的数据。

这里是我们想要解决的问题。

我们需要开发一种方法。

- 提供存储、组织和更新包含我们程序将管理的数据的*项目集合*的方法（例如，我们希望有一个评论的*集合*
为我们的餐厅评论应用程序）。

- 我们的解决方案应该允许我们根据需要保留单个数据项的少量或多量实例。我们事先不知道这个数字，而且它可能会随着时间的推移而改变。我们不希望被限制在一个*固定数量的项目上*。
- 当新的数据项目被添加到我们的*收集中时*，应*按需*保留空间。这是为了避免通过预先保留大量的空间来浪费计算机存储。换句话说，我们的存储解决方案应该是*可扩展的*。
- 我们的解决方案应使我们能够*搜索*、*访问*、*修改*和*删除*任何个人数据我们的*收藏品*中的*项目*。

7.- 容器和列表

*容器*是一种构造（我们所构建的东西），它提供了一种存储、组织和访问特定类型的数据项*集合的方法*。请注意，这是一个非常笼统的定义--
它没有指定数据将如何被组织，没有指定数据将如何被存储在内存中（如果我们想做一个持久的拷贝，则存储在磁盘中），也没有说我们将如何实现函数来访问和修改*集合*中的数据项。

*数组*是一个非常简单但有限的*容器*。我们在上面讨论了当我们事先不知道我们的程序要处理多少数据时，鼓励我们开发一个更好的解决方案来存储数据的限制。

现在让我们看看什么是可能是最简单的*容器*。

- 允许我们保留一个数据项的*集合*。
- 收藏品的规模可以随着时间的推移而增长或缩小。
- 项目的内存是*按需*保留的，只有在需要添加一个新的项目到*收集*。
- 它提供了一种*寻找特定项目*的*方法*，以及*添加*、*删除或修改现有项目*。

我们正在谈论的*容器*被称为*列表*，它有以下属性。

- 它提供了一种*方法*来存储特定类型的数据项的*集合*。
- 数据项是按顺序存储的，一个接一个，对于每个项目，我们可以知道下一个项目是什么（如果有的话）。

这也是一个相当笼统的定义--
是故意的这个定义的目的是只提供一个列表的基本定义，适用于你可以为它编写的任何实现。这一点很重要，因为不管*你如何实现列表*，或者*用什么编程语言*，或者*它包含什么类型的数据*，它仍然是一个列表！这一点很重要。

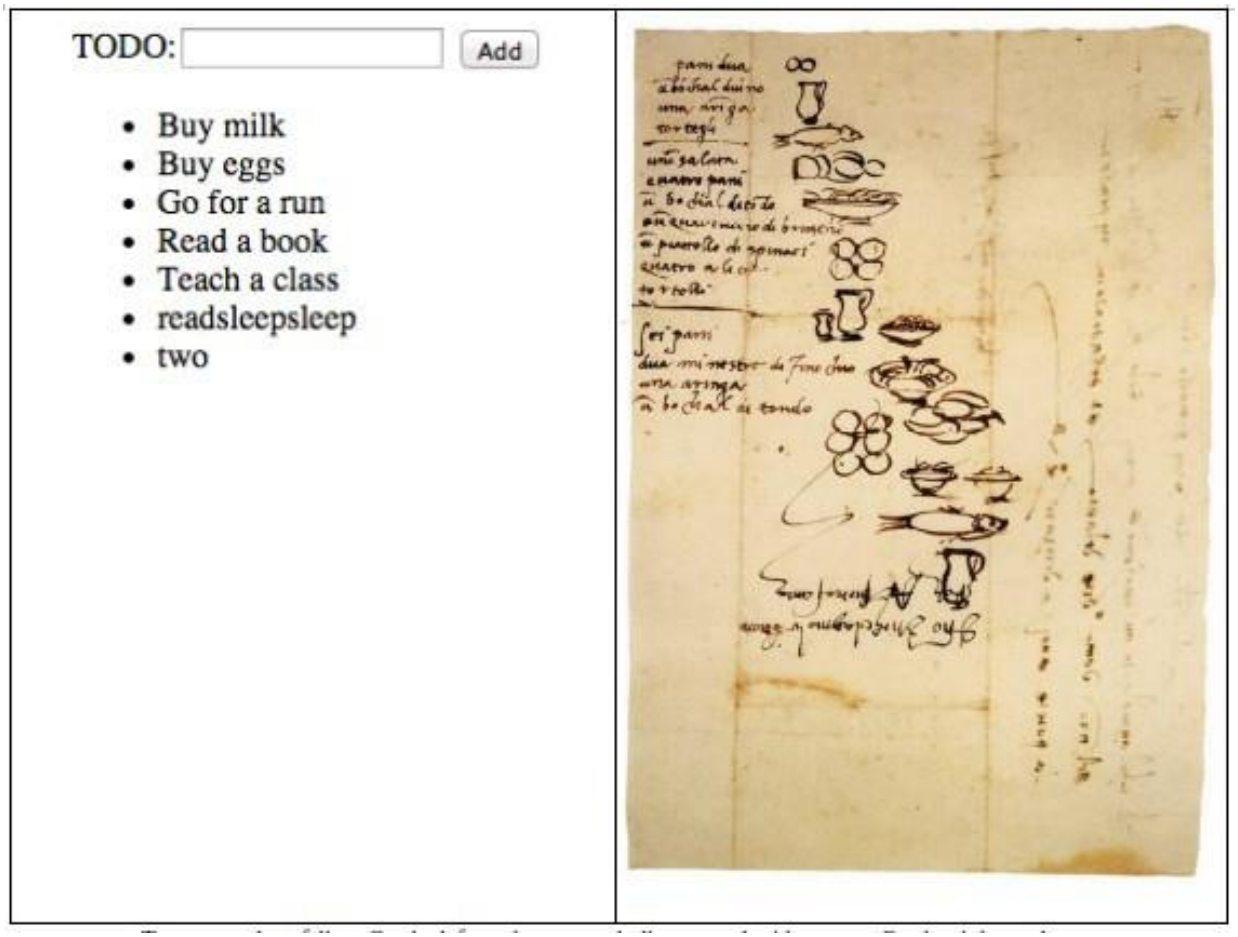


插图4：两个清单的例子。左边是一个用应用程序创建的待办事项清单。右边是16世纪米开朗基罗的购物清单。图片：（左）Lubaouchan，维基共享资源，CC-SA 4.0；（右）Michelangelo，维基共享资源，公共领域

为了非常清楚地说明这一点，在上图中你可以看到两种非常不同的清单*实现方式*--手写的与电脑制作的，购物清单与待办事项清单，以及意大利语与英语。清单是如何创建的，或者它包含什么，这些细节并不重要。它们都有相同的*关键属性*，即有一个项目的集合，按顺序排列，所以它们都是*清单*的例子。

让我们把这个概念更接近于描述一个我们可以用计算机实际实现的列表。

列表抽象数据类型（列表ADT）

*List抽象数据类型*通过指定列表必须提供的操作，扩展了我们对*列表*容器的定义。也就是说，除了代表一个*按*顺序排列的数据项*集合外*，*List ADT*还要求实现以下操作。

- *声明*一个新的（空）列表
- *将*项目*添加*到现有列表中
- 从列表中*删除*项目
- 搜索一个特定的数据项

需要进行*搜索*操作，以便我们能够找到并修改我们集合中的特定项目。例如，在我们的餐厅评论应用程序中，我们可能想更新已经在列表中的一家餐厅的分数。我们还应该检查我们正在输入评论的餐厅是否已经包含在我们的集合中。

上面的定义有很多变化。你可以找到包含其他操作的*List ADT*版本，例如，获得列表的长度，或者在列表的特定位置插入项目（常见的选项包括在前面和后面）。*我们在这里提供的定义包含了你在编程时发现或使用的几乎所有列表的基本操作。*

为什么这被称为"抽象"数据类型？

这是一个特别重要的点。我们上面定义的 List ADT 被称为*抽象的*，因为*它没有指定如何实现 List ADT 和它的操作*。我们可以用许多可能的方法来构建我们的列表，我们可以用我们知道的任何编程语言来实现它。List ADT的*实现*可以是完全不同的，然而，任何知道*List ADT*包括什么的人都会知道应该期待什么：一个*数据项*的*集合*，*按顺序排列*，支持*声明一个新的列表，添加和删除项目，以及搜索*。

这一点非常重要，因为它意味着一旦你知道如何以及何时使用*List ADT*来存储和组织数据，你就可以在*任何编程语言中*使用*该ADT的任何实现*，而不必担心实现的细节问题。

抽象数据类型是计算机科学中解决问题的一个基本组成部分--
它们允许你从数据的组织方式和对数据进行的操作方面进行思考--
因此你可以确定对于你需要解决的任何具体问题，什么是存储和管理信息的最佳方式--
而不必担心实施细节。

如果你继续学习CS，你将在第二年的*软件设计（CSCB07）*和*数据结构设计与分析（CSCB63）*课程中学习更多关于***ADT的***知识。所以***不要忘记。ADTs定义了存储、访问、修改和组织数据的独立实现方式。***

我们如何使用ADT？

当我们在解决一个问题时，我们必须做出的一个关键的早期决定是，*我们将如何存储和组织我们的程序需要处理的数据。*

这意味着你必须考虑你所知道的不同*容器*，它们有什么属性，提供什么操作，以及它们的*效率如何*；然后选择最适合你的特定问题的容器。

例如，在这一点上你知道如何使用*数组*，现在你知道*List ADT*。当决定如何组织餐厅评论应用程序的数据时，你会考虑这两种方法，找出最适合这项特殊任务的方法，然后使用它。我们已经看到，在这种情况下，数组解决方案有很多缺点和限制，而看起来*List ADT*提供了我们需要的属性和我们的应用程序需要的操作，所以我们会选择*List ADT*。

一旦你选择了你想用来解决问题的ADT，你就可以继续解决这个问题，*假设你可以依靠ADT的实现来提供所有需要的操作和存储特性。*

随着时间的推移，你会学到许多不同的***ADT***，每一种都有自己的属性、优势和局限性，每一种都更适合于特定类型的问题。*效率*问题将变得非常重要（事实上，我们很快就会看一看它！）。*你的任务是了解这些ADT是如何工作的，什么时候使用它们最好，以及它们带来了哪些优势/限制，这样你就可以为你需要解决的任何问题选择最好的方法来组织数据。*

现在，让我们假设我们已经决定使用*List ADT*来存储我们的餐馆评论。***现在我们将使用List ADT的规范来提出一个我们可以在程序中使用的实现。***

8.- 链接列表

也许拥有List *ADT*定义的所有属性和操作的List *ADT的*最简单实现是*链表*。为了理解链表是如何工作的，我们可以回到我们最初的比喻，即内存是一个非常大的房间，里面都是编号的储物柜。

下面是一个真实的例子，说明我们建立一个链表的过程。

假设你到达苏黎世进行一次小小的观光旅行。因为你只停留几个小时，所以你懒得预订酒店房间，而是决定把行李放在火车站的一个储物柜里。因此，你找到一个空的储物柜，支付费用，把你的行李放在储物柜里，然后拿到编号的钥匙（假设你拿到的是1342号储物柜）。

你去了我们的城市，开始探索这个城市。这是一个非常有趣的城市，你买了一些东西带回家。首先，你买了一些瑞士巧克力，为了避免它在你走动时融化，你决定回到火车站，把它留在另一个储物柜里。你找到一个空的，付了钱，把巧克力放进去，拿着你的编号钥匙（#0789）。

接下来，你会发现一些有趣的怀表，买一块，为了不随身携带，你会发现

回到车站，把它放在自己的储物柜里（与之前的过程相同），并拿上编号的钥匙（#3519）。

这个过程重复进行，你获得了一些书（留在6134号储物柜里），一个新的数码相机（你把旧的留在2156号储物柜里），一些更多的巧克力！（0178号储物柜）和一些T恤（9781号储物柜）。(#0178号储物柜)，还有几件T恤衫(#9781号储物柜)。

在这一点上，你注意到你脖子上挂着一串钥匙在走动。这可不好玩。所以你你开始想。我怎样才能把我所有的东西（它不适合放在较少的储物柜里）以这样一种方式存放起来，使我在任何时候都只需要携带***一把钥匙***，但我仍然可以随时去取我的任何物品？

在思考了一段时间后，你想出了这个方案。

写下你拥有的所有储物柜的清单（按照你得到它们的顺序）。#1342,

#0789, #3519, #6134, #2156, #0178, #9781

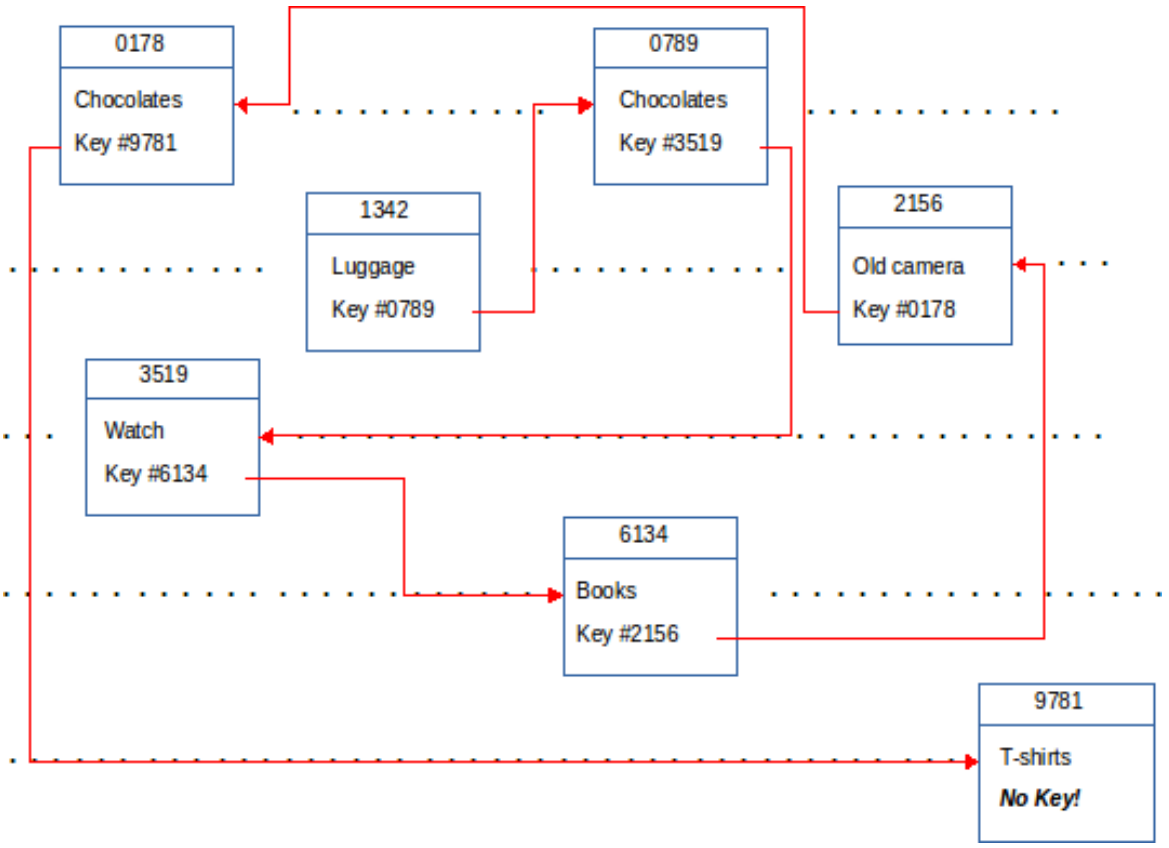
现在，去0178号储物柜，***把9781号储物柜的钥匙放在里面***
去2156号储物柜旁边，***把0178号储物柜的钥匙放在那里***
去6134号储物柜，把***2156号储物柜的钥匙放在那里***
走到3519号储物柜，***把6134号储物柜的钥匙放在那里***
移动到0789号储物柜，***把3519号储物柜的钥匙放在那里***
到第一个储物柜1342号，***把0789号储物柜的钥匙放在那里***。
现在你可以再次走到外面，只带着1342号储物柜的钥匙!

你在这里完成的是创建一个储物柜的排列，其中你只有第一个储物柜的钥匙，而在每个储物柜内你可以找到下一个储物柜的钥匙。这就是一个***链接列表***

在这个例子中，***链接***是***打开集合中下一个储物柜的钥匙***。

列表中的第一个储物柜，也就是我们携带钥匙的那个，被称为列表的***头***。最后一个储物柜，即***里面没有钥匙的***那个，被称为链表的***尾部***。

让我们看一下更衣室的插图，看看发生了什么事



上图中需要注意的重要事项。

- 储物柜是按顺序排列的（但不是按储物柜数量递增的顺序！）。顺序是根据它们被*添加到你的收藏中的*时间来确定的。
- 除最后一个柜子外，每个柜子都有一个*独特的继任者*，其*编号的钥匙*是柜子里的一部分内容。
- 最后一个储物柜*没有钥匙*。
- 第一个储物柜的钥匙不存放在*任何储物柜*中，而是由你保管。

问题。

1) 你是否希望储物柜是按照储物柜编号的增加值来排序的？

2) 哪个储物柜是6134号储物柜的*继承者*？

3) 储物柜的顺序是否*有意义*（它是否提供了关于储物柜中存放的任何信息）？

在寻找什么？

假设现在你已经走了一段时间，拍了一些照片。你的新相机没电了，但幸运的是，你记得你买了一个备用的，并把它放在装旧相机的柜子里。

问题：要从装有旧相机的柜子里*取回*备用电池，你必须采取什么行动顺序？

由于*链接列表*的结构，每当我们要寻找一个特定的项目时，我们需要*遍历*列表，*依次*寻找*每个锁定器*，直到找到我们要找的项目，或者我们到达列表的末端（在这种情况下，项目不在那里！）。

在这种情况下，我们将不得不执行以下行动。

- 用你的钥匙到1342号储物柜，看看里面。这不是你需要的储物柜，所以用存放在那里的钥匙打开下一个储物柜#0789（别忘了在关闭#1342之前把钥匙放回去！）。
- 在0789号储物柜中寻找。巧克力!但我们需要一个电池，所以用存放在那里的钥匙来打开下一个储物柜，#3519。
- 看看3519号柜子，手表不是我们要找的东西，所以用存放在那里的钥匙打开下一个柜子，6134号。
- 看看6134号储物柜，里面是书!不是我们要找的东西。所以拿着那里的钥匙，用它来打开2156号柜子。
- 看看2156号储物柜里面。是那台旧相机!中奖了!拿到备用电池，关上柜子，然后出去。

正如你所知道的，这需要一些工作。

请记住。*每当你使用一个链接列表来保存一个项目集合时，搜索一个特定的项目将需要遍历列表直到我们找到它。与数组不同，我们不能简单地去找列表中的任何一个任意项目--我们需要钥匙，而钥匙存放在另一个柜子里。要找到一个特定的项目，唯一的方法就是沿着链*

锻炼。结果发现所有的行走都让你有点出汗，所以你决定换掉你的衬衫。写下你从你的储物柜中拿一件干净的T恤所需的行动顺序。

如果我们需要储存更多的东西怎么办？

假设你发现一幅漂亮的瑞士风景画，你想带回家。你买了它，并把它带回了车站。

问题。我们怎样才能在我们的收藏***中插入***（添加）另一个储物柜？

我们可以通过几种方式向我们的收藏品添加新物品。无论我们选择哪种方式，我们都必须至少执行这三个步骤。

- 找一个新的储物柜来存放东西，我们会拿到这个储物柜的***钥匙***。
- 把我们需要储存的东西放在新获得的储物柜里。
- ***将***新的储物柜***链接***到我们的集合。这是确保我们的链接列表保持连接的关键步骤。
- 如何将新的储物柜链接到列表中，取决于我们想把它***插入到列表的哪个位置***。

例子。 让我们把新买的画作存放在一个储物柜中，并把新的储物柜***插入***我们的集合中，放在***链接列表***的***头部***。

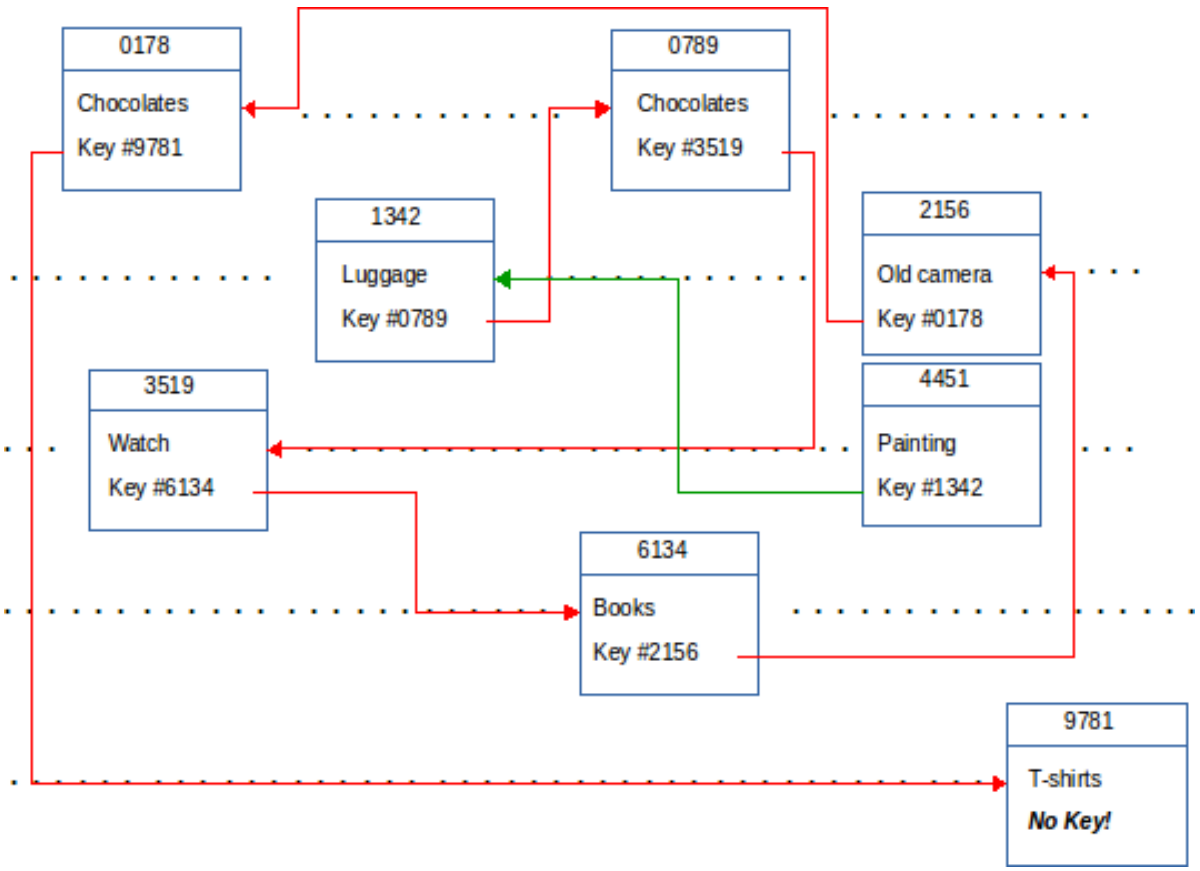
- 预订一个新的储物柜（#4451）。
- 把画放进柜子里（我们很幸运，它正好合适！）。
- 将新的锁定器***链接***到***头部***的现有***链接列表***。
    * 这意味着新的储物柜将成为我们列表中的第一个储物柜，即新的***头***。
    * ***当前的头部***将成为列表中的***第二个储物柜***。
    * 我们有***当前头的钥匙***（#1342）。所以这个过程是

    将***当前头部***(#1342)***的钥匙***存放在***新的储物柜***(#4451)中。
    储物柜#4451现在是***头部***，所以我们把钥匙放在身边。完成了!

执行上述步骤后，我们的集合看起来如下图所示（从***链表的新头***到***链表的旧头所***增加的新链接显示为绿色）。

同样的过程将允许我们在列表的***头部***添加任何数量的项目。列表将从前端开始增长。

练习。从一个空列表开始，显示一个链接列表的示意图，说明在我们插入*巧克力*（2215号柜）、*瑞士奶酪*（0117号柜）、*咕咕钟*（4152号柜）和一堆*明信片*（1890号柜）之后，依次***将每个项目插入到列表的头部，***看起来是什么样子。



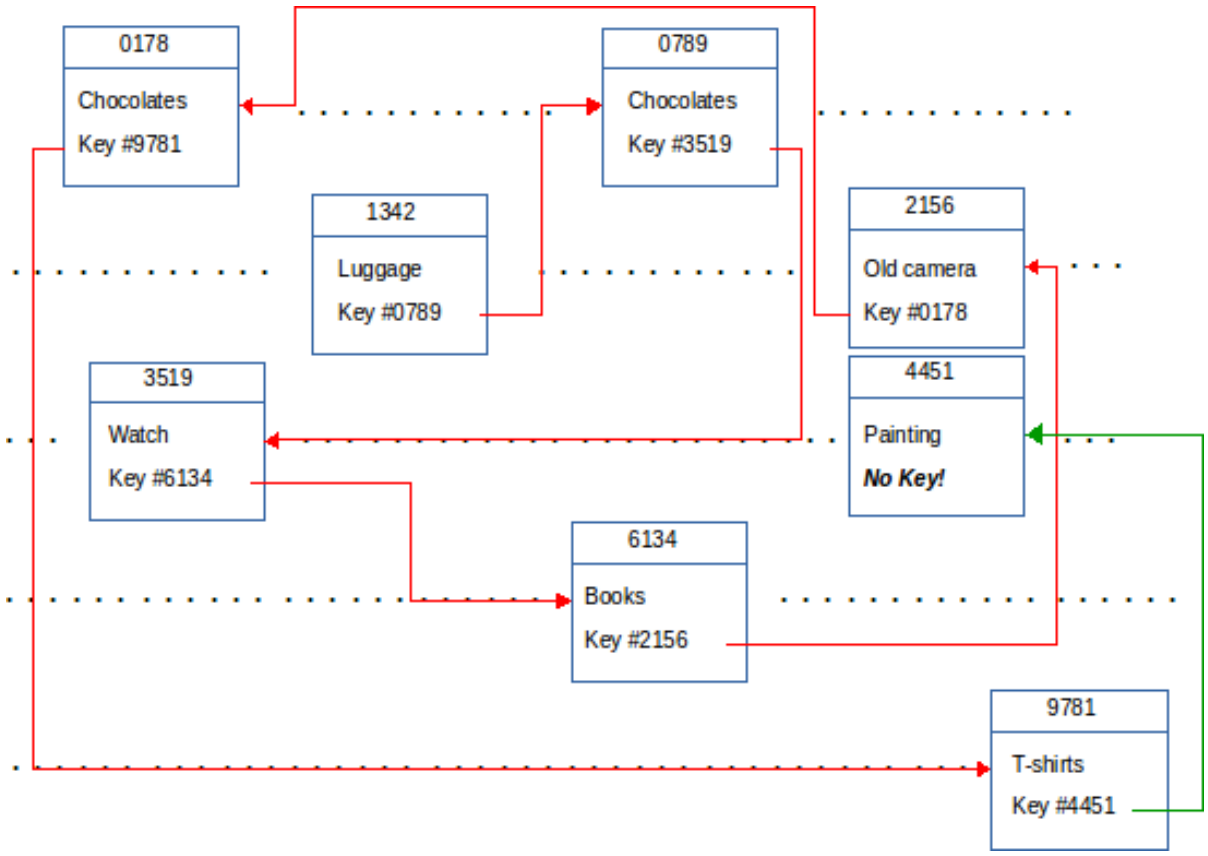
在列表的尾部插入一个新项目

在*头部*插入是在列表中插入一个新项目的最直接（最少的努力）的方法。然而，这并不是唯一的选择。我们可以通过更多的工作，在列表的*尾部*插入一个新项目，这样列表就会从尾部开始增长。

假设我们想用我们的绘画（#4451）添加新的储物柜，而不是在*头部*，而是在列表的*尾部*。我们将不得不。

- 拿到我们储物柜的钥匙（#4451）。
- 将画作存放在储物柜中，注意这个储物柜将*不*包含*钥匙*，因为它将是列表中最后一个。
- 遍*历*链接列表，直到我们到达最后一个储物柜（因为它*没有钥匙*而被识别），在上面的例子中，这将是储物柜#9781。这个储物柜是列表中的*当前尾部*。
- 把新储物柜的钥匙放在列表的*当前尾部*里面。

新添加的项目成为*列表的尾部*。如果我们选择将新的储物柜添加到*尾部*而不是列表的*头部*，我们的列表将如下所示。



不要忘记。在列表的***尾部***添加一个项目需要***遍历整个列表***。这可能是一个很大的工作!那么*我们为什么要这样做呢？*想一想这个小问题，我们很快就会看到在列表尾部添加项目的应用是非常合理的

练习。从一个***空列表***开始，说明如果你进行以下操作，链表会是什么样子（每项都注明了你得到的锁子）。

- 将巧克力（#0008）***插在***列表的***前面***
  (这与此时在尾部插入巧克力是一样的吗？)
- ***在***列表的***尾部插入***一个装有羊角面包的袋子（#9501）。
- 将一袋书(#0546)***插在***名单的***前面***
- ***在***列表的***前面插入***一对T恤衫（#6121）。
- ***在***列表的***尾部插入***一双鞋（#2222）。

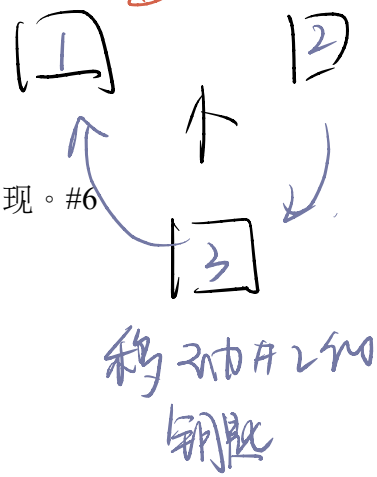
#6121
#0546
#0008
#9501
#2222

问题。
在进行了上述操作后。
- 列表中的***头部***是什么？(储物柜编号。 6121)
- 列表的***尾部***是什么？(储物柜号码。 2222)

在现有项目之间的一个位置插入

插入新物品的最后一个选项是将它们放置在我们列表中现有物品之间的某个地方。这是最复杂的操作（尽管我们将看到每一步都是有意义的，如果你考虑到需要如何组织储物柜的话）。就像在*尾部*插入一样，这是一种对特殊应用有意义的插入方式。让我们看看它是如何完成的。

假设我们想*在书本之后*（或者，相当于同样的事情，在*旧相机之前*）就有画。这个过程会是这样的。



- 获得一个新的画柜（#4451）。
- 把画放在那个柜子里。
- 遍*历*链接列表，直到我们找到包含书籍的储物柜（#6134）。
    * 在这一点上，我们需要确保储物柜最终以这样的顺序出现。#6134号（书）→4451号（画）→2156号（旧相机）。
    * 我们有4451号的钥匙（我们预订储物柜时得到的）。
    * 6134号储物柜包含2156号储物柜的钥匙，所以。

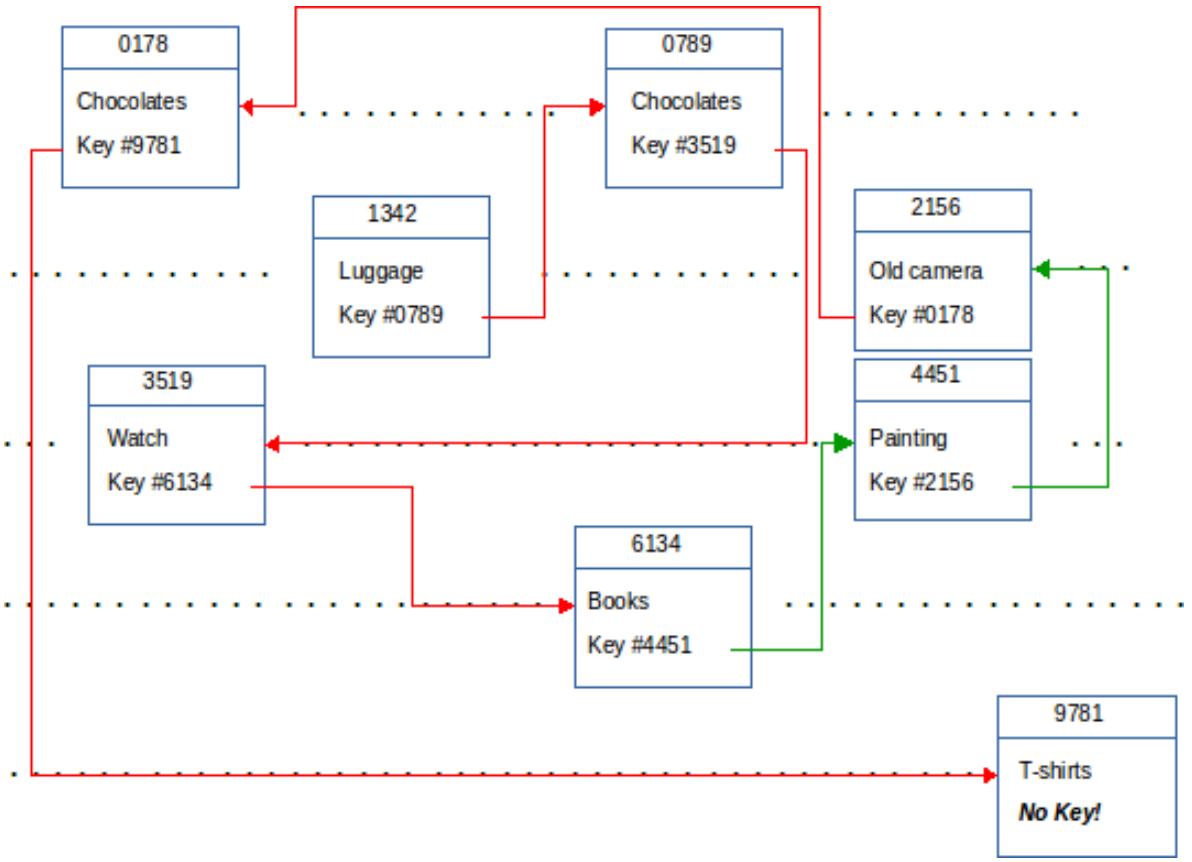
- 我们从6134号储物柜（书籍）中取出2156号储物柜（旧相机）的钥匙。
- 我们将4451号储物柜（绘画）的钥匙存放在6134号储物柜（图书）中。
- 我们将2156号储物柜（旧相机）的钥匙存放在4451号储物柜（绘画）中。

就这样了。请注意，我们不必对2156号储物柜的内容做任何处理，就该储物柜而言，什么都没有发生！这就是我们的工作。

在这个过程中，唯一需要真正小心的部分是我们在*移动钥匙的*时候。然而，如果你花点时间真正理解*上述步骤的原因*，你就能在需要的时候*弄清楚这些步骤*，而且你不需要*背诵*任何东西!

如果我们执行上述步骤，将这幅画添加到我们的收藏中，我们的***链接列表***将如下所示（更新的链接显示为绿色）。

练习。列出*将*一袋*瑞士无咖啡因咖啡*放入我们的收藏品中所需要的步骤--*在巧克力和T恤衫之间*。请确保列出每一个步骤，并展示所产生的清单是什么样子的。



在这一点上，你应该能接受的事情。

- 一个链接列表是如何组织的。
- 如何在一个链接列表中搜索一个特定的项目。
- 如何在*头*、*尾*或*现有项目之间*插入一个新项目。

有一个最后的操作，我们可以在一个链表上执行，我们应该看一下，然后我们可以继续写一个C语言的工作链表的实现。

从我们的收藏中移除（删除）项目

所有四处走动获取东西并将它们存放在储物柜中的井然有序的链接列表中的工作，使你非常饥饿。你决定吃掉你一个储物柜里的所有巧克力，你记得有两块，你确实非常饿，所以你决定吃掉你收藏的第一个巧克力。

- 你回到储物柜前，*遍历*你的链接列表，直到找到巧克力。
    * 从1342号储物柜（行李）开始，拿到0789号储物柜的钥匙
    * 去0789号储物柜（巧克力）。找到他们了!*吃掉所有的巧克力!*

吃完巧克力后，储物柜已经空了，所以你决定把钥匙还给储物柜租赁办公室，***但首先你要确定剩下的储物柜还是一个链接的列表！你要把钥匙还给他们。***

我们目前的情况是这样的。

#1342号（行李，0789号的钥匙）→0789号（无物品，3519号的钥匙）→3519号（手表....）

如果我们删除#0789，我们需要确保#1342号储物柜成为与#3519号储物柜的***链接***，后者是紧随被吃掉的巧克力之后的储物柜。

因此，*为了*从列表中***删除***一个项目，我们

- 转到我们要删除的项目的***前身***（紧接着的项目）。
- 用我们要删除的项目的***后继者的链接替换前身中的链接***（这个链接与我们要删除的项目一起存储！）。

在上面的案例中，我们需要从#0789中取出正在被移除的钥匙，并将其存放在储物柜#1342中。这将导致以下情况的发生。

#1342（行李，3519号的钥匙）→#3519（手表....）

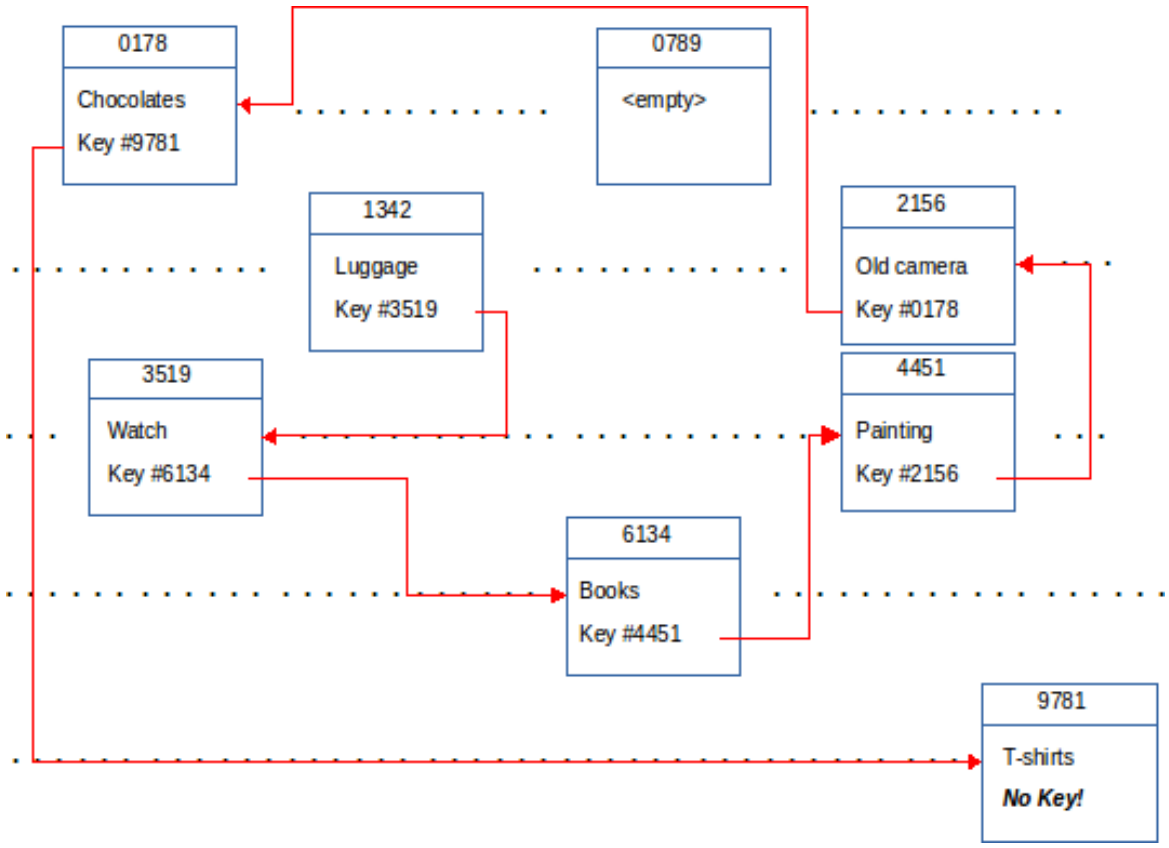
0789号储物柜已不在我们的名单中，*我们可以把钥匙还给租赁办公室，这样就可以重新使用该储物柜。*

因为我们的***链接列表***是关于按需获得储物柜，并且能够获得我们所需的储物柜数量，所以我们应该成为好公民，*永远不要忘记归还我们不再需要的储物柜，这样它就可以在以后的时间里被其他人或自己重新使用。*

如果我们把装有巧克力的储物柜去掉，我们的清单将如下所示。

问题。
如果我们要***删除链表的尾部，***同样的过程是否可行***？***
如果我们要***删除链表的头部，***同样的过程是否可行***？***

你已经看完了用锁存器实现链表的整个例子！你可能想知道为什么我们这样做--
没有任何实际的代码。你可能想知道为什么我们这样做--
没有任何实际的代码。原因是，***同样的过程适用于链表，与你用什么语言编程无关，也与你在那里存储什么项目无关***。因此，理解列表是如何独立于代码工作的，将允许你在任何语言中实现一个链接列表，用于任何应用程序，并用于存储任何类型的数据。***这正是你应该一直努力实现的理解***。实现链表将有助于完善和巩固你的理解，但不要忘记。链表的***概念、过程和组织***比任何具体的实现更重要。



9.- 用C语言实现一个关联列表

到此为止，我们一直在概念层面上讨论链接列表，作为一种***抽象数据类型***，它可以在任何编程语言中以许多不同的方式实现。现在是时候让我们看看*链接列表 ADT* 的实际实现了。

一个ADT的具体实现被称为数据结构。两者的区别很重要：可能有许多不同的方法来实现一个特定的ADT（甚至使用相同的编程语言），不同语言中的同一个ADT的实现可能看起来完全不同。另一方面，***数据结构***是特定于编程语言的，并且依赖于实现。***数据结构和ADT都描述了组织数据的相同方式，以及可以对这些数据进行的操作。***

我们要做的是在C语言中创建一个*链表数据结构*，这包括以下内容步骤。

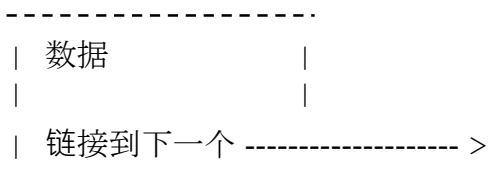
- 设置一个新的数据类型来存储列表中的***一个项目***。每个单独的项目通常被称为列表中的***一个节点***。
- 设置一个指针来跟踪***列表的头部***。
- 编写一个函数，按要求创建一个***新的空节点***。

- 编写一个函数将新的节点*插入*到列表中。
- 编写一个函数来搜索一个特定的*项目*。
- 编写一个函数来*删除*列表中的项目。

这似乎有点费事，但正如我们将看到的，这个过程是相同的，与链表所包含的内容无关，所以*一旦你知道如何对一种类型的项目进行操作，你就可以对任何其他类型的项目进行操作*。

让我们从*节点的*数据类型开始。

链接列表中的节点的一般结构是



正如你所看到的，*节点*只是一个盒子，而这个盒子有*两个部分*。第一部分，即*数据*部分，由你实际想要存储在链表中的信息组成。它可以是

- 一个简单的数据类型，如*int*、*float*或*string*。
- 一个复合数据类型，比如我们之前定义的*Review*数据类型
- 多个数据类型的组合（实际上，它可以定义一个新的复合数据类型）。
- *一个指向存储在其他地方的信息的指针*

这里要说明的是，我们建立和使用链表的方式是一样的，无论其中存储的是什么数据。

*节点*中的第二个组件是指向列表中下一个节点的*链接*（还记得我们上面的长例子中指向下一个锁子的钥匙吗！）。在C语言中，这只是一个*指针变量*，包含我们列表中下一个节点的地址。

我们必须使用指针，因为正如我们已经学到的那样

- 我们不知道一个新的*节点*将被放置在内存的什么地方。
- 我们将按需申请*节点*空间，需要多少就申请多少，但不能超过。
- 在 C
  语言中，我们需要*指针*来允许函数访问/改变在其范围之外声明的变量。所有在链表上工作的函数都必须这样做，所以我们需要指针。

让我们看看我们如何为一个简单的数据类型定义一个链接列表节点。

例子。定义一个***链接列表节点***，每个节点存储一个***int***值。

```
typedef struct int_list_node
{
     int stored_integer;            // DATA
     struct int_list_node *next; // 链接到下一个条目
} int_node;
```

我们已经看到，我们使用*typedef*来创建新的数据类型。链接列表 ***节点***是一种新的数据类型，其定义方式完全相同。第一行

```
typedef struct int_list_node
```

告诉编译器我们要定义一个新的*复合数据类型*，叫做*int_list_node*（一个包含整数的链接列表的节点）。

接下来的几行

```
     int stored_integer;            // DATA
     struct int_list_node *next; // 链接到下一个条目
```

定义这个节点的内容：一个名为 *stored_integer 的 int* 值，以及一个指向链接列表中下一个节点的*指针*（其类型为 *int_list_node*）。最后一行

```
} int_node;
```

告诉编译器我们要调用我们的新数据类型*int_node*。此后，我们可以继续为我们的链表中的节点声明变量，具体做法是

```
int_node  a;          // 一个int_node类型的变量
int_node  *头。       // 一个指向int_node的指针
```

让我们看看如何在一个小程序中使用我们的新*数据类型吧*

```
#include<stdio.h>
#include<stdlib.h>

typedef struct int_list_node
{
     int stored_integer;              // 数据
     struct int_list_node *next;      // 链接到下一个条目
} int_node;

int main()
{
    int_node     a_node。
    int_node *p_node, *another_node;
```

```
    a_node.stored_integer=21;
    a_node.next=NULL。

    node_ptr=&a_node。
    node_ptr->stored_integer=17。

    printf("节点中包含的值是%d\n",\ node_ptr-
           >stored_integer)。

    返回0。
```

编译并运行上面的代码，我们得到了。

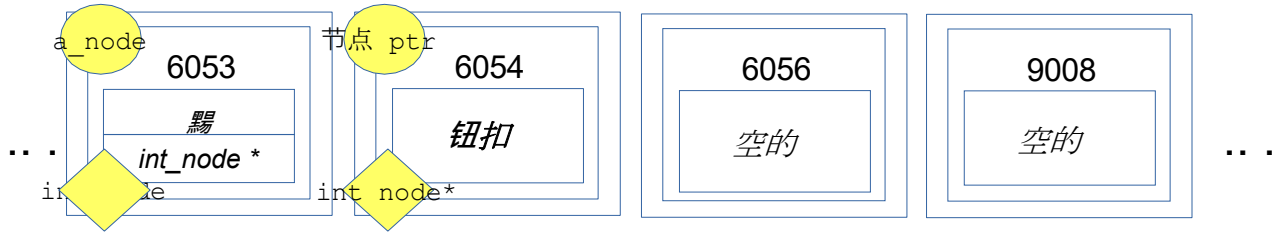
```
.../a.exe
节点中包含的值是17
```

让我们看看这在内存中的作用，以充分了解出小程序。

首先，*main()*声明了两个变量

```
int_node     a_node;
int_node     *node_ptr=NULL
。
```

第一个是一个叫做'*a_node*'的***链接列表节点***，第二个是一个指向'*int_node*'类型变量的***指针***。在内存中，这将保留一个大小合适的盒子来存放一个*int_node*，还有一个盒子用来存放一个*指针，如下*所示



请注意，。

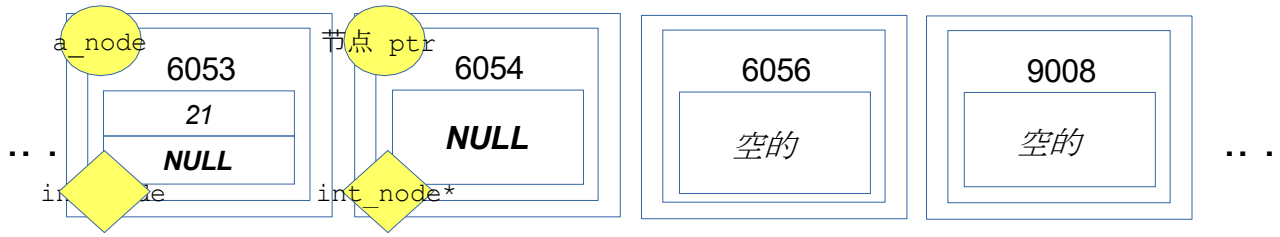
- 包含列表节点的盒子有两部分：一个*int*，和一个指向*int_node的*指针，因此我们可以将这个盒子链接到一个列表中。
- 另一方面，*node_ptr**只是一个指针***，尽管它是一个指向*int_node*类型的变量的指针，但它并没有两个组成部分。最初它是***NULL***，表示它没有指向任何东西。

接下来，程序*将*数据*填入*'*a_node*'中。

`a_node.stored_integer=21`。

```
a_node.next=NULL。
```

在记忆中，我们现在有这样的东西。



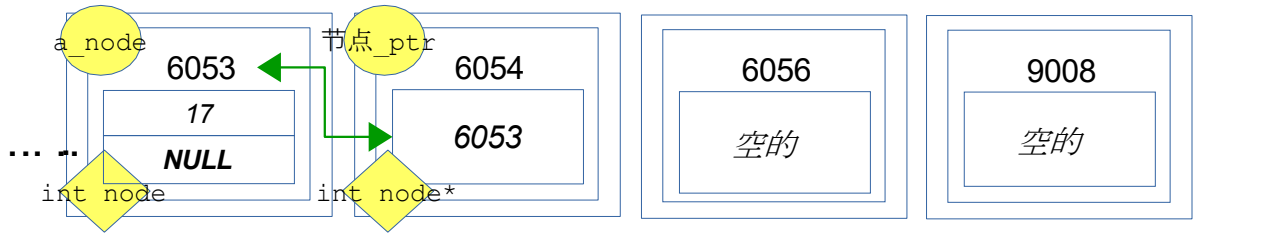
请注意，。
- 我们将*a_node.next*的值设置为***NULL***，表示这个节点目前*没有链接到任何东西*。

*一定要确保新创建的列表节点内的指针被设置为**NULL***，否则，正如你所知，为节点保留的实际内存将包含*垃圾*，而你的代码将把这些垃圾当作实际的指针。这将在你的代码中产生一个难以修复的错误。

接下来我们得到一个指向我们新创建的节点的指针，用它来改变节点中数据的值，并打印出节点的数据内容。

```
node_ptr=&a_node。
node_ptr->stored_integer=17。

printf("节点中包含的值是%d\n",\ node_ptr-
       >stored_integer)。
```

第一行读作
"*获得'a_node'的地址并将其存储在'node_ptr'中*"，然后我们使用我们的指针访问节点的内容（记住，当我们有一个指向复合数据类型的指针时，我们可以使用'->'操作符访问其不同部分）。在这种情况下，这一行写道
"使地址在'node_pointer'中的节点的'store_integer'的值等于17"。最后一行打印出节点的存储整数（使用指针来访问它！）。正如预期的那样，它打印出17。在内存中我们现在有



你可以看到*'node_ptr'*只包含*'a_node'*的地址，如果我们有一个函数需要访问/修改*'a_node'*中

的数据，我们可以传递给它*'node_ptr'*。

上面的例子是向你展示我们如何定义一个 ***节点***数据类型，以及我们如何声明这种类型的变量和指针，并使用它们来访问和修改节点中的数据。然而，我们在本节开始时说，***我们希望能够按需创建节点，***因为新的数据项被添加到一个集合中。我们不能用写在代码中的变量声明来做到这一点。

接下来，我们将看到如何***按需创建节点***（这被称为*动态内存分配*，这只不过是一个花哨的术语，用于*在你需要时为数据获得空间*）。我们将看到，处理这种数据的唯一方法是使用指针。在这一点上，我们应该想到我们最初的餐馆评论应用程序，所以让我们应用我们所知道的来创建一个*餐馆评论*的链接列表，并看看我们如何*按需生成新的餐馆评论*来放在我们的列表中。

声明一个用于审查的链接列表节点

请记住我们的*Review*数据类型（我们已经看到它是如何工作的，以及如何使用它）。

```
#define MAX_STRING_LENGTH 1024

Typedef struct Restaurant_Score
{
       char restaurant_name[MAX_STRING_LENGTH];
       char restaurant_address[MAX_STRING_LENGTH];
       int score;

}审查。
```

现在让我们建立一个 ***节点***，在一个链接列表中存储评论。就像以前一样，我们的 ***节点***将包含两个部分。一个变量用来保存一个*评论*，和一个*指向*链表中*下一个节点的指针*。

```
typedef 结构 Review_List_Node
{
      审查修订。
      结构 Review_List_Node *next;
}Review_Node;
```

这将创建一个名为 *"Review_Node "的*新数据类型，其中包含一个*Review*，以及一个指向链接列表中下一条目的指针。

花点时间将这个节点定义与上面的*int_list_node*的定义进行比较，你会发现唯一的变化是节点的*数据组件*现在是一个*Review*类型的变量。除此以外，它的工作方式完全相同。*这表明为一个链接列表创建节点对任何数据类型的工作方式都是一样的。*

按需创建节点

由于我们必须能够*按需*创建 *节点*，我们需要写一个小函数，将

- 为一个新的 *"Review_Node "*保留空间。
- 初始化新保留节点的内容。
- 为我们的程序提供 *一个*指向新节点的*指针*，这样我们就可以*访问/修改*里面的数据，
  这样我们就可以*把它链接到一个列表中*。

下面是我们如何为审查节点做的，但请注意，同样的过程将适用于包含任何其他数据类型的节点。

```
Review_Node *new_Review_Node(void)
{
      Review_Node *new_review=NULL;          //新节点的指针


      new_review=(Review_Node *)calloc(1, sizeof(Review_Node))。

      // 初始化新节点的内容，其值显示为
      //它还没有被填满。在我们的例子中，我们把分数设为-1。
      // 将地址和餐厅名称都改为空字符串 ""
      // 非常重要的是!将'下一个'指针设置为NULL

      new_review->rev.score=-1;
      strcpy(new_review->rev.restaurant_name,"")
      。
      strcpy(new_review->rev.restaurant_address,");
      new_review->next=NULL。
      返回new_review。
}
```

让我们详细看看上面的代码--
它很重要，因为它显示了你将如何为你用C语言编写的任何链表（以及许多其他数据结构）按需创建节点。

首先是函数声明。

```
Review_Node *new_Review_Node(void)
```

这说明名为*'new_Review_Node'*的函数*没有输入参数*，并返回*一个指向Review_Node的指针*。

在函数的主体中，我们有一个变量声明。

```
Review_Node *new_review=NULL;     //指向新节点的指针。
```

这只是一个指向*'Review_Node'的*指针，它最初被设置为***NULL***，表示它没有被分配。

分配一个新的 *"Review_Node "*的实际工作在这里完成。

`new_review=(Review_Node *)calloc(1, sizeof(Review_Node))`。

这里的语法需要注意一下才能理解。函数*calloc()*是一个库函数，用于*按需保留内存*。它接收了两个参数。

```
calloc( # of items , size of each item in bytes)
```

在上面的例子中，我们正在请求*一个项目*，其大小是*一个Review_Node的大小*。幸运的是，我们有一个有用的*sizeof()*函数，它可以返回我们程序已知的任何数据类型的字节大小。

*calloc()*的作用是什么？

- 它在内存中找到一个具有请求容量的可用位置
- 它保留了我们所要求的确切数量的空间
- 它将该内存空间的内容用零清除掉
- 它返回一个指向我们保留的内存块的指针

函数*calloc()*返回一个*没有任何附加数据类型的*指针，所以这一行

`new_review=(Review_Node *)calloc(1, sizeof(Review_Node))`。

取出返回的指针，将其*类型转换*为*Review_Node*类型变量的指针，并将其存储在我们的指针变量*"new_review "中*。

这是很难接受的!因此，让我们分步骤慢慢回顾。

- 我们声明了一个*指向new_review的指针*
- 我们用*calloc()*为*new_review 节点*保留了内存空间。它给了我们一个*指针*
  到我们*新保留的节点*。
- 我们存储这个指针，以便我们知道我们的节点在哪里

我们一会儿就会看到这些东西在内存中是如何工作的!让我们完成*new_Review_Node()*函数。这个函数的最后部分*初始化*（填充）了我们新获得的*Review_Node的*值，*表明该节点还没有被实际数据更新*。

这是一个重要的步骤，有助于我们避免因访问已经创建但仍未包含有效信息的节点中的信息而造成的错误。

在上面的例子中，代码将*'score'*设置为*'-1'*，并将餐厅的名称和地址初始化为空字符串（""）。然后，它***将'next'指针设置为NULL***。这是一个重要的步骤，因为它可以确保你不会把其他东西留在内存中的*垃圾*误认为是指向链表中一

个节点的*有效指针*。***始终将新创建的节点的指针初始化为NULL***。

为了充分理解上述函数的作用，让我们看看在内存中会发生什么，如果我们运行一个

这个小程序创建了一个*单一的Review_Node，在*新节点中填入信息，并将这些信息打印出来。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAX_STRING_LENGTH 1024

Typedef struct Restaurant_Score
{
       char restaurant_name[MAX_STRING_LENGTH];
       char restaurant_address[MAX_STRING_LENGTH];
       int score;

}审查。

typedef 结构 Review_List_Node
{
     审查修订。
     结构 Review_List_Node *next;
}Review_Node;

Review_Node *new_Review_Node(void)
{
     Review_Node *new_review=NULL;          //新节点的指针


     new_review=(Review_Node *)calloc(1, sizeof(Review_Node))。

     // 初始化新节点的内容，其值显示为
     //它还没有被填满。在我们的例子中，我们把分数设为-1。
     // 将地址和餐厅名称都改为空字符串 ""
     // 非常重要的是!将'下一个'指针设置为NULL

     new_review->rev.score=-1;
     strcpy(new_review->rev.restaurant_name,"")
     。
     strcpy(new_review->rev.restaurant_address,");
     new_review→next=NULL。
     返回new_review。
}

int main()
{
    Review_Node *my_node=NULL;
```

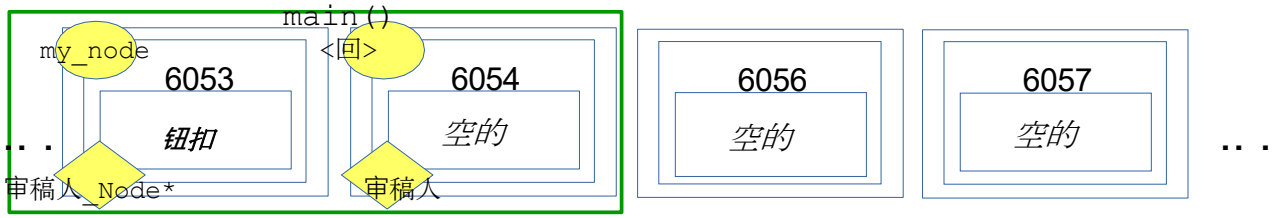
```
my_node=new_Review_Node()。

strcpy(my_node->rev.restaurant_name, "Veggie Goodness");
strcpy(my_node->rev.restaurant_address, "The Toronto Zoo, Section
C");
```

```
    my_node->rev.score=3。

    printf("The review node contains:\n");
    printf("Name=%s\n",my_node->rev.restaurant_name);
    printf("Address=%s\n",my_node->rev.restaurant_address);
    printf("Score=%d\n",my_node->rev.score)

    free(my_node);
    返回0。
}
```

编译和运行上面的代码会产生。

```
.../a.exe
审查节点包含。
名字=Veggie Goodness
地址=多伦多动物园，C区 得分=3分
链接=(nil)
```

让我们看看当我们运行上面的代码时到底发生了什么。首先，*main()*声明了一个指向"*Review_Node*"*的指针变量*。这意味着无论这里存储的是什么内存地址，我们都可以期望在这个位置找到构成*'Review_Node'的*所有信息。



需要注意的事项

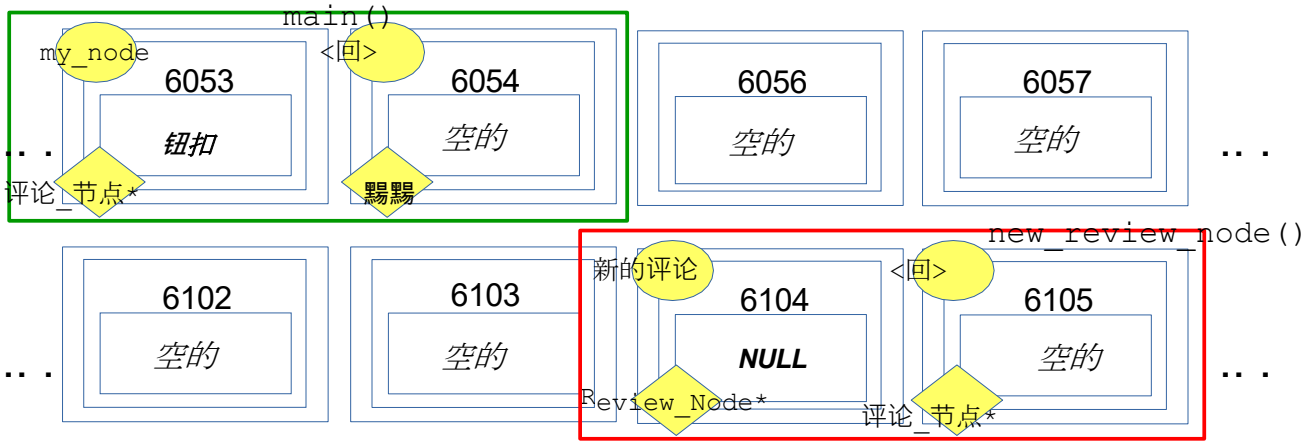
- 指针*'my_node'*是*main()*中唯一声明的变量。
- 它***不是***一个*'审查节点'*。
- 它被初始化为***NULL***，表示它是未分配的。
- 让我们不要忘记main()的返回值!接下来

我们有一个对*new_Review_Node()的*调用

。

`my_node=new_Review_Node()`。

这个函数声明了一个指向 *"Review_Node "*的单一*指针变量*，并有一个*返回值*是*一个指向 "Review_Node "的指针*。这些需要保留在内存中。



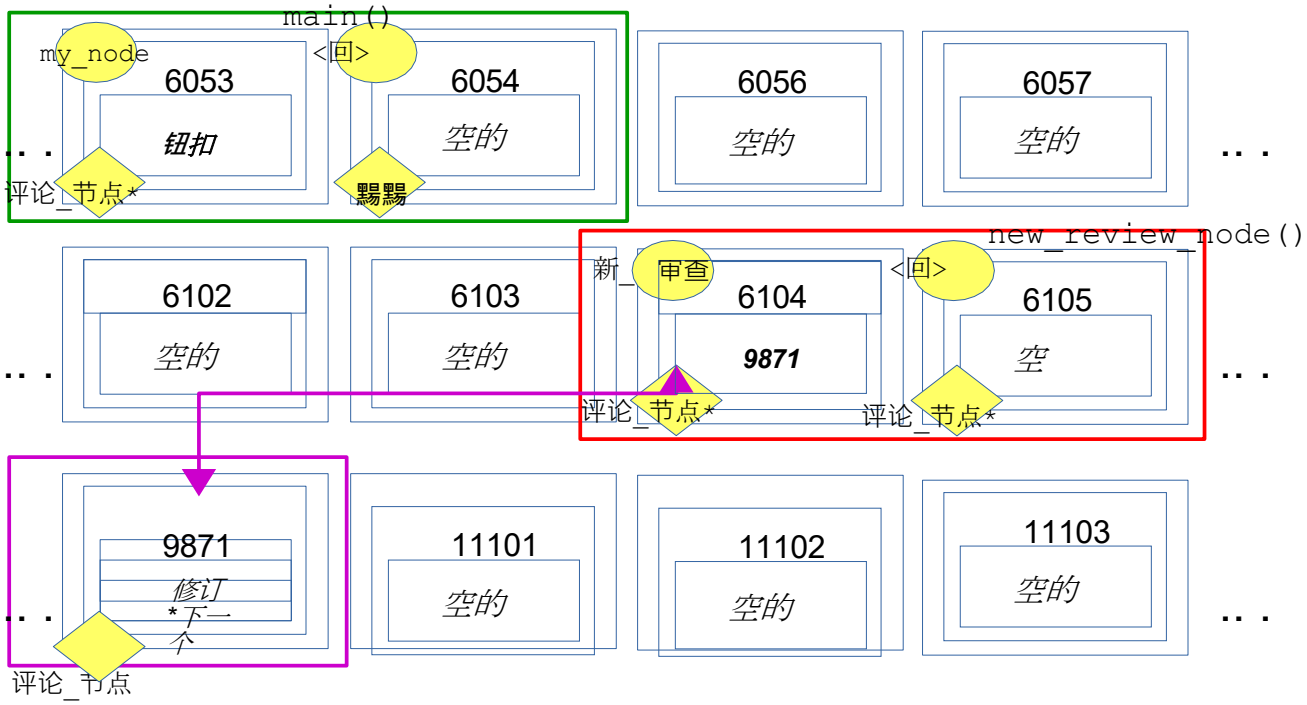
需要注意的事项。

- 为*main()*保留的所有空间都在绿框内。
- 为*new_Review_Node()*保留的所有空间都在红框内
- 这两个函数都没有声明一个*Review_Node*变量!

在*new_Review_Node()*中，内存被保留给新节点，我们得到一个指向新保留空间的指针

`new_review=(Review_Node *)calloc(1, sizeof(Review_Node))`。

在记忆中，结果会是这样的。



需要注意的事项。

- 新保留的*Review_Node*显示在紫色方框内。
- 它*没有名字标签*，因为它不是*我们在程序中声明的*一个*变量*。
- 因为它没有名字标签，所以唯一的方法是通过指针中的地址（#9871）来获得它。这就是为什么对*calloc()的*调用

```
new_review=(Review_Node *)calloc(1, sizeof(Review_Node));
```

返回新保留空间的地址。我们的指针*'new_review'*有新创建节点的地址。
- 新节点*不属于任何函数*。它*不是一个局部变量*。
- 新的*'Review_Node'*有两个字段，正如预期的那样：一个是*Review*类型的字段，即我们称之为 *"rev"*，而一个指向链表中下一个节点的指针，我们称之为 *"next"*。

*new_Review_Node()*中接下来的几行初始化了新节点的内容，以显示它没有被填充有效的数据

```
new_review->rev.score=-1;
strcpy(new_review->rev.restaurant_name,"")
。
strcpy(new_review->rev.restaurant_address,");
new_review->next=NULL。
```

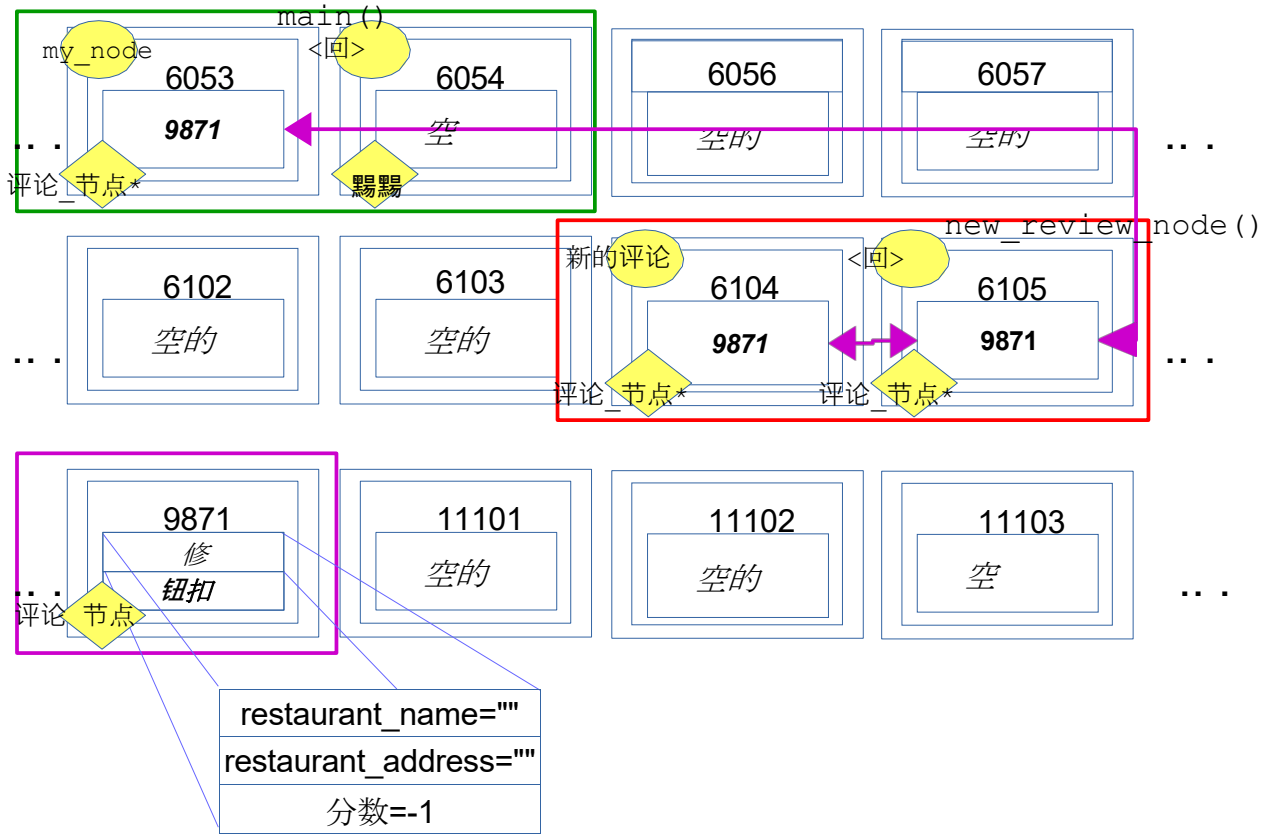
这将改变内存中新节点的内容

正如你在上图中看到的，*'rev'*有它的*三个字段*，所有这些都被初始化为合理的值，表明该节点没有被真实数据更新。

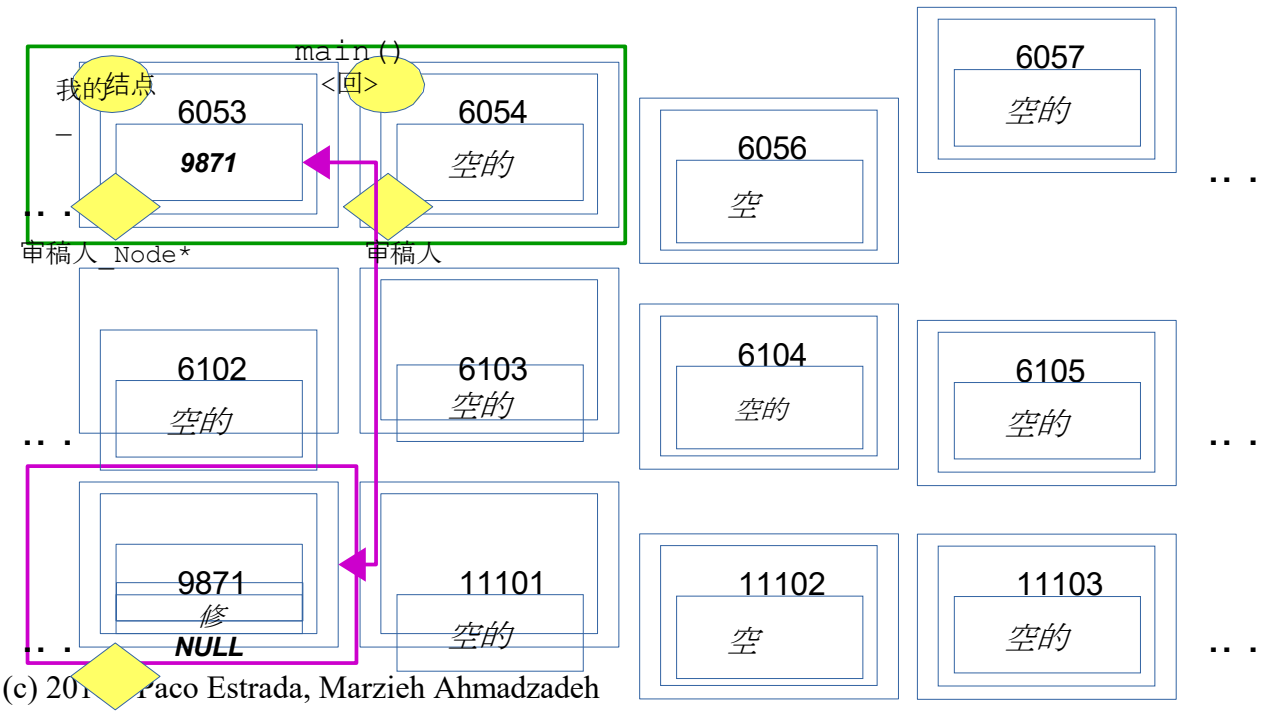
最后，*new_Review_Node()*返回一个*指向新创建节点的指针*。这意味着将节点的地址复制到*返回值中*，然后将其分配给*main()*的*'my_review'*。

```
my_node=new_Review_Node()。
```

因此，*在调用new_Review_Node()返回指向新节点的指针时*，内存的内容看起来像。



所以现在*main()*有一个指向新分配的 *"Review_Node "*的指针。最后，*new_Review_Node()*所保留的空间被释放，以便重新使用（你知道这发生在一个函数结束之后）。



评论_节点

需要注意的事项。

- 一旦对*new_Review_Node()*的调用完成，我们就在内存的某个地方有一个新的*'Review_Node'*，初始化后显示它目前没有实际数据。
- *main()*有一个指向该节点的指针，因此它可以访问和改变该节点的信息。
- 尽管新节点是由*new_Review_Node()*创建的，但当该函数结束时，它并没有被删除。新节点不属于任何函数，它也不是一个局部变量。*它将一直存在，直到我们决定释放它所使用的空间。*
- 因为新的节点没有名字标签，所以唯一的方法就是用指针来访问它。
- 结果是，*如果我们丢失了指针，我们就再也找不到这个节点了。*

*main()*中接下来的几行使用我们拥有的指针，为*存储在节点中的餐厅评论的字段*分配有意义的值。让我们只看其中一个。

```
my_node->rev.score=3。
```

这里的符号值得注意一下。首先，*'my_node'*是一个指向*'Review_Node'的*指针。评论节点有*两个字段：'rev'*是实际的评论，*'next'是*指向一个列表中下一个节点的指针。我们想更新*'rev'*中的评论内容，我们使用的是一个指针，所以要访问*'rev'*我们使用

```
my_node->rev
```

但这还不够，因为*'rev'*本身有*三个字段：餐厅名称*、*餐厅地址*和*分数*。因为*'rev'*是一个变量（不是一个指针！），我们使用点'.'操作符来访问它的字段。因此，把所有东西放在一起，这一行

```
my_node->rev.score=3。
```

可以理解为
"*进入地址存储在my_node的节点的'rev'字段，将'rev'中的'score'字段更新为等于3"。*

*看看main()*是如何更新餐厅名称和地址的，以及如何打印评论的内容的。所有这些都使用了我们对这个节点的指针。

一旦*main()*完成了对节点的处理，还有一个小细节需要处理。在*main()*退出之前，我们需要返回为 *"Review_Node "*保留的内存。

```
free(my_node)。
```

这告诉计算机我们已经用完了存储在*'my_node'*中的地址所保留的空间，并想释放这个空间。***你应该始终确保释放（free）你用calloc()申请的所有内存。***不释放你获得的内存被称为

"内存泄漏"，它可能会因为占用你的计算机内存而给你带来麻烦。

截至目前的总结

- 我们创建了一个名为 "*Review "的*复合数据类型来存储餐厅评论。
- 我们已经创建了一个 *"Review_Node*
  *"*数据类型，可以用来建立一个餐厅评论的链接列表
- 我们已经实现了一个为新的 *"Review_Nodes*
  *"*按需保留空间的函数，我们已经*详细地*看到了这个函数是如何工作的，空间是如何被保留的，以及我们如何使用指针来访问已经*按需*保留的内存。

*我们为什么要费这么大的劲呢？*在不同的课程中，我们写的代码可能会以更紧凑的方式来解释。特别是，这一行

```
my_node=new_Review_Node()。
```

可以用 *"函数new_Review_Node()分配了一个新的Review_Node，并返回其地址"来*解释。这是一个准确的陈述，但它并不能帮助你真正理解当我们按需保留内存时发生了什么，或者当你不得不（你将不得不！）编写创建和初始化不同类型数据项的代码时，你必须遵循的过程。因此，值得把整个过程详细地看一遍，并确保你能理解每一个步骤，这些步骤都是有意义的，从逻辑上看是需要完成的，而且当你经历分配和初始化一个新的数据项的过程时，你能直观地看到内存中发生了什么。

在这一点上，考虑到我们所做的关于如何处理变量、指针、复合类型和函数调用的所有例子，你应该对在C语言中执行一系列操作时发生的事情有相当好的理解。所以，从现在开始，我们将花更少的时间来研究运行代码时内存中的事物是如何变化的这些非常低级的细节，而开始在更高的层次上关注我们所做的事情的概念方面的程序--
这是不可避免的，因为我们将研究更复杂的程序，而研究其中的每一个步骤将比你整个学位的长度还要长

但不要忘记，C语言是一种非常简单明了的语言，它不会做任何你没有要求它做的事情。如果你用内存中对应于你的程序正在处理的数据的盒子，以及对这些盒子的操作来思考，你*总是*可以准确地弄清楚正在发生什么。每当你不确定发生了什么，就*拿一张白纸和一支铅笔，画一张内存图，并确保你明白你的代码在做什么!*

*练习。*编写一个小程序，用于
- 创建一个*int_node*数据类型，它代表一个链接列表中的节点，其中的数据项是单个整数值。
- 有一个函数用于分配和初始化*new_int_node()*。
- 在*main()*中创建一个新的*int_node*，并使用一个指针将其整数值更新为42。
- 使用指针来打印出*int_node*的内容。
- 在退出前释放*int_node*的内存。

建立一个链接的评论列表

在这一点上，我们拥有创建餐厅评论链接列表所需的一切。

- 我们知道如何定义一个复合数据类型来存储单个审查的数据
- 我们知道如何定义一个链接列表的节点类型，我们可以用它来将评论链接到一个列表中去
- 我们知道如何编写一个函数，按需分配新的审查节点，并返回一个指向新创建节点的指针，以便我们可以访问/修改其中的信息。
- 我们知道如何从终端读取用户的输入，所以我们可以获得信息来填补我们的评论。

现在是时候了，我们把所有的东西放在一起，变成一个小程序，能够从终端读取评论信息，把用户输入的信息填入按需分配的评论节点，并把这些节点连接起来，形成一个链接列表。

为了完成这一计划，我们将需要。

- 初始化一个新的（空）链表的代码
- 在链接列表中插入一个新创建的审查节点的代码
- 只要我们想，就可以打印列表中的评论的代码
- *main()*中的代码允许用户根据需要输入评论数量

我们将在一个更高的、更具概念性的层面上看这个代码，只有当这些细节说明了我们以前没有见过的新想法时，才会停下来看细节。

练习。用*伪码写*出我们需要在*main()*中实现的步骤，以便我们的程序能够

- 给用户提供了以下选择
    a) 输入一个新的评论
    b) 打印出到目前为止输入的所有评论
    c) 退出程序

- 如果用户选择a），程序应该执行所有需要的步骤，将新的评论添加到评论的链接列表中。

- 如果用户选择b），程序将*遍历*列表，依次打印出每条评论。

- 如果用户选择c），程序就会释放分配给链表的所有内存，并退出。

让我们来看看我们如何在*main()*中实现上述步骤。

```
int main()
{
```

```
    Review_Node *head=NULL;
    Review_Node *one_review=NULL;
    char name[MAX_STRING_LENGTH];
    char address[MAX_STRING_LENGTH];
    int score;
    int choice=1。

    while (choice!=3)
    {
        printf("Please choose one of the following:\n");
        printf("1 - Add a new review\n")。
        printf("2 - 打印现有评论/n"); printf("3 -
        退出此程序/n")。

        scanf("%d",&choice);
        getchar();

        如果（选择==1）
        {
            // 在这里，我们需要代码将一个新的评论添加到链接列表中。
        }
        否则，如果（选择==2）
        {
            // 这里我们将添加代码来打印现有的评论。
        }
    }

    // 用户选择#3 - 释放内存并退出程序。
}
```

上面的代码并不完整。它包含了我们需要完成的不同部分，以实现所有要求的功能。*然而*，它做了两件重要的事情。

- 它声明了一个新的、空的链表。该行

```
Review_Node *head=NULL。
```

声明了一个指向 *"Review_Node "的指针*，并将该指针设置为***NULL***。这就是我们如何在C语言中创建一个空的链表!

问题。我们如何检查一个链表是否为空？

- 它提供了一个循环，提示用户从1到3选择一个数字。根据用户的选择，执行不同的

代码。如果用户选择了'3'，循环就退出。

问题。如果用户输入的不是1-3中的数值，那么这个循环会做什么？

因此，我们项目的核心已经在那里了。

当你写一个复杂的程序时，写一个小的*驱动程序是*个好主意，它有一个像上面那样的循环，允许你分别测试程序的不同组件，并选择哪一个被测试，以什么顺序测试组件，以及向每个组件传递什么信息。

现在让我们填入细节。首先，让我们看一下选项'1'的代码，它应该在我们的链接列表中插入一个新的评论。

```
如果（选择==1）
{
   // 获得一个新的评论节点
   one_review=new_Review_Node()。

   //从终端读取信息来填写这个评论 printf("Please enter the restaurant's
   name/n")。
   fgets(name, MAX_STRING_LENGTH, stdin);
   printf("Please enter the restaurant's address\n");
   fgets(address, MAX_STRING_LENGTH, stdin);
   printf("Please enter the restaurant's score\n");
   scanf("%d", &score)。
   getchar()。

   //在新的评论节点中填入数据 strcpy(one_review-
   >rev.restaurant_name,name); strcpy(one_review-
   >rev.restaurant_address,address); one_review-
   >rev.score=score。

   //将新的评论插入到链接列表中 head=insert_at_head(head,one_review)。
}
```

上面的代码使用我们之前写的函数*new_Review_Node()*来分配和初始化一个新的*'Review_Node'*。你已经知道这个函数是如何工作的，以及当我们调用它时内存中会发生什么。在上面的代码中，我们可以简单地认为，我们获得了*一个指向新分配的'Review_Node'的指针*。

下一步是从用户那里获得信息来填写评论。一旦我们得到这些数据，我们就可以更新*'Rev'*变量里面的*字段，这些变量包含在'Review_Node'里面。记住!*我们在便当盒内有便当盒。*Review_Node'*包含一个名为*'rev'的'评论'*，该评论包含餐厅的*名称*、*地址*和*得分*。

最后一步是将*新节点插入到链表中*。为此我们有一个叫做 *insert_at_head()* 的函数（尚未实现！）。记得我们在上面谈到了我们可以通过三种不同的方式将一个节点插入

到列表中：*在头部*、*在尾部或在现有节点之间*。

这里我们将在头部插入新的节点，因为我们的程序不需要*以某种有意义的方式对评论进行排序*。这意味着列表中节点的顺序并不重要，而且我们知道在头部插入一个节点的*工作量最小*。

让我们看看如何实现*insert_at_head()*函数，看一个在最初的空列表中插入几个节点的例子。

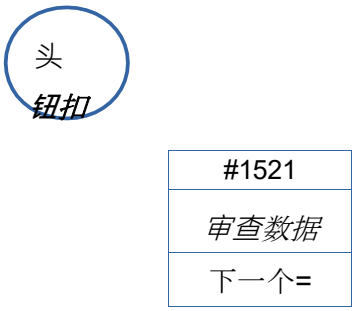
例子。在图中显示我们插入两条评论后的链接列表的样子。该列表最初是空的。

1) 初始状态。

我们有一个*指向列表头部的指针*，但最初它是***NULL***，所以列表是空的。

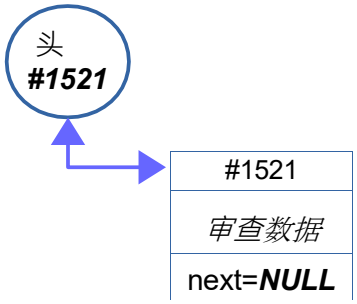
2) 插入第一个审查节点。

我们的程序将为新的审查分配和填入数据



新的*'Review_Node'*已经包含一个有效的评论，由用户输入。我们需要将其链接到列表中。记住在列表头部链接一个新节点的过程。

- 我们将当前在*头部指针*中的地址复制到*新节点*中的*'下一个'*指针上
- 我们把*新节点*的地址（我们得到了它的指针！）复制到*头部指针*。



我们现在有了一个链表!它只有一个节点，***即头部节点***，但它是一个适当的链表。

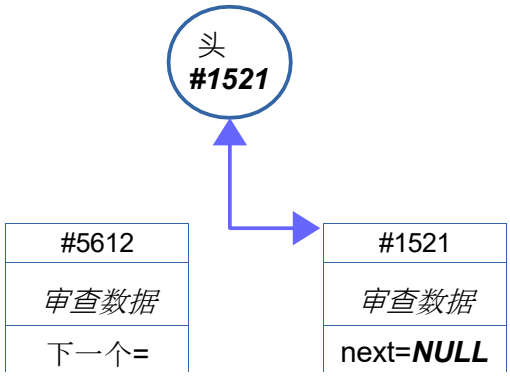
需要注意的事项。

- 我们的***头部节点***中的'*下一个*'指针是***NULL***，因为我们把之前的头部指针复制到了它上面，而这个指针是***NULL***。
- ***头部指针不是列表中的一个节点***，它只是列表中第一个节点的地址。
- ***不要把头部指针和头部节点混淆起来!***

重要提示：确保*列表中最后一个节点*的*"下一个"*指针是***空的，这***一点至关重要。如果它包含*垃圾*，或者是以前的指针值，那么任何使用链表的程序都会认为有更多的节点，并在*"下一个"*指针中找到的任何地址去寻找它们。这是一种不好的错误类型--它会产生不可预测的行为，或者，如果你幸运的话，会使你的程序崩溃。如果你在使用链接列

3) 在列表中插入第二个评论。

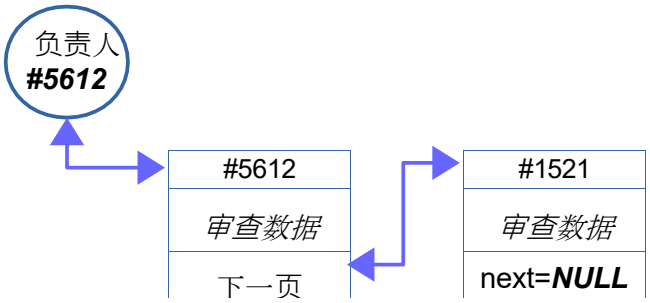
再一次，我们的程序将分配并填入一个新的*"Review_Node"*的数据。就在我们将新节点链接到列表之前，我们有这样的情况。



我们再次

- 取*当前头部指针*中的地址，并将其复制到*新节点的*'*下一个*'指针中。
- 然后我们把*新节点*的地址复制到*头部指针上*

这使我们的名单看起来像这样。



同样的过程将使我们能够向我们的链表添加尽可能多的节点。

练习。显示我们再插入两条评论后的列表，第一条评论的地址是#3141，第二条评论的地址是#9811。

在了解了插入过程的工作原理后，让我们写一个函数来向列表中插入一个新节点。

```
Review_Node *insert_at_head(Review_Node *head, Review_Node *new_node)
{
      // 这个函数在列表的头部添加一个新的节点。
      // 输入参数。
      //head      。指向当前列表头部的指针
      //new_node 。指向新节点的指针
      // 返回。
      /           /新的头部指针

      new_node->next=head;
      return new_node;
}
```

正如你所看到的，这是一个相当短的函数!-
它将*当前头部节点的地址*复制到*新节点的'next'*指针（使用'->'操作符，因为*'new_node'*是一个指针）。然后它返回*新头部节点的地址*--它包含在指针*'new_node'中*。

练习。画一个内存图，显示当我们调用*insert_at_head()*时发生了什么。功能。你的图表应该显示

- 来自*main()*的*头部*指针变量
- 当前在内存中某处的*头部节点*
- *main()*中的*new_*node指针变量

- 内存中某处的*新节点*
- *insert_at_head()*的参数和返回值
- 所有指针的*最终*值（在对*insert_at_head()的*调用完成后）。

这就完成了选项'1'，在列表中插入一个节点。现在让我们看看如何实现选项'2'--打印当前列表中的所有评论。

我们为实现选项'2'而必须进行的过程是你必须对链表进行的*最常见的操作之一。遍历链表，*同时在每个节点进行一些特殊的操作。这里的操作只是简单地打印出内容，但在更复杂的应用中，你的链表包含各种复杂的信息，操作本身可能是相当复杂的。无论进行什么操作，*列表的遍历过程都是相同的。*请确保你完全理解它是如何工作的!

遍历一个链表

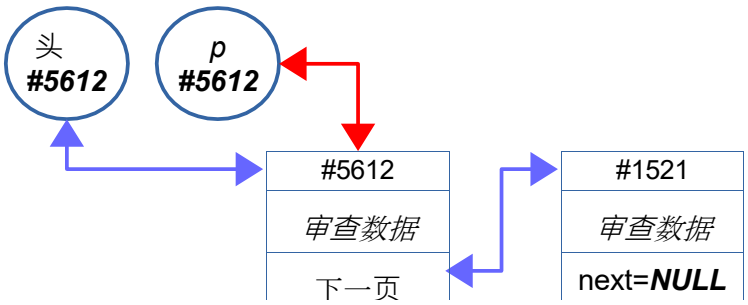
这个过程要求你。

- 设置一个*指针*，当我们在列表中向下移动时，该*指针*将被更新，以指向当前正在处理的节点。
- 将*遍历指针*初始化为列表*头部节点*的*地址*。
- 写一个循环，即。
    * 处理地址在当前*遍历指针*中的节点
    * 更新*遍历指针*以指向列表中的下一个节点
    * 当*遍历指针*为***NULL***时，循环结束。这表明已经到达了列表的末端。

让我们应用上述方法编写一个小函数，打印出列表中的所有评论。

```
void print_reviews(Review_Node *head)
{
      审阅_节点       *p=NULL;      // 遍历指针

      p=head;                  //将遍历指针初始化为
                                     // 指向头部节点
      while (p!=NULL)
      {
         // 打印出这个节点的评论
         printf("Restaurant Name: %s\n",p->rev.restaurant_name);
         printf("Restaurant Address: %s\n",p->rev.restaurant_address);
         printf("Restaurant Score: %d\n",p->rev.score)。

         // 更新遍历指针以指向下一个节点 p=p->next。
      }
}
```

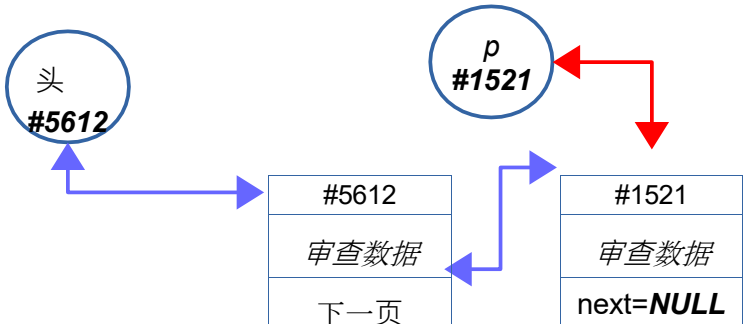
这值得我们思考一下。让我们看看它是如何与我们上面使用的示例链表一起工作的。



打印函数声明了一个名为 *"p "的遍历指针*，并将其初始化为指向*头部节点*（其地址由*头部*指针提供：#5612）。

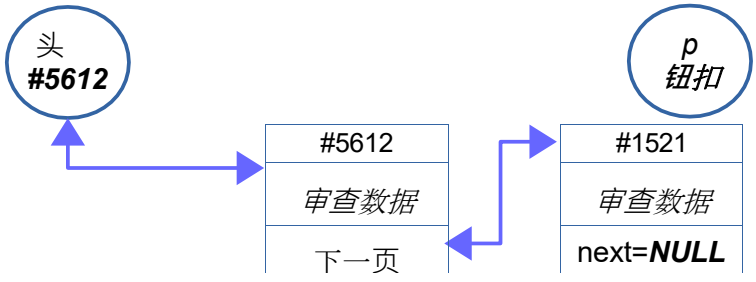
现在，该函数进入了一个循环。
- 打印出'*p*'所指向的节点的内容（#5612）。
<打印一家餐厅的信息>。
- 然后它用节点（#1521）的'*下一个*'指针中的地址更新'*p*'，所以我们有

。



我们再次经历了这个循环。
- 打印出'*p*'所指向的节点的内容（#1521）。
<打印了不同餐厅的信息>。
- 然后它用节点的'*下一个*'指针中的地址更新'*p*'（***NULL***），现在我们有

了。



在这一点上，循环退出（我们已经到达了列表的末端）。

你应该从中得到什么。

- 遍历一个链接列表需要使用一个*遍历指针*，该*指针*被更新以指向列表中的每个连续节点。
- 遍历过程是直截了当的。
    * 将*遍历指针*初始化为*头部节点的地址*
    * 循环，直到*遍历指针为**NULL***。
        - 在节点上执行所需的操作
        - 更新*遍历指针*以指向 *下一个节点*

你将会经常这样做!因此，要确保你对遍历过程有一个非常扎实的了解。

问题。如果我们向打印函数传递*一个空列表，*会发生什么？它是正常工作还是会使我们的程序崩溃？

我们的小程序的最后部分涉及选项'3'，当用户希望退出时。如果不是为了*释放我们在列表中为审查申请的所有内存*这个小细节，这将是微不足道的。

事实证明，释放链接列表的内存只是我们上面讨论的*列表遍历过程的另一种应用*只是在这种情况下，我们不是打印节点的内容，而是释放分配给该节点的内存。这里有一个小函数，在我们的程序之后进行清理。

```
void delete_list(Review_Node *head)
{
      Review_Node *p=NULL;
      Review_Node *q=NULL。

      p=头。
      while (p!=NULL)
      {
            q=p->next;
            free(p);
            p=q;
      }
}
```

你应该认识到列表遍历过程中的所有步骤。唯一奇怪的细节是，我们有两个指针，*'p'*和*'q'*。为什么需要这样做呢？

- 该循环将*释放*分配给地址在*'p'*中的节点的内存。
- 然而，*下一个节点的地址*是存储在我们要删除的节点中的。

- 如果我们*在释放*这个节点的内存*后*试图访问*'p->next'*，我们的程序就会崩溃。

- 所以我们使用*'q'*来临时存储列表中*下一个节点的*地址。然后我们可以删除该节点，并用我们保存在*'q'*中的地址来更新*遍历指针*。

把我们上面建立的所有东西放到一个完整的程序中，我们得到下面的清单。

```
/*
   CSC A48 - 第3单元 - 容器、ADT和关联列表

   这个程序实现了一个餐厅评论的链接列表。该程序允许用户根据需要输入评论，打印
   现有的评论，完成后，在退出前释放分配给列表的所有内存。

   (c) 2018 - F. Estrada & M. Ahmadzadeh。
*/

#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define MAX_STRING_LENGTH 1024

Typedef struct Restaurant_Score
{
       char restaurant_name[MAX_STRING_LENGTH];
       char restaurant_address[MAX_STRING_LENGTH];
       int score;

}审查。

typedef 结构 Review_List_Node
{
     审查修订。
     结构 R

Review_Node *new_Review_Node(void)
{
     Review_Node *new_review=NULL          // 指向新节点的指针

     new_review=(Review_Node *)calloc(1, sizeof(Review_Node))。

     // 初始化新节点的内容，其值显示为
     //它还没有被填满。在我们的例子中，我们把分数设为-1。
     // 将地址和餐厅名称都改为空字符串 ""
```

```
      new_review->rev.score=-1;
      strcpy(new_review->rev.restaurant_name,"")
      。
      strcpy(new_review->rev.restaurant_address,");
      new_review→next=NULL。
      返回new_review。
}

Review_Node *insert_at_head(Review_Node *head, Review_Node *new_node)
{
      // 这个函数在列表的头部添加一个新的节点。
      // 输入参数。
      /          /head : 指向当前列表头部的指针
      //new_node。指向新节点的指针
      // 返回。
      /          /新的头部指针

      new_node->next=head;
      return new_node;
}

void print_reviews(Review_Node *head)
{
      审阅_节点       *p=NULL;      // 遍历指针

      p=head;                 //将遍历指针初始化为
                                    // 指向头部节点
      while (p!=NULL)
      {
         //打印出这个节点的评论
         printf("*********************************************\n");
         printf("Restaurant Name: %s\n",p->rev.restaurant_name);
         printf("Restaurant Address: %s\n",p->rev.restaurant_address);
         printf("Restaurant Score: %d\n", p->rev.score);
         // 更新遍历指针以指向下一个节点 p=p->next。
      }
}

void delete_list(Review_Node *head)
{
      Review_Node *p=NULL;
      Review_Node *q=NULL。
```

```
p=头。
while (p!=NULL)
{
```

```
            q=p->next;
            free(p);
            p=q;
      }
}

int main()
{
      Review_Node *head=NULL;
      Review_Node *one_review=NULL;
      char name[MAX_STRING_LENGTH];
      char address[MAX_STRING_LENGTH];
      int score;
      int choice=1。

      while (choice!=3)
      {
            printf("Please choose one of the following:\n");
            printf("1 - Add a new review\n")。
            printf("2 - 打印现有评论/n"); printf("3 -
            退出此程序/n")。

            scanf("%d",&choice);
            getchar();

            如果（选择==1）
            {
               // 获得一个新的评论节点 one_review=new_Review_Node()
               。

               //从终端读取信息来填写这个评论 printf("Please enter the
               restaurant's name/n")。
               fgets(name, MAX_STRING_LENGTH, stdin);
               printf("Please enter the restaurant's address\n");
               fgets(address, MAX_STRING_LENGTH, stdin);
               printf("Please enter the restaurant's score\n");
               scanf("%d", &score)。
               getchar()。

               //在新的评论节点中填入数据 strcpy(one_review-
               >rev.restaurant_name,name); strcpy(one_review-
               >rev.restaurant_address,address); one_review-
               >rev.score=score。

               //将新的评论插入到链接列表中
```

```
        head=insert_at_head(head,one_review)。
    }
```

否则，如果（选择==2）

```
            {
                print_reviews(head)。
            }
        }

        // 用户选择#3 - 释放内存并退出程序。 delete_list(head)。
        返回0。
}
```

到目前为止，我们所取得的成就

在这一点上，你知道如何建立一个包含复合数据类型项目的链表。这是一件大事--有大量的应用程序依靠链表来组织和处理信息！你会发现链表有各种不同的风格和不同的编程语言。你会发现链接列表有各种不同的风格，在不同的编程语言中。因此，请记住以下几点。

- 不管它包含什么，你使用什么编程语言，以及它要支持什么应用，链表的组织都是一样的。
- 在列表中插入新节点的过程也是如此。实际的实现会根据你的编程语言而改变，但步骤是一样的。
- *列表遍历*的过程也与编程语言、列表内容或应用程序无关。

因此，确保你已经理解了这三件事背后的*概念和过程*，它们是我们到现在为止所做工作的最重要部分。

你应该对我们上面开发的程序的代码感到满意。确保你明白每一步是怎么回事，以及其中的每个函数是如何工作的。检查你是否理解的一个好方法是*向别人解释代码是如何工作的*，或者*用你自己的话为自己写一个总结，解释代码在做什么和为什么*。

下一步是什么？

在链表上有两个主要的操作我们还没有学会：*搜索一个特定的项目*，以及从列表中*删除项目*。让我们来看看这些操作，以完成我们对链表的学习。

10.- 在大型数据集中搜索特定项目

我们开始这一节的目的是了解如何组织、存储和操作大量的信息集合。也许这样做的最重要方面是能够在数据集合中搜索感兴趣的项目。考虑一下这个词你有多少次。

- 使用谷歌查找文件、课堂笔记、新闻或图片
- 使用在线零售商店的搜索功能来寻找你想要的物品
- 按歌曲名称或艺术家姓名搜索特定的音乐视频

现实世界中大量的应用程序都有一个内置的搜索功能，可以让你找到并探索存储在一个大集合中的特定数据项。在很大程度上，这些应用程序的有用性与它们能够如何有效和准确地找到用户需要的信息有关。

在计算机科学中，人们投入了大量的精力来弄清楚什么是组织信息的最佳方式，以便我们能够*快速搜索非常大的集合*。在本课程中，我们将开始研究这个问题，看看我们能在多大程度上使用链表，并了解搜索一个被组织为链表的大集合需要多少工作。

这将为我们打开一扇门，让我们开始从特定*算法*或特定*数据结构*的*效率*方面进行思考，从而允许我们*在实现同一ADT的不同数据结构之间进行*选择，*和/或在不同的ADT之间*选择提供*最佳性能的ADT*（正如我们将看到的，*性能*的定义取决于我们想通过程序实现的目标）。

在一个链接列表中进行搜索

在一个链接列表上的搜索过程只是一种*列表遍历的*形式。我们已经看到了列表遍历的工作原理，当我们进行*搜索*时，唯一的区别是在一个节点上进行的操作是*搜索键*和*存储在列表节点中的值*之间的*比较*。搜索过程将是

- 找到请求的*搜索键*，并返回一个指向包含该键的节点的指针。
- 或 -
- 遍历整个列表，没有找到钥匙，并返回***NULL***

让我们看看如何为我们的餐厅评论链接列表编写一个搜索函数，以便我们能够*更新列表中某个特定餐厅的分数*。这个搜索函数应该接受一个餐厅名称作为*搜索键*，并返回一个指向包含该餐厅评论的节点的指针，否则就返回***NULL***，表示我们的列表中没有该名称的餐厅。

```
Review_Node *search_by_name(Review_Node *head, const char name_key[] )
{
    // 在链接列表中寻找一个包含有
    //对一家名称与'name_key'相符的餐厅进行评论
    // 如果找到，返回一个指向有评论的节点的指针。否则
    //返回NULL。

    审阅_节点        *p=NULL;      // 遍历指针
```

```
        p=头。
        while (p!=NULL)
        {
          如果(strcmp(p->rev.restaurant_name,name_key)==0)
          {
              // 找到了钥匙!返回这个节点的一个指针 return p;
           }
           p=p->next。
        }
        return NULL;      // 没有找到搜索关键词!
}
```

上面的代码浏览了链接列表，在每个节点，它将存储在该节点的评论中的餐厅名称与*搜索键*进行比较，如果它们相等，它将返回该节点的指针。

注意事项。

1- 这是一个代码的例子，在这个例子中，提前退出循环是非常有意义的--一旦我们找到了*搜索键*，我们就返回到我们找到它的节点的指针。想象一下，一个有数百万个条目的列表，在我们找到我们要找的东西之后，继续遍历每个节点是*没有意义的*。

2- 你可能已经注意到函数声明中有一个*我们以前没有见过的关键词*。`Review_Node *search_by_name(Review_Node *head, const char name_key[])`的部分，我们声明*搜索*键为*'const char name_key[]'*。正如你所知，字符串*的数组*，所以我们可以通过声明一个参数*"char"*，将一个字符串传入搜索函数。*name_key[]'*。然而，你也知道，如果我们给一个函数一个指向数组的指针，*该函数可以去改变该数组的内容*

由于我们*希望搜索函数**不**修改搜索键*，所以在编写函数时，将其输入参数声明为*'const char name_key[]'是*很好的编程实践。***const***"关键字指定该数组的内容是*恒定的，在函数中不能被改变*（这个想法你在A08和Python中应该很熟悉--在C语言中声明为*"const"*的数据项是*不可变的*。然而，在C语言中，没有任何数据类型是固有的可变或不可变的，这取决于你决定何时和如何使用*'const'*）。

让数组成为*常数*可以实现两件事。

- 它向使用你的代码的任何人保证，*search_by_name()*函数***不会***改变你传递给它的任何字符串。
- 它可以确保你不会因为改变了一个不属于你的输入参数而引入一个错误。

将被你的函数修改。

const "关键字可以与任何数据类型一起使用，包括由以下声明的复合类型
你。

我们现在可以修改我们原来的程序--
处理餐馆评论的程序，使它允许用户选择更新已经添加到列表中的评论。这需要我们对*main()*中的选项列表做一些修改。

```
printf("Please choose one of the following:\n");
printf("1 - Add a new review\n")。
printf("2 - Print existing reviews\n"); printf("3
- Update review for one restaurant\n"); printf("4
- Exit this program\n") 。

scanf("%d",&choice);
getchar();
```

而我们需要添加代码，使用我们的搜索功能来寻找一个特定的餐厅，并更新它的分数。

```
否则，如果（选择==3）
{
  printf("Which restaurant's score do you want to update?\n");
  fgets(name,MAX_STRING_LENGTH, stdin);
  one_review=search_by_name(head,name);
  如果(one_review==NULL)
  {
       printf("对不起，那家餐厅似乎不在本网站内。
list\n")
  。
  }
  否则
  {
    printf("Please enter new score for the restaurant/n");
    scanf("%d",&one_review->rev.score)。
    getchar()。
  }
}
```

将这些改进添加到我们的代码中，使我们能够更新已经添加到我们链接列表中的餐馆的评论。完整的程序清单出现在下面。

```
/*
   CSC A48 - 第3单元 - 容器、ADT和关联列表
   这个程序实现了一个餐厅评论的链接列表。该程序允许用户根据需要输入尽可能多的
```

```
   来打印现有的评论，完成后，它在退出前释放分配给列表的所有内存。

   (c) 2018 - F. Estrada & M. Ahmadzadeh.
*/

#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define MAX_STRING_LENGTH 1024

Typedef struct Restaurant_Score
{
       char restaurant_name[MAX_STRING_LENGTH];
       char restaurant_address[MAX_STRING_LENGTH];
       int score;

}审查。

typedef 结构 Review_List_Node
{
      审查修订。
      结构 Review_List_Node *next;
}Review_Node;

Review_Node *new_Review_Node(void)
{
      Review_Node *new_review=NULL;       //新节点的指针


      new_review=(Review_Node *)calloc(1, sizeof(Review_Node))。

      // 初始化新节点的内容，其值显示为
      //它还没有被填满。在我们的例子中，我们把分数设为-1。
      // 将地址和餐厅名称都改为空字符串 ""
      // 非常重要的是!将'下一个'指针设置为NULL

      new_review->rev.score=-1;
      strcpy(new_review->rev.restaurant_name,"")
      。
      strcpy(new_review->rev.restaurant_address,");
      new_review→next=NULL。
      返回new_review。
}

Review_Node *insert_at_head(Review_Node *head, Review_Node *new_node)
{
     // 这个函数在列表的头部添加一个新的节点。
```

```
// 输入参数。
/           /head : 指向当前列表头部的指针
//new_node 。指向新节点的指针
// 返回。
/            /新的头部指针
```

```
      new_node->next=head;
      return new_node;
}

void print_reviews(Review_Node *head)
{
       Review_Node*p=NULL      ; // 遍历指针


       p=head;                   //将遍历指针初始化为
                                        // 指向头部节点
       while (p!=NULL)
       {
          //打印出这个节点的评论
           printf("*********************************************\n");
           printf("Restaurant Name: %s\n",p->rev.restaurant_name);
           printf("Restaurant Address: %s\n",p->rev.restaurant_address);
           printf("Restaurant Score: %d\n", p->rev.score);
           // 更新遍历指针以指向下一个节点 p=p->next。
       }
}

void delete_list(Review_Node *head)
{
       Review_Node *p=NULL;
       Review_Node *q=NULL。


       p=头。
       while (p!=NULL)
       {
            q=p->next;
            free(p);
            p=q;
       }
}

Review_Node *search_by_name(Review_Node *head,\
                              const char name_key[MAX_STRING_LENGTH])
{
       // 在链接列表中寻找一个包含有
       //对一家名称与'name_key'相符的餐厅进行评论
       // 如果找到，返回一个指向有评论的节点的指针。否则
       //返回NULL。

       Review_Node*p=NULL      ; // 遍历指针p=head。
       while (p!=NULL)
       {
         如果(strcmp(p->rev.restaurant_name,name_key)==0)
         {
              // 找到了钥匙!
```

```
            返回p。
        }
         p=p->next。
      }
      return NULL;       // 没有找到搜索关键词!
}

int main()
{
      Review_Node *head=NULL;
      Review_Node *one_review=NULL;
      char name[MAX_STRING_LENGTH];
      char address[MAX_STRING_LENGTH];
      int score;
      int choice=1。

      while (choice!=4)
      {
            printf("Please choose one of the following:\n");
            printf("1 - Add a new review\n")。
            printf("2 - 打印现有的评论/n")。
            printf("3 - Update review for one restaurant\n");
            printf("4 - Exit this program\n")。

            scanf("%d",&choice);
            getchar();

            如果（选择==1）
            {
               // 获得一个新的评论节点
               one_review=new_Review_Node()。

               //从终端读取信息来填写这个评论 printf("Please enter the restaurant's
               name/n")。
               fgets(name, MAX_STRING_LENGTH, stdin);
               printf("Please enter the restaurant's address\n");
               fgets(address, MAX_STRING_LENGTH, stdin);
               printf("Please enter the restaurant's score\n");
               scanf("%d", &score)。
               getchar()。

               //在新的评论节点中填入数据 strcpy(one_review-
               >rev.restaurant_name,name); strcpy(one_review-
               >rev.restaurant_address,address); one_review-
               >rev.score=score。

               //将新的评论插入到链接列表中
               head=insert_at_head(head,one_review)。
            }
```

```
否则，如果（选择==2）
{
   print_reviews(head)。
}
```

```
            否则，如果（选择==3）
            {
               printf("Which restaurant's score do you want to update?\n");
               fgets(name,MAX_STRING_LENGTH, stdin);
               one_review=search_by_name(head,name);
               如果(one_review==NULL)
               {
                  printf("对不起，那家餐厅似乎不在本网站内。
list\n")。
               }
               否则
               {
                  printf("Please enter new score for the restaurant/n");
                  scanf("%d",&one_review->rev.score)。
                  getchar()。
               }
            }
      }

      // 用户选择#3 - 释放内存并退出程序。 delete_list(head)。
      返回0
```

练习。编译并运行上面的代码，插入一些评论，打印列表中的评论，然后修改其中一个评论。一定要*测试在什么情况下会发生什么。*

- 你试图打印一个空的列表
- 你试图修改一个不存在的餐厅的评论（还没有），在列表中。
- 当提示时，你选择了一个不在1-4的选项

尝试*破坏*程序。看看你能做什么来使它表现得奇怪或崩溃，然后想想*你将如何防止用户以这种方式破坏程序。*

练习。写一个搜索函数*search_by_address()*，允许你通过搜索餐厅的地址来修改该餐厅的分数。在*main()*中的菜单上添加一个选项，允许用户这样做，并实现更新分数的代码。***测试***你的代码，确保它是可靠的，更新正确的评论，并且在用户输入一个不存在的地址时不会中断。

练习。编写一个搜索函数，*打印出所有评论分数>=大于指定搜索键值的餐馆。*如果用户想查看分数等于或大于指定值的餐馆，这将是非常有用的。

问题。假设我们要在一个有1,000,000个节点的列表中按名字搜索一个特定的餐馆。在*最坏的情况下*（也就是我们的程序必须做最多工作的情况！），我们要检查多少个节点才能找到想要的餐馆或者确定它不在列表中？

如果列表中有10,000,000个节点，这个数字会有什么变化？如果是100,000,000个节点呢？

你应该从上述内容中得到什么。

- 在一个链表上搜索只是一个*列表遍历*，检查每个节点的*搜索关键字*
- 因为它是一个*列表遍历*，我们可能要在整个列表中寻找一个节点
- 这意味着我们在*搜索*过程中的工作量会随着列表的长度而增加（还记得你在苏黎世的储物柜中寻找T恤衫的步骤吗！）。
- 我们说，*搜索*操作在链表上具有*线性复杂性*。这意味着搜索函数的工作量可以用*k*n*来描述，其中*k*是某个常数，*n*是列表中的节点数。

一个具有*线性复杂度*的算法或函数就目前而言还不算太糟糕，但如果你考虑到非常大的数据集合（例如谷歌索引的数百万个文件），你应该很清楚，一个链接列表根本不是一个足够快的数据结构来组织数据。想象一下，如果你每次在谷歌上输入一个搜索关键词，它都要在一个长达几十亿个节点的列表中寻找它，那得花多长时间啊!

我们将需要*一种更快的方法来*对大型集合*进行搜索*。不幸的是，我们没有什么办法使在链表上的搜索更快，所以我们必须想出更聪明的数据结构。但这是稍后的事。现在，让我们定下几个与搜索有关的重要想法，这些想法以后会很重要（例如，如果你选择花时间研究和使用数据库）。

关于搜索的思考

我们应该花点时间思考我们使用的*搜索*键来查看我们的评论列表。

*问题。*一个*好的*搜索关键词应该有什么属性？

思考一下*餐厅的名字*。乍一看，这可能是一个很好的选择，而且对我们这个由一个用户输入一些评论的小程序来说，它很有效。但是想想看。

- 如果用户输入"*麦克唐纳*"作为搜索关键词会怎样？
    * 我们是否会期望有*一个单一的节点*来代表MacDonald's？
    * 我们是否期望找到多个条目？(例如，每个不同的地点都有一个)
    * 如果一个*搜索键*有多个匹配，程序应该怎么做？
        一个一个地更新？
        要求提供更多的信息来单列出一个地点？放弃并拒绝更新？

这里要说明的是，*尽管我们可以使用任何字段来搜索信息*，但一个好的搜索引擎会有办法来唯一地识别一个集合中的每个条目。

在设计数据库时，必须进行的基本任务之一是确定数据库的*模式*--即包含数据库将记录和处理的信息的*字段*列表，以及可用于搜索数据库内信息的*键*列表。

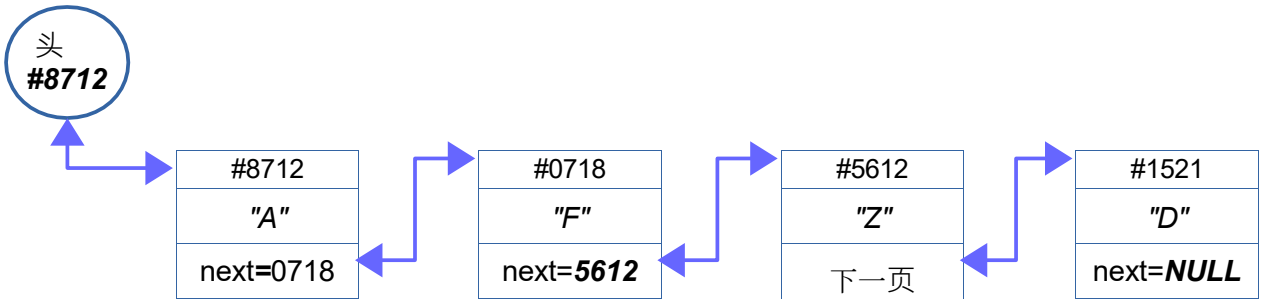
生成识别数据库中每个项目的*唯一键*对于保持信息的一致性至关重要--你在加入大学时得到一个唯一的*学生号是有*原因的！这也是为什么你会得到一个唯一的*学生号*。

如果你参加我们的数据库课程CSC C43，你可以学到更多关于*键*、*记录和数据库*的知识。就目前而言，你只需考虑如果用户想插入多个具有相同餐馆名称（不同地址）的评论，或不同名称但相同地址的评论，你的程序应该怎么做。

11.- 从一个链表中删除节点

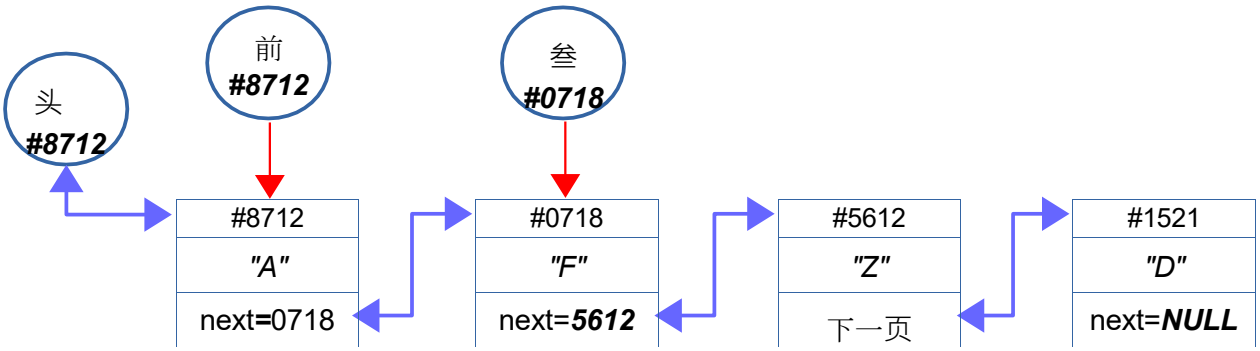
我们需要实现的最后一个操作是*删除*或*移除*操作，以完成我们的链表。顾名思义，它从一个列表中删除一个特定的节点。因为它寻找的是一个特定的项目，所以它涉及到一个*稍微修改的搜索过程*--所以它本质上是一个*列表遍历*操作。

让我们先看看一张图，它显示了如果我们想删除一个链表中的特定节点，我们需要做什么。



假设我们想*删除*包含 *"Z "*项的节点。我们将使用一个稍加修改的*列表遍历来*完成这个任务，它使用两个指针来寻找*我们要删除的节点*和*它的前身*。我们需要跟踪一个节点的前身，因为我们需要把它与刚刚被删除的节点连接起来。在上面的例子中，我们要删除带有 *"Z "*的节点，我们需要把它的*前身*（带有 *"F "*的节点）和它的*后继者*（带有 *"D "*的节点）联系起来。

遍历从遍历指针开始，如下图所示。

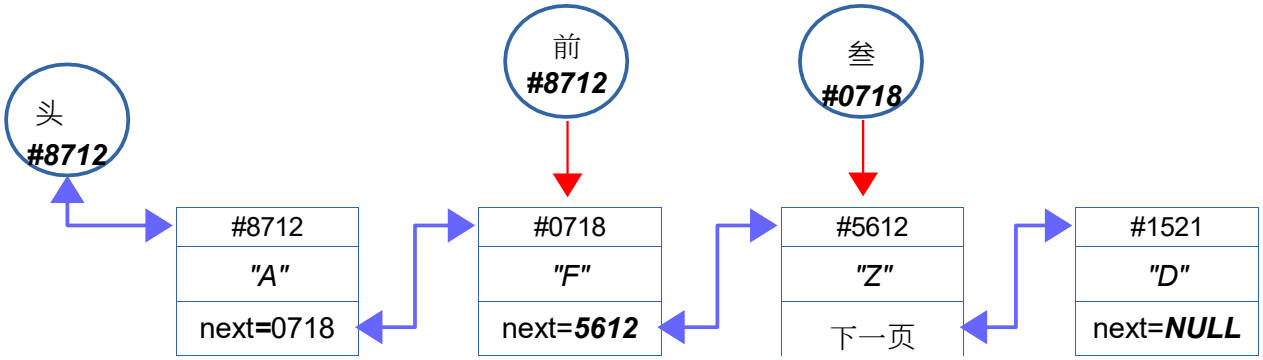


*前任指针'pre'*指向*头部节点*，*遍历指针'tr'*指向*'pre'*之后的一个节点。

然后*我们进行循环，直到找到我们想要的节点，或者'tr'为空*。

- 检查*'tr'*处的节点是否包含我们想要删除的项目
  * 如果是，我们就删除该节点并退出循环。
- 否则，将*'pre'*和*'tr'*都移到下一个节点。

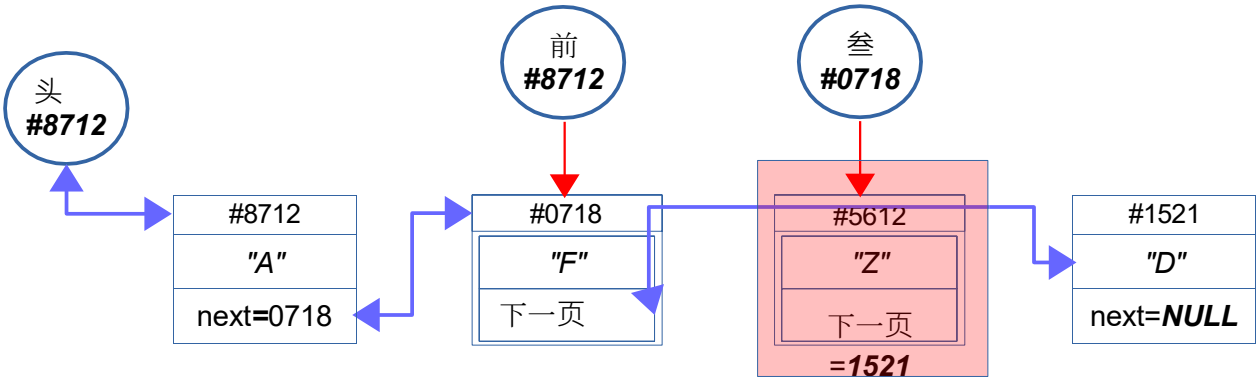
在上面的例子中，我们正在寻找带有 *"Z "*的节点，它不在*'tr'*处，所以我们将两个指针移到下一个节点。



现在我们在*'tr'*处有一个带有 *"Z "的*节点，所以我们继续删除该节点。这涉及到。

- 将*'下一个'*指针从*'tr'*处的节点复制到*'pre'*处的*'下一个'*指针。在上面的例子中，它将#1521复制到有 *"F "*的节点的*'下一个'*字段。这有效地连接了被删除节点的*前任*和*后任*。
- 然后我们*释放*被删除的节点。
- 我们退出这个循环。

这使我们的名单处于以下状态。



列表仍然是正确的链接，而被删除的节点显示为红色。

这个过程对*头部节点以外的*任何节点都有效。头部节点是一个特例，因为*它没有一个前身！它*的前身是什么？

问题。*删除*列表中的*头部节点*的过程是什么？让我们看看删除节

点的函数是什么样子的。

```
Review_Node *delete_by_name(Review_Node *head, const char name_key[] )
{
      // 这个函数从链接列表中删除包含以下内容的节点
的
      //有匹配的餐厅名称的评论。Review_Node *tr=NULL。
      Review_Node *pre=NULL。

      if (head==NULL) return NULL;      //空的链表！

      // 设置前身和遍历指针，使之指向
首先
      // 列表中的两个节点。  pre=head。
      tr=head->next。

      // 检查我们是否必须删除头部节点
      如果(strcmp(head->rev.restaurant_name, name_key)==0)
      {
            free(pre);          //删除列表中的第一个节点
            return tr;          //返回第二个节点的指针（新头！）。
      }

      while(tr!=NULL)
      {
```

```
        如果(strcmp(tr->rev.restaurant_name, name_key)==0)
```

```
        {
             // 找到了我们要删除的节点
             pre->next=tr->next;          //更新前任指针 free(tr);      //
             删除节点
             break;                       // Done!
        }
        tr=tr->next;
        pre=pre->next
        。
    }
    returns head;          // Head did not change
}
```

上面的代码实现了我们看过的列表遍历，有一个*前身指针*和一个*遍历指针*。它检查我们是否要删除*头部节点*，如果是，它返回更新的*头部节点指针*。否则，它通过循环找到并删除包含指定*搜索键的*节点。

要完成我们这个存储、组织和更新餐馆评论的小程序，只需要在*main()*中增加一个选项，允许用户删除指定餐馆的评论。这意味着从菜单中多了一个选择，以及调用删除函数的代码。

```
否则，如果（选择==4）
{
   printf("Which restaurant's review do you want to delete?\n");
   fgets(name,MAX_STRING_LENGTH, stdin);
   head=delete_by_name(head,name)。
}
```

该方案的完整清单可在本节注释的最后找到。试试吧
出来了!

练习。写一个删除函数，允许用户通过指定餐厅地址来删除评论。

练习。编写一个删除函数，允许用户删除*所有*具有指定分数的*评论*（例如，我们可能想从我们的列表中删除所有分数非常差的餐馆，如1）。

12.- 列表ADT的一个变种

在我们结束这一节之前，值得探索*List ADT*的一个重要变化。它为广泛的有趣的应用提供了正确的组织形式。

队列ADT

Queue

*ADT*定义了一个集合，其中的项目是*按顺序排列的*（就像在一个列表中）。该集合支持以下操作。

- *Enqueue（插入）--在队列的末端*添加一个新的项目
- *Dequeue (移除)* - 移除*当前在队列前面的*项目进行处理

此外，其他操作也经常被定义，例如，获得
排队。

队列是无处不在的（它们到处都出现！）。例如，一台网络打印机会有一个*作业队列*。打印作业可以在任何时间到达，来自可能的大量用户中的任何一个。打印作业是按照它们到达的顺序进行的。

队列在模拟调度操作的软件中也很重要。例如，空中交通管制软件会有离港航班、进港航班和排队降落的飞机的队列。

最后，对于使用图来表示和处理信息的应用程序来说，队列是必不可少的。图是数据组织/存储结构，其中项目由节点表示，而节点可以以编码数据项目之间特定关系的方式相互连接。一个例子是*社交网络*，它的节点代表*用户*，而链接表示用户之间的联系。

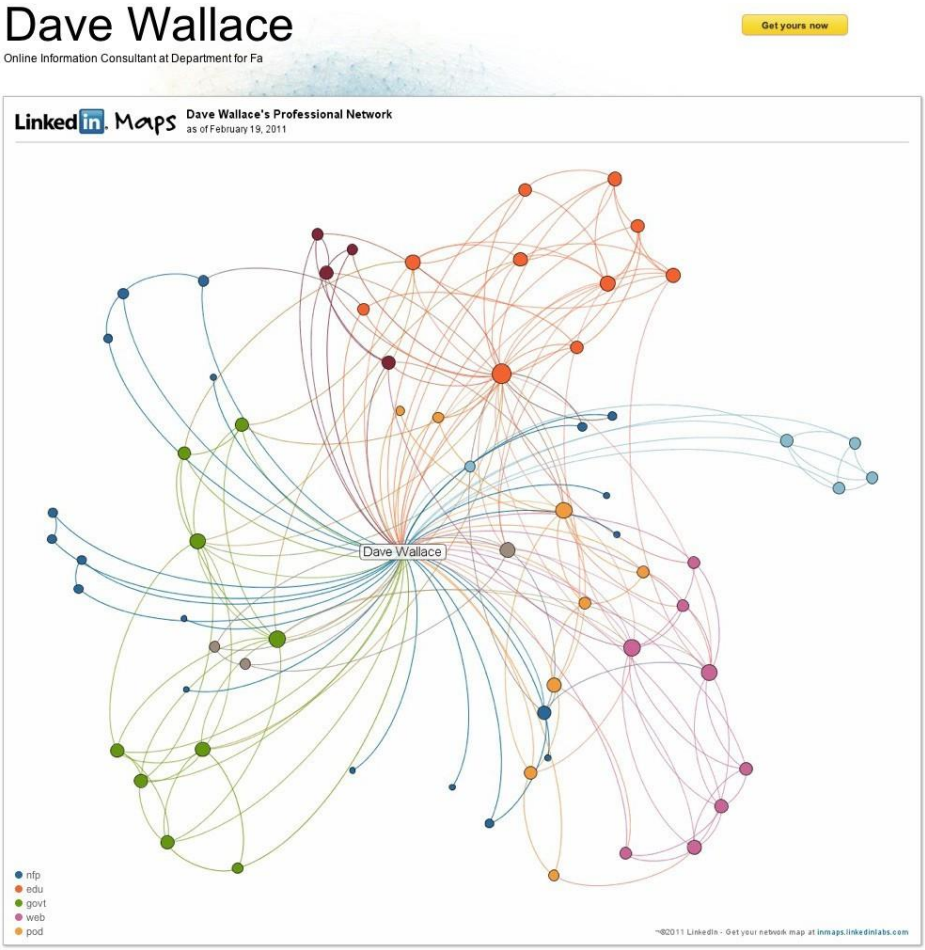


插图5：一个专业网络的样本，显示了一个用户和其他用户之间的连接，按领域进行了颜色编码。照片。Dave Wallace, Flickr, CC-SA 2.0

处理图中的信息往往涉及将节点放在队列中。人工智能搜索方法也使用队列来完成任务，如寻路（想想你手机中的地图应用）、调度、寻找受限优化问题的解决方案等等。

你将有机会探索队列的更多应用，所以不要忘记什么是队列。*排队的ADT*看起来像!

现在是本节最精彩的部分。***你已经知道如何实现一个队列ADT了***

练习。想一想你将如何使用我们为*List ADT*开发的*链表数据结构来*实现*Queue ADT*。

13.- 收尾和总结

在学习这些笔记并完成所有练习后，你应该学到什么？

- 你应该能够解释为什么我们需要仔细思考如何存储和组织数据集合。
- 你应该能够解释为什么我们需要能够为数据项目保留空间
  按需使用（也就是说，你应该知道数组的限制是什么，以及为什么我们不能将其
  用于开始时没有指定数据量的应用）。
- 你应该知道如何为数据创建复合数据类型（便当盒！），以便我们能够表示复
  杂的项目。
- 你应该知道如何创建和操作复杂数据类型的变量和指针。
- 你应该能够解释什么是容器，以及我们如何用它来存储、组织和访问物品的集合。
- 你应该知道*List ADT*指定了什么，它如何组织数据，以及它支持哪些操作。
- 从概念上讲，你应该了解链表的工作原理。在一个链接列表中搜索一个项目，添加
  一个项目，和删除一个项目需要什么。
- 你应该能够在C语言中实现一个*链表数据结构*，这包括。
    * 定义复合数据类型来保存我们需要的信息
    * 定义一个*节点*数据类型，我们可以用它来建立列表
    * 在头部、尾部或节点之间实现*插入*功能
    * 实现*搜索*功能，查找和更新特定项目
    * 实现*删除*函数，根据需要删除节点
    * 释放我们分配给列表中的节点的内存
- 你应该理解在列表遍历中涉及多少工作。你应该能够解释为什么我们说*遍历*的工作量
  与列表中的节点数是*线性关系*。
- 你应该能够解释*ADT*和*数据结构*之间的区别。
- 你应该能够通过本节末尾的程序清单，了解它所做的一切。
- 除此之外，你应该已经学会了C语言实现的各个方面，我们用它来实现餐厅评论应
  用程序。
    * 如何从终端读取输入
    * 如何使用*calloc()*为一个列表节点按需分配内存？
    * 如何编写一个小的驱动程序，让你测试你的代码的一部分
    * 如何以及何时传递和返回指针，以便函数可以在链表上工作

请确保你已经实现了本节的上述所有学习目标。如果有什么不清楚的地方，请在办公时间内拜访你的助教或课程教师，并带来你所有的问题

为什么本节很重要

我们开始这一节需要一个方法。

- 存储、组织和管理可能是大量的复杂数据项目集合
- 能够按需获得数据项目的空间
- 能够搜索特定的项目，并根据需要增加或缩减我们的收藏。

我们讨论了集合的概念，这是一个适用于几乎所有使用计算机管理信息的应用程序的信息存储的一般原则。然后我们看了一个特殊的容器类型--*List ADT*，我们看到了如何使用一个*链表*来实现*List ADT*，并且我们花时间来实现一个*链表数据结构*。

我们现在有了一个有效的*链接列表*的实现，我们可以创建这个列表的变体来处理几乎所有我们可能需要存储和保持组织的数据类型这是我们在这部分课程中取得的第一个成就。

事实上，*链接列表*或其变体将出现在绝大多数的应用程序中。给你举几个例子，你可能会觉得很有趣。

- 图形渲染程序保留了用于创建图像的大多数数据项目的列表：要渲染的对象、光源、纹理、动画关键帧等。
- 音乐合成器会保留一个正在播放的音符列表，以反馈给声音合成引擎。也有数字效果的列表，甚至整首歌曲的列表保存在内存中，以便播放
- 一个在线零售商的购物车可以通过一个链接列表快速而容易地实现

这些只是几个应用，还有很多很多。然而，在这一点上你也知道，*链接列表*有一个缺点，即*搜索*（从而更新特定节点的信息）、*删除*和*列表遍历*需要相当多的工作--我们可能要依次检查整个列表中的每个节点来找到我们想要的东西。

这意味着对于将处理非常大的数据量的应用来说，*链接列表*将导致需要进行数千次的基本操作的等待时间过长，令人无法接受。

因此，虽然列表为我们提供了一种满足我们在本节开始时设定的数据组织和存储目标的方法，但我们现在知道，如果我们想确保*以最快的速度访问*可能*非常大量的数据，我们*需要找到一种更聪明的方法来组织数据。

这将是本课程下一单元的主题。现在，让我们看看在学习了容器和列表之后，我们能解决什么样的问题吧!

问题的解决

正如我们在课程开始时所说，A48是关于学习用于解决计算机科学问题的一般技术。在这个单元中，我们学习了容器和列表。我们这样做的动机是需要了解我们如何组织、存储和管理可能的大量复杂信息，以便使其在程序中发挥作用。

下面的问题让你有机会*测试你*对本节材料的*理解*。

并练习解决问题的能力。

解决CS中编程相关问题的建议方法

- *仔细阅读*问题描述。如果问题说明中有不清楚的地方，一定要问你的助教或课程老师。
- 考虑问题的*输入*--也就是说，你要处理哪些数据？
  你是否从一开始就知道它的数量，或者数量是否会随着时间的推移而变化（而且可能会增长），以及输入数据的任何特定特征。也要考虑是否需要*用户输入*。
  写下你对数据的假设!
  * 你认为将需要的数据类型，你必须创建的新复合类型
  * 特殊条件（如输入值的范围，或有效输入的描述）。
  * 唯一性约束（例如，如果一个输入字段包含的值*必须*是唯一的，如学生编号）。
  * 你可能期望处理的数据量
  * 你认为将需要什么样的存储结构（例如，阵列与列表）？
- 考虑问题要求你解决的*任务*。为了找到一个好的编程方案，你需要了解一旦*输入*数据进入你的程序后会发生什么，记下将对*输入数据进行什么操作或处理*，以及是否将应用于*全部或大部分数据*或*个别项目*。
- 考虑问题的*输出*：这意味着要考虑你的解决方案需要*计算*或*产生*什么。*输出*是否只用于显示（如打印到终端），还是要作为程序的不同部分的*输入*。根据这一点，你需要考虑如何存储*输出*。
  写下你对输出的假设!
- 用简单的语言（不是代码）写下*解决方案*。在这一点上，你要确保你理解问题的解决方案，并能想到所涉及的每一个步骤。
- *设计并实施*解决方案。设计*必须*以你对*输入的*分析为*依据*和*输出*到程序，以及将对输入数据做什么*处理*。
- *彻底测试*你的解决方案，确保它能以*合理的*输入解决这个问题。这可能涉及到*用不同的可能输入来多次运行你的代码*，这些输入是精心选择的，以涵盖不同的*可能但有效的*程序输入。*它必须每次都工作。解决测试中发现的任何问题。*
- *测试*你的解决方案的特殊情况，例如，空的或缺失的输入值，错误的数据类型的输入，打破你对数据的最初假设的输入（这就是为什么我们把它们写下来！）。*解决测试中发现的任何问题。*
- *现在试着破坏这个程序。*看看你是否能想出导致你的程序崩溃或做错事的输入。这可能包括无效的输入、空字段、使用特殊字符等等。这里的目标是找出潜在的问题，并思考如何使你的解决方案更加健全。
- 最后--
  如果你的解决方案的输出将被用作*程序的不同部分的输入*，*请*确保输出能被需要它的任何代码正确*访问*。

这个过程很重要--在没有完全理解的情况下就砍掉一个解决方案。

问题很可能会

- *让你更难想出一个好的解决方案*
- *让你觉得C语言很困难--*
  因为你很难实现解决方案，问题不在于语言，而在于你还没有想到解决方案应该是什么!
- *产生质量较差的解决方案*
- *产生的解决方案缺乏组织性，更难测试和维护*
- *产生一个需要重新工作的解决方案，因为它没有做它应该做的事情*
- *导致代码脆弱，容易被破坏*

习惯于*有条不紊地*完成一个解决方案，并***在开始编码前***仔细思考解决方案的每个方面。

记住：能够想出一个可靠的、经过深思熟虑的问题解决方案，要比仅仅能够实施一个已经存在的解决方案更有价值。

当你遇到麻烦时，该如何处理。

- 弄清楚你在*材料的哪一部分遇到*了困难。
- 仔细*复习*笔记中的材料。通过为自己以外的人写下解释来检查你对事情的理解。
- 当你遇到困难时，请在办公时间来找我们谈一谈!我们在那里提供帮助!

注意：我们***不会***为下面的所有问题提供解决方案。它们应该是*由你自己解决的*。然而，有些问题将*在辅导课上讨论*，我们将在你解决问题时***提供你需要的所有帮助***。每周*在实验室的实践*课是你解决这些问题的完美空间，你可以接触到助教，助教可以帮助你澄清问题的任何部分，或者可以在实施解决方案的技术方面帮助你。*充分利用实践环节!*但*不要向你的助教询问解决方案，也不要在论坛上发布解决方案。*

记住：这些问题的目的是让你思考，并找出你还没有掌握的材料！如果这些问题看起来有难度，不要紧张。-
如果它们看起来具有挑战性，请不要紧张。它们确实具有挑战性，但只要稍加指导和集中学习，你就能想出办法，并最终解决它们。

涉及容器和列表的问题

P0 -
在实践中，我们经常需要找出一个链接列表的长度（例如，我们需要知道在某一时刻我们的系统中有多少条餐馆评论）。

a) 计算一个链表的长度

编写一个小的C函数，将一个链表的头部作为输入，并返回该链表的长度（如果该链表为空，则为0）。假设列表中的节点有一个指针 "next"，指向列表中的下一个条目。

列表，就像我们上面做的所有例子一样。

P1 **-**
你正在为一家在线商店的购物车开发一个*结账*模块。因为典型的用户在任何一次访问中只会往购物车里添加一些东西，所以商店的在线系统将当前在购物车中的物品保存在一个链接列表中。链接列表中的每个 "*Item_Node "都*包含。

```
项目       item_info;
Item_Node *next;
```

项目"本身包含

```
intitem_id;            //每个 项目的唯一标识符  char name[1024];
                       // 项目的名称
float price;           //  项目的价格。
floatdiscount_pct;     //折扣百分比在 [0, .5] (0% 到 50%) intquantity;
                       //  购物车中的物品数量。
```

a) 在C语言中完成 "项目 "和 "项目_节点 "的定义。

这基本上是为了练习你对定义新数据类型所需的语法的掌握。

b) 实现计算购物车中物品总价的函数

首先，用简单的语言写下解决方案的步骤，并检查你的解决方案是否有意义，并考虑到每个项目的 *"数量 "*和 *"折扣率"*，计算出正确的总数。

然后用C语言编写一个函数，计算并返回当前购物车中物品的总价格。你可以假设该函数将接收一个指向购物车的链接列表头部的指针。

P2 **-**
你已经在*多伦多*中央*公共图书馆*找到一份暑期工作。图书馆一直在扩大其*数字收藏*，包括电子书、电影、录音和照片。图书馆的数字馆藏存储在一个中央服务器上，你有一个可用的项目*链接列表*。

库中的每个*'Item_Node'都*包含。

```
Typedef struct Item_List_Node{
      intitem_id;            //  唯一的标识符 char
      title[1024];           //这个项目的标题
```

```
    inttype;                // 资源的类型
                            // 0 - 电子书，1 - 视频。
                            // 2 - 音乐，3 - 照片

    // 这里还有很多字段，我们不需要这个问题。
```

```
    struct Item_List_Node *next;     // 指向列表中下一个节点的指针。
}Item_Node;
```

你的问题是这样的。每个地方的图书馆分馆都有自己的视频和音乐收藏（这些是以实际的DVD和CD的形式）。由于大多数用户宁愿在他们的手持设备上以电子方式访问同样的内容，所以现在很少有人访问这些内容。因此，图书馆决定从每个分馆移除*已经属于中央数字收藏的任何视频或音乐记录。*

图书馆人员已经对每个分馆的内容进行了编目，并将其存储在一个（你猜对了！）*链接列表*中。

第一部分）寻找重复的内容

写下一个算法的步骤，该算法将*两个"项目节点"的链接列表*作为输入（一个用于中央数字收藏，一个用于本地分支收藏），并打印出*任何重复的视频或音乐条目*，以便从本地分支收藏中删除这些重复的内容。

请确保写下你对项目节点中的字段所做的任何假设。

用浅显的语言写出你的解决方案，要有足够的细节，让*别人可以实现它。*
在C.

一旦你对你的解决方案感到满意，用C语言写一个实现。

b部分）思考数据的表示方法

请注意，不管是谁设计了库的链接列表的*数据表示*，都没有费心为每个项目的信息建立一个单独的数据类型。相反，他们把所有的东西都放在*'Item_Node'*里。写下你认为会是什么。

以这种方式表示项目的优点。

以这种方式表示项目的缺点。

P3 -
你正在做一个网络浏览器的*开源*项目，该项目为用户提供了对提供给网站的个人信息的数量和类型的完全控制。任何网络浏览器的关键组成部分之一是*书签*部分。为了简单起见，书签被组织成一个简单的链接列表。新书签被插入到列表的*头部。*

然而，用户可以选择以许多不同的方式来组织书签。特别是，他们可以选择通过按下浏览器主窗口上的一个按钮，按*网址*对书签进行排序。

a) 实现一个建立排序链表的函数

写下以下所需的步骤
- 取一个未经排序的书签链接列表
- 创建一个*新的*链接列表，其中的书签是*按网址排序的*，方法是将原始输入列表中的每个节点*按照其排序顺序插入排序后的列表中的正确位置。(*你可能想复习一下插入式排序，或者请助教用扑克牌做个演示！）。

使用平实的语言，但要确保你的解决方案足够详细，*别人可以用C语言实现它*。

*用图表说明*你的解决方案是如何工作的。现在，

假设链表节点包含。

```
char       url[1024];
Url_Node   *next;
```

用C语言编写一个函数，该函数接收未排序链表*'Url_Nodes'*的头部作为输入，建立一个*排序链表'Url_Nodes'*，并返回一个指向*已排序链表头部的*指针。你可以假设你已经写了一个函数

```
Url_Node *copyUrlNode(Url_Node orig)。
```

它接受一个指向*'Url_Node'的*指针作为输入，并创建一个具有*相同URL的*新节点，但*'next'*指针设置为NULL（因此你可以将其插入不断增长的排序链表）。

*b部分）**更具挑战性--***
实现一个函数，该函数接收一个未经排序的输入链接的URL列表，并对其进行排序而不产生一个新的列表。

和a)部分一样，你应该先用平实的语言写出你的解决步骤，画一张图来说明这个过程是如何在输入列表的一个节点上工作的，最后用C语言写出一个实现。

P4 **-**
你被聘为大学卫生网络的实习生。看到你在A48课程中学习了C语言，他们决定给你一个任务，为一个新的系统设计存储框架，以记录在急诊室就诊的病人的顺序。

通过解决这个问题，你将实现什么？

- 你将经历设计和实施一个适用于现实世界的数据组织/存储/管理问题的解决方案的完整过程。
- 你会发现你对本单元教材的理解有任何不足之处
- 你将练习本单元所涉及的每一个概念，并将其应用于问题的解决。

- 你将练习在C代码中实现处理复合数据类型和数据集合。

要求你。

- 开发一个合适的数据表示法，以记录分诊护士所采集的每个病人的信息。
- 开发存储框架（用什么数据结构来保存信息），以及所需的功能。
    - 在病人到达时添加病人
    - 一旦病人被医生看完，或者他们离开，就把他们带走。
    - 按姓名搜索特定病人
    - 按照病人预期看病的顺序打印出一份病人名单

第一部分）--为患者设计数据表述

分诊站的护士将采集以下信息。
- 患者的姓名（姓、名、中）。
- 患者的街道地址
- 病人的邮政编码
- 电话
- 健康卡号码
- 体温（摄氏度
- 对问题的简短描述

设计一个*数据表示模型*，使你的程序能够组织和存储*一个病人*的信息。这个模型将是你的分诊系统的基础。

1) 显示一个*数据字段*和它们的*数据类型*的列表。你应该说明（解释）为什么数据类型适合每个字段。如果你添加了护士所捕获的字段之外的字段，请解释*为什么*需要这些字段以及它们将*如何被*使用。
2) 指出你可以为每个字段确定的*特殊约束*（例如，值的范围，唯一性约束等）。
3) 标明*哪些字段*将用于搜索特定项目，例如删除特定项目，或实现系统要求的功能
4) 用*C语言*写一个你的数据表示模型的实现。

第二部分）--设计分诊病人管理系统的核心部分

必要的功能。

- 该系统必须允许你在病人到达分诊处时添加病人
    * 病人应该按照他们到达的顺序就诊。
    * 除非他们的体温>40.5C，在这种情况下必须先去看病。
    * 护士必须能够在任何时候将病人推到名单的前面，如果他们

认为病人需要立即关注。

* 病人一旦看完病，或如果他们离开，就必须从系统中删除（他们可以，也可以不通知护士）。
* 目前的病人名单按其就诊顺序被打印到一个屏幕，以便分诊护士可以随时跟踪正在发生的事情。

你需要提供。

- 对解决方案的总体描述。
    * 你将使用什么数据结构，解释你为什么需要这些结构
    * 你将如何把你的解决方案分成可以作为独立功能实现的*模块*。
    * 主函数的伪代码描述显示了所发生的情况
        - 当病人到达时
        - 当一个病人被看到或离开时
        - 当护士把一个病人推到名单前面时
    * 对你的解决方案中增加一个新病人的部分进行伪代码描述
    * 对你的解决方案中把病人移到列表前面的部分进行伪代码描述。

*在这一点上，你已经解决了问题!*剩下的就是实施解决方案。

c) 部分 - 实施并测试你的解决方案!

你可以把你的实现（或部分实现）带到实践环节，向你的TA展示（或获得帮助）。