考虑到设置和删除这种多维数组需要的样板代码量很多，强烈建议将这种功能封装到一个可重用的类中。这个问题留作练习，当读者在后续章节中学习了如何创建自己的类后可以解决这个问题。

提示：

我们在这里使用了一种简单的技术来创建动态多维数组，但这不是最高效的方法。它在自由存储区中分别分配所有行的内存，这意味着这些内存可能不是连续的。程序在连续内存上进行处理时运行速度一般快得多。因此，封装多维数组的类一般只分配一个包含 rows * columns 个元素的数组，并使用公式 i * columns + j 来将对行 i 和列 j 的访问映射到一个索引。

6.9 通过指针选择成员

指针可以存储类类型的对象的地址，例如 vector<T>容器。对象常常包含操作它的成员函数，例如，vector<T>容器的 at()函数可以访问元素，push_back()函数可以添加元素。假定 pdata 是一个 vector<>容器的指针。下面的语句在自由存储区中创建了这个容器：

```
auto* pdata {new std::vector<int>{}};
```

但也可以是使用地址运算符获取的局部对象的地址：

```
std::vector<int> data;
auto* pdata = &data;
```

在这两种情况中，编译器都能够推断出 pdata 的类型为 vector<int>*，即指向 int 元素向量的指针。对于接下来要介绍的内容，vector<>在自由存储区中创建，还是作为局部对象在栈上创建并不重要。要添加元素，应调用 vector<int>对象的 push_back()函数，前面介绍了如何使用表示向量的变量和成员函数名之间的句点。要使用指针访问向量对象，必须使用解除引用运算符，所以添加元素的语句如下：

```
(*pdata).push_back(66);         // Add an element containing 66
```

这条语句要编译，包围*pdata 的圆括号就是必不可少的，因为•运算符的优先级高于*运算符。这是一个看起来很笨拙的表达式，但在处理对象时常常要使用它，所以 C++提供了一个运算符，先解除对象指针的引用，再选择对象的成员。前面的语句可以写作：

```
pdata->push_back(66);       // Add an element containing 66
```

->运算符由一个减号和一个大于号组成，称为箭头运算符或间接成员选择运算符。箭头足以表示其含义。本书后面常常使用这个运算符。

6.10 动态内存分配的危险

在使用 new 动态分配内存时，可能出现许多严重的问题。本节将讨论几种常见的问题。任何使用过 new 和 delete 的程序员都会承认，这些危险是真实存在的。C++开发人员处理的大部分比较严重的 bug 常常源自对动态内存管理不当。

本节将呈现一个看起来十分阴暗的场景，以说明各种与动态内存有关的危险。但是读者不要丧失信心。我们很快就会说明如何走出这个满是危险的雷区。6.11 节将列出经过验证的一些准则和工具，很容易避免这里提到的大部分(甚至全部)问题。事实上，读者已经知道这样的一个工具：std::vector<>容器。该容器几乎总是比使用 new[]直接分配动态内存更好。后面将介绍标准库提供的其他用来更好地管理动态内存的工具。在此之前，先来了解与动态内存有关的一些风险和隐患。

6.10.1 悬挂指针和多次释放

悬挂指针指的是这样一种指针变量：在使用 delete 或 delete[]释放分配的内存后，指针变量仍然包含自由存储区