当使用Pimpl惯用法，请在实现文件中定义特殊成员函数

如果你曾经与过多的编译次数斗争过，你会对`Pimpl`(Pointer to implementation)惯用法很熟悉。 凭借这样一种技巧，你可以将一个**类数据成员**替换成一个指向包含具体实现的类或结构体的指针, 并将放在主类(primary class)的数据成员们移动到实现类去(implementation class), 而这些数据成员的访问将通过指针间接访问。 举个例子，假如有一个类`Widget`看起来如下:

```
class Widget()      //定义在头文件`widget.h`
{
public:
    Widget();
    ...
private:
    std::string name;
    std::vector<double> data;
    Gadget g1, g2, g3;  //Gadget是用户自定义的类型
}
```

因为类`Widget`的数据成员包含有类型`std::string`， `std::vector`和`Gadget`， 定义有这些类型的头文件在类`Widget`编译的时候，必须被包含进来，这意味着类`Widget`的使用者必须要`#include <string>,<vector>`以及`gadget.h`。 这些头文件将会增加类`Widget`使用者的编译时间，并且让这些使用者依赖于这些头文件。 如果一个头文件的内容变了，类`Widget`使用者也必须要重新编译。 标准库文件`<string>`和`<vector>`不是很常变，但是`gadget.h`可能会经常修订。

在C++98中使用`Pimpl`惯用法，可以把`Widget`的数据成员替换成一个原始指针(raw pointer)，指向一个已经被声明过却还未被定义的类，如下:

```
class Widget        //仍然在"Widget.h"中
{
public:
    Widget();
    ~Widget();       //析构函数在后面会分析
    ...

    private:
    struct Impl;     //声明一个 实现结构体
    Impl *pImpl;     //以及指向它的指针
}
```

因为类`Widget`不再提到类型`std:::string`,`std::vector`以及`Gadget`,`Widget`的使用者不再需要为了这些类型而引入头文件。 这可以加速编译，并且意味着，如果这些头文件中有所变动，`Widget`的使用者不会受到影响。

一个已经被声明，却还未被实现的类型，被称为**未完成类型**(incomplete type)。 `Widget::Impl`就是这种类型。 你能对一个未完成类型做的事很少，但是声明一个指向它指针是可以的。 `Pimpl`手法利用了这一点。

`Pimpl`惯用法的第一步，是声明一个数据成员，它是个指针，指向一个未完成类型。 第二步是动态分配(dynamic allocation)和回收一个对象，该对象包含那些以前在原来的类中的数据成员。 内存分配和回收的代码都写在**实现文件**(implementation file)里，比如，对于类`Widget`而言，写在`Widget.cpp`里:

```
#include "widget.h"      //以下代码均在实现文件 widget.cpp里
```

```
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl      //之前在Widget中声明的Widget::Impl类型的定义
{
    std::string name;
    std::vector<double> data;
    Gadget g1,g2,g3;
}

Widget::Widget()         //为此Widget对象分配数据成员
: pImpl(new Impl)
{}

Widget::~Widget()
{delete pImpl;}          //销毁数据成员
```

在这里我把`#include`命令写出来是为了明确一点，对于头文件`std::string`,`std::vector`和`Gadget`的整体依赖依然存在。 然而，这些依赖从头文件`widget.h`(它被所有`Widget`类的使用者包含，并且对他们可见）移动到了`widget.cpp`（该文件只被`Widget`类的实现者包含，并只对它可见）。 我高亮了其中动态分配和回收`Impl`对象的部分(markdown高亮不了，实际是`new`和`delete`两部分——译者注)。这就是为什么我们需要`Widget`的析构函数——我们需要回收该对象。

但是，我展示给你们看的是一段C++98的代码，散发着一股已经过去了几千年的腐朽气息。 它使用了原始指针，原始的`new`和原始的`delete`，一切都让它如此的...原始。这一章建立在"智能指针比原始指针更好"的主题上，并且，如果我们想要的只是在类`Widget`的构造函数动态分配`Widget::impl`对象，在`Widget`对象销毁时一并销毁它， `std::unique_ptr`（见Item 18)是最合适的工具。 在头文件中用`std::unique_ptr`替代原始指针，就有了如下代码:

```
class Widget       //在"Widget.h"中
{
public:
    Widget();
    ...

    private:
    struct Impl;    //声明一个 实现结构体
    std::unique_ptr<Impl> pImpl;   //使用智能指针而不是原始指针
}
```

实现文件也可以改成如下:

```
#include "widget.h"     //以下代码均在实现文件 widget.cpp里
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl     //跟之前一样
{
    std::string name;
    std::vector<double> data;
    Gadget g1,g2,g3;
}

Widget::Widget()        //根据Item 21， 通过std::make_shared来创建std::unique_ptr
: pImpl(std::make_unique<Imple>())
```

```
{}
```

你会注意到，`Widget`的析构函数不存在了。这是因为我们没有代码加在里面了。`std::unique_ptr`在自身析构时，会自动销毁它所指向的对象，所以我们自己无需手动销毁任何东西。这就是智能指针的众多优点之一：它使我们从手动资源释放中解放出来。

以上的代码能编译，但是，最普通的`Widget`用法却会导致编译出错：

```
#include "widget.h"

Wdiget w;       //编译出错
```

你所看到的错误信息根据编译器不同会有所不同，但是其文本一般会提到一些**有关于把`sizeof`和`delete`应用到未完成类型`incomplete type`上**的信息。对于未完成类型，使用以上操作是禁止的。

在`Pimpl`惯用法中使用`std::unique_ptr`会抛出错误，有点惊悚，因为第一`std::unique_ptr`宣称它支持未完成类型，第二`Pimpl`惯用法是`std::unique_ptr`的最常见的用法。 幸运的是，让这段代码能正常运行很简单。 只需要对是什么导致以上代码编译出错有一个基础的认识就可以了。

在对象`w`被析构时，例如离开了作用域(scope)，问题出现了。在这个时候，它的析构函数被调用。我们在类的定义里使用了`std::unique_ptr`，所以我们没有声明一个析构函数，因为我们并没有任何代码需要写在里面。根据编译器自动生成的*特殊成员函数*的规则(见 Item 17)，编译器会自动为我们生成一个析构函数。 在这个析构函数里，编译器会插入一些代码来调用类`Widget`的数据成员`Pimpl`的析构函数。`Pimpl`是一个`std::unique_ptr<Widget::Impl>`,也就是说，一个带有默认销毁器(default deleter)的`std::unique_ptr`。 默认销毁器(default deleter)是一个函数，它使用`delete`来销毁内置于`std::unique_ptr`的原始指针。然而，在使用`delete`之前，通常会使默认销毁器使用C++11的特性`static_assert`来确保原始指针指向的类型不是一个未完成类型。 当编译器为`Widget w`的析构生成代码时，它会遇到`static_assert`检查并且失败，这通常是错误信息的来源。 这些错误信息只在对象`w`销毁的地方出现，因为类`Widget`的析构函数，正如其他的编译器生成的特殊成员函数一样，是暗含`inline`属性的。 错误信息自身往往指向对象`w`被创建的那行，因为这行代码明确地构造了这个对象，导致了后面潜在的析构。

为了解决这个问题，你只需要确保在编译器生成销毁`std::unique_ptr<Widget::Imple>`的代码之前， `Widget::Impl`已经是一个完成类型(complete type)。 当编译器"看到"它的定义的时候，该类型就成为*完成类型*了。 但是`Widget::Impl`的定义在`wideget.cpp`里。成功编译的关键，就是，在`widget.cpp`文件内，让编译器在"看到" `Widget`的析构函数实现之前（也即编译器自动插入销毁`std::unique_ptr`的数据成员的位置），先定义`Wdiget::Impl`。

做出这样的调整很容易。只需要在先在`widget.h`里，只**声明**(declare)类`Widget`的析构函数，却不要在这里**定义**(define)它:

```
class Widget {      // as before, in "widget.h"
public:
  Widget();
  ~Widget();     // declaration only
  ...

private:     // as before
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};
```

在`widget.cpp`文件中，在结构体`Widget::Impl`被定义之后，再定义析构函数:

```
#include "widget.h"      //以下代码均在实现文件 widget.cpp里
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl      //跟之前一样,定义Widget::Impl
{
    std::string name;
    std::vector<double> data;
    Gadget g1,g2,g3;
}

Widget::Widget()         //根据Item 21,  通过std::make_shared来创建std::unique_ptr
: pImpl(std::make_unique<Imple>())
{}

Widget::~Widget()        //析构函数的定义(译者注: 这里高亮)
{}
```

这样就可以了，并且这样增加的代码也最少，但是，如果你想要强调编译器自动生成的析构函数会工作的很好——你声明`Widget`的析构函数的唯一原因，是确保它会在`Widget`的实现文件内(指`widget.cpp`，译者注)被自动生成，你可以把析构函数体直接定义为`=default`:

```
    Widget::~Widget() = default;        //同上述代码效果一致
```

使用了`Pimpl`惯用法的类自然适合支持移动操作，因为编译器自动生成的移动操作正合我们所意： 对隐藏的`std::unique_ptr`进行移动。 正如`Item 17`所解释的那样，声明一个类`Widget`的析构函数会阻止编译器生成移动操作，所以如果你想要支持移动操作，你必须自己声明相关的函数。考虑到编译器自动生成的版本能够正常功能，你可能会被诱使着来这样实现:

```
class Widget        //在"Widget.h"中
{
public:
    Widget();
    ~Widget();
    ...

    Widget(Widget&& rhs) = default;              //思路正确，但代码错误
    Widget& operator=(Widget&& rhs) = default;


    private:
    struct Impl;         //如上
    std::unique_ptr<Impl> pImpl;
}
```

这样的做法会导致同样的错误，和之前的声明一个不带析构函数的类的错误一样，并且是因为同样的原因。 编译器生成的移动赋值操作符(move assignment operator)，在重新赋值之前，需要先销毁指针`pImpl`指向的对象。然而在`Widget`的头文件里，`pImpl`指针指向的是一个未完成类型。情况和移动构造函数(move constructor)有所不同。 移动构造函数的问题是编译器自动生成的代码里，包含有抛出异常的事件，在这个事件里会生成销毁`pImpl`的代码。然而，销毁`pImpl`需要`Impl`是一个完成类型。

因为这个问题同上面一致，所以解决方案也一样——把移动操作的定义移动到实现文件里:

```
class Widget        //在"Widget.h"中
{
public:
    Widget();
    ~Widget();
    ...

    Widget(Widget&& rhs);   //仅声明
    Widget& operator=(Widget&& rhs);


    private:
    struct Impl;        //如上
    std::unique_ptr<Impl> pImpl;
}
```

```
#include "widget.h"     //以下代码均在实现文件 widget.cpp里
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl     //跟之前一样,定义Widget::Impl
{
    std::string name;
    std::vector<double> data;
    Gadget g1,g2,g3;
}

Widget::Widget()        //根据Item 21,  通过std::make_shared来创建std::unique_ptr
: pImpl(std::make_unique<Imple>())
{}

Widget::~Widget() = default;

Widget(Widget&& rhs) = default;             //在这里定义
Widget& operator=(Widget&& rhs) = default;
```

`pImpl`惯用法是用来减少类实现者和类使用者之间的编译依赖的一种方法，但是，从概念而言，使用这种惯用法并不改变这个类的表现。 原来的类`Widget`包含有`std::string`,`std::vector`和`Gadget`数据成员，并且，假设类型`Gadget`，如同`std::string`和`std::vector`一样,允许复制操作，所以类`Widget`支持复制操作也很合理。 我们必须要自己来写这些函数，因为第一，对包含有**只可移动(move-only)**类型，如`std::unique_ptr`的类，编译器不会生成复制操作;第二，即使编译器帮我们生成了，生成的复制操作也只会复制`std::unique_ptr`(也即浅复制(shallow copy)),而实际上我们需要复制指针所指向的对象(也即深复制(deep copy))。

使用我们已经熟悉的方法，我们在头文件里声明函数，而在实现文件里去实现他们:

```
class Widget        //在"Widget.h"中
{
public:
    Widget();
    ~Widget();
    ...

    Widget(const Widget& rhs);   //仅声明
    Widget& operator=(const Widget& rhs);
```

```
    private:
    struct Impl;        //如上
    std::unique_ptr<Impl> pImpl;
}
```

```
#include "widget.h"     //以下代码均在实现文件 widget.cpp里
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl     //跟之前一样,定义Widget::Impl
{
    ...
}

Widget::Widget()        //根据Item 21, 通过std::make_shared来创建std::unique_ptr
: pImpl(std::make_unique<Imple>())
{}

Widget::~Widget() = default;
    ...
Widget::Widget(const Widget& rhs)
:pImpl(std::make_unique<Impl>(*rhs.pImpl))
{}

Widget& Widget::operator=(const Widget& rhs)
{
    *pImpl = *rhs.pImpl;
    return *this;
}
```

两个函数的实现都比较中规中矩。 在每个情况中，我们都只从源对象(rhs)中，复制了结构体`Impl`的内容到目标对象中(*this)。我们利用了编译器会为我们自动生成结构体`Impl`的复制操作函数的机制，而不是逐一复制结构体`Impl`的成员，自动生成的复制操作能自动复制每一个成员。 因此我们通过调用`Widget::Impl`的编译器生成的复制操作函数来实现了类`Widget`的复制操作。 在复制构造函数中，注意，我们仍然遵从了Item 21的建议，使用`std::make_unique`而非直接使用`new`。

为了实现`Pimpl`惯用法，`std::unique_ptr`是我们使用的智能指针，因为位于对象内部的`pImpl`指针(例如，在类`Widget`内部)，对所指向的对应实现的对象的享有独占所有权(exclusive ownership)。然而，有趣的是，如果我们使用`std::shared_ptr`而不是`std::unique_ptr`来做`pImpl`指针， 我们会发现本节的建议不再适用。 我们不需要在类`Widget`里声明析构函数，也不用用户定义析构函数，编译器将会愉快地生成移动操作，并且将会如我们所期望般工作。代码如下:

```
//在Widget.h中
class Widget{
public:
    Widget();
    ...         //没有对移动操作和析构函数的声明
    private:
    struct Impl;
    std::shared_ptr<Impl> pImpl;    //使用std::shared_ptr而非std::unique_ptr
}
```

而类`Widget`的使用者，使用`#include widget.h`，可以使用如下代码

```
Widget w1;
auto w2(std::move(w1));  //移动构造w2
w1 = std::move(w2);      //移动赋值w1
```

这些都能编译，并且工作地如我们所望: `w1`将会被默认构造，它的值会被移动进`w2`，随后值将会被移动回`w1`，然后两者都会被销毁(因此导致指向的`Widget::Impl`对象一并也被销毁)。

`std::unique_ptr`和`std::shared_ptr`在`pImpl`指针上的表现上的区别的深层原因在于，他们支持自定义销毁器(custom deleter)的方式不同。 对`std::unique_ptr`而言，销毁器的类型是`unique_ptr`的一部分，这让编译器有可能生成更小的运行时数据结构和更快的运行代码。 这种更高效率的后果之一就是`unique_ptr`指向的类型，在编译器的生成特殊成员函数被调用时(如析构函数，移动操作)时，必须已经是一个完成类型。 而对`std::shared_ptr`而言，销毁器的类型不是该智能指针的一部分，这让它会生成更大的运行时数据结构和稍微慢点的代码，但是当编译器生成的特殊成员函数被使用的时候，指向的对象不必是一个完成类型。(译者注: 知道`unique_ptr`和`shared_ptr`的实现，这一段才比较容易理解。)

对于`pImpl`惯用法而言，在`std::unique_ptr`和`std::shared_ptr`的特性之间，没有一个比较好的折中。 因为对于类`Widget`以及`Widget::Impl`而言，他们是独享占有权关系，这让`std::unique_ptr`使用起来很合适。 然而，有必要知道，在其他情况中，当共享所有权(shared ownership)存在时，`std::shared_ptr`是很适用的选择的时候，没有必要使用`std::unique_ptr`所必需的**声明——定义**(function-definition)这样的麻烦事了。

记住

- `pImpl`惯用法通过减少在类实现和类使用者之间的编译依赖来减少编译时间。
- 对于`std::unique_ptr`类型的`pImpl`指针，需要在头文件的类里声明特殊的成员函数，但是在实现文件里面来实现他们。即使是编译器自动生成的代码可以工作，也要这么做。
- 以上的建议只适用于`std::unique_ptr`，不适用于`std::shared_ptr`。