附录C　模拟虚构造函数

在一个构造函数调用期间，虚机制并不工作（出现早期绑定），有时这很让人为难。

另外，我们可能想组织我们的代码，使得在创建一个对象时不用选择一个确切的构造函数类型，也就是，我们想说：“我不准确地知道我们是哪种类型的对象，但这与我们自己创建自己的信息有关。”这个附录介绍了两种“虚构造函数”方法。第一个是一种充分发展的技术，它在堆栈上和堆上都可以工作，但实现起来较为复杂。第二种则简单得多，但我们只能在堆上创建对象。

C.1　全功能的虚构造函数

让我们来看一个经常被引用的“shapes”例子。对于一个对象，我们可能想在构造函数内设置每件事，然后调用draw()来画出这个对象。draw()应该是一个虚函数，是一条它应该准确画它自身的消息，依赖于它是一个圆、矩形、线等。然而，这些在构造函数内并不能工作，原因就是第14章介绍的：当在构造函数中调用一个虚函数时，虚函数将分解为本地的函数体。

如果我们想在构造函数中调用一个虚函数并让它能正确工作，我们必须用一种模拟虚构造函数的技术。这是一个棘手的问题。记住虚函数是这样一种函数：我们送一条消息到一个对象，然后让这个对象算出该做些什么。但构造函数用来创建一个对象。所以一个虚构造函数好象在说：“我不确切地知道你是哪种类型的对象，那么你自己创建吧。”在一般的构造函数中，`编译器必须知道哪个VTABLE的地址绑到了这个VPTR上和它是否已存在，一个虚构造函数并不能做这些，因为它在编译时并不知道全部的类型信息。因此说一个构造函数不能是虚函数是有道理的，因为它是一种必须知道对象类型全部信息的函数。

仍然有一些时候，我们想要一些东西类似于虚构造函数的行为。

在shape这个例子中，在构造函数的参数表中传递一些特殊信息，并让构造函数在没有更多信息的情况下创建特殊的形状（圆、矩形或三角形）。通常，我们不得不显式地调用circle、square或triangle的构造函数。

Coplien [1] 称他对这个问题的解决方案为“信封和信纸”类。信封类是基类，一个包含指向基类对象的指针的外壳。“信封类”的构造函数决定（在运行时间，即当构造函数被调用时，而不是编译时间，即当类型检查已完成时）创建哪种特定类型，然后创建这种类型的对象（在堆上），并将这个对象赋给它的指针。所有的函数调用都由基类通过它的指针来完成。

下面是shape例子的一个简化版：

```
//: SSHAPE.CPP -- "Virtual constructors"
// Used in a simple "shape" framework
#include <iostream.h>
#include "..\14\tstash.h"

class shape {
```

[1] James O.Coplien，《高级C++编程风格与习语》，Addison_Wesley, 1992。

```
  shape* S;
  // Prevent copy-construction & operator=
  shape(shape&);
  shape operator=(shape&);
protected:
  shape() { S = 0; };
public:
  enum type { Circle, Square, Triangle };
  shape(type);  // "Virtual" constructor
  virtual void draw() { S->draw(); }
  virtual ~shape() {
    cout << "~shape\n";
    delete S;
  }
};
class circle : public shape {
  // Prevent copy-construction & operator=
  circle(circle&);
  circle operator=(circle&);
public:
  circle() {}
  void draw() { cout << "circle::draw\n"; }
  ~circle() { cout << "~circle\n"; }
};

class square : public shape {
  // Prevent copy-construction & operator=
  square(square&);
  square operator=(square&);
public:
  square() {}
  void draw() { cout << "square::draw\n"; }
  ~square() { cout << "~square\n"; }
};

class triangle : public shape {
  // Prevent copy-construction & operator=
  triangle(triangle&);
  triangle operator=(triangle&);
public:
  triangle() {}
  void draw() { cout << "triangle::draw\n"; }
  ~triangle() { cout << "~triangle\n"; }
};

shape::shape(type t) {
```

```
  switch(t) {
    case Circle: S = new circle; break;
    case Square: S = new square; break;
    case Triangle: S = new triangle; break;
  }
  draw();  // Virtual call in the constructor
}

main() {
  tstash<shape> shapes; // Default to ownership
  cout << "virtual constructor calls:" << endl;
  shapes.add(new shape(shape::Circle));
  shapes.add(new shape(shape::Square));
  shapes.add(new shape(shape::Triangle));
  cout << "virtual function calls:" << endl;
  for(int i = 0; i < shapes.count(); i++)
    shapes[i]->draw();
  shape c(shape::Circle); // Can create on stack
}
```

基类shape包含一个指向shape类型对象的指针，这也是它唯一的数据成员。当我们建立了一个“虚构造函数”配置时，我们必须确保这个指针已经被初始化为指向一个已存在的对象。

type枚举在类shape内部和shape的构造函数要在所有的派生类之后定义是这种方法的两个限制。每次我们从shape类中派生出新的子类型，我们必须回过头来把这个类型的名字加到 type枚举列表中。然后我们必须修改 shape的构造函数以处理新情况。其不利的一面是现在 shape类和所有 shape的派生类之间存在着依赖关系。有利的一面是这种一般情况下出现在程序体中（可能不止一处，这使得可维护性差）的关系被限制在一个类中。另外这还产生了一种像第 3章中介绍的“ cheshire cat ”的效果。在这种情况下，所有特定形状类的定义可以隐藏在实现文件中。这样，基类的接口是用户可以看到的唯一的东西。

在这个例子中，我们必须传递给构造函数的关于创建对象类型的信息是很清楚的，它是类型的一个枚举值。当然我们可以用其他信息——比方说，在语法分析程序中扫描器的输出可能被传递给“虚构造函数”，然后用这个文本串来决定准确的是要创建什么。

“虚构造函数”shape（type）只能在类中声明，它只有到这个基类的一切都声明之后才能定义。当然在class shape内可以定义缺省构造函数，但这应该声明为 protected，这样就不能产生临时shape对象。这种缺省构造函数只能被子派生类的构造函数调用。我们必须显式地产生缺省构造函数，因为只有在没有定义构造函数时编译器才能为我们自动产生一个缺省构造函数。因为我们定义了shape(type)，所以我们也必须定义shape（）。

在这个例子中，缺省构造函数至少有一个非常重要的工作——它必须将S指针的值置为0。这乍听起来很奇怪，但记住，缺省构造函数将作为创建实际对象——用Coplien的话来说是“信纸”，不是“信封”——的一部分来调用。因为“信纸”是从信封继承来的，所以它也继承了数据成员S。在“信封”中，S是很重要的，因为它指向一个实际的对象，但在“信纸”中，S只是一个多余的口袋。然而即使是多余的口袋，也应该初始化，如果“信纸”在调用缺省构造函数时没有把S置0，事情将变得很糟糕（后面我们将看到）。

“虚构造函数”完全由它的参数来决定对象的类型。虽然这种类型信息直到运行时才会被读取来起作用，而一般情况下，编译器在编译时就必须知道确切的类型（这是这个系统有效地模拟虚构造函数的另一个原因）。

在虚构造函数里有一个switch语句，这里用参数来构造实际的对象，然后将它指定给“信封”里的指针。在这之后，“信纸”创建就完成了，所以任何虚调用都会被正常地引导。

作为一个例子，我们来看在虚构造函数中调用 draw()，如果跟踪这个调用（手工或用一个调试器），我们可以看到它是从基类 shape中的draw()函数开始的，这个函数用“信封”中S指针指向的“信纸”的 draw()。所有从shape中派生的类都共享同一接口，所以这个虚调用被正确地执行，即使它似乎是在一个构造函数内部（实际上这个“信纸”的构造函数已经完成）。只要基类中所有的虚函数的调用都通过指向“信纸”的指针来完成的话，系统就会正确运行。

为了理解它是如何工作的，现在来看 main()中的代码。为了产生数组 s[]，“虚构造函数”shape被调用。一般在这种情况下，我们就要调用实际类型的构造函数，而这个类型的 VPTR应当被安装在这个对象中。然而现在，每种情况下使用的 VPTR都是shape类的，而不是特定的circle、square 或triangle类的。

在for循环中，每个shape调用draw()函数，这个虚函数通过VPTR来决定相应的类型。当然这里每个情况下都是 shape。实际上，我们可能奇怪为什么 draw被做成完全虚的，下一步显示了其理由：基类的draw()函数通过“信纸”指针S来调用了“信纸”的draw()函数。这时函数调用决定针对实际的对象类型，而不仅是基类 shape。因此，使用“虚构造函数”的运行时间支出比我们每次调用虚函数时要多。

剩下的问题

这里所讲述的“虚构造函数”中，析构函数内部的虚函数调用是在编译时用一般的方法确定的。我们可能认为，通过在析构函数中巧妙地调用基类的虚函数，我们可以回避这种限制。不幸的是，这会导致灾难性的结果。基类中的函数确实要调用。然而它的 this指针是指向“信纸”而不是“信封”的。所以当基类的函数被调用时——记住，基类中的虚函数总是以为它正在处理“信封”——它将用S指针来完成调用。对于“信纸”，S是0，由protected的缺省构造函数设置。当然我们可以通过在基类中的每个虚函数中加入一段代码来检查 S是不是为0，从而防止上述0指针调用，或者我们在使用“虚构造函数”时遵守下面两条原则：

1) 在派生类函数中不要显式地调用根类的虚函数。

2) 总是重定义根类中所有的虚函数。

这两条规则起因于同样的问题，如果我们在一个“信纸”内调用一个虚函数，这个函数获得“信纸”的this指针。如果虚函数没有重载，这个调用将调用基类中的虚函数，然后调用将通过“信纸”的this指针指向“信纸”的S，而这又是0。

我们可以看出，用这种方法费时、受限而且危险，这就是为什么不想让它在所有时候都作为编程环境的一部分的原因。它是属于那种解决某些问题时非常有用，但我们不想在所有时候都运用的特征。幸运的是，C++允许我们在需要它时，再把它放入，而标准的方法是最安全的，最终也是最容易的。

• 析构操作

析构操作也很棘手。为了便于理解，让我逐句跟踪当我们对于指向创建在堆上的 shape对象的指针调用delete时——特别是对于一个square时——会发生什么。（这比在堆栈中创建的对

象更复杂)。这个delete将穿越多态接口，就像在main()中的delete s [i]语句一样。

指针S[i]的类型是基类shape，所以编译器让所有的调用都通过shape类。一般情况下，我们可能说这是虚调用，所以square的析构函数将被调用。但用这种“虚构造函数”方案时，编译器正在产生一个实际的shape对象，即使构造函数初始化“信纸”的指针指向一特定类型。虚机制被使用，但shape对象中的VPTR是shape的VPTR，而不是square的。这样就决定了对应于shape的析构函数，它对于“信纸”的指针S 调用delete，而S实际指向一个square对象。这又是一个虚调用，但这次是调用square的析构函数。

当然在继承层次中的所有析构函数都会被调用， square的析构函数首先被调用，紧接着是其他中间的析构函数，按次序，一直到基类的析构函数被调用。在基类析构函数中有 delete S语句。当这个函数最初被调用时，它是为“信封” S的，但现在它是为“信纸” S，这是因为“信纸”是从“信封”中继承过来的，而不是因为它包含了什么。因此这个 delete调用不做任何事。

为了解决这个问题，可以让“信纸”的 S指针等于零。然后当“信纸”基类析构函数被调用时，得到delete 0，它被定义为不做任何事。因为缺省构造函数是protected，它只能在“信纸”的构造期间被调用，所以这是将S置零的唯一情况。

C.2 更简单的选择

“虚构造函数”对于大多数应用来说并不一定很复杂。如果我们能限制我们自己只在堆上产生对象，事情可以变得简单得多。我们所需要的是某个函数（我把它称为对象制造函数），我们可以向它发送一串信息，它将会产生正确的对象。如果它可以产生一个基类指针而不是对象本身，那么这个函数不一定是构造函数。下面是用对象制造函数重新设计的“信封——信纸”例子：

```
//: SSHAPE2.CPP -- Alternative to SSHAPE.CPP
#include <iostream.h>
#include "..\14\tstash.h"

class shape {
  shape(shape&); // Prevent copy-construction
protected:
  shape() {} // Prevent stack objects
  // But allow access to derived constructors
public:
  enum type { Circle, Square, Triangle };
  virtual void draw() = 0;
  virtual ~shape() { cout << "~shape\n"; }
  static shape* make(type);
};

class circle : public shape {
  circle(circle&); // No copy-construction
  circle operator=(circle&); // No operator=
protected:
  circle() {};
```

```
public:
  void draw() { cout << "circle::draw\n"; }
  ~circle() { cout << "~circle\n"; }
  friend shape* shape::make(type t);
};

class square : public shape {
  square(square&); // No copy-construction
  square operator=(square&); // No operator=
protected:
  square() {};
public:
  void draw() { cout << "square::draw\n"; }
  ~square() { cout << "~square\n"; }
  friend shape* shape::make(type t);
};

class triangle : public shape {
  triangle(triangle&); // No copy-construction
  triangle operator=(triangle&); // Prevent
protected:
  triangle() {};
public:
  void draw() { cout << "triangle::draw\n"; }
  ~triangle() { cout << "~triangle\n"; }
  friend shape* shape::make(type t);
};

shape* shape::make(type t) {
  shape* S;
  switch(t) {
    case Circle: S = new circle; break;
    case Square: S = new square; break;
    case Triangle: S = new triangle; break;
  }
  S->draw(); // Virtual function call
  return S;
}

main() {
  tstash<shape> shapes; // Default to ownership
  shapes.add(shape::make(shape::Circle));
  shapes.add(shape::make(shape::Square));
  shapes.add(shape::make(shape::Triangle));
  cout << "virtual function calls:\n";
  for(int i = 0; i < shapes.count(); i++)
```

```
    shapes[i]->draw();
//Circle c; // error: can't create on stack
}
```

可以看到，所有的事情都变得简单了，而且不用担心内部指针引起的奇怪情况。唯一的限制是基类中的enum在我们每次派生一个新类时都必须更改（用信封——信纸的方法也是如此），而且对象必须在堆中创建。这后一条是通过使所有构造函数为 protected来强制完成的，所以用户不能创建这个类的对象，但派生类的对象在对象创建期间可以访问基类的构造函数。这是protected关键字不可缺少的一个很好的例证。