第18章　运行时类型识别

运行时类型识别（Run-time type identification, RTTI）是在我们只有一个指向基类的指针或引用时确定一个对象的准确类型。

这可以被看作C++的第二大特征，在我们茫然不知所措时，这确实是一个很有用的工具。一般情况下，我们并不需要知道一个类的确切类型，虚函数机制可以实现那种类型的正确行为。但是有些时候，我们有指向某个对象的基类指针，确定该对象的准确类型是很有用的。这些信息让我们更高效地完成一个特定情况下的操作，防止基类的接口变得很笨拙。大多数的类库用一些虚函数来提供运行时的类型信息。当异常处理功能加入到 C++时，它要求知道有关这个对象的准确类型信息。下一步是在语言中访问这些信息，这很容易实现。

这一章解释RTTI是干什么的以及怎样使用它。另外，这一章还解释了 C++新的映射语法是什么，它和RTTI很相似。

18.1　例子——shape

这里有一个使用多态性的类层次的例子。基类为 shape，三个派生类分别为circle、square和triangle；

下面是一个典型的类继承关系图，基类在上，派生类向下生长。面向对象程序设计的一般目标就是用代码体管理指向基类的指针，所以如果想增加一个新类来扩充程序（比如从 shape中派生出rhomboid），代码体部分并不受影响。在本例中， shape接口部分的虚函数是draw()，其目的就是让用户程序员通过一个shape指针来调用draw()，draw()在所有的派生类中都被重新定义。由于它是一个虚函数，所以即使是用一个 shape()型的指针来调用它，它仍然会被正确调用。

因此，创建一个特定的对象 (circle、square或triangle)，取它的地址并把它映射到 shape*（忘掉对象的实际类型），然后在程序的其他地方用这个匿名指针。继承关系图如上所示，所以这种从多个派生类到基类的映射叫做向上映射。

18.2　什么是RTTI

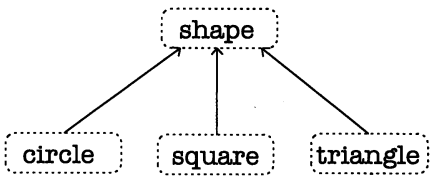

图　18-1

假如在编程中遇到了特殊的问题，而只要我们知道了一个一般指针的准确类型它就会迎刃而解，我们该怎么办？比如，假设允许我们的用户将任一形状变成紫色来表示加亮。用这种方法，他们可以发现屏幕上的所有三角形都被加亮。我们可能自然地想到用虚函数，像 TurnColorIfYouAreA()，它允许一些种类颜色的枚举型参数和 shape::circle、shape::square或shape::triangle参数。

为了解决这种问题，多数类库设计者把虚函数放在基类中，使运行时返回特定对象的类型信息。我们可能见过一些名字为 isA()和type Of() 之类的成员函数，这些就是开发商定义的RTTI函数。使用这些函数，当处理一个对象列表时就可以说：“ 如果这个对象是triangle类的，

就把它变成紫色。”

当C++中引进异常处理时，它的实现要求把一些运行时间类型信息放在虚函数表中。这意味着只要对语言作一点小小的扩充，程序员就能获得有关一个对象的运行时间类型信息。所有的开发商都在自己的类库中加入了RTTI，所以它已包含在C++语言中。

RTTI与异常一样，依赖驻留在虚函数表中的类型信息。如果试图在一个没有虚函数的类上用RTTI，就得不到预期的结果。

RTTI的两种使用方法

使用RTTI有两种不同的方法。第一种就像sizeof()，因为它看上就像一个函数。但实际上它是由编译器实现的。typeid()带有一个参数，它可以是一个对象引用或指针，返回全局typeinfo类的常量对象的一个引用。可以用运算符“==”和“!=”来互相比较这些对象。也可以用name()来获得类型的名称。注意，如果给typeid()传递一个shape*型参数，它会认为类型为shape*，所以如果想知道一个指针所指对象的精确类型，我们必须逆向引用这个指针。比如，s是个shape*，

```
cout << typeid(*s).name()<<endl;
```

将显示出s所指向的对象类型。

也可以用before(typeinfo&)查询一个typeinfo对象是否在另一个typeinfo对象的前面（以定义实现的排列顺序），它将返回true或false。如果写：

```
if(typeid(me).before(typeid(you))) //...
```

那么表示我们正在查询me在排列顺序中是否在you之前。

RTTI的第二个用法叫“安全类型向下映射”。之所以用“向下映射”这个词也是由于类继承的排列顺序。如果映射一个circle*到shape*叫向上映射的话，那么将一个shape*映射成一个circle*就叫向下映射了。当然一个circle*也是一个shape*，编译器允许任意的向上映射，但一个shape*不一定就是circle*，所以编译器在没有明确的类型映射时并不允许我们完成一个向下映射任务。当然可以用原有的C风格的类型映射或C++的静态映射（static_cast,将在本章末介绍）来强制执行，这等于在说：“我希望它实际上是一个circle*，而且我打算要求它是。”由于并没有明确地知道它实际上是circle，因此这样做是很危险的。在开发商制定的RTTI中一般的方法是：创建一个函数来试着将shape*指派为一个circle*(在本例中)，检查执行过程中的数据类型。如果这个函数返回一个地址，则成功；如果返回null，说明我们并没有一个circle*对象。

C++的RTTI的“安全类型向下映射”就是按照这种“试探映射”函数的格式，但它（非常合理地）用模板语法来产生这个特殊的动态映射函数（dynamic_cast），所以本例变成：

```
shape* sp = new circle;
circle* cp = dynamic_cast<circle*>(sp);
if(cp) cout << "cast successful";
```

动态映射的模板参数是我们想要该函数创建的数据类型，也就是这个函数的返回值。函数参数是我们试图映射的源数据类型。

通常只要对一种类型作这种工作（比如将三角型变成紫色），但如果想算出各种shape的数目，可以用面下例子的框架：

```
circle* cp = dynamic_cast<circle*>(sh);
square* sp = dynamic_cast<square*>(sh);
triangle* tp = dynamic_cast<triangle*>(sh);
```

当然这是人为的——我们可能已经在各个类型中放了一个静态数据成员并在构造函数中对

它自增。如果可以控制类的源代码并可以修改它，当然可以这样做。下面这个例子用来计算shape的个数，它用了静态数据成员和动态映射两种方法：

```
//: RTSHAPES.CPP -- Counting shapes
#include <iostream.h>
#include <time.h>
#include <typeinfo.h>
#include "..\14\tstash.h"

class shape {
protected:
  static int count;
public:
  shape() { count++; }
  virtual ~shape() = 0 { count--; }
  virtual void draw() const = 0;
  static int quantity() { return count; }
};

int shape::count = 0;

class rectangle : public shape {
  void operator=(rectangle&); // Disallow
protected:
  static int count;
public:
  rectangle() { count++; }
  rectangle(const rectangle&) { count++;}
  ~rectangle() { count--; }
  void draw() const {
    cout << "rectangle::draw()" << endl;
  }
  static int quantity() { return count; }
};

int rectangle::count = 0;

class ellipse : public shape {
  void operator=(ellipse&); // Disallow
protected:
  static int count;
public:
  ellipse() { count++; }
  ellipse(const ellipse&) { count++; }
  ~ellipse() { count--; }
  void draw() const {
    cout << "ellipse::draw()" << endl;
```

```
  }
  static int quantity() { return count; }
};

int ellipse::count = 0;

class circle : public ellipse {
  void operator=(circle&); // Disallow
protected:
  static int count;
public:
  circle() { count++; }
  circle(const circle&) { count++; }
  ~circle() { count--; }
  void draw() const {
    cout << "circle::draw()" << endl;
  }
  static int quantity() { return count; }
};

int circle::count = 0;

main() {
  tstash<shape> shapes;
  time_t t;
  // Seed random number generator:
  srand((unsigned)time(&t));
  const mod = 12;
  for(int i = 0; i < rand() % mod; i++)
    shapes.add(new rectangle);
  for(int j = 0; j < rand() % mod; j++)
    shapes.add(new ellipse);
  for(int k = 0; k < rand() % mod; k++)
    shapes.add(new circle);
  int Ncircles = 0;
  int Nellipses = 0;
  int Nrects = 0;
  int Nshapes = 0;
  for(int u = 0; u < shapes.count(); u++) {
    shapes[u]->draw();
    if(dynamic_cast<circle*>(shapes[u]))
      Ncircles++;
    if(dynamic_cast<ellipse*>(shapes[u]))
      Nellipses++;
    if(dynamic_cast<rectangle*>(shapes[u]))
      Nrects++;
```

```
    if(dynamic_cast<shape*>(shapes[u]))
      Nshapes++;
  }
  cout << endl << endl
       << "circles = " << Ncircles << endl
       << "ellipses = " << Nellipses << endl
       << "rectangles = " << Nrects << endl
       << "shapes = " << Nshapes << endl
       << endl
       << "circle::quantity() = "
       << circle::quantity() << endl
       << "ellipse::quantity() = "
       << ellipse::quantity() << endl
       << "rectangle::quantity() = "
       << rectangle::quantity() << endl
       << "shape::quantity() = "
       << shape::quantity() << endl;
}
```

对于这个例子，两种方法都是可行的，但静态数据成员方法只能用于我们拥有源代码并已安装了静态数据成员和成员函数时（或者开发商已为我们提供了这些），另外RTTI可能在不同的类中用法不同。

18.3 语法细节

本节详细介绍RTTI的两种形式是如何运行的以及两者之间的不同。

18.3.1 对于内部类型的typeid()

为了保持一致性，typeid()也可以运用于内部类型，所以下面的表达式结果为true：

```
typeid(47) == typeid(int)
typeid(0) == typeid(int)
int i;
typeid(i) == typeid(int)
typeid(&i) == typeid(int*)
```

18.3.2 产生合适的类型名字

typeid()必须在所有的状况下都可以运行，比方说，下面的类中包含了一个嵌套类：

```
//: RNEST.CPP -- Nesting and RTTI
#include <iostream.h>
#include <typeinfo.h>

class one {
  class nested {};
  nested* n;
public:
```

```
    one() { n = new nested; }
    ~one() { delete n; }
    nested* N() { return n; }
};

main() {
    one O;
    cout << typeid(*O.N()).name() << endl;
}
```

typeinfo::name()成员函数还是提供了适当的类名，其结果为one::nested。

18.3.3 非多态类型

虽然typeid( )可以运用于非多态类型（基类中没有虚函数），但我们用这种方法获得的信息是值得怀疑的。假设类层次如下：

```
class X {
  int i;
public:
  // ...
};
class Y : public X {
  int j;
public:
  // ...
};
```

如果创建一个派生类对象并向上映射它：

```
X* xp = new Y;
```

typeid()运算符将返回一个结果，但可能不是我们想要的。因为没有非多态机制，所以可以使用静态类型信息：

```
typeid(*xp) == typeid(X)
typeid(*xp) != typeid(Y)
```

一般希望RTTI用于多态类。

18.3.4 映射到中间级

动态映射不仅可用来确定准确的类型，也可用于多层次继承关系中的中间类型。如下例：

```
class d1 {
public:
  virtual void foo() {}
};
class d2 {
public:
  virtual void bar() {}
};
class mi : public d1, public d2 { };
```

```
class mi2 : public mi {};

d2* D2 = new mi2;
mi2* MI2 = dynamic_cast<mi2*>(D2);
mi* MI = dynamic_cast<mi*>(D2);
```

由于多重继承，问题变得更复杂了。如果我们创建了一个 mi2并将向上映射到根类（在这种情况下，从两种可能的根类中选出一种），然后成功地动态映射回派生类mi或mi2。

我们甚至可以从一个根类映射到另一个：

```
d1* D1 = dynamic_cast<d1*>(D2);
```

这可以成功地映射，因为D2实际上指向一个mi2对象，它包含了类型d1的一个子对象。

映射到中间级在dynamic_cast与typeid()之间产生了一个有趣的差异。typeid()总是产生一个typeinfo 对象的引用来描述一个对象的准确类型，因此它不会给出中间层次的信息。在下面的表达式中（它的值是true），typeid()并不像dynamic_cast那样把D2看作一个指向派生类的指针：

```
typeid(D2) != typeid(mi2*)
```

D2的类型就是指针的准确类型：

```
typeid(D2) == typeid(d2*)
```

18.3.5 void指针

运行时类型的识别对一个void型指针不起作用：

```
//: VOIDRTTI.CPP -- RTTI & void pointers
#include <iostream.h>
#include <typeinfo.h>

class stimpy {
public:
  virtual void happy() {}
  virtual void joy() {}
};

main() {
  void* v = new stimpy;
  // Error:
//!  stimpy* s = dynamic_cast<stimpy*>(v);
  // Error:
//!  cout << typeid(*v).name() << endl;
}
```

void*确实意味着“根本没有类型信息”。

18.3.6 用模板来使用RTTI

模板产生许多不同类的名字，而有时希望指出有关下面使用的对象是哪个类的。RTTI提供

了一个实现此功能的方便方法。下面的例子修改了第13章的代码，它没有用预处理宏显示了构造函数和析构函数调用的顺序。

```
//: INHORDER.CPP -- Order of constructor calls
#include <iostream.h>
#include <typeinfo.h>

template<int id> class announce {
public:
  announce() {
    cout << typeid(*this).name()
         << " constructor " << endl;
  }
  ~announce() {
    cout << typeid(*this).name()
         << " destructor " << endl;
  }
};

class X : public announce<0> {
  announce<1> m1;
  announce<2> m2;
public:
  X() { cout << "X::X()" << endl; }
  ~X() { cout << "X::~X()" << endl; }
};

main() { X x; }
```

TYPEINFO.H头文件必须被包含，以便于调用typeid()返回的typeinfo对象的任何成员函数。这个模板用了一个整型常量来区分两个类，但用类参数也行。在构造函数与析构函数的内部，RTTI的信息用来产生类的名字并显示，类X既用了继承又用了组合来创建一个类，这里构造函数与析构函数调用的顺序很有趣。

这种技术有助于我们理解C++语言是如何工作的。

18.4 引用

RTTI必须能与引用一起工作。指针与引用存在明显不同，因为引用总是由编译器逆向引用，而一个指针的类型或它指向的类型可能要检测。请看下例：

```
class B {
public:
  virtual float f() { return 1.0;}
};
class D : public B { /* ... */ };
B* p = new D;
B& r = *p;
```

typeid()看到的指针类型是基类而不是派生类，而它看到的引用类型则是派生类：

```
typeid(p) == typeid(B*)
typeid(p) != typeid(D*)
typeid(r) == typeid(D)
```

与此相反，指针指向的类型在 typeid()看来是派生类而不是基类，而用一个引用的地址时产生的是基类而不是派生类。

```
typeid(*p) == typeid(D)
typeid(*p) != typeid(B)
typeid(&r) == typeid(B*)
typeid(&r) != typeid(D*)
```

表达式也可以用typeid()运算符，因为它们也有一个类型：

```
typeid(r.f()) == typeid(float)
```

异常

对一个引用完成了一个动态映射，其结果还必须被指定到一个引用上。但如果映射失败则会产生什么呢？因为不能有空的引用，所以这里是抛出一个异常的合适地方。在标准 C++中异常类型为bad-cast，但在下面的例子中，用一个处理块来捕获所有异常：

```
class X {};

mi MI;
d1 & D1 = MI; // upcast to reference
try {
  X& xr = dynamic_cast<X&>(D1);
} catch(...) {
  cout << "dynamic_cast<X&>(D1) failed"
       << endl;
}
```

失败的原因当然是因为D1实际上并不指向一个X对象，如果这里没有抛出一个异常， xr就会没有边界，所有被创建的对象或引用的安全保障都可能被打破。

如果在调用typeid()时试图去除一个空指针的引用，也会引起一个异常，在标准 C++中，这个异常叫bad-typeid:

```
B* bp = 0;
try {
  typeid(*bp); // throws exception
} catch(bad_typeid) {
  cout << "Bad typeid() expression" << endl;
}
```

这里可以在typeid操作之前检查指针是否为空来避免异常的产生（不像上面的那个引用的例子），这是最好的方法。

18.5 多重继承

当然，RTTI机制必须适用于任何复杂的多重继承，包括virtual基类：

```
//: MIRTTI.CPP -- MI & RTTI
#include <iostream.h>
#include <typeinfo.h>

class BB {
public:
 virtual void f() {}
};
class B1 : virtual public BB {};
class B2 : virtual public BB {};
class MI : public B1, public B2 {};

main() {
  BB* bbp = new MI; // Upcast
  // Proper name detection:
  cout << typeid(*bbp).name() << endl;
  // Dynamic_cast works properly:
  MI* mip = dynamic_cast<MI*>(bbp);
  // Can't force old-style cast:
  //! MI* mip2 = (MI*)bbp; // Compile error
}
```

即使只提供一个virtual基类指针，typeid()也能准确地检测出实际对象的名字。用动态映射同样也会工作得很好，但编译器将不允许我们试图用原来的方法强制映射：

```
MI* mip = (MI*)bbp; // compile-time error
```

编译器知道这不可能正确，所以它要求我们用动态映射。

18.6 合理使用RTTI

因为RTTI可以让我们用一个匿名的多态指针来发现类型信息，所以它常常被初学者滥用，因为它可能在虚函数完成之前就有意义了。

对于许多来自过程编程背景的人来说，要他们不把程序组织成为一组 switch语句是非常困难的。他们可能会用 RTTI完成这些，但这样会在代码开发维护阶段丢失多态性的非常重要的价值。C++的意图是：尽可能地使用虚函数，必要时才使用 RTTI。

当然，要想以我们所想的那样使用虚函数，我们必须控制基类的定义，因为随着程序的不断扩大，有时我们可能发现基类并没有我们想要的虚函数，如果基类来自类库或其他由别人控制的来源，就可以用 RTTI作为一种解决办法：我们可以继承一个新类并加上我们的成员函数。在代码的其他地方我们可以检测到我们的新增类型和调用的那个成员函数。这不会破坏多态性和程序逻辑的可扩展性，因为加一个新类并不要求我们寻找 switch语句。当然如果在主程序中增加新的代码时用到了这个新类，我们就必须检测这个特定类型。

把一个特征放在一个基类中可能意味着为了某个特定类的利益，所有从该类派生出的类都保留了一些无意义的虚函数的残留。这使得接口变得不清晰，使那些必须重新定义纯虚函数的人当他们从这个类派生新类时感到很不方便。比方说，假设在第 14章（14.6）节的WINDS.CPP程序中，我们想清除管弦乐队中所有乐器的无用值。一种方法是在基类 instrument

中放一个虚函数ClearSpitvalve()，但这就会引起混乱，因为它暗示percussion和electronic乐器也有无用值。RTTI提供了一个更合理的方法，因为可以把函数放在一个合适的特定类中（这里是wind）。

最后，RTTI有时可以解决效率问题。如果代码用一种好的方法来用多态机制，但结果是这种通用代码对某个对象起反作用，使其运行效率低下。我们可以用 RTTI将这种类型找出来，并写出针对特定情况的代码以提高效率。

回顾垃圾再生器例子

下面是第15章垃圾再生例子（trash recycling）的一个相似的版本，这里我们没有在类层次中建立类信息，而是采用了RTTI：

```
//: RECYCLE2.CPP -- Chapter 14 example w/ RTTI
#include <fstream.h>
#include <stdlib.h>
#include <time.h>
#include <typeinfo.h>
#include "..\14\tstack.h"
ofstream out("recycle2.out");

class trash {
  float Weight;
public:
  trash(float Wt) : Weight(Wt) {}
  virtual float value() const = 0;
  float weight() const { return Weight; }
  virtual ~trash() {}
};

class aluminum : public trash {
  static float val;
public:
  aluminum(float Wt) : trash(Wt) {}
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float aluminum::val = 1.67;
class paper : public trash {
  static float val;
public:
  paper(float Wt) : trash(Wt) {}
  float value() const { return val; }
  static void value(int newval) {
```

```
    val = newval;
  }
};

float paper::val = 0.10;

class glass : public trash {
  static float val;
public:
  glass(float Wt) : trash(Wt) {}
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float glass::val = 0.23;

// Sums up the value of the trash in a bin:
template<class T> void
SumValue(const tstack<T>& bin, ostream& os) {
  tstackIterator<T> tally(bin);
  float val = 0;
  while(tally) {
    val += tally->weight() * tally->value();
    os << "weight of "
       << typeid(*tally.current()).name()
       << " = " << tally->weight() << endl;
    tally++;
  }
  os << "Total value = " << val << endl;
}

main() {
  // Seed the random number generator
  time_t t;
  srand((unsigned)time(&t));

  tstack<trash> bin; // Default to ownership
  // Fill up the trash bin:
  for(int i = 0; i < 30; i++)
    switch(rand() % 3) {
      case 0 :
        bin.push(new aluminum(rand() % 100));
        break;
      case 1 :
```

```
        bin.push(new paper(rand() % 100));
        break;
      case 2 :
        bin.push(new glass(rand() % 100));
        break;
    }
  // Note difference w/ chapter 14: Bins hold
  // exact type of object, not base type:
  tstack<glass> glassbin(0); // No ownership
  tstack<paper> paperbin(0);
  tstack<aluminum> ALbin(0);
  tstackIterator<trash> sorter(bin);
  // Sort the trash:
  while(sorter) {
    aluminum* ap =
      dynamic_cast<aluminum*>(sorter.current());
    paper* pp =
      dynamic_cast<paper*>(sorter.current());
    glass* gp =
      dynamic_cast<glass*>(sorter.current());
    if(ap) ALbin.push(ap);
    if(pp) paperbin.push(pp);
    if(gp) glassbin.push(gp);
    sorter++;
  }
  SumValue(ALbin, out);
  SumValue(paperbin, out);
  SumValue(glassbin, out);
  SumValue(bin, out);
}
```

这个问题的本质是这些垃圾被扔进了一个没有分类的单一的垃圾箱中，所以特定的类型信息被丢失了。但之后特定类型信息必须恢复以便对垃圾准确分类，所以 RTTI被用上了。在第15章中，一个RTTI系统插入到类的继承关系中，但正如我们在这里看到的那样，用 C++预定义的RTTI更方便。

18.7 RTTI的机制及花费

典型的RTTI是通过在VTABLE中放一个额外的指针来实现的。这个指针指向一个描述该特定类型的typeinfo结构（每个新类只产生一个typeinfo的实例），所以typeid()表达式的作用实际上很简单。VPTR用来取typeinfo的指针，然后产生一个结果typeinfo结构的一个引用——这是一个决定性的步骤——我们已经知道它要花多少时间。

对于dynamic_cast<目标*><源指针>，多数情况下是很容易的，先恢复源指针的 RTTI信息再取出目标*的类型RTTI信息，然后调用库中的一个例程判断源指针是否与目标 *相同或者是目标*类型的基类。它可能对返回的指针做了一点小的改动，因为目的指针类可能存在多重继承的情况，而源指针类型并不是派生类的第一个基类。在多重继承时情况会变得复杂些，因为

一个基类在继承层次中可能出现一次以上，并且可能有虚基类。

用于动态映射的库例程必须检查一串长长的基类列表，所以动态映射的开销比 typeid()要大（当然我们得到的信息也不同，这对于我们的问题来说可能很关键），并且这是非确定性的，因为查找一个基类要比查找一个派生类花更多的时间。另外动态映射允许我们比较任何类型，不限于在同一个继承层次中比较两个类。这使得动态映射调用的库例程开销更高了。

18.8 创建我们自己的RTTI

如果编译器还不支持RTTI，可以在类库中很容易地建立自己的RTTI。这是很有意义的事情，因为在人们发现所有的类库实际上都有要用到某种形式的 RTTI之后才在C++引入RTTI。（在异常处理被加入到C++后，人感觉“自由”一些了，因为异常处理要求有关类的准确信息）。

从本质上说，RTTI只要两个函数就行了，一个用来指明类的准确类型的虚函数，一个取得基类的指针并将它向下映射成派生类，这个函数必须产生一个指向更加派生类的指针（我们可能希望也能处理引用）。有许多方法来实现我们自己的RTTI，但都要求每个类有一个唯一的标识符和一个能产生类型信息的虚函数。下例用了一个叫 dynacast()的静态成员函数，它调用一个类型信息函数dynamic_type()，这两个函数都必须在每个新派生类中重新定义：

```
//: SELFRTTI.CPP -- Your own RTTI system
#include "..\14\tstack.h"
#include <iostream.h>
class security {
protected:
  enum { baseID = 1000 };
public:
  virtual int dynamic_type(int ID) {
    if(ID == baseID) return 1;
    return 0;
  }
};

class stock : public security {
protected:
  enum { typeID = baseID + 1 };
public:
  int dynamic_type(int ID) {
    if(ID == typeID) return 1;
    return security::dynamic_type(ID);
  }
  static stock* dynacast(security* s) {
    if(s->dynamic_type(typeID))
      return (stock*)s;
    return 0;
  }
};

class bond : public security {
```

```
protected:
  enum { typeID = baseID + 2 };
public:
  int dynamic_type(int ID) {
    if(ID == typeID) return 1;
    return security::dynamic_type(ID);
  }
  static bond* dynacast(security* s) {
    if(s->dynamic_type(typeID))
      return (bond*)s;
    return 0;
  }
};

class commodity : public security {
protected:
  enum { typeID = baseID + 3};
public:
  int dynamic_type(int ID) {
    if(ID == typeID) return 1;
    return security::dynamic_type(ID);
  }
  static commodity* dynacast(security* s) {
    if(s->dynamic_type(typeID))
      return (commodity*)s;
    return 0;
  }
  void special() {
    cout << "special commodity function\n";
  }
};

class metal : public commodity {
protected:
  enum { typeID = baseID + 4};
public:
  int dynamic_type(int ID) {
    if(ID == typeID) return 1;
    return commodity::dynamic_type(ID);
  }
  static metal* dynacast(security* s) {
    if(s->dynamic_type(typeID))
      return (metal*)s;
    return 0;
  }
```

```
};

main() {
  tstack<security> portfolio;
  portfolio.push(new metal);
  portfolio.push(new commodity);
  portfolio.push(new bond);
  portfolio.push(new stock);
  tstackIterator<security> it(portfolio);
  while(it) {
    commodity* cm =
      commodity::dynacast(it.current());
    if(cm) cm->special();
    else cout << "not a commodity" << endl;
    it++;
  }
  cout << "cast from intermediate pointer:\n";
  security* sp = new metal;
  commodity* cp = commodity::dynacast(sp);
  if(cp) cout << "it's a commodity\n";
  metal* mp = metal::dynacast(sp);
  if(mp) cout << "it's a metal too!\n";
}
```

每个子类必须创建它自己的typeID，重新定义虚函数dynamic_type()来返回这个typeID，并定义一个静态成员调用 dynacast()，它用一个基类指针作为参数（或继承层次中任意层上的一个指针——在这种情况下，指针被简单地向上映射）。

在从security派生出的类中，可以看到每个类都定义了自己的 typeID，并加到baseID中。baseID可以从派生类中直接访问，这一点是关键所在，因为enum必须在编译时计算出值的大小，所以采用内联函数的方法来读一个私有数据成员的方法不会成功。这是一个需要 protected成员的一个典型事例。

enum baseID为所有从security派生出的类建立了一个基本的标识符，这样如果一个 ID值与已有ID值发生冲突，可能改变一下基值就可以改变所有的ID值（因为这个例子中并不比较不同的继承树，所以不可能发生 ID冲突）。在所有的类中，类的ID值都是protected，所以它可以被派生类访问，但终端用户则不能访问它们。

这个例子说明了创建RTTI需要处理哪些事情。我们不仅要确定对象的准确类型，还要能判断这个类是不是从我们要找的类中派生出来的。比如： metal是从commodity派生出来的，commodity有一个叫special()的函数，所以如果有一个metal类的对象，就可以调用special()。如果dynamic_type()只告诉我们这个对象的准确类型，当我们问它一个 metal对象是否是commodity对象时，它会说“不是”，而这实际上是不正确的。所以 RTTI还必须能合理地在继承层次中将一种类型映射到某一中间类型上和准确类型上。

dynacast()函数通过调用虚函数dynamic_type()来确定类型信息。这个函数用一个我们正试图映射到的类的 typeID为参数。它是一个虚函数，所以函数体在对象的准确类型中。每个dynamic_type()函数首先检查传入的 typeID是否与自己的类型匹配，它检查是否与基类匹配，

这只要调用基类的dynamic_type()函数就行了。就像一个循环函数调用，每个dynamic_type()都检查传入的参数是否与自己的ID值相等，如不匹配，它调用基类的dynamic_type()，并将它结果返回。当一直找到继承树的根部时，它将返回零，表示没有匹配的类。

如果dynamic_type()返回1(true)，则指针指向的对象要么就是我们要找的类型，要么是这个类的派生类，然后dynacast()用security指针作参数，并把它映射成想要的类型，如果返回值是false,，dynacast()返回零，表示映射不成功，用这种方法使它看上去就像 C++中的dynamic_cast运算符一样。

C++的动态映射运算符比上面的例子多一项功能：它可以比较两个继承层次中的类型，这两个继承层次可以是完全分开的。这就增加了系统的通用性，使它适用于跨层次体系的类型比较，当然这也增加了系统的复杂性。

现在我们很容易想像出怎样创建一个使用上面方案并允许更容易转换成内置 dynamic_cast运算符的DYNAMIC-CAST宏来。

18.9 新的映射语法

无论什么时候用类型映射，都是在打破类型系统 [1]，这实际上是在告诉编译器，即使知道一个对象的确切类型，还是可以假定认为它是另外一种类型。这本身就是一种很危险的事情，也是一个容易发生错误的地方。

不幸的是，每一种类型映射都是不同的：它是用括号括起来的目标类型的名字。所以如果我们的一段代码不能正确工作，而我们知道应该检查所有的类型映射看它们是否是产生错误的原因。我们怎么保证可以找出所有的类型映射呢？在一个 C程序中无法做到这一点。因为C编译器并不总是要求类型映射（它可以用一个 void指针指向不同的类型而不必强迫使用映射），而映射表现不同，所以我们不知道我们是不是已经找出所有的映射了。

为了解决这个问题， C++用保留字dynamic_cast(本章第一部分的主题)、const_cast、static_cast和reinterpret_cast来提供了一个统一的类型映射语法。当需要进行动态映射时，这就提供了一个解决问题的可能。这意味着那些已有的映射语法已经被重载得太多了，不能再支持任何其他的功能了。

通过使用这些映射来代替原有的（ newtype）语法，我们可以在任何程序中很容易地找出所有的映射。为了支持已有的代码，大多数编译器都可以产生不同级别的错误或警告，并可由用户对错误或警告产生选择打开或关闭。如果把新类型映射的全部错误打开的话，就可以确保我们找出项目中所有的类型映射，这使得查找错误变得很容易。

下表指出了四个不同形式的映射的含义：

static_cast	为了“行为良好”和“行为较好”而使用的映射，包括一些我们可能现在不用的映射（如向上映射和自动类型转换）
const_cast	用于映射常量和变量（const和volatile）
dynamic_cast	为了安全类型的向下映射（本章前面已经介绍）
reinterpret_cast	为了映射到一个完全不同的意思。这个关键词在我们需要把类型映射回原有类型时要用到它。我们映射到的类型仅仅是为了故弄玄虚和其他目的。这是所有映射中最危险的

三个新映射将在后面小节中完整地介绍。

[1] 参看Josée Lajoie“The new cast notation and the bool data type” ，C++报告，1994年9月，pp.46-51.

18.9.1 static_cast

static_cast可以用于所有良定义转换。这些包括“安全”转换与次安全转换，“安全”转换是编译器允许我们不用映射就能完成的转换。次安全转换也是良定义的。由 static_cast覆盖的变换的类型包括典型的无映射变换、窄化变换（丢失信息）、用void*的强制变换、隐式类型变换和类层次的静态导航。

```
//: STATCAST.CPP -- Examples of static_cast

class base { /* ... */ };
class derived : public base {
public:
  // ...
  // Automatic type conversion:
  operator int() { return 1; }
};

void func(int) {}

class other {};

main() {
  int i = 0x7fff; // Max pos value = 32767
  long l;
  float f;
  //(1) typical castless conversions:
  l = i;
  f = i;
  // Also works:
  l = static_cast<long>(i);
  f = static_cast<float>(i);

  //(2) narrowing conversions:
  i = l; // May lose digits
  i = f; // May lose info
 // Says "I know," eliminates warnings:
 i = static_cast<int>(l);
 i = static_cast<int>(f);
 char c = static_cast<char>(i);

 //(3) forcing a conversion from void* :
 void* vp = &i;
 // Old way produces a dangerous conversion:
 float* fp = (float*)vp;
 // The new way is equally dangerous:
 fp = static_cast<float*>(vp);
```

```
  //(4) implicit type conversions, normally
  // Performed by the compiler:
  derived d;
  base* bp = &d; // Upcast: normal and OK
  bp = static_cast<base*>(&d); // More explicit
  int x = d; // Automatic type conversion
  x = static_cast<int>(d); // More explicit
  func(d); // Automatic type conversion
  func(static_cast<int>(d)); // More explicit

  //(5) Static Navigation of class hierarchies:
  derived* dp = static_cast<derived*>(bp);
  // ONLY an efficiency hack. dynamic_cast is
  // Always safer. However:
  // Other* op = static_cast<other*>(bp);
  // Conveniently gives an error message, while
  other* op2 = (other*)bp;
  // Does not.
}
```

在第（1）段中，我们看到了在C语言中使用过的各种类型的转换，用映射的或不用映射的。从一个int到一个long或float当然不成问题，因为后者可以存放 int包含的全部值，可以用一个static_cast来使这些转换变得醒目一些，虽然并不是必须要这样做。

在第（2）段中，我们可以看到转换回原有类型的方法。这里可能丢失部分数据，因为一个int没有一个long或float那么“宽”。因此这些被称为“窄化转换”，编译器仍可以完成这些转换，但会给我们一个警告信息。可以去掉这些警告，并用一个映射指出我们确实需要映射。

在C++中，从void*向外赋值是不允许的（这与C中不同），请看第（3）段。这样做是很危险的，它要求程序员清楚地知道他正在干什么。当我们查找错误时， static_cast比老式的标准映射更容易定位。

第（4）段显示了几种隐式类型转换，这些通常是由编译器自动完成的。它们是自动的和不要求映射的，用static_cast会使它变得醒目，以便于我们以后需要查找它或明确它们的含义。

如果在一个类层次中没有虚函数或者如果我们有其他允许我们安全地向下映射的信息，则用静态向下映射比 dynamic_cast稍微快一些，就像在第（ 5）段中显示的那样。另外，static_cast不允许我们映射到类层次之外，就像传统的映射一样，所以它更安全。然而静态地导航类层次总是冒险的，因此我们应该用dynamic_cast，除非特殊情况。

18.9.2 const_cast

如果想把一个 const 转换为非const，或把一个volatile转换成一个非 volatile，就要用到const_cast。这是可以用const_cast的唯一转换。如果还有其他的转换牵涉进来，它必须分开来指定，否则会有一个编译错误。

```
//: CONSTCST.CPP -- Const casts

main() {
```

```
  const int i = 0;
  int* j = (int*)&i; // Deprecated form
  j  = const_cast<int*>(&i); // Preferred
  // Can't do simultaneous additional casting:
//! long* l = const_cast<long*>(&i); // Error
  volatile int k = 0;
  int* u = const_cast<int*>(&k);
}

class X {
  int i;
// mutable int i; // a better approach
public:
  void f() const {
    // Casting away const-ness:
    (const_cast<X*>(this))->i = 1;
  }
};
```

如果用了一个const对象的地址创建一个指针指向一个 const，在没有一个映射时不能把它赋给一个非const的指针中。老式风格的映射可以完成这个，但使用 const_cast更合适。这同样适用于volatile。

如果想在一个const成员函数内部改变一个类成员，传统的方法就是用（ X*）this映射掉常量性质。现在也可以使用较好的 const_cast来映射掉常量性质，但一个更好的方法是使那些特殊的数据成员成为 mutable，这样在类定义时它更清楚，不会在成员函数定义中被隐藏掉，而且这些成员可以在const成员函数中改变。

18.9.3 reinterpret_cast

这是一种不太安全的类型映射机制，也是最容易引起错误的一种。一般情况下，编译器都包含了一组开关，允许我们强制使用 const_cast和reinterpret_cast，它们可以定位那些不安全的类型映射。

reinterpret_cast假设一个对象仅仅是一个比特模式，它可以被当作完全不同的对象对待（为了某种模糊的目的）。这是低层处理，在C中已经很不好了。实际上在我们用它做别的事情之前，总是要用reinterpret_cast将其映射回原来的类型。

```
//: REINTERP.CPP -- Reinterpret_cast
// Example depends on VPTR location,
// Which may differ between compilers.
#include <string.h>
#include <fstream.h>
ofstream out("reinterp.out");

class X {
  enum { sz = 5 };
  int a[sz];
```

```
public:
  X() { memset(a, 0, sz * sizeof(int)); }
  virtual void f() {}
  // Size of all the data members:
  int membsize() { return sizeof(a); }
  friend ostream&
    operator<<(ostream& os, const X& x) {
      for(int i = 0; i < sz; i++)
        os << x.a[i] << ' ';
      return os;
  }
};

main() {
  X x;
  out << x << endl; // Initialized to zeroes
  int* xp = reinterpret_cast<int*>(&x);
  xp[1] = 47;
  out << x << endl; // Oops!

  X x2;
  const vptr_size = sizeof(X) - x2.membsize();
  long l = reinterpret_cast<long>(&x2);
  // *IF* the VPTR is first in the object:
  l += vptr_size; // Move past VPTR
  xp = reinterpret_cast<int*>(l);
  xp[1] = 47;
  out << x2 << endl;
}
```

类X包含一些数据和一个虚函数，在main()中，一个X的对象被打印出以显示出它已被初始化为零了，然后它的地址用reinterpret_cast映射为一个int*，假设它是一个int*，这个对象被索引成像一个数组，并且成员1被置为47（理论上），但在这里输出结果[1]却是：

```
0 0  0  0  0
47 0  0  0  0
```

很明显，认为对象的第一个数据存放在对象的起始地址处这一假定是不安全的。事实上，这个编译器把VPTR放在对象的开始处，所以如果用xp[0]而不是用xp[1],就会使VPTR变得毫无价值。

为了更正这个错误，可以用对象的大小减去数据成员的大小算出 VPTR的大小，然后对象的地址被映射为一个 long型(用reinterpret_cast)。假定VPTR是放在对象的开始处的，这样，实际数据的开始地址就被计算出来了。结果数字映被射回 int*，现在索引值可以产生想要的结果了：

```
0  47  0  0  0
```

[1] 对于特定的编译器，结果可能不同。

当然这种方法不值得推荐，而且可移植性差。这是一个 reinterpret_cast指示器能做的事情之一，但当我们决定我们必须要用它时，它是可用的。

18.10 小结

RTTI是一个很方便的额外特征，就像蛋糕上加了一层糖衣。虽然一般都是把一个指针向上映射为一个基类指针，然后使用基类的接口（通过虚函数），但是偶尔需要知道一个基类指针指向的对象的确切类型来提高程序的效率，这时如果我们束手无策，就可使用 RTTI。因为基于虚函数的RTTI已经出现在几乎所有的类库中，所以这是一个非常有用的特征，因为它意味着：

1) 我们并不需要把它建在其他类库中。
2) 我们不用担心它是否将建在其他库中。
3) 在继承过程中我们不需要有额外的编程花费用来管理 RTTI配置。
4) 语法是一致的，我们不需要为每个新库来重新配置。

因为RTTI使用很方便，像C++的多数特征一样，所以它可能被滥用，包括那些天真的或有决心的程序员。最常见的滥用可能来自那些不理解虚函数的程序员，他们用 RTTI去做类型检查编码。C++的哲学似乎是提供强有力的工具并维护类型的完整和防止类型的违规，但我们如果有意滥用某一个特征的话，没有什么可以阻止我们。有时走点小弯路是获取经验的最快途径。

新的类型映射语法在调试阶段对我们有很大的帮助，因为这种类型映射在我们的类型系统中开了一个小洞，并允许错误流进去。而这种新的语法使我们更容易定位这些错误入口通道。

18.11 练习

1. 用RTTI帮助程序调试，即打印一个使用了 typeid()的模板的确切名称。用不同的类型将这个模板实例化，然后看看结果是什么。

2. 用RTTI实现本章前面讲的TurnColorIfYouAreA()函数。

3. 将第14章的WIND5.CPP拷贝到一个新的位置，然后修改其中的 instrument的层次。在wind类中加一个虚函数ClearSpitValve()并在所有的wind的派生类中重新定义它。让 tstash的一个实例拥有一些instrument指针，把用new创建的各类instrument对象赋给它们。现在用RTTI巡视这个包容器，在所有的对象中找类 wind或它的派生类的对象，为这些对象调用ClearSpitValve()函数。注意，如果instrument基类中已经包含了ClearSpitValve()函数，它可能会引起混乱。