第22章　运行时类型标识与强制转换运算符

标准C++包含两个有助于支持面向对象的现代编程方法的功能：即运行时类型标识（RTTI）和四个附加的强制转换运算符。这两个功能在最初的C++规范说明中没有提供，它们是为了支持运行时的多态性后来添加的。RTTI使你可以在程序执行期间指定一个对象的类型，而强制转换运算符提供了更安全、更可控的转换方法。由于强制转换操作中的dynamic_cast与RTTI直接相关，所以本书把它们放在同一章讨论。

22.1　运行时类型标识

因为在诸如C这样的非多态性语言中没有运行时类型的内容，所以对你来说它们可能是新内容。在非多态性语言中，因为在编译时（也就是在编写程序时）才知道每个对象的类型，所以不需要运行时类型信息。然而，在诸如C++这样的多态性语言中，因为在程序执行之前没有确定对象的精确属性，所以可能出现编译时不知道对象类型的情况。本书曾在第17章讲到，C++利用对象的层次性、虚函数以及基类指针实现了多态性。由于基类指针可用于指向该基类的对象或任何从它派生出来的对象，所以在某个特定时间不可能总是能够及时知道基类指针指向的对象类型，类型的确定要在运行时利用运行时类型标识做出。

为了获得一个对象的类型，可使用typeid函数。为了使用typeid，必须包含头文件<typeinfo>。最常用的typeid函数形式如下：

```
typeid(object)
```

其中，object是将要获得的类型的对象，它可以是任何类型的对象，其中包括内置类型和你创建的类类型。typeid函数返回一个对type_info类对象的引用，该类描述了对象类型。

type_info类定义了下列公有成员：

```
bool operator==(const type_info &ob);
bool operator!=(const type_info &ob);
bool before(const type_info &ob);
const char *name( );
```

重载运算符==和!=用于类型比较。如果对象在用做一个参数之前被调用，before( )函数将返回true（该函数通常只作为内部使用，其返回值对继承和类层次结构而言没有任何用处）。name( )函数返回一个指向类型名的指针。

下面是一个使用typeid的简单例子：

```
// A simple example that uses typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class myclass1 {
```

```
  // ...
};

class myclass2 {
  // ...
};

int main()
{
  int i, j;
  float f;
  char *p;
  myclass1 ob1;
  myclass2 ob2;

  cout << "The type of i is: " << typeid(i).name();
  cout << endl;
  cout << "The type of f is: " << typeid(f).name();
  cout << endl;
  cout << "The type of p is: " << typeid(p).name();
  cout << endl;

  cout << "The type of ob1 is: " << typeid(ob1).name();
  cout << endl;
  cout << "The type of ob2 is: " << typeid(ob2).name();
  cout << "\n\n";

  if(typeid(i) == typeid(j))
    cout << "The types of i and j are the same\n";

  if(typeid(i) != typeid(f))
    cout << "The types of i and f are not the same\n";
  if(typeid(ob1) != typeid(ob2))
    cout << "ob1 and ob2 are of differing types\n";

  return 0;
}
```

上面这个程序的输出如下所示：

```
The type of i is: int
The type of f is: float
The type of p is: char *
The type of ob1 is: class myclass1
The type of ob2 is: class myclass2

The types of i and j are the same
The types of i and f are not the same
ob1 and ob2 are of differing types
```

typeid最重要的用途体现在通过一个多态的基类指针使用它时。在这种情况下，它将自动返回指针指向的实际对象的类型，它或许是一个基类对象，也可能是一个从基类中派生出来的对象（记住，基类指针可以指向该基类的对象或从它派生出来的任何类）。因此，利用typeid可以在运行时确定基类指针所指的对象类型。下面的程序说明了这个原理：

```
// An example that uses typeid on a polymorphic class hierarchy.
#include <iostream>
#include <typeinfo>
using namespace std;

class Mammal {
public:
  virtual bool lays_eggs() { return false; } // Mammal is polymorphic
  // ...
};

class Cat: public Mammal {
public:
  // ...
};

class Platypus: public Mammal {
public:
  bool lays_eggs() { return true; }
  // ...
};

int main()
{
  Mammal *p, AnyMammal;
  Cat cat;
  Platypus platypus;

  p = &AnyMammal;
  cout << "p is pointing to an object of type ";
  cout << typeid(*p).name() << endl;

  p = &cat;
  cout << "p is pointing to an object of type ";
  cout << typeid(*p).name() << endl;

  p = &platypus;
  cout << "p is pointing to an object of type ";
  cout << typeid(*p).name() << endl;

  return 0;
}
```

程序的输出如下所示：

```
p is pointing to an object of type class Mammal
p is pointing to an object of type class Cat
p is pointing to an object of type class Platypus
```

正如前面介绍的那样，当typeid应用于一个多态类型的基类指针时，指针所指的对象类型将在运行时确定，就像上面程序的输出显示的那样。

无论哪一种情况，只要将typeid应用于一个非多态的类层次结构的指针，就可以获得指针的基类。也就是说，没有做出指针实际指向何种类型的决定。例如，可在Mammal中的lays_eggs()函数前加上关键字virtual，然后再编译并运行该程序。这个程序的输出如下所示：

```
p is pointing to an object of type class Mammal
p is pointing to an object of type class Mammal
p is pointing to an object of type class Mammal
```

由于Mammal不再是多态的类，所以每一个对象类型都将是Mammal，因为它是指针的类型。

因为typeid经常被用于一个非关联的指针（即，已经应用了运算符*的指针），所以创建了一个专门的异常来处理这种情况，其中，非关联的指针为空指针。在这种情况下，typeid抛出bad_typeid异常。对一个多态类的引用的工作方式与指针相同。当typeid被用于一个多态类对象的引用时，它将返回实际引用的对象类型，该类型可以是派生类型。最常使用该功能的场合是通过引用给函数传递对象时。例如，在下面的程序中，函数WhatMammal()声明了一个针对Mammal类型对象的引用参数，这意味着WhatMammal()通过该引用参数可以传递Mammal的对象或从Mammal派生的任何类。当把typeid运算符用于该参数时，它将返回被传递的对象的真正类型。

```
// Use a reference with typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class Mammal {
public:
  virtual bool lays_eggs() { return false; } // Mammal is polymorphic
  // ...
};

class Cat: public Mammal {
public:
  // ...
};

class Platypus: public Mammal {
public:
  bool lays_eggs() { return true; }
  // ...
};

  // Demonstrate typeid with a reference parameter.
void WhatMammal(Mammal &ob)
{
  cout << "ob is referencing an object of type ";
  cout << typeid(ob).name() << endl;
}

int main()
{
  Mammal AnyMammal;
  Cat cat;
  Platypus platypus;

  WhatMammal(AnyMammal);
  WhatMammal(cat);
```

```
  WhatMammal(platypus);

  return 0;
}
```

上面程序的输出如下所示：

```
ob is referencing an object of type class Mammal
ob is referencing an object of type class Cat
ob is referencing an object of type class Platypus
```

typeid 的另一种形式将类型名作为参数，该函数形式如下所示：

typeid(*type-name*)

例如，下面的语句是完全可以接受的：

```
cout << typeid(int).name();
```

这种形式的typeid主要用于获取一个可描述指定类型的type_info对象，因此它可以用在一个类型比较语句中。例如，这种形式的 WhatMammal( )可以报告说猫和水不一样：

```
void WhatMammal(Mammal &ob)
{
  cout << "ob is referencing an object of type ";
  cout << typeid(ob).name() << endl;
  if(typeid(ob) == typeid(Cat))
    cout << "Cats don't like water.\n";
}
```

一个简单的 RTTI 的应用程序

下面的程序对 RTTI 的能力给出了示意性说明。在这个程序中，叫做 factory( )的函数创建由 Mammal 派生的各种对象类型的实例（产生对象的函数有时称为对象工厂）。被创建的特定的对象类型通过调用 rand( )(C++ 的随机数生成函数)的结果确定，因此无法提前知道将生成何种对象类型。该程序将创建10个对象并统计每一种Mammal类型的个数。由于所有 Mammal 的类型都可以通过调用 factory( )生成，所以该程序依赖 typeid 确定哪一种对象类型被真正生成。

```
// Demonstrating run-time type id.
#include <iostream>
using namespace std;

class Mammal {
public:
  virtual bool lays_eggs() { return false; } // Mammal is polymorphic
  // ...
};

class Cat: public Mammal {
public:
  // ...
};

class Platypus: public Mammal {
public:
  bool lays_eggs() { return true; }
```

```
  // ...
};

class Dog: public Mammal {
public:
  // ...
};

// A factory for objects derived from Mammal.
Mammal *factory()
{
  switch(rand() % 3 ) {
    case 0: return new Dog;
    case 1: return new Cat;
    case 2: return new Platypus;
  }
  return 0;
}

int main()
{
  Mammal *ptr; // pointer to base class
  int i;
  int c=0, d=0, p=0;

  // generate and count objects
  for(i=0; i<10; i++) {
    ptr = factory(); // generate an object

    cout << "Object is " << typeid(*ptr).name();
    cout << endl;

    // count it
    if(typeid(*ptr) == typeid(Dog)) d++;
    if(typeid(*ptr) == typeid(Cat)) c++;
    if(typeid(*ptr) == typeid(Platypus)) p++;
  }

  cout << endl;
  cout << "Animals generated:\n";
  cout << " Dogs: " << d << endl;
  cout << " Cats: " << c << endl;
  cout << " Platypuses: " << p << endl;

  return 0;
}
```

样例输出如下所示：

```
Object is class Platypus
Object is class Platypus
Object is class Cat
Object is class Cat
Object is class Platypus
Object is class Cat
Object is class Dog
```

```
Object is class Dog
Object is class Cat
Object is class Platypus

Animals generated:
  Dogs: 2
  Cats: 4
  Platypuses: 4
```

22.1.1 将 typeid 用于模板类

typeid运算符可被用于模板类。从某种程度上说，作为一个模板类实例的对象类型是由对象被实例化时用做一般数据的数据所决定的，因此，用不同数据创建的两个相同模板类的实例具有不同的类型。下面是一个简单的例子：

```
// Using typeid with templates.
#include <iostream>
using namespace std;

template <class T> class myclass {
  T a;
public:
  myclass(T i) { a = i; }
  // ...
};

int main()
{
  myclass<int> o1(10), o2(9);
  myclass<double> o3(7.2);

  cout << "Type of o1 is ";
  cout << typeid(o1).name() << endl;

  cout << "Type of o2 is ";
  cout << typeid(o2).name() << endl;

  cout << "Type of o3 is ";
  cout << typeid(o3).name() << endl;

  cout << endl;

  if(typeid(o1) == typeid(o2))
    cout << "o1 and o2 are the same type\n";

  if(typeid(o1) == typeid(o3))
    cout << "Error\n";
  else
    cout << "o1 and o3 are different types\n";

  return 0;
}
```

上面这个程序的输出如下所示：

```
Type of o1 is class myclass<int>
Type of o2 is class myclass<int>
```

```
Type of o3 is class myclass<double>

o1 and o2 are the same type
o1 and o3 are different types
```

正如我们看到的那样，即使两个对象具有相同的模板类类型，如果它们的参数化数据不匹配，其类型也不相同。在上面的程序中，o1的类型是myclass<int>，o3的类型是myclass<double>，可见，它们的类型并不相同。

并不是所有的程序都要用到运行时类型标识，然而，当处理多态类型时，它可以使你了解在任何特定情况下所操作的对象类型。

22.2 强制转换运算符

C++定义了5个强制转换运算符。第一个是从C继承而来的传统形式的强制转换运算符，其他4个是几年前才添加的运算符，它们是dynamic_cast，const_cast，reinterpret_cast和static_cast。这些运算符对强制转换的方式提供了进一步的控制。

22.3 dynamic_cast

在新添加的强制转换运算符中，最重要的一个或许就是dynamic_cast。dynamic_cast执行一个运行时转换，该转换验证一个强制转换的有效性。如果该强制转换在执行dynamic_cast时是无效的，则这个转换失败。dynamic_cast的一般形式如下所示：

dynamic_cast<*target-type*> (*expr*)

其中，target-type指定强制转换的目标类型，expr是被强制转换为新类型的表达式。目标类型必须是一个指针或者是一个引用类型，被强制转换的表达式的值必须是一个指针或引用。因此，dynamic_cast可以被用来将一种指针类型强制转换为另一种指针类型，或者是将一种引用类型强制转换为另一种引用类型。dynamic_cast的目的是执行多态类型的强制转换。例如，假定有两个多态类分别为B和D，其中D从B中派生而来，dynamic_cast总是可以把指针D*强制转换为指针B*。这是因为基类指针总是可以指向它的派生对象，而只有当基类指针所指的对象真是一个D类对象时，dynamic_cast才可以把指针B*强制转换为指针D*。一般来说，如果被强制转换的指针（或引用）指向目标类型的对象或指向从目标类型派生的对象，dynamic_cast将会成功执行。否则，强制转换操作将会失败。如果转换失败，若该转换涉及到指针，那么dynamic_cast的值将为空；如果一个涉及到引用类型的dynamic_cast以失败告终，则会抛出一个bad_cast异常。

下面的程序是一个简单的例子（假设Base是一个多态类，Derived是Base的派生类）：

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // base pointer points to Derived object
dp = dynamic_cast<Derived *> (bp); // cast to derived pointer OK
if(dp) cout << "Cast OK";
```

其中，因为bp是真正指向一个Derived对象的指针，所以从基类指针bp到派生指针dp的强制转换可以成功执行。因此，该程序片段将显示Cast OK。但是在下面的程序片段中，因为

bp是指向一个Base对象的指针，而且将基类对象强制转换为其派生对象是非法的，所以该转换操作失败。

```
bp = &b_ob; // base pointer points to Base object
dp = dynamic_cast<Derived *> (bp); // error
if(!dp) cout << "Cast Fails";
```

因为这个强制转换以失败告终，所以该程序片段将显示Cast Fails。下面的程序演示了dynamic_cast可以处理的各种情况。

```
// Demonstrate dynamic_cast.
#include <iostream>
using namespace std;

class Base {
public:
  virtual void f() { cout << "Inside Base\n"; }
  // ...
};

class Derived : public Base {
public:
  void f() { cout << "Inside Derived\n"; }
};

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  dp = dynamic_cast<Derived *> (&d_ob);
  if(dp) {
    cout << "Cast from Derived * to Derived * OK.\n";
    dp->f();
  } else
    cout << "Error\n";

  cout << endl;

  bp = dynamic_cast<Base *> (&d_ob);
  if(bp) {
    cout << "Cast from Derived * to Base * OK.\n";
    bp->f();
  } else
    cout << "Error\n";

cout << endl;

bp = dynamic_cast<Base *> (&b_ob);
if(bp) {
  cout << "Cast from Base * to Base * OK.\n";
  bp->f();
} else
  cout << "Error\n";

  cout << endl;
```

```
  dp = dynamic_cast<Derived *> (&b_ob);
  if(dp)
    cout << "Error\n";
  else
    cout << "Cast from Base * to Derived * not OK.\n";

  cout << endl;

  bp = &d_ob; // bp points to Derived object
  dp = dynamic_cast<Derived *> (bp);
  if(dp) {
    cout << "Casting bp to a Derived * OK\n" <<
      "because bp is really pointing\n" <<
      "to a Derived object.\n";
    dp->f();
  } else
    cout << "Error\n";

  cout << endl;
  bp = &b_ob; // bp points to Base object
  dp = dynamic_cast<Derived *> (bp);
  if(dp)
    cout << "Error";
  else {
    cout << "Now casting bp to a Derived *\n" <<
      "is not OK because bp is really \n" <<
      "pointing to a Base object.\n";
  }

  cout << endl;

  dp = &d_ob; // dp points to Derived object
  bp = dynamic_cast<Base *> (dp);
  if(bp) {
    cout << "Casting dp to a Base * is OK.\n";
    bp->f();
  } else
    cout << "Error\n";

  return 0;
}
```

上面这个程序的输出如下所示：

```
Cast from Derived * to Derived * OK.
Inside Derived

Cast from Derived * to Base * OK.
Inside Derived

Cast from Base * to Base * OK.
Inside Base

Cast from Base * to Derived * not OK.
Casting bp to a Derived * OK
because bp is really pointing
```

```
to a Derived object.
Inside Derived

Now casting bp to a Derived *
is not OK because bp is really
pointing to a Base object.

Casting dp to a Base * is OK.
Inside Derived
```

22.3.1 用dynamic_cast替代typeid

在某些情况下，可以用dynamic_cast运算符代替typeid。例如，我们再次假设Base是Derived的多态基类。当且仅当某对象是一个真正的Derived对象时，下面的程序片段将把bp所指的对象的地址赋给dp。

```
Base *bp;
Derived *dp;
// ...
if(typeid(*bp) == typeid(Derived)) dp = (Derived *) bp;
```

在这种情况下，传统形式的强制转换被用于实际的转换操作。这样做很安全，因为if语句在真正进行强制转换之前先利用typeid检查该转换的合法性。然而，还有一种更好的方法也可以达到这个目的，这就是用dynamic_cast取代typeid运算符和if语句。

```
dp = dynamic_cast<Derived *> (bp);
```

因为只有当被转换的对象是一个目标类型的对象或者是一个从目标类型派生出来的对象时dynamic_cast才能成功执行，所以在这个语句执行之后，dp要么是一个空指针，要么是一个指向Derived类型的对象的指针。因为只有当强制转换操作合法时，dynamic_cast才能成功执行，所以可以在某些情况下简化逻辑操作。下面的程序说明了怎样用dynamic_cast取代typeid，该程序将同样的操作执行了两次——先利用typeid，然后利用dynamic_cast。

```
// Use dynamic_cast to replace typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
public:
  virtual void f() {}
};

class Derived : public Base {
public:
  void derivedOnly() {
    cout << "Is a Derived Object.\n";
  }
};

int main()
{
  Base *bp, b_ob;
```

```
  Derived *dp, d_ob;

  // ************************************
  // use typeid
  // ************************************
  bp = &b_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else
    cout << "Cast from Base to Derived failed.\n";

  bp = &d_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else
    cout << "Error, cast should work!\n";

  // ************************************
  // use dynamic_cast
  // ************************************
  bp = &b_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
    cout << "Cast from Base to Derived failed.\n";
  bp = &d_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
    cout << "Error, cast should work!\n";

  return 0;
}
```

就像你看到的那样，利用dynamic_cast简化了把一个基类指针强制转换为一个派生指针所需要的逻辑操作。该程序的输出如下所示：

```
Cast from Base to Derived failed.
Is a Derived Object.
Cast from Base to Derived failed.
Is a Derived Object.
```

22.3.2 将 dynamic_cast 与模板类一同使用

dynamic_cast 运算符还可以与模板类一同使用，例如：

```
// Demonstrate dynamic_cast on template classes.
#include <iostream>
using namespace std;
```

```
template <class T> class Num {
protected:
  T val;
public:
  Num(T x) { val = x; }
  virtual T getval() { return val; }
  // ...
};

template <class T> class SqrNum : public Num<T> {
public:
  SqrNum(T x) : Num<T>(x) { }
  T getval() { return val * val; }
};

int main()
{
  Num<int> *bp, numInt_ob(2);
  SqrNum<int> *dp, sqrInt_ob(3);
  Num<double> numDouble_ob(3.3);

  bp = dynamic_cast<Num<int> *> (&sqrInt_ob);
  if(bp) {
    cout << "Cast from SqrNum<int>* to Num<int>* OK.\n";
    cout << "Value is " << bp->getval() << endl;
  } else
    cout << "Error\n";

  cout << endl;

  dp = dynamic_cast<SqrNum<int> *> (&numInt_ob);
  if(dp)
    cout << "Error\n";
  else {
    cout << "Cast from Num<int>* to SqrNum<int>* not OK.\n";
    cout << "Can't cast a pointer to a base object into\n";
    cout << "a pointer to a derived object.\n";
  }
  cout << endl;

  bp = dynamic_cast<Num<int> *> (&numDouble_ob);
  if(bp)
    cout << "Error\n";
  else
    cout << "Can't cast from Num<double>* to Num<int>*.\n";
    cout << "These are two different types.\n";

    return 0;
}
```

上面这个程序的输出如下所示：

```
Cast from SqrNum<int>* to Num<int>* OK.
Value is 9

Cast from Num<int>* to SqrNum<int>* not OK.
```

```
Can't cast a pointer to a base object into
a pointer to a derived object.

Can't cast from Num<double>* to Num<int>*.
These are two different types.
```

这个例子说明了一个关键问题，即利用dynamic_cast把一个指向模板实例类型的指针强制转换为指向另一个实例类型的指针是不可能的。记住，一个模板类对象的精确类型将由创建该模板实例的数据类型决定，因此，Num<double>和Num<int>是两种不同的类型。

22.3.3 const_cast

const_cast运算符用于在一个强制转换操作中显式地重载const和/或volatile。其目标类型必须与源类型相同（const或volatile属性的改变除外）。const_cast最常见的用途是删除const属性。const_cast的一般形式如下所示：

const_cast<*type*> (*expr*)

其中，type指定强制转换的目标类型，expr是将被转换为新类型的表达式。

下面的程序演示了const_cast。

```
// Demonstrate const_cast.
#include <iostream>
using namespace std;

void sqrval(const int *val)
{
  int *p;

  // cast away const-ness.
  p = const_cast<int *> (val);

  *p = *val * *val; // now, modify object through v
}

int main()
{
  int x = 10;

  cout << "x before call: " << x << endl;
  sqrval(&x);
  cout << "x after call: " << x << endl;

  return 0;
}
```

这个程序的输出如下所示：

```
x before call: 10
x after call: 100
```

可以看到，即使sqrval()的参数被指定为一个const指针，x仍然被sqrval()所修改。

const_cast还可以用于从一个const引用中抛掉其const属性。例如，下面是对前面程序重写之后的程序，其中要平方的值被作为一个const引用传递。

```
// Use const_cast on a const reference.
```

```
#include <iostream>
using namespace std;

void sqrval(const int &val)
{
  // cast away const on val
  const_cast<int &> (val) = val * val;
}

int main()
{
  int x = 10;

  cout << "x before call: " << x << endl;
  sqrval(x);
  cout << "x after call: " << x << endl;

  return 0;
}
```

这个程序与前面的程序产生的输出相同。它之所以工作，只是因为const_cast临时从val中删除了const属性，从而使其被用于给调用参数（这里是x）分配一个新值。

必须强调的是：使用const_cast删除const属性（const-ness）是一种具有潜在危险的功能，所以使用时要格外小心。

另外，只有const_cast可以丢弃const属性。也就是说，无论是dynamic_cast，static_cast还是reinterpret_cast，它们都不能改变一个对象的const属性。

22.3.4 static_cast

static_cast运算符执行非多态转换，任何标准转换都可以使用它，并且不执行运行时检查。它的一般形式是：

static_cast<*type*> (*expr*)

其中，type指定转换的目标类型，expr是将被转换为新类型的表达式。

static_cast运算符本质上是初始强制转换运算符的替代品，它简单地执行一个非多态转换。例如，下面把一个int值强制转换。

```
// Use static_cast.
#include <iostream>
using namespace std;

int main()
{
  int i;

  for(i=0; i<10; i++)
    cout << static_cast<double> (i) / 3 << " ";

  return 0;
}
```

22.3.5 reinterpret_cast

reinterpret_cast运算符把一种类型转换成一个不同的类型。例如，它可以把一个指针改为一个整数，一个整数改为一个指针。我们也可以使用它来强制转换不兼容的指针类型。它的一般形式是：

reinterpret_cast<*type*> (*expr*)

其中，type 指定强制转换的目标类型，expr 是将被转换为新类型的表达式。

下面的程序演示了 reinterpret_cast 的使用情况：

```
// An example that uses reinterpret_cast.
#include <iostream>
using namespace std;

int main()
{
  int i;
  char *p = "This is a string";

  i = reinterpret_cast<int> (p); // cast pointer to integer

  cout << i;

  return 0;
}
```

其中，reinterpret_cast 把指针 p 转换为整型。这个转换代表一种基本类型转变和 reinterpret_cast 的一种有效的使用方法。