Introduction to RTTI

Jonathan Hoyle

Eastman Kodak

11/30/00

Overview

- What is RTTI?
- typeid( ) function
- Polymorphism
- dynamic_cast< > operation
- RTTI Gotcha’s
- Demo

What is RTTI?

- Run-Time Type Identification
- Dynamically determining an object’s type
- Two RTTI operations:
  - typeid( ) function *(generic types)*
  - dynamic_cast< > *(polymorphic types)*
- Useful for type specialization in code
- Very useful for templates
- Good debugging tool

typeid( ) function

- Returns a `type_info` describing that type
- `typeid()` can be used on any variable or type
- `name()` returns the type name as a string
- `type_info`'s can be compared using the `==` and `!=` operators
- It is "polymorphic-friendly"
- Must #include the following header:

```
        #include <typeinfo.h>
```

typeid( ) Example #1

```
// Prints the type name to screen
template <class T>
void WhatAmI(T x)
{ cout << typeid(x).name() << endl; }

WhatAmI(1);         // prints “int”
WhatAmI(1.0);       // prints “double”
WhatAmI(“Hi!”);     // prints “char *”

MyClass     x;
WhatAmI(x);         // “MyClass” (CW & Borland)
                    // or “class MyClass” (VC++)
```

typeid( ) Example #2

```
// Test if a given object is a basic numeric type
template <class T>
bool IsNumericType(T x)
{
  if (typeid(x) == typeid(short))   return true;
  if (typeid(x) == typeid(long))    return true;
  if (typeid(x) == typeid(int))     return true;
  if (typeid(x) == typeid(double))  return true;
  ...

  return false;
}
```

Polymorphism

- A class which declares or inherits a virtual function is called a *polymorphic class*:

```
class Av
{
    public:
          virtual void foo();
};

class Bv : public Av
{
    ...
};
```

Polymorphism

- Polymorphic classes call the “intended” virtual function despite variable type:

```
class A             {... void foo(); ...};
class B  : public A {... void foo(); ...};

class Av            {... virtual void foo(); ...};
class Bv : public Av {... virtual void foo(); ...};


A  *aPtr  = new B;     // non-polymorphic class
Av *avPtr = new Bv;    // polymorphic class

aPtr->foo();           // calls A::foo()
avPtr->foo();          // calls Bv::foo()
```

typeid( ) respects polymorphism

```
class A              {... void foo(); ...};
class B  : public A  {... void foo(); ...};

class Av             {... virtual void foo(); ...};
class Bv : public Av {... virtual void foo(); ...};

A  *aPtr  = new B;      // non-polymorphic class
Av *avPtr = new Bv;     // polymorphic class

cout << typeid(*aPtr).name(); // prints “A”
cout << typeid(*avPtr).name();// prints “Bv”

cout << typeid(aPtr).name();  // prints “A *”
cout << typeid(avPtr).name(); // prints “Av *”
```

Polymorphic RTTI Example

```
class ZCommunications                            { ... };
class ZParallel : public ZCommunications  { ... };
class ZSCSI     : public ZCommunications  { ... };
class ZFirewire : public ZCommunications  { ... };

void InitCommunications(ZCommunications &inComm)
{
  if (typeid(inComm) == typeid(ZParallel))
      InitParallelCommunications(inComm);

  if (typeid(inComm) == typeid(ZSCSI))
      InitSCSICommunications(inComm);

  if (typeid(inComm) == typeid(ZFirewire))
      InitFirewireCommunications(inComm);
}
```

typeid( ) Tips

- You cannot determine the "real" type of an object pointed to by a `void *`.
- For *non-polymorphic* typed variables, `typeid()` gives info on the variable type.
- For *polymorphic* typed variables, `typeid()` gives info on the underlying "real" type.
- You cannot determine the name of an object's base class.

dynamic_cast< >

- dynamic_cast< > is used to cast one polymorphic type to another type within its inheritance chain
- dynamic_cast< > performs a safe "down cast".
- dynamic_cast< > operations must be used on polymorphic *pointers* or *references* only.

dynamic_cast< > syntax

- dynamic_cast< > operation on polymorphic *pointers* and *references*:

```
// pointer cast
T  *ptr1 = (T *) ptr2;
T  *ptr1 = dynamic_cast<T *>(ptr2);

// reference cast
T  object1 = (T) object2;
T  object1 = dynamic_cast<T &>(object2);

// compiler error
T  object1 = dynamic_cast<T>(object2);
```

dynamic_cast< > with pointers

- An incompatible pointer cast returns NULL:

```
Av  *aPtr = new Av;
Bv  *bPtr = new Bv;
foo(aPtr, bPtr);
...

void foo(Av *aPtr1, Av *aPtr2)
{
    // bPtr1 set to NULL
    Bv  *bPtr1 = dynamic_cast<Bv *>(aPtr1);

    // bPtr2 set to a valid Bv pointer
    Bv  *bPtr2 = dynamic_cast<Bv *>(aPtr2);
}
```

dynamic_cast< > with references

- An incompatible reference cast throws a `bad_cast` exception:

```
Av  aObject;
Bv  bObject;
foo(aObject, bObject);

void foo(Av &aObject1, Av &aObject2)
{
    // bObject1 throws a bad_cast exception
    Bv  bObject1 = dynamic_cast<Bv &>(aObject1);

    // bObject2 set to a valid Bv object
    Bv  bObject2 = dynamic_cast<Bv &>(aObject2);
}
```

dynamic_cast< > Example

```
class ZCommunications                          { ... };
class ZParallel : public ZCommunications  { ... };
class ZSCSI     : public ZCommunications  { ... };
class ZFirewire : public ZCommunications  { ... };

void InitCommunications(ZCommunications *inPtr)
{
  ZParallel *pPtr = dynamic_cast<ZParallel *>(inPtr);
  if (pPtr) InitParallelCommunications(inPtr);

  ZSCSI     *sPtr = dynamic_cast<ZSCSI *>(inPtr);
  if (sPtr) InitSCSICommunications(inPtr);

  ZFirewire *fPtr = dynamic_cast<ZFirewire *>(inPtr);
  if (fPtr) InitFirewireCommunications(inPtr);
}
```

dynamic_cast< > vs. typeid( )

```
class ZCommunications                           { ... };
class ZSCSI     : public ZCommunications  { ... };
class ZSCSI_TSP : public ZSCSI                  { ... };

void InitCommunications(ZCommunications &inComm)
{ // will not work if inComm is a ZSCSI_TSP
  if (typeid(inComm) == typeid(ZSCSI))
      InitSCSICommunications(inComm);
}

void InitCommunications(ZCommunications *inPtr)
{ // will work if inComm is a ZSCSI_TSP *
  ZSCSI    *sPtr = dynamic_cast<ZSCSI *>(inPtr);
  if (sPtr) InitSCSICommunications(inPtr);
}
```

dynamic_cast< > Tips

- If you use `dynamic_cast<>` in the wrong place, you will get a compiler error.
- You cannot `dynamic_cast<>` a `void *`.
- You cannot `dynamic_cast<>` any non-polymorphic type.
- If you have a non-polymorphic class heierarchy, use `static_cast<>`.

RTTI Gotcha's

- For `typeid()`, always `#include <typeinfo.h>`.
- The string returned in `typeid(`*object*`).name()` may differ slightly from compiler to compiler.
- Use `dynamic_cast<>` for polymorphic classes
- `typeid()` resolves polymorphic types for *objects* only, not *pointers*.
- `dynamic_cast<>`-ing works for *pointers* and *references* to polymorphic types, not object types
- Can't be used with Visual C++ v1.5.2
- RTTI is defaulted off on Visual C++ 6

Demo