Chapter 22

Run-Time Type ID and the Casting Operators

Standard C++ contains two features that help support modern, object-oriented programming: run-time type identification (RTTI for short) and a set of four additional casting operators. Neither of these were part of the original specification for C++, but both were added to provide enhanced support for run-time polymorphism. RTTI allows you to identify the type of an object during the execution of your program. The casting operators give you safer, more controlled ways to cast. Since one of the casting operators, **dynamic_cast**, relates directly to RTTI, it makes sense to discuss them in the same chapter.

Run-Time Type Identification (RTTI)

Run-time type information may be new to you because it is not found in nonpolymorphic languages, such as C. In nonpolymorphic languages there is no need for run-time type information because the type of each object is known at compile time (i.e., when the program is written). However, in polymorphic languages such as C++, there can be situations in which the type of an object is unknown at compile time because the precise nature of that object is not determined until the program is executed. As explained in Chapter 17, C++ implements polymorphism through the use of class hierarchies, virtual functions, and base-class pointers. Since base-class pointers may be used to point to objects of the base class or *any object derived from that base*, it is not always possible to know in advance what type of object will be pointed to by a base pointer at any given moment in time. This determination must be made at run time, using run-time type identification.

To obtain an object's type, use **typeid**. You must include the header **<typeinfo>** in order to use **typeid**. Its most commonly used form is shown here:

typeid(*object*)

Here, *object* is the object whose type you will be obtaining. It may be of any type, including the built-in types and class types that you create. **typeid** returns a reference to an object of type **type_info** that describes the type of *object*.

The **type_info** class defines the following public members:

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

The overloaded == and != provide for the comparison of types. The **before( )** function returns true if the invoking object is before the object used as a parameter in collation order. (This function is mostly for internal use only. Its return value has nothing to do with inheritance or class hierarchies.) The **name( )** function returns a pointer to the name of the type.

Here is a simple example that uses **typeid**.

```
// 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;
}
```

The output produced by this program is shown here:

```
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
```

The most important use of **typeid** occurs when it is applied through a pointer of a polymorphic base class. In this case, it will automatically return the type of the actual object being pointed to, which may be a base-class object or an object derived from that base. (Remember, a base-class pointer can point to objects of the base class or of any class derived from that base.) Thus, using **typeid**, you can determine at run time the type of the object that is being pointed to by a base-class pointer. The following program demonstrates this principle.

```
// 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;
}
```

The output produced by this program is shown here:

```
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
```

As explained, when **typeid** is applied to a base-class pointer of a polymorphic type, the type of object pointed to will be determined at run time, as shown by the output produced by the program.

In all cases, when **typeid** is applied to a pointer of a nonpolymorphic class hierarchy, then the base type of the pointer is obtained. That is, no determination of what that pointer is actually pointing to is made. For example, comment out the **virtual** keyword before the function **lays_eggs( )** in **Mammal** and then compile and run the program. You will see the following output.

```
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
```

Since **Mammal** is no longer a polymorphic class, the type of each object will be **Mammal** because that is the type of the pointer.

Since **typeid** is commonly applied to a dereferenced pointer (i.e., one to which the * operator has been applied), a special exception has been created to handle the situation in which the pointer being dereferenced is null. In this case, **typeid** throws **bad_typeid**.

References to an object of a polymorphic class hierarchy work the same as pointers. When **typeid** is applied to a reference to an object of a polymorphic class, it will return the type of the object actually being referred to, which may be of a derived type. The circumstance where you will most often make use of this feature is when objects are passed to functions by reference. For example, in the following program, the function **WhatMammal( )** declares a reference parameter to objects of type **Mammal**. This means that **WhatMammal( )** can be passed references to objects of type **Mammal** or any class derived from **Mammal**. When the **typeid** operator is applied to this parameter, it returns the actual type of the object being passed.

```
// 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;
}
```

The output produced by this program is shown here:

```
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
```

There is a second form of **typeid** that takes a type name as its argument. This form is shown here:

typeid(*type-name*)

For example, the following statement is perfectly acceptable:

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

The main use of this form of **typeid** is to obtain a **type_info** object that describes the specified type so that it can be used in a type comparison statement. For example, this form of **WhatMammal( )** reports that cats don't like water:

```
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";
}
```

A Simple Application of Run-Time Type ID

The following program hints at the power of RTTI. In the program, the function called **factory( )** creates instances of various types of objects derived from the class **Mammal**. (A function that produces objects is sometimes called an *object factory*.) The specific type of object created is determined by the outcome of a call to **rand( )**, C++'s random number generator. Thus, there is no way to know in advance what type of object will be generated. The program creates 10 objects and counts the number of each type of mammal. Since any type of mammal may be generated by a call to **factory( )**, the program relies upon **typeid** to determine which type of object has actually been made.

```
// 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;
}
```

Sample output is shown here.

```
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
```

typeid Can Be Applied to Template Classes

The **typeid** operator can be applied to template classes. The type of an object that is an instance of a template class is in part determined by what data is used for its generic data when the object is instantiated. Two instances of the same template class that are created using different data are therefore different types. Here is a simple example:

```
// 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;
}
```

The output produced by this program is shown here.

```
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
```

As you can see, even though two objects are of the same template class type, if their parameterized data does not match, they are not equivalent types. In the program, **o1** is of type **myclass<int>** and **o3** is of type **myclass<double>**. Thus, they are of different types.

Run-time type identification is not something that every program will use. However, when you are working with polymorphic types, it allows you to know what type of object is being operated upon in any given situation.

The Casting Operators

C++ defines five casting operators. The first is the traditional-style cast inherited from C. The remaining four were added a few years ago. They are **dynamic_cast**, **const_cast**, **reinterpret_cast**, and **static_cast**. These operators give you additional control over how casting takes place.

dynamic_cast

Perhaps the most important of the new casting operators is **dynamic_cast**. The **dynamic_cast** performs a run-time cast that verifies the validity of a cast. If the cast is invalid at the time **dynamic_cast** is executed, then the cast fails. The general form of **dynamic_cast** is shown here:

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

Here, *target-type* specifies the target type of the cast, and *expr* is the expression being cast into the new type. The target type must be a pointer or reference type, and the expression being cast must evaluate to a pointer or reference. Thus, **dynamic_cast** may be used to cast one type of pointer into another or one type of reference into another.

The purpose of **dynamic_cast** is to perform casts on polymorphic types. For example, given two polymorphic classes B and D, with D derived from B, a **dynamic_cast** can always cast a D* pointer into a B* pointer. This is because a base pointer can always point to a derived object. But a **dynamic_cast** can cast a B* pointer into a D* pointer only if the object being pointed to *actually is* a D object. In general, **dynamic_cast** will succeed if the pointer (or reference) being cast is a pointer (or reference) to either an object of the target type or an object derived from the target type. Otherwise, the cast will fail.If the cast fails, then **dynamic_cast** evaluates to null if the cast involves pointers. If a **dynamic_cast** on reference types fails, a **bad_cast** exception is thrown.

Here is a simple example. Assume that **Base** is a polymorphic class and that **Derived** is derived from **Base**.

```
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";
```

Here, the cast from the base pointer **bp** to the derived pointer **dp** works because **bp** is actually pointing to a **Derived** object. Thus, this fragment displays **Cast OK**. But in the next fragment, the cast fails because **bp** is pointing to a **Base** object and it is illegal to cast a base object into a derived object.

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

Because the cast fails, this fragment displays **Cast Fails**.

The following program demonstrates the various situations that **dynamic_cast** can handle.

```
// 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;
}
```

The program produces the following output:

```
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
```

Replacing typeid with dynamic_cast

The **dynamic_cast** operator can sometimes be used instead of **typeid** in certain cases. For example, again assume that **Base** is a polymorphic base class for **Derived**. The following fragment will assign **dp** the address of the object pointed to by **bp** if and only if the object really is a **Derived** object.

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

In this case, a traditional-style cast is used to actually perform the cast. This is safe because the **if** statement checks the legality of the cast using **typeid** before the cast actually occurs. However, a better way to accomplish this is to replace the **typeid** operators and **if** statement with this **dynamic_cast**.

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

Since **dynamic_cast** succeeds only if the object being cast is either an object of the target type or an object derived from the target type, after this statement executes **dp** will contain either a null or a pointer to an object of type **Derived**. Since **dynamic_cast** succeeds only if the cast is legal, it can simplify the logic in certain situations. The following program illustrates how a **dynamic_cast** can be used to replace **typeid**. It performs the same set of operations twice—first with **typeid**, then using **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;
}
```

As you can see, the use of **dynamic_cast** simplifies the logic required to cast a base pointer into a derived pointer. The output from the program is shown here:

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

Using dynamic_cast with Template Classes

The **dynamic_cast** operator can also be used with template classes. For example,

```
// 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;
}
```

The output from this program is shown here:

```
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.
```

A key point illustrated by this example is that it is not possible to use **dynamic_cast** to cast a pointer to one type of template instantiation into a pointer to another type of instance. Remember, the precise type of an object of a template class is determined by the type of data used to create an instance of the template. Thus, **Num<double>** and **Num<int>** are two different types.

const_cast

The **const_cast** operator is used to explicitly override **const** and/or **volatile** in a cast. The target type must be the same as the source type except for the alteration of its **const** or **volatile** attributes. The most common use of **const_cast** is to remove **const**-ness. The general form of **const_cast** is shown here.

const_cast<*type*> (*expr*)

Here, *type* specifies the target type of the cast, and *expr* is the expression being cast into the new type.

The following program demonstrates **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;
}
```

The output produced by this program is shown here:

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

As you can see, **x** was modified by **sqrval( )** even though the parameter to **sqrval( )** was specified as a **const** pointer.

const_cast can also be used to cast away **const**-ness from a **const** reference. For example, here is the preceding program reworked so that the value being squared is passed as a **const** reference.

```
// 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;
}
```

This program produces the same output as before. Again, it works only because the **const_cast** temporarily removes the **const** attribute from **val**, allowing it to be used to assign a new value to the calling argument (in this case, **x**).

It must be stressed that the use of **const_cast** to cast way **const**-ness is a potentially dangerous feature. Use it with care.

One other point: Only **const_cast** can cast away **const**-ness. That is, neither **dynamic_cast, static_cast** nor **reinterpret_cast** can alter the **const**-ness of an object.

static_cast

The **static_cast** operator performs a nonpolymorphic cast. It can be used for any standard conversion. No run-time checks are performed. Its general form is

static_cast<*type*> (*expr*)

Here, *type* specifies the target type of the cast, and *expr* is the expression being cast into the new type.

The **static_cast** operator is essentially a substitute for the original cast operator. It simply performs a nonpolymorphic cast. For example, the following casts an **int** value into a **double**.

```
// 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;
}
```

reinterpret_cast

The **reinterpret_cast** operator converts one type into a fundamentally different type. For example, it can change a pointer into an integer and an integer into a pointer. It can also be used for casting inherently incompatible pointer types. Its general form is

reinterpret_cast<*type*> (*expr*)

Here, *type* specifies the target type of the cast, and *expr* is the expression being cast into the new type.

The following program demonstrates the use of **reinterpret_cast**:

```
// 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;
}
```

Here, **reinterpret_cast** converts the pointer **p** into an integer. This conversion represents a fundamental type change and is a good use of **reinterpret_cast**.