APPENDIX I

C++ Casts and Run-Time Type Identification

Introduction

There are many times when a programmer needs to use type casts to tell the compiler to convert the type of an expression to another type. For example, suppose that you have the sum of `n` integer values in an integer variable `sum`, and would like to compute the average and store it in a variable `average` of type `float`. In this case, you must tell the compiler to first convert `sum` to `double` before performing the division by using a type cast:

```
double average;
average = (double)sum / n;
```

This type of casting is called the *C- style* of casting. Alternatively, you can use the newer C++ *functional style* of casting:

```
double average;
average = double(sum) / n;
```

The functional style of casting will work for primitive built-in types such as `char`, `float`, `int` and `double`, as well as for user-defined class types that provide convert constructors. The effect of functional style casting is in most cases the same as that of C-style casting.

Another example of casting occurs frequently in reading from an I/O device (perhaps a modem, a network connection, or a disk) from which data arrives as a stream of bytes. In this case, to store the data in a data item of some type, say a structure

```
struct PersonData
{
   char name[20];
   int age;
};
PersonData data;
```

the address of data must be treated as the address of a location where a sequence of bytes will be stored. In other words, a pointer to data must be treated as a pointer to a byte. Since C++ treats bytes as if they were characters, this just means that a pointer to the type PersonData must be regarded as if it were a pointer to char. Again, this is accomplished through the use of a type cast:

```
char *pChar = (char *)&data;
```

This statement takes a pointer to the structure data, treats it as a pointer to char using a type cast, and uses the resulting value to initialize the newly created pointer variable pChar.

It is clear that these two examples of casting are different, not only in their purpose, but also in the way that they are implemented by the compiler. In converting an int to a double, the compiler must change the internal bit representation of an integer into the bit representation of a double, whereas in the second example, no transformation of the bit representation is performed. The designers of the C++ standard decided to introduce different casting notations to represent these two different kinds of casts as well as two other kinds of casts also used in the language. The designers introduced four different ways of casting:

1. static_cast
2. reinterpret_cast
3. const_cast
4. dynamic_cast

Using the new style casts clarifies the programmer's intent to the compiler and enables the compiler to catch casting errors.

In this appendix, we will discuss the four ANSI C++ style type casts and briefly touch on the related topic of using the typeid operator.

static_cast

The most commonly used type of cast is the static_cast, which is used to convert an expression to a value of a specified type. For example, to convert an expression expr1 to a value of type Type1 and assign it to a variable var1, you would write

```
type1 var1 = static_cast< type1 >(expr1);
```

in place of the traditional C-style cast

```
type1 var1 = (type1) expr1;
```

For example, to perform the conversion of int to double mentioned in the first example of the introductory section, you would write

```
double average = static_cast< double >(sum) / n;
```

A static_cast is used to convert an expression of one type to a second type when the compiler has information on how to perform the indicated conversion. For example, static_cast can be used to perform all kinds of conversions between char, int, long, float, and double because the knowledge for performing those types of conversions is already built into the compiler. A static_cast can also be used if information on how to perform the conversion has been provided by the programmer through type conversion operators. For example, consider the class

```
class FeetInches
{
   private:
    int feet;
    int inches;
   public:
    FeetInches(int f, int i)
    {
       feet = f;
       inches = i;
    }
    // type conversion operator
     operator double()
    {
       return feet + inches/12.0;
    }
};
```

This class provides information to the compiler on how to convert a `FeetInches` object to a `double`. As a result, the following `main` function will compile correctly and print 3.5 and 3 when it executes.

```
#include <iostream>
#include "FeetInchesEx.h"
using namespace std;

int main()
{
   FeetInches ftObject(3, 6);
   // use static_cast to convert to double
   double ft = static_cast< double >(ftObject);
   cout << ft;
   // use static_cast to convert to int
   cout << endl << static_cast< int >(ftObject);
   return 0;
}
```

Here we have assumed that the class declaration is stored in the indicated header file. The static cast to `double` succeeds because of the presence of the type conversion operator, while the cast to `int` succeeds because the compiler already knows how to convert `double` to `int`.

An example of an improper use of `static_cast` might be instructive. Assuming the same declaration of `FeetInches` as earlier, the following `main` function will be rejected by the compiler:

```
int main()
{
   FeetInches ftObject(3, 6);
   // illegal use of static_cast
   char *pInt = static_cast< int * >(&ftObject);
   cout << *pInt;
   return 0;
}
```

The program fails to compile because the compiler has been given no information on how to convert a pointer to a `FeetInches` object to a pointer to `int`.

Finally, a `static_cast` can be used to cast a pointer to a base class to a pointer to a derived class. We will see an example of this in the last section of this appendix.

reinterpret_cast

A `reinterpret_cast` is used to force the compiler to treat a value as if it were of a different type when the compiler knows of no way to perform the type conversion. A `reinterpret_cast` is mostly used with pointers, when a pointer to one type needs to be treated as if it were a pointer to a different type. In other words, `reinterpret_cast` is useful with the second kind of casting discussed in the introductory section. No change in the bit representation takes place: The value being cast is just used as is.

The notation for `reinterpret_cast` is similar to that for `static_cast`. To force expression `expr1` to be regarded as a value of type `type1`, we would write

```
type1 var1 = reinterpret_cast< type1 >(expr1);
```

instead of the old C-style cast. For example, if for some reason we needed to treat a `FeetInches` object as a pair of integers, we could set a pointer to `int` to point to the object, and then access the integer components of the object by dereferencing the pointer. The `reinterpret_cast` would be used to force the change of type in the pointer. The following `main` program would print 3 on one line and 6 on the next.

```
int main()
{
   FeetInches ftObject(3, 6);
   // point to beginning of object
   int *p = reinterpret_cast< int * >(&ftObject);
   cout << *p << endl;
   // advance the pointer by size of one int
   p++;
   cout << *p;
   return 0;
}
```

The compiler will reject the use of `reinterpret_cast` where there is already adequate information on how to perform the type conversion. In particular, the following statement generates a compiler error:

```
cout << reinterpret_cast< int >(ftObject);
```

Well-designed programs that do not work directly with hardware should have little need for this type of casting. Indeed, a need to use `reinterpret_cast` may well be an indication of a design error.

const_cast

This type of casting is only used with *pointers to constants,* and is used to treat a pointer to a constant as though it were a regular pointer. A pointer to a constant may not be used to change the memory location it points to. For example, we may define a pair of integer variables and a pointer to a constant `int` as follows:

```
int  k = 4;
int m = 20;
const int *pToC;
```

We may then make the pointer `pToC` point to different integer variables, as in

```
pToC = &k;
cout << *pToC; // prints 4
pToC = &m;
cout << *pToC; // prints 20
```

but we cannot use `pToC` to change whatever variable `pToC` points to. For example, the code

```
*pToC = 23;
```

is illegal. Moreover, you cannot assign the value of a pointer to a constant to another pointer that is not itself a pointer to a constant, because the constant might then be changed through the second pointer:

```
int *p1;       // not a pointer to constant
p1 = pToC;     // error!!
```

For the same reason, a pointer to a constant can only be passed to a function if the corresponding formal parameter is a pointer to a constant. Thus, with the function definition

```
void print(int *p)
{
   cout << *p;
}
```

the call

```
print(pToC);
```

would be illegal. Such a call, however, would be all right if the function were modified to take a parameter that is a pointer to a constant. Thus, in the presence of

```
void constPrint(const int *p)
{
   cout << *p;
}
```

the call

```
constPrint(pToC);
```

would be alowed.

We have purposely kept these examples simple. In real programs, the pointer to a constant might be returned by a member function of a class, and point to a member of the class that needs to be protected from modification. This pointer might need to be passed to a function such as `print` above which perhaps through poor planning was not written to take a pointer to a constant. In this case, the compiler can be persuaded to accept the call by "casting away" the "constness" of the pointer using a `const_cast`:

```
print( const_cast< int * >(pToC) );
```

Naturally, `const_cast` can also be used to allow assignment of a pointer to a constant to a regular pointer:

```
int *p = const_cast< int * >(pToC);
```

As in the case of `reinterpret_cast`, we note that the use of `const_cast` should not be necessary in most well-designed programs.

dynamic_cast

Polymorphic code is code that is capable of being invoked with objects belonging to different classes within the same inheritance hierarchy. Because objects of different classes have different storage requirements, polymorphic code does not use the objects directly. Instead, it accesses them through references or pointers. In this appendix, we will deal mainly with the access of polymorphic objects through pointers: access through references is similar.

Polymorphic code processes objects of different classes by treating them as belonging to the same base class. At times, however, it is necessary to determine at run time the specific derived class of the object, so that its methods can be invoked. Objects designed to be processed by polymorphic code carry type information within them to make run-time type identification possible. In C++, such objects must belong to a polymorphic class. A polymorphic class is a class with at least one virtual member function, or one that is derived from such a class.

In C++, a `dynamic_cast` is used to take a pointer (or reference) to an object of a polymorphic class, determine whether the object is of a specified *target class*, and if so, return that pointer cast as a pointer to the target class. If the object cannot be regarded as belonging to the target class, `dynamic_cast` returns the special pointer value `0`.

A typical use of `dynamic_cast` is as follows. Let `pExpr` be a pointer to an object of some derived class of a polymorphic class `PolyClass`, and let `DerivedClass` be one of several classes derived from `PolyClass`. We can determine whether the object pointed to by `pExpr` is a `DerivedClass` object by writing

```
DerivedClass *dP = dynamic_cast<DerivedClass *>(pExpr);
if (dP)
 {
    // the object *dP belongs to DerivedClass
    …
 }
else
 {
    // *dp does not belong to DerivedClass
    …
 }
```

Here `DerivedClass` is what we have called the specified target class.

As an example, consider a farm that keeps cows for milk as well as a number of dogs to guard the homestead. All the animals eat (have an `eat` member function), the cows give milk (have a `giveMilk` member function), and the dogs keep watch (have a `guardhouse` member function). We can describe all of these by using the following hierarchy of classes:

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

class DomesticAnimal
{
   public:
     virtual void eat()
      {
        cout << "Animal eating: Munch munch." << endl;
      }
};

class Cow:public DomesticAnimal
{
  public:
    void giveMilk()
    {
      cout << "Cow giving milk." << endl;
    }
};

class Dog:public DomesticAnimal
{
  public:
    void guardHouse()
    {
      cout << "Dog guarding house." << endl;
    }
};
```

Note that the `eat` member function has been declared as a virtual member function in order to make all the classes polymorphic.

Many applications work with collections, usually arrays of objects belonging to the same inheritance hierarchy. For example, our farm may have two cows and two dogs, which can be stored in an array as follows:

```
DomesticAnimal *a[4] = {new Dog, new Cow,
                        new Dog, new Cow
                       };
```

We have to use an array of pointers since an array of `DomesticAnimal` would not be able to hold `Dog` or `Cow` objects, either of which would normally require more storage than a `DomesticAnimal`. Now suppose that we wanted to go through the entire array of animals and milk all the cows. We couldn't just go through the array with a loop such as

```
for (int k = 0; k < 4; k++)
  a[k]->giveMilk();
```

for then we would cause run-time errors whenever `a[k]` points to a `Dog`. A `dynamic_cast` will take a pointer such as `a[k]`, look at the actual type of the object being pointed to, and return the address of the object if the object matches the target type of the `dynamic_cast`. If the object does not match the target type of the cast, then `0` is returned in place of the

address. As mentioned, the general format for casting an expression `expr1` to a target type `TargetType` is

```
dynamic_cast< TargetType >(expr1);
```

where `TargetType` must be a pointer or reference type. In our case, to determine if a domestic animal pointed to by `a[k]` is a cow we would write

```
Cow *pC = dynamic_cast< Cow * >(a[k]);
```

and then test `pC` to see if it was `0`. If it is `0`, we know the animal is not a cow and that it is useless to attempt to milk it; otherwise, we can milk the animal by invoking

```
a[k]->giveMilk();
```

Here is a `main` function that puts all of this together.

```
int main()
{
   DomesticAnimal *a[4] = {new Dog, new Cow,
                           new Dog, new Cow,
                          };
   for (int k = 0; k < 4; k++)
   {
     Cow *pC = dynamic_cast<Cow *>(a[k]);
     if (pC)
       {
         // pC not 0, so we found a cow
         pC->giveMilk();
       }
     else
      {
        cout << "This animal is not a cow." << endl;
      }
   }
   return 0;
}
```

When executed, the output will be

```
This animal is not a cow.
Cow giving milk.
This animal is not a cow.
Cow giving milk.
```

The `dynamic_cast` is so called because the type of the object of a polymorphic class cannot always be determined statically, that is, at compile time without running the program. For example, in the statement

```
a[k]->giveMilk();
```

it is impossible to determine at compile time whether `a[k]` points to a `Dog` or a `Cow` since it can point to objects of either type. In contrast, `static_cast` uses information available at compile time.

Run-Time Type Identification

As we have seen, `dynamic_cast` can be used to identify the class type of a polymorphic object within an inheritance hierarchy at run time. More generally, the `typeid` operator can be used to identify the type of any expression at run time. The `typeid` operator can be applied to both data and type expressions:

```
typeid(data_expression)
```

or

```
typeid(type_expression)
```

For example, if we have the definitions

```
int i;
Cow c;
Cow *pC;
```

then we could apply the `typeid` operator to the data items `i+12`, `c`, and `pC`, giving

```
typeid(i+12)
typeid(c)
typeid(pC)
```

which would respectively be equal to the results of applying `typeid` to the corresponding type expressions

```
typeid(int)
typeid(Cow)
typeid(Cow *)
```

In the program in the last section, evaluating the expression

```
typeid(a[k]) == typeid(DomesticAnimal)
```

would always yield the value true. To find out if the animal pointed to by `a[k]` is a cow, we would test the expression

```
typeid(*a[k]) == typeid(Cow)
```

to see if it was true. If it was true, we could then use a `static_cast` to cast `a[k]` to the appropriate type in order to milk the cow:

```
static_cast<Cow *>(a[k])->giveMilk();
```

The cast is necessary since without it, the statement

```
a[k]->giveMilk();
```

will not compile. This is because the type of `a[k]` is a pointer to a `DomesticAnimal`, and domestic animals do not have a `giveMilk()` member function.