To fix this, you should check for self-assignment at the start of any overloaded assignment operator. You can do this in most cases by comparing the address of the object for which the operator was called with the address of its argument. If the addresses are the same, the objects are identical and you should return immediately. (You don't need to assign one to the other; they're already the same.) For example, in STRIMEM2, you can insert the lines

```
if(this == &S)
   return *this;
```

at the start of `operator=()`. That should solve the problem.

Dynamic Type Information

It's possible to find out information about an object's class and even change the class of an object at runtime. We'll look briefly at two mechanisms: the `dynamic_cast` operator, and the `typeid` operator. These are advanced capabilities, but you may find them useful someday.

These capabilities are usually used in situations where a variety of classes are descended (sometimes in complicated ways) from a base class. For dynamic casts to work, the base class must be polymorphic; that is, it must have at least one virtual function.

For both `dynamic_cast` and `typeid` to work, your compiler must enable Run-Time Type Information (RTTI). Borland C++Builder has this capability enabled by default, but in Microsoft Visual C++ you'll need to turn it on overtly. See Appendix C, "Microsoft Visual C++," for details on how this is done. You'll also need to include the header file TYPEINFO.

Checking the Type of a Class with `dynamic_cast`

Suppose some other program sends your program an object (as the operating system might do with a callback function). It's supposed to be a certain type of object, but you want to check it to be sure. How can you tell if an object is a certain type? The `dynamic_cast` operator provides a way, assuming that the classes whose objects you want to check are all descended from a common ancestor. The DYNCAST1 program shows how this looks.

```
//dyncast1.cpp
//dynamic cast used to test type of object
//RTTI must be enabled in compiler
#include <iostream>
#include <typeinfo>           //for dynamic_cast
using namespace std;
////////////////////////////////////////////////////////////////
class Base
   {
   virtual void vertFunc()    //needed for dynamic cast
      {  }
   };
```

```
class Derv1 : public Base
   {  };
class Derv2 : public Base
   {  };
////////////////////////////////////////////////////////////////
//checks if pUnknown points to a Derv1
bool isDerv1(Base* pUnknown)  //unknown subclass of Base
   {
   Derv1* pDerv1;
   if( pDerv1 = dynamic_cast<Derv1*>(pUnknown) )
      return true;
   else
      return false;
   }
//--------------------------------------------------------------
int main()
   {
   Derv1* d1 = new Derv1;
   Derv2* d2 = new Derv2;

   if( isDerv1(d1) )
      cout << "d1 is a member of the Derv1 class\n";
   else
      cout << "d1 is not a member of the Derv1 class\n";

   if( isDerv1(d2) )
      cout << "d2 is a member of the Derv1 class\n";
   else
      cout << "d2 is not a member of the Derv1 class\n";
   return 0;
   }
```

Here we have a base class `Base` and two derived classes `Derv1` and `Derv2`. There's also a function, `isDerv1()`, which returns true if the pointer it received as an argument points to an object of class `Derv1`. This argument is of class `Base`, so the object passed can be either `Derv1` or `Derv2`. The `dynamic_cast` operator attempts to convert this unknown pointer `pUnknown` to type `Derv1`. If the result is not zero, `pUnknown` did point to a `Derv1` object. If the result is zero, it pointed to something else.

Changing Pointer Types with `dynamic_cast`

The `dynamic_cast` operator allows you to cast upward and downward in the inheritance tree. However, it allows such casting only in limited ways. The DYNCAST2 program shows examples of such casts.

```
//dyncast2.cpp
//tests dynamic casts
//RTTI must be enabled in compiler
#include <iostream>
#include <typeinfo>            //for dynamic_cast
using namespace std;
////////////////////////////////////////////////////////////////
class Base
   {
   protected:
      int ba;
   public:
      Base() : ba(0)
         {  }
      Base(int b) : ba(b)
         {  }
      virtual void vertFunc()  //needed for dynamic_cast
         {  }
      void show()
         { cout << "Base: ba=" << ba << endl; }
   };
////////////////////////////////////////////////////////////////
class Derv : public Base
   {
   private:
      int da;
   public:
      Derv(int b, int d) : da(d)
         { ba = b; }
      void show()
         { cout << "Derv: ba=" << ba << ", da=" << da << endl; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Base* pBase = new Base(10);           //pointer to Base
   Derv* pDerv = new Derv(21, 22);       //pointer to Derv

   //derived-to-base: upcast -- points to Base subobject of Derv
   pBase = dynamic_cast<Base*>(pDerv);
   pBase->show();                        //"Base: ba=21"

   pBase = new Derv(31, 32);             //normal
   //base-to-derived: downcast -- (pBase must point to a Derv)
   pDerv = dynamic_cast<Derv*>(pBase);
   pDerv->show();                        //"Derv: ba=31, da=32"
   return 0;
   }
```

Here we have a base and a derived class. We've given each of these classes a data item to better demonstrate the effects of dynamic casts.

In an upcast you attempt to change a derived-class object into a base-class object. What you get is the base part of the derived class object. In the example we make an object of class `Derv`. The base class part of this object holds member data `ba`, which has a value of 21, and the derived part holds data member `da`, which has the value 22. After the cast, `pBase` points to the base-class part of this `Derv` class object, so when called upon to display itself, it prints `Base: ba=21`. Upcasts are fine if all you want is the base part of the object.

In a downcast we put a derived class object, which is pointed to by a base-class pointer, into a derived-class pointer.

The `typeid` Operator

Sometimes you want more information about an object than simple verification that it's of a certain class. You can obtain information about the type of an unknown object, such as its class name, using the `typeid` operator. The TYPEID program demonstrates how it works.

```
// typeid.cpp
// demonstrates typeid() function
// RTTI must be enabled in compiler
#include <iostream>
#include <typeinfo>           //for typeid()
using namespace std;
////////////////////////////////////////////////////////////////
class Base
   {
   virtual void virtFunc()    //needed for typeid
      {  }
   };
class Derv1 : public Base
   { };
class Derv2 : public Base
   { };
////////////////////////////////////////////////////////////////
void displayName(Base* pB)
   {
   cout << "pointer to an object of ";  //display name of class
   cout << typeid(*pB).name() << endl;  //pointed to by pB
   }
//--------------------------------------------------------------
int main()
   {
   Base* pBase = new Derv1;
   displayName(pBase);   //"pointer to an object of class Derv1"

   pBase = new Derv2;
   displayName(pBase);   //"pointer to an object of class Derv2"
   return 0;
   }
```

In this example the `displayName()` function displays the name of the class of the object passed to it. To do this, it uses the `name` member of the `type_info` class, along with the `typeid` operator. In `main()` we pass this function two objects of class `Derv1` and `Derv2` respectively, and the program's output is

```
pointer to an object of class Derv1
pointer to an object of class Derv2
```

Besides its name, other information about a class is available using `typeid`. For example, you can check for equality of classes using an overloaded == operator. We'll show an example of this in the EMPL_IO program in Chapter 12, "Streams and Files." Although the examples in this section have used pointers, `dynamic_cast` and `typeid` work equally well with references.

Summary

Virtual functions provide a way for a program to decide while it is running what function to call. Ordinarily such decisions are made at compile time. Virtual functions make possible greater flexibility in performing the same kind of action on different kinds of objects. In particular, they allow the use of functions called from an array of type pointer-to-base that actually holds pointers (or references) to a variety of derived types. This is an example of *polymorphism*. Typically a function is declared virtual in the base class, and other functions with the same name are declared in derived classes.

The use of one or more pure virtual functions in a class makes the class *abstract*, which means that no objects can be instantiated from it.

A `friend` function can access a class's private data, even though it is not a member function of the class. This is useful when one function must have access to two or more unrelated classes and when an overloaded operator must use, on its left side, a value of a class other than the one of which it is a member. `friend`s are also used to facilitate functional notation.

A `static` function is one that operates on the class in general, rather than on objects of the class. In particular it can operate on static variables. It can be called with the class name and scope-resolution operator.