Chapter 16: Abstract Classes and Multiple Inheritance

From the Holy Bible, New King James Version, Proverbs 22:6.

> Train up a child in the way he should go, and when he is old he will not depart from it.

An abstract class declares a set of behaviors (function prototypes) that all of its descendents must follow (or implement). Documentation accompanying the abstract class must explain the purpose of each function and how each interacts with other parts of the class. Derived classes must implement all of the functions, and should obey the guidelines explained in the documentation.

16.1 An Abstract Class Defines Expectations

Abstract classes. When a base class includes even one prototype for a pure virtual function, it is an *abstract class* which cannot be used to create objects. However, such classes do have a purpose and they are important in the process of developing a large system. An abstract class lets us define and enforce a common interface, or behavior, for a set of related classes. Class derivation, combined with methods defined in the derived class(es) make the abstraction useful.

An abstract class specifies a set of virtual representation-dependent function prototypes for which definitions will be required in future derived classes. By doing so, it enables a large system to be developed in a top-down style.

The first development step is to define the major modules. The interface that each module provides is then written The compiler will ensure that no necessary part of A is forgotten, and that all functions conform to the prototypes that were promised.

Large systems developed by teams of programmers are built this way. First, each major system component is identified. As soon as its role in the system is clear, its public interface can be defined in the form of an abstract class declaration. The abstract class forms a *contract* for the programming team that will develop the component, and different people (or teams) will work on different subsystems simultaneously.

Suppose a system designer has specified modules named A, B, and C. Since all the prototypes provided by module A are defined, programmers working on modules B and C can begin to write code that calls the functions of A, even before A is fully implemented. Derivation is used to connect the abstract interfaces to their implementation modules. When modules are complete, the subsystems can be easily integrated because the established prototypes guarantee that functions in one module will be able to call functions in another module that were programmed by a different person.

Definitions and rules.

- The opposite of abstract is *concrete*. A concrete class can have virtual functions, but all of those functions must have methods defined within the class itself or its ancestor classes.
- An abstract class cannot be `instantiated`, that is, used to construct any objects.
- Any class that has one or more pure virtual functions is called an *abstract class*.
- Abstract classes is polymorphic if more than one concrete class is derived from it.

16.2 Abstraction Example: an Ordered Type

Two abstract classes are used in this chapter's program: `Container` (from the previous chapter) and `Ordered`.

- `Container` defines the interface that should be presented by any container class: a way to put data into the container (put()), and find it when needed (get()), take it out of the container (remove()), write the contents of the container to a stream (print()).

- Ordered defines prototypes for functions that are needed when you sort data items: comparison functions, sentinels, and a way to access the key field of the data.

```
1  // ---------------------------------------------------------------------
2  // Ordered base class -- An abstract class
3  // A. Fischer  June 8, 1998                              file: ordered.hpp
4  //
5  #ifndef ORDERED_H
6  #define ORDERED_H
7  #include <limits.h>
8  #include <iostream.h>
9  // ---------------------------------------------------------------------
10 class Ordered {
11   public:
12     virtual ~Ordered(){}
13     virtual KeyType key() const                          =0;
14     virtual bool    operator <  (const KeyType&) const  =0;
15     virtual bool    operator == (const KeyType&) const  =0;
16 };
17 #endif
```

Specifying and enforcing requirements. The purpose of a container is to store a collection of items. The data stored in an item is not important; we use the class name Item as a representative of any kind of object that a containers might store. However, a few properties of an Item are essential for use with a sorted container:

- The Item must contains a key field and a key() function that returns the key.
- The operators `<` and `==` must be defined to compare two Items. Items will be compared using one or both of these operators. They will be stored in the container in ascending order, as defined by the operator `<`.
- A programmer who creates an Ordered class must supply the appropriate typedef for KeyType and appropriate definitions for the operators and sentinels that define the minimum and maximum possible values for a KeyType object.

The first two properties can be specified by defining an abstract class, Ordered, that gives prototypes (but no definitions) for the three required functions. We can enforce these requirements in a data class by deriving the data class from Ordered. When we do this, we instruct the compiler to guarantee that the derived data class does implement every function listed by Ordered. If one of the functions is missing, the compiler will give an error comment.

16.3 Multiple Inheritance

A class may be derived from more than one parent class. (We must `#include` the header files for each parent class.) The purpose of such *multiple inheritance* is:

- Simple form: to inherit properties from one parent and constraints from another.
- General form: inherit properties from two parent classes

The syntax is a simple extension of ordinary derivation. We use it here (line 30) to combine the properties of the Exam class from the previous chapter with the abstract Ordered class. The new class is a *wrapper* for Exam that provides more functions than the original class but does not duplicate the ones that Exam supplies (`Print()` and `operator<<`).

16.3.1 Item: The Data Class

In the previous chapter, we used a `typedef` to make `Item` a synonym for the `Exam` class. In this chapter, we do more with `Item`: we use multiple inheritance to add constraints and functionality to the original `Exam` class.

```
18  //-------------------------------------------------------------------------
19  // Class declaration for data items.
20  // A. Fischer, May 29, 2001                                  file: item.hpp
21  //
22  #ifndef ITEM_H
23  #define ITEM_H
24  #include <limits.h>
25
26  typedef int KeyType;
27  #include "exam.hpp"
28  #include "ordered.hpp"
29  //-------------------------------------------------------------------------
30  class Item : public Exam, public Ordered {
31    public:
32      static const KeyType max_sentinel = KeyType(INT_MAX);
33      static const KeyType min_sentinel = KeyType(INT_MIN);
34
35      Item(char* init, int sc): Exam(init, sc){}
36      ~Item() { cerr <<"Deleting Item " << key() <<"\n"; }
37
38      KeyType key() const                      { return Score; }
39      bool operator==(const KeyType& k) const { return key() == k; }
40      bool operator< (const KeyType& k) const { return key() < k; }
41      bool operator< (const Item& s)    const { return key() < s.key(); }
42  };
43  #endif
```

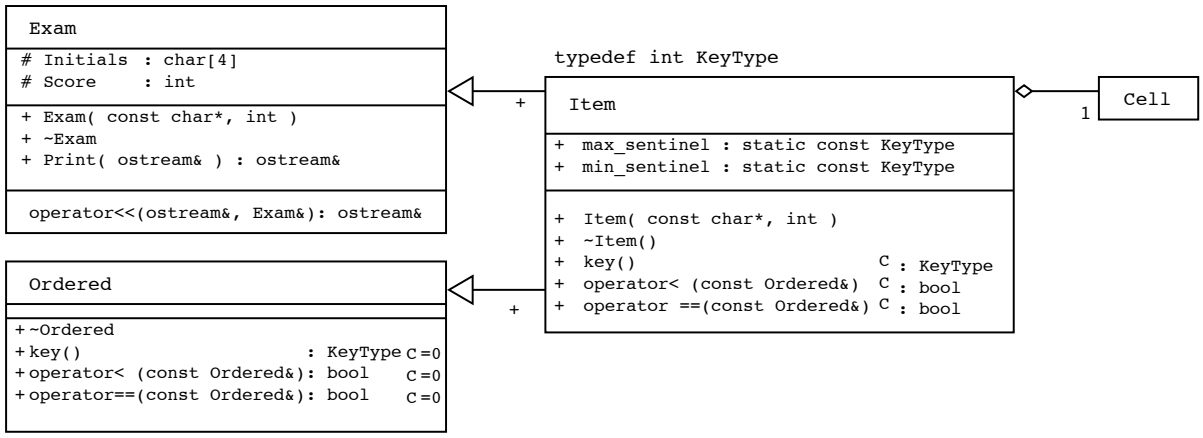

Figure 16.1: Inheriting both functionality and constraints.

Notes on the Item class. Two comparison operators and the key() function are required for Item because it was derived from Ordered. We define these three functions (lines 38–40) in such a way that Score (inherited from Exam) is the key field and the exams will be sorted in ascending order by Score. A third comparison function (line 41) is added for the convenience of client classes.

The Item class also defines two constants that are often needed for sorting algorithms: the maximum and minimum values of type KeyType. Line 24 tells us that KeyType is a synonym for int; the int values used here are supplied by the file `<limits.h>` in the standard library.

The Item constructor does nothing but pass its arguments through to the Exam constructor because it has no variables of its own that need initialization. The virtual destructor in the Ordered class is necessary to avoid warning comments in the Item class. For example, without that seemingly useless function, we get a warning comment about line 42:

```
item.hpp:42: warning:
    'class Item' has virtual functions but non-virtual destructor
```

With these definitions, Item fulfills all the inherited obligations, the class is concrete and can be used

normally. This class will be used with Linear and Cell from the prior chapter to build two new container classes: List and Priority Queue.

16.4 Linear Containers You Can Search

In this chapter we develop two new container classes from Linear. In a stack or a queue, all insertions and deletions are at one of the ends of the container; we never need to locate a spot in the middle. In contrast, a priority queue requires all insertions to be made in sorted order, and a simple list requires a search whenever an item is removed. To develop these classes in a general way, we assume that each Cell will contain an Item that is derived from Ordered, and we use the functions promised by Ordered to define three new functions in the Linear class:

```
bool Linear:: operator < ( Cell* cp ) { return (*cp->data < *here->data); }
bool Linear:: operator < ( KeyType k ){ return *here->data < k; }
bool Linear:: operator== ( KeyType k ){ return *here->data == k; }
```

These functions allow us to search or sort a linear container according to the key field of the Item.

16.4.1 PQueue: a Sorted Linear Container

Notes on the PQueue code Items are deleted from a priority queue at the head of the list, just like an ordinary queue. Preparation for a deletion is easy because Linear provides the reset() function to position its pointers at the head of the list.

In a priority queue, items are inserted in priority order in the list and removed from the head of the list. To do the insertion, we must scan the list to locate the correct insertion spot: the item at `prior` should have higher priority (a higher key number than the new item) and the item at `here` the same or lower priority. The loop on lines 60–62 performs such a scan using the `<` operator and list traversal functions (reset(), end(), and `++`) that are provided by Linear. Each reference to `*this` calls an operator defined by Linear and inherited by PQueue. When the right place is found, control is returned to Linear to do the actual insertion.

The only other functions here are a null constructor and a null destructor. Neither is necessary because the compiler will supply them by default. (Compare this class to List, below, in which the constructor and destructor have been omitted.) We write explicit functions because it is good style.

```
44  // ------------------------------------------------------------------------
45  // Priority queues: derived from  Container-<--Linear-<--PQueue
46  // A. Fischer   June 9, 2001                              file: pqueue.hpp
47  // ------------------------------------------------------------------------
48  #ifndef PQUEUE_H
49  #define PQUEUE_H
50  #include "linear.hpp"
51
52  class PQueue : public Linear {
53    public:          // ---------------------------------------------------
54      PQueue(){}
55      ~PQueue(){}
56      void focus(){ reset(); }    // Priority queue deletion is at the head.
57
58      // ------------------------ Insert new Cell in ascending sorted order.
59      void PQueue::insert( Cell* cp ) {
60          for (reset(); !end(); ++*this) {    // locate insertion spot.
61              if ( !(*this < cp) )break;
62          }
63          Linear::insert( cp );               // do the insertion.
64      }
65  };
66  #endif
```

16.4.2 List: An Unordered Container

Notes on the List code The List class provides a container with no special rules for insertion, deletion, or internal order. Since the order of items in the list does not matter, we use the easiest possible insertion method: insertion at the head, as in Stack. However, removing an item creates two new problems: how can we specify which item to remove, and how can we find it? The removal function required by Container does not have a parameter, but to remove an item from a List, we must know the key of the desired item. We solve the problem here by making the required focus() function interactive; it asks the operator to input a key. In a real program, the List class would probably have another public function that could be called with a parameter.

Once we know the key to remove, we search the list sequentially for a matching key. Since the list is unsorted, we must search the entire list before we know whether or not the key is in the list. If it is not, the pointer `here` will be NULL when we return from this function to Linear::remove(); the remove() will pass on the NULL it received to its caller, main().

```
67  // ------------------------------------------------------------------------
68  // Unsorted list: derived from  Container-<--Linear-<--List
69  // A. Fischer   June 9, 2001                                  file: list.hpp
70  // ------------------------------------------------------------------------
71  #ifndef LIST_H
72  #define LIST_H
73  #include "linear.hpp"
74  #include "item.hpp"
75
76  // ------------------------------------------------------------------------
77  class List : public Linear {
78    public:
79      void  insert( Cell* cp ) { reset(); Linear::insert(cp); }
80
81      // --------------------------------------------------------------------
82      void  focus(){
83          KeyType k;
84          cout <<"\n  What key would you like to remove? ";
85          cin >> k;
86          for (reset(); !end(); ++*this) if (*this == k) break;
87      }
88  };
89  #endif
```

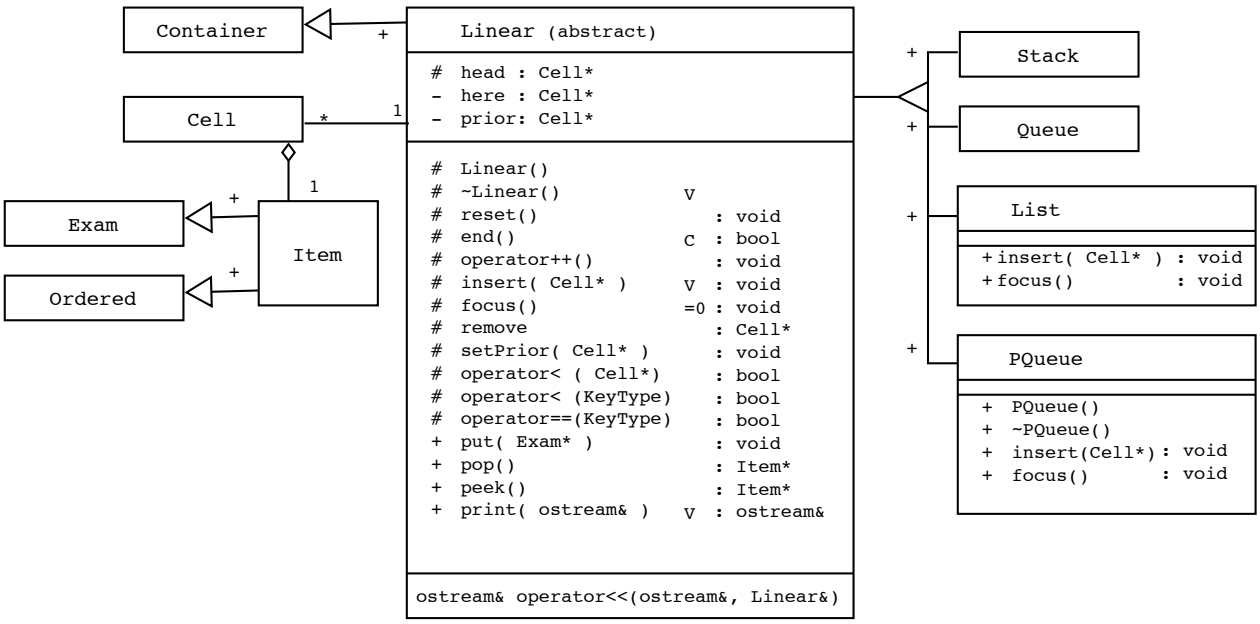

Figure 16.2: UML for all the Linear classes.

16.4.3 The Main Program

This main program has only one purpose: to test the two new classes. It puts meaningless data into two containers, takes some back out, and prints the results. The test results shown on the next page prove that insertions into the PQueue are implemented in the correct order, and that removal from the middle of the list works correctly. Other test runs verified that removal from head and tail of list, and attempted removal of a key that did not exist, also work properly. Trace comments after termination show that 9 cells (7 data cells and 2 dummy headers) and 7 Items were deleted by the destructors.

```
90   // ------------------------------------------------------------------------
91   // Demonstration of derived classes with virtual functionL.
92   // A. Fischer  June 15, 1998                                file: main.cpp
93   // ------------------------------------------------------------------------
94   #include "tools.hpp"
95   #include "pqueue.hpp"
96   #include "list.hpp"
97   #include "item.hpp"
98   // ------------------------------------------------------------------------
99   int main( void ) {
100    PQueue P;
101    List L;
102    cerr << "  Print the empty List L.\n" <<L;
103    cerr << "\n  Putting 3 items onto List L: 99, 77, 88.\n" ;
104    L.put( new Item("Ned", 29) );     //cerr << L << endl;
105    L.put( new Item("Leo", 37) );     //cerr << L << endl;
106    L.put( new Item("Max", 18) );       cerr << L << endl;
107
108    cerr << "  Putting 3 items onto PQueue P: 22, 11, 44.\n" ;
109    P.put( new Item("Bea",22) );      //cerr << P << endl;
110    P.put( new Item("Ali",11) );      //cerr << P << endl;
111    P.put( new Item("Dan",44) );        cerr << P << endl;
112
113
114    cerr << "  Remove one item from L and queue on P." ;
115    P.put( L.pop()) ;
116
117    cerr << "\n  Dequeue one item from P and push onto L." ;
118    L.put( P.pop()) ;
119
120    cerr << "\n  Pushing 33 onto P.\n" ;
121    cerr << "\n  Peek at P: " << *P.peek() <<" \n";
122    cerr <<"\n  The list contains: \n" << L;
123    P.put( new Item("Cil",33) );
124    cerr <<"\n  The priority queue contains: \n" << P << endl;
125    bye();
126  }
```

The output.

```
  Print the empty List L.
  <[
  ]>

  Putting 3 items onto List L: 99, 77, 88.
  <[
Cell 0x0x33650 [Max: 18  , 0x33630]
Cell 0x0x33630 [Leo: 37  , 0x33610]
Cell 0x0x33610 [Ned: 29  , 0x0]
  ]>

  Putting 3 items onto PQueue P: 22, 11, 44.
  <[
Cell 0x0x336b0 [Dan: 44  , 0x33670]
Cell 0x0x33670 [Bea: 22  , 0x33690]
Cell 0x0x33690 [Ali: 11  , 0x0]
  ]>

  Remove one item from L and queue on P.
  What key would you like to remove? 37
```

```
  Deleting Cell 0x0x33630...
  Dequeue one item from P and push onto L.
  Deleting Cell 0x0x336b0...
  Pushing 33 onto P.

  Peek at P: Leo: 37

  The list contains:
  <[
Cell 0x0x336b0 [Dan: 44  , 0x33650]
Cell 0x0x33650 [Max: 18  , 0x33610]
Cell 0x0x33610 [Ned: 29  , 0x0]
  ]>

  The priority queue contains:
  <[
Cell 0x0x33630 [Leo: 37  , 0x376f0]
Cell 0x0x376f0 [Cil: 33  , 0x33670]
Cell 0x0x33670 [Bea: 22  , 0x33690]
Cell 0x0x33690 [Ali: 11  , 0x0]
  ]>


Normal termination.

  Deleting Cell 0x0x335f0...Deleting Item 44
    Deleting Score Dan...
  Deleting Cell 0x0x336b0...Deleting Item 18
    Deleting Score Max...
  Deleting Cell 0x0x33650...Deleting Item 29
    Deleting Score Ned...
  Deleting Cell 0x0x33610...
  Deleting Cell 0x0x335e0...Deleting Item 37
    Deleting Score Leo...
  Deleting Cell 0x0x33630...Deleting Item 33
    Deleting Score Cil...
  Deleting Cell 0x0x376f0...Deleting Item 22
    Deleting Score Bea...
  Deleting Cell 0x0x33670...Deleting Item 11
    Deleting Score Ali...
  Deleting Cell 0x0x33690...
multiple has exited with status 0.
```

16.5 C++ Has Four Kinds of Casts

16.5.1 Static Casts

A static cast is an ordinary type conversion. It converts a value of one type to a value with approximately the same meaning in another type. The conversion can be a lengthening (short to long), a shortening (int to char), or a change in representation (float to int). The C and C++ languages support the built in type conversions shown in Figure 16.5.1 These are called "static" casts because the compiler finds out that they are needed at compile time and generates unconditional conversion code at that time.

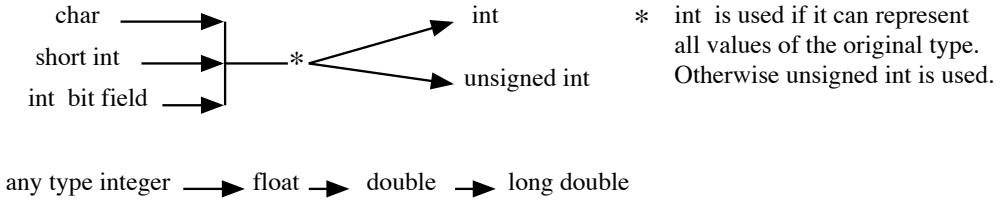

Figure 16.3: Built-in type conversions in C and C++.

Explicit casts. A static cast can be called explicitly using ordinary C syntax. In addition, C++ has two new ways to call a cast:

```
int k, m, *ip;
float f, *fp;
f = (float)k;                  // traditional C syntax.
```

```
f = float(k);                   // function-call syntax.
f = static_cast<float>(k);      // explicit C++ syntax.
```

Coercion. Coercion, or automatic type conversion, happens when a function call is encountered, and the type of an argument does not match the declared type of its parameter. In this case, the argument will be converted to the parameter type, if the compiler has a method for doing so. Coercion is also used to make operands match the type-requirements of operators. In C, coercion is limited to primitive types. However, in C++, it can also apply to a class type, say Cls:

- If an object of type T is used where a Cls object is needed, and the class Cls contains a constructor with one parameter of type T, the constructor will be used to coerce the T value to a value of type Cls.
- If an object of type Cls is used where a T object is needed, and the class Cls contains a cast operator whose result is type T, the cast operator will be used to coerce the Cls object so that the context makes sense.

16.5.2 Reinterpret Casts

A reinterpret cast is a type trick performed with pointers. It relabels the pointer's base type without changing any bits of either the pointer or its referent. This is like putting lamb's clothing on a wolf. Using a reinterpret cast, a program can access a value of one type using a pointer of a different type, without compiler warnings. (See Figure 16.5.2, line 147.) This lets us perform nonsensical operations such as adding incompatible values. For example, the reinterpret cast in the following program lets us relabel the integer 987654321 as a float, then add 1.0 to it to produce garbage:

```
140  #include <iostream.h>
141  using namespace std;
142  // ------------------------------------------------------------ File:  "assign.cpp"
143  int main( void )
144  {
145      int    my_int = 987654321;
146      int*   p_int = & my_int;
147      float* p_float  = (float*) p_int;
148      float  answer = *p_float + 1.0;
149      cerr <<answer <<"\n";
150  }
```

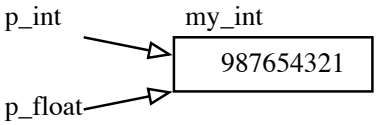

Figure 16.4: Relabeling the type of an object.

The answer is garbage: 1.0017, not 987654322 because The integer value and the floating point 1.0 were added to each other without any type conversion. The primary applications for reinterpret casts are hash functions and input conversion functions like strtod() and strtol().

Alternative syntax. Ordinary C syntax or C++ syntax with angle brackets can be used to invoke a reinterpret cast:

```
fp= (float*)p_int                   // ordinary C syntax.
fp= reinterpret_cast<float*>(ip);   // explicit C++ syntax.
```

16.5.3 Const Casts

A const cast provides a way to remove the const property from a pointer variable just long enough to change the value of its referent. This lets us use a constructor (or any class function) instead of a ctor to initialize a const class member.

```
127  #include <iostream.h>
128  int main( void )
129  {
130      int w = 99;
131      const int* cip = &w;
132
133      cout <<"  &w is " <<&w <<" referent of cip is " <<cip <<endl;
134      cout <<"  w= " <<w  <<"  *cip= " <<*cip <<endl;
135
136      * const_cast<int*>(cip) = 33;
137      //*cip = 33;
138      cout <<"  w= " <<w  <<"  *cip= " <<*cip <<endl;
139  }
```

When we write `*cip = 33;` without a const cast, we get a const violation error:

```
const.cpp: In function ‘int main ()’:
const.cpp:165: assignment of read-only location
```

With a const cast, we are permitted to change the location and we get output:

```
&w is 0xbffff714 referent of cip is 0xbffff714
w= 99  *cip= 99
w= 33  *cip= 33
```

16.5.4 Dynamic Casts

Dynamic casts are used with polymorphic classes, and can cast either pointers or references. Dynamic casts can move either upward or downward on the derivation tree. In Figure 16.6, a cast from B to D would be a downward cast, a cast from B to A would be an upward cast.

Suppose class D is derived from class B, as in the figure, and suppose p is a pointer (or a reference) to an object of class D. Then we can use a dynamic cast to relable p as a B pointer (or reference). An upward dynamic cast (from D to B) is always meaningful because any derived-class object includes a base-class object as part of itself.

According to my compiler, no downward casts are permitted at all. Accordiing to the Schildt text, some downward casts are permitted, but they are more complex and require a run-time check. Here is what Schildt says: If a pointer, p, has type A* it could be pointing at an object of any one of the four types A, B, C, or D. A down-cast of p from A* to B* would be meaningful if p's referent were actually type B or D, because these classes actually have all the members required by type B. However, such a down-cast would not be meaningful if p's referent were actually type A or C because some of B's members would be missing. It could cause a run-time crash if such a cast were permitted. To prevent such problems, the legality of every down-cast is checked at run time. A bad pointer down-cast will return a NULL result. A bad reference down-cast will throw an exception.

When do I use a dynamic-cast? You don't need to use an explicit dynamic cast to move up a derivation tree. The dynamic cast is done automatically whenever you use a derived-class object in a base-class context, or set a base-class pointer to point at a derived-class object. For this reason, it is hard to find a good use for an explicit dynamic cast in a simple program.

The rules for using dynamic casts are complicated by private parts, private derivation, and multiple inheritance. Some examples are given in the next section to illustrate these issues and show what kind of dynamic casts are and are not permitted in a multiple-inheritance situation.

16.6 Virtual Inheritance and Dynamic Casts

As long as derivation is used singly—so that a derived class has only one parent with data members—inheritance works smoothly, the storage model is easy to implement, and it is not hard to understand how it works. However, when a class inherits data members from two parents, two serious problems can occur:

- The `this` pointer points at the beginning of the entire object, consisting of the parts of the first parent, followed by the parts of the second, and finally, the parts of the derived class. To apply a function inherited from the second class, the compiler must compute where `this` should point for that part of the object. Dynamic casts are related to this problem.
- A class could inherit the same grandparent from two parents. If the grandparent has data members, are two copies of each inherited? Virtual inheritance exists to prevent this. When used in this way, the word `virtual` has nothing to do with virtual functions.

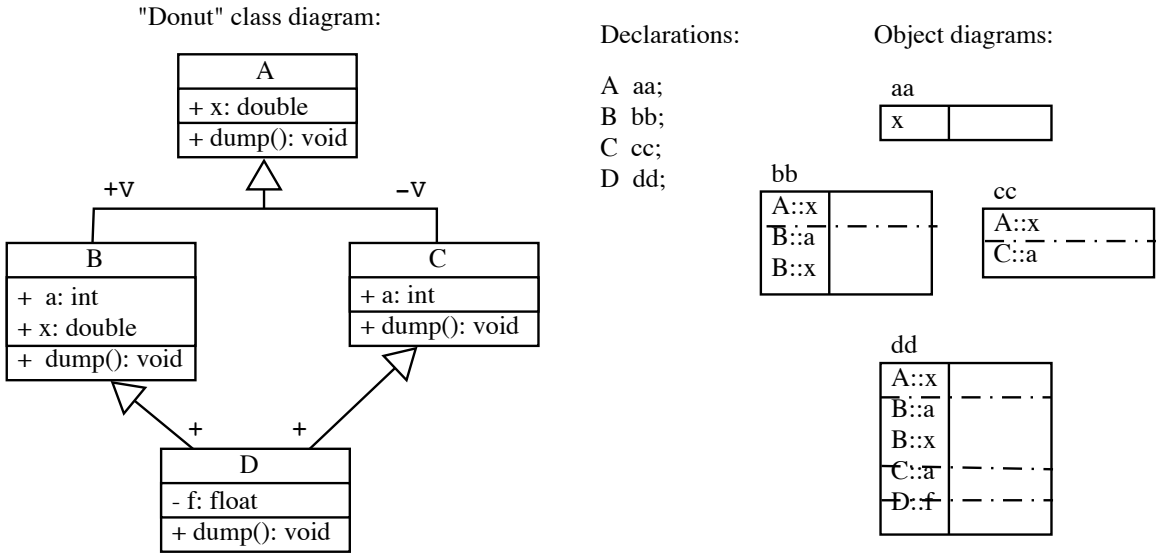

Figure 16.5: Double (donut) inheritance.

16.6.1 Virtual Inheritance

The keyword `virtual` can be used in a derivation declaration to prevent inheriting the same members from two parents. It is only relevant when multiple inheritance will be used, and when there is more than one path through the UML diagram from an ancestor class that has data members to some derived class. A class may have both virtual and nonvirtual base classes. When virtual derivation is used, an object of a derived class will have exactly one copy of the members of each ancestor. If derivation is not virtual, each derivation path will produce its own copy of any common ancestor, resulting in two data members with the same name in the derived-class object.

The simplest situation that illustrates the rules for virtual derivation is a class hierarchy in the shape of a "donut". The following set of four classes creates the "donut" shown in Figure 16.6. Each class has either one or two data members, a constructor, and a dump function. Data is made public to make it easier to show what is going on.

The data diagrams on the right show how storage would be allocated for objects of each of the four classes. The data member of A is inherited by classes B and C and becomes the first member of objects bb and cc. class D inherits all of the data members of B and all of the data members of C. Nonetheless, dd has only one sub-object of class A, because virtual derivation was used to derive B and C from A.

Naming rules. Two rules govern the meaning of a name in a donut situation:

- A class can inherit two members with the same name from different parent classes. When this happens, the name is *ambiguous* and you must use the :: to denote which one you want. For example, within class D, you would write B::a to refer to the member named 'a' inherited from B or C::a for the member inherited from C.
- If there are more than three levels in an inheritance hierarchy, the virtual property must be redeclared at each level that has multiple inheritance.

```
151  // ------------------------------------------------ File:  "donut.hpp"
152  #ifndef DONUT_H
153  #define DONUT_H
154  #include <iostream>
155  #include <iomanip>
156  using namespace std;
157  class A { //---------------------------------------- Grandparent Class
158      public:
159          double x;
160          A(): x(11.1) {}
161          virtual ~A(){};
162          virtual void dump(){ cerr <<"    A::x = " << x <<"\n"; }
163  };
164
165  class B: virtual public A { //----------------------- First Parent of D
166      public:
167          int a;
168          double x;
169          B(): a(20), x(22.2) {}
170          virtual ~B(){};
171          virtual void dump(){
172              A::dump();
173              cerr <<"    B::x = " << x <<"   "<<"B::a = " << a <<"\n";
174          }
175  };
176
177  class C: virtual private A { //---------------------- Second Parent of D
178      public:
179          int a;
180          C(): a(30) {}
181          virtual ~C(){};
182          virtual void dump(){ A::dump(); cerr <<"    C::a = " << a <<"\n"; }
183  };
184
185  class D: public B, public C { //---------------------- Grandchild Class
186          float f;
187      public:
188          D(): f(44.4) {}
189          void dump(){
190              A::dump();
191              B::dump();
192              C::dump();
193              cerr <<"    D::f = " << f <<"\n";
194          }
195  };
196  #endif
```

- A is a parent class of B, and both define a member named x. In classes A and C, only A::x is visible, so writing x in these contexts will always mean A::x. For functions in classes B and D, both A::x and B::x are visible, but B::x *dominates* A::x because it is "closer" to these classes. The dominant member will be used when x is written without the double colon. To refer to the non-dominant member, the full name A::x must be used.

The following brief program shows how the naming works and how visibility interacts with private inheritance. Two lines are commented out because they caused compilation errors:

- Line 204 produced this compiler error:

  ```
  double A::x  is inaccessible within this context in 'C' due to private inheritance.
  ```

- Line 210 was ambiguous. Neither B::a nor C::a is dominant here because the two classes B and C are equally close ancestors of D. To use either one in class D, we must qualify the member name as in lines 208 and 209.

```
197  // ------------------------------------------------------ file: naming.cpp
198  #include "donut.hpp"
199  int main( void )
200  {
201      C cc;                           // Object has one base class.
202      D dd;                           // Object has three base classes.
203      cout <<" &cc.a=    " <<&cc.a <<"  value= " <<cc.a <<endl;
204      //cout <<" cc.A::x= " <<&cc.A::x <<"  cc.x=" <<cc.x <<endl;
205
206      cout <<" &dd.x=    " <<&dd.x    <<"  value= " <<dd.x <<endl;
207      cout <<" &dd.A::x= " <<&dd.A::x <<"  value= " <<dd.A::x <<endl;
208      cout <<" &dd.B::a= " <<&dd.B::a <<"  value= " <<dd.B::a <<endl;
209      cout <<" &dd.C::a= " <<&dd.C::a <<"  value= " <<dd.C::a <<endl;
210      //cout <<" &dd.a= "  <<&dd.a    <<"  value= " <<dd.a <<endl;
211  }
```

Here is the output:

```
&cc.a=    0xbffff864  value= 30
&dd.x=    0xbffff850  value= 22.2
&dd.A::x= 0xbffff840  value= 11.1
&dd.B::a= 0xbffff84c  value= 20
&dd.C::a= 0xbffff834  value= 30
```

16.6.2 Dynamic Casts on the Donut

The final program in this chapter demonstrates dynamic casts and polymorphism in the context of our donut-shaped derivation tree.

```
212  #include "donut.hpp"
213  // ------------------------------------------------------------ File:  "donut.cpp"
214  int main( void )
215  {
216      A aa;   // Data member: x
217      C cc;   // Data members: A::x, C::a
218      D dd;   // Data members: A::x, B::a, B::x, C::a, D::f
219
220      cerr <<"Dumping aa\n";  aa.dump();
221      cerr <<"Dumping cc\n";  cc.dump();
222      cerr <<"Dumping the B part of dd\n";  dynamic_cast<B*>(&dd)->dump();
223
224      // Upward casts. ------------------------------------------------------------------
225      B* bp = &dd;                        // Explicit Low->high pointer cast not needed.
226      B& br = dd;                         // Explicit Low->high reference cast not needed.
227      cerr <<"Dumping br\n";  br.dump();  // Use the reference variable to dump.
228
229      A* ap = dynamic_cast<A*>(bp);       // Upward cast IS OK if derivation is public.
230      cerr <<"Dumping ap\n";  ap->dump(); // Show result of dynamic cast.
231      //dynamic_cast<A*>(&cc)->dump();    // Upward cast NOT OK if derivation is private.
232
233      // Downward casts. ----------------------------------------------------------------
234      cerr <<"Dumping dp after up and down casts, D->B->A->D .\n";
235      D* dp = dynamic_cast<D*>(bp);     // Can dynamic_cast downward where type is true.
236      dp->dump();                       // Start with D and return to a D.
237
238      cerr <<"Dumping cp after up and down casts, D->B->A->C .\n";
239      C* cp = dynamic_cast<C*>(ap);     // Can dynamic_cast downward where type is true.
240      cp->dump();                       // A D object has all the parts of a C object.
241
242      cerr <<"Dumping after down casts, A->C .\n";
243      ap = &aa;
244      //cp = dynamic_cast<C*>(&aa);     // Cannot dynamic_cast down to wrong type.
245      cp = dynamic_cast<C*>(ap);        // Cannot dynamic_cast down to wrong type.
246      cerr <<"No exception was thrown.\n";
247      cp->dump();
248  }
```

Polymorphism. The file `donut.hpp` defines a base class named A with two derived classes B, and C, and a class D derived from both B and C. We can create a donut diagram like this with or without polymorphism; a class only becomes polymorphic when it has virtual functions. In this example, Classs A, B, and C are polymorphic because they have virtual `dump()` functions. This forces us to also define virtual destructors.

Class D is the end of the derivation chain and the last class in the polymorphic family of classes. It is last because its derivation is not virtual and its functions are not virtual. Because these properties end in class D, it should not be used for further derivation.

A class is polymorphic if it has even one virtual function. If a class is polymorphic or is derived from a polymorphic class, the true type of every class object must be stored as part of the object at run time. I call this a "type tag". Whenever a virtual function is called, this type tag is used to select the most appropriate method for the function. If class is not part of a polymorphic family, no run-time type tag is attached to its objects, and no run-time function dispatching happens.

Virtual derivation

- Lines 216 through 222 create and print three variables. The output is:

```
Dumping aa
    A::x = 11.1
Dumping cc
    A::x = 11.1
    C::a = 30
Dumping the B part of dd
    A::x = 11.1
    B::x = 22.2   B::a = 20
```

- In this example, both B and C are derived virtually from A. If we omit the "virtual" from either one of the lines 163 and 174, or from both, we get this error comment.

```
donut.hpp: In method ‘void D::dump()’:
donut.hpp:37: cannot convert a pointer of type ‘D’ to a pointer of type ‘A’
donut.hpp:37: because ‘A’ is an ambiguous base class
```

Upward dynamic casts.

- Lines 225 and 226 perform implicit upward dynamic pointer and reference casts from class D to class B. Line 227 shows how to use the reference variable. The output is:

```
Dumping br
    A::x = 11.1
    B::x = 22.2   B::a = 20
```

- On line 222, an explicit cast was used because it was simplest. I tried an implicit cast there but it had the wrong prececence in relation to the `->` operator: `(B*)(&dd)->dump()`
- Line 229 shows another explicit dynamic cast, and line 230 prints the result. The output is:

```
Dumping ap
    A::x = 11.1
```

- Line 231 is commented out because it caused a privacy error:

```
Error in function ‘int main()’ of donut.cpp:
donut.cpp:20: dynamic_cast from ‘C’ to private base class ‘A’
A* ap = dynamic_cast<A*>(&cc);
```

Polymorphic downward dynamic casts.

- Lines 235, 239, 244, and 245 do explicit downward dynamic casts. If Classes A, B, and C did *not* have virtual functions, all of these lines would cause compile-time errors. The dynamic down-cast uses the run-time type information stored with every polymorphic object. But when a class is not polymorphic, the type tag is not there, and a dynamic down-cast cannot be done. Here is the error comment:

```
donut.cpp:24: cannot dynamic_cast ‘ap’ (of type ‘class A *’) to type ‘class C *’
```

- Because donut.hpp *does* defines a polymorphic class, lines 235, 239, 244, and 245 compile. The first of these lines is fine: we started with an object of type D, up-cast it, then down-cast it again. On line 235, we finished with class D, where we started. Clearly, every step in this casting-process was meaningful. The output is not surprising:

```
Dumping dp after up and down casts, D->B->A->D .
    A::x = 11.1
    A::x = 11.1
    B::x = 22.2   B::a = 20
    A::x = 11.1
    C::a = 30
    D::f = 44.4
```

- On Line 239, we downcast the same object to class C, which is also meaningful, since every D object contains a C object as part of itself. Even though the program now thinks it has a C object, the object itself retains its true type identity, and when we dump it, we get class D's version of dump, just as we did on line 236.

```
Dumping cp after up and down casts, D->B->A->C .
    A::x = 11.1
    A::x = 11.1
    B::x = 22.2   B::a = 20
    A::x = 11.1
    C::a = 30
    D::f = 44.4
```

- Line 244 is commented out because it gives a warning message:

```
donut.cpp:33: warning: dynamic_cast of ‘class A aa’ to ‘class C *’ can never succeed
```

  This warning happens because aa is an object, not a pointer, and we know that it does not have all the members that a C object needs. No casting magic can create the missing parts.

- On line 245, we down-cast an A* instead of an A&. There is no warning here, because the compiler cannot predict the actual type of an object that an A* pointer might point at. The result is a run-time malfunction:

```
Dumping after down casts, A->C .
No exception was thrown.
Bus error
```

  The Schildt text says that this downcast should cause an exception to be thrown, and since this program does not attempt to catch exceptions, the program should be terminated. (Exceptions are covered in Chapter 17.) Clearly, since control reached line 246, this did not happen. The compiled code did not check for an illegal downcast; it performed it. The result was a bus error (segmentation error on a second machine) when the print function tried to access a member of the object that never existed.