Abstraction is selective ignorance.

Andrew Koeing

6.1 Introduction

We will look at an example which implements multiple and repeated inheritance concept. This example allows us to create a vector (one-dimensional array) class of integers such that high level operations like array addition should be supported. It also allows us to change the size of the vector dynamically.

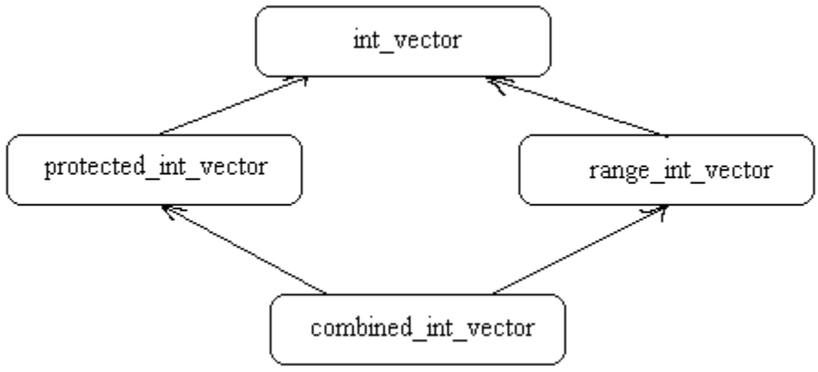

Fig. 6.1 Multiple and repeated inheritance

There are 4 classes as shown in Fig. 6.1. The base class is int_vector, which contains all the common functions used by the derived classes protected_int_vector and range_int_vector. There is a class combined_int_vector which is created which is an extension of both protected_int_vector and range_int_vector.

6.1.1 int_vector

The header file, "int_vec.h" contains the class definition. The class contains 3 data members.

sz - to define the size of the array
init_value - initial value assigned to an array
*ip - a pointer data which has memory allocated based on the size. After the memory is allocated, the init_value data is assigned to each element of the array.

In addition it has a constructor, destructor, resize function, output operator, subscript operator, assignment operator etc.

6.1 Code Listing

```
# include <iostream.h>
# include <stdlib.h>

# ifndef  INT_VECTOR_H
# define INT_VECTOR_H

class int_vector
{
   private:
      int sz
```

```
      int init_value;
      int *ip;

   protected:
      // This function returns the offset given the array index
      virtual int get_offset(int x)
      {
         return x;
      }

   public:
      int_vector(int size = 0, int default_value = 0);
      ~int_vector();
      int size()
      {
         return sz;
      }

      void resize(int new_size);
      int_vector& operator = (int_vector &);
      int& operator [](int x);
      friend int_vector operator + (int_vector & a, int_vector & b);
      friend ostream & operator << (ostream & o, int_vector & v);
};
#endif
```

The implementation of the member function follows (int_vec.cpp file).

Code Listing continued

The constructor function allocates an array of the requested size and initializes all the elements of the array to the initial value.

```
int_vector :: int_vector(int size, int initial)
{
   sz = size;
   init_value = initial;
   ip = new int[size];

   for(int i = 0; i < size; i++)
      ip[i] = initial;
}
```

The destructor function returns our array back to the free pool.

```
int_vector :: ~int_vector()
{
   delete ip;
}
```

The subscript operator returns a value from the array pointed to by the ip.

```
int& int_vector :: operator[](int x)
{
   return ip[get_offset(x)];
}
```

The overloaded assignment operator deletes the array in the destination, allocates a new array and copies the source array to it.

```
int_vector& int_vector :: operator = (int_vector &v)
{
```

```
   if(this != &v)
   {
      delete ip;
      ip = new int[sz = v.size()];

      for(int i = 0; i < sz; i++)
         ip[i] = v.ip[i];
   }
   return *this;
}
```

Implement the copy constructor?

The resize function allocates memory for a new array, copies the old values and reclaims the old array.

```
void int_vector :: resize(int size)
{
   int *new_p = new int[size];

   for(int i = 0; i < size; i++)
      new_p[i] = (i < sz) ? ip[i] : init_value;

   delete ip;
   sz = size;
   ip = new_p;
}
```

The operator + adds two vectors of the same size. It adds the corresponding elements and returns a vector containing the sum.

```
int_vector operator +(int_vector &a, int_vector &b)
{
   if(a.size() != b.size())
   {
      cout<<"Attempt to add int vectors of unequal sizes !\n";
      exit(1);
   }

   int_vector c(a.size(), 0);

   for(int i = 0; i < a.size(); i++)
      c.ip[i] = a.ip[i] + b.ip[i];

   return c;
}
```

Modify the operator + to add vectors of unequal size.

The output operator outputs the data separated by comma (,) surrounded by square brackets.

```
ostream & operator << (ostream & o, int_vector & v)
{
   o << "[";

   for(int i = 0; i < v.size(); i++)
   {
      if(i > 0)
         o << ",";
      o << v.ip[i];
   }
   return(o << "]");
}
```

A program to test the use of int_vector class.

```
# include "int_vec.h"

void main()
{
   int_vector a(10, 1), b(10, 2);
   cout << "Changing A[3] to 3" << endl;
   a[3] = 3;
   cout << " A is " << a << " B is " << b << endl;
   int_vector c = a + b;
   cout << "A+B is" << c << endl;
   cout << "Shrinking A to 5 elements" << endl;
   a.resize(5);
   cout << "A is" << a << endl;
   cout << "Growing A to 15 elements" << endl;
   a.resize(15);
   cout << "A is" << a << endl;
}
```

6.1.2 protected_int_vector

This class adds a protection feature to the int vector class. References to out of bound elements should return an error value. We have defined a new class so as not to affect users of the int_vector class.

6.2 Code Listing

```
# include "int_vec.h"

# ifndef  PROTECTED_INT_VECTOR_H
# define  PROTECTED_INT_VECTOR_H

class protected_int_vector :  public virtual int_vector
{
   private:
      int error_value;

   public:
      protected_int_vector(int size = 0, int initial = 0, int error = -1);
      int& operator[](int x);
};
#endif;
```

Code Listing continued

The implementation contains a constructor definition and the overridden subscript operator.

The constructor calls the base class constructor and then initializes the error value.

```
protected_int_vector :: protected_int_vector(int size, int initial, int error)
: int_vector(size, initial)
{
   error_value = error;
}
```

The subscript operator is used to check the validity of the subscript. If the index is in the range, we call the base class subscript, otherwise return an error value.

```
int & protected_int_vector :: operator[](int x)
{
   int x1 = get_offset(x);
```

```
   return((x1 >= 0 && x1< size()) ? int_vector :: operator[](x)
   : error_value);
}
```

A program to test the use of protected_int_vector class.

```
# include "pro_vec.h"

void main()
{
   int_vector a(10, 1);
   protected_int_vector b(10, 2, -1);
   cout << "A is" << a << endl;
   cout << "B is" << b << endl;
   int_vector c = a + b;
   cout << "A + B is" << c << endl;
   cout << "Invalid subscript" <<endl;
   cout << "B[25] is" << b[25];
}
```

6.1.3 range_int_vector

This class adds yet another feature to the int_vector class. Instead of having the subscript index from 0 to n, we can have a subscript defined from x to y. The output operator should indicate our vectors new starting point. Because of this requirement, our base class (int_vector), has a function defined called get_offset, which converts a subscript to the appropriate offset in the array. This has also resulted in the protected subscript operator to make a call to the get_offset() function, before an array out of bounds check is made. The get_offset() is declared as virtual, so the subscript function will choose the right version at execution time.

6.3 Code Listing

```
# include "int_vec.h"

# ifndef  RANGE_INT_VECTOR_H
# define RANGE_INT_VECTOR_H

class range_int_vector : virtual public int_vector
{
   private:
      int offset;
      int get_offset(int x);
   public:
      range_int_vector(int = 0, int = 0, int = 0);
      friend ostream & operator << (ostream &, range_int_vector &);
};
#endif
```

Code Listing continued

The implementation of the class (ran_vec.cpp) follows. It contains a constructor definition, get_offset and the overridden output operator.
The constructor calls the base class constructor and then initializes the offset value.

```
range_int_vector :: range_int_vector(int low, int high, int initial_value) :
   int_vector(high - low + 1, initial_value)
{
   offset = low;
}
```

The get_offset routine converts the subscript passed into the real offset into the array.

```
int range_int_vector :: get_offset(int x)
{
   return (x - offset);
}
```

Our output operator outputs the offset and a colon. It then casts the vector to class int_vector, so that the existing print routine will handle the output of the data.

```
ostream & operator << (ostream & o, range_int_vector & v)
{
   int_vector & iv = v;
   return o << v.offset << ":" << iv;
}
```

A program to test the use of range_int_vector class.

```
# include <iostream.h>
# include "pro_vec.h"
# include "ran_vec.h"

void main()
{
   range_int_vector a(3, 12, 1);
   protected_int_vector b(10, 2, -1);
   cout << "A is " << a << endl;
   cout << "Changing A[3] to 3 " << endl;

   a[3] = 3;

   cout << "A is" << a << endl;
   cout << "B is" << b <<endl;
   int_vector c = a + b;
   cout << "A+B is" << c << endl;
}
```

6.1.4 combined_int_vector

This class has the dynamic resizing and the vector definition from the int_vector class. The subscript checking from the protected_int_vector class and the arbitrary starting subscript from the range_int_vector class.

The combined_int_vector class inherits every data member and member function from all its parents. This can lead to ambiguity as the same data member and the member function can be defined in more than one class. If a name is defined in the current class or inherited from only one ancestor class, it can be used without any qualification. However, whenever ambiguity exists, the programmer must resolve it by qualifying the name.

6.4 Code Listing

```
# include "pro_vec.h"
# include "ran_vec.h"
# ifndef COMBINED_INT_VECTOR_H
# define COMBINED_INT_VECTOR_H

class combined_int_vector : public protected_int_vector,
   public range_int_vector
{
   public:
      combined_int_vector(int l = 0, int h = 0, int i = 0, int e = 0);
};
```

```
#endif;

combined_int_vector :: combined_int_vector(int l, int h, int i, int e) :
   protected_int_vector(h - l +1, i, e),
   range_int_vector(l, h, i), int_vector(h - l + 1, i)
{
   // Nothing here
}
```

The structure of the objects is as shown in Fig. 6.2

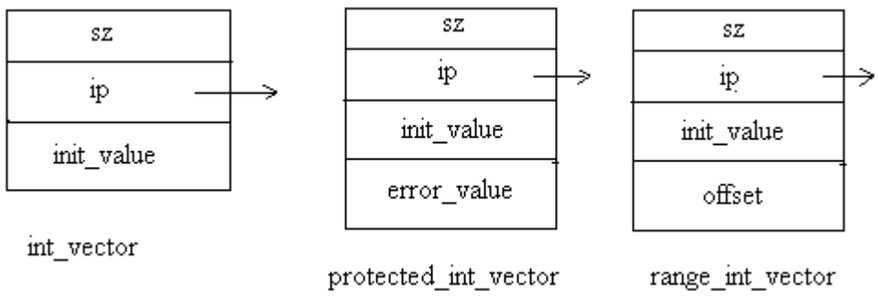

Fig. 6.2 Structure of objects

With multiple inheritance involving a common parent (repeated inheritance), we may get multiple copies of the parent class structure. Refer to Fig. 6.3. We solve this problem by declaring the parent class virtual. The structure of combined_int_vector after declaring the class virtual is as shown in Fig. 6.4

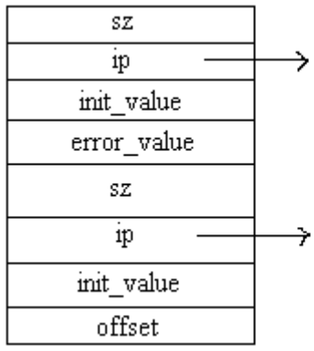

Fig. 6.3 combined_int_vector without virtual

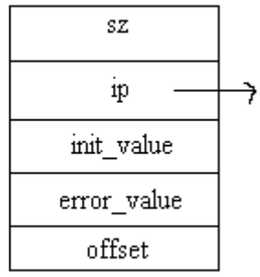

Fig. 6.4 combined_int_vector with virtual

6.2 Virtual base classes

Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritance. Each nonvirtual object contains a copy of the data members defined in the base class. This duplication wastes space and requires you to specify which copy of the base class members you want whenever you access them. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use it as a virtual base. Virtual inheritance provides significant size benefits when compared with nonvirtual inheritance. However, it can introduce extra processing overhead.

The **virtual** keyword appears in the base lists of the derived classes, not the base class.

The virtual declaration must appear in a class if a child of that class is to have a single copy of a parent of that class. Because of the above reason, the protected_int_vector and the range_int_vector class are declared with the virtual keyword as shown below.

```
class protected_int_vector : public virtual int_vector { };

class range_int_vector : public virtual int_vector { };

class combined_int_vector : public protected_int_vector, public range_int_vector { };
```

However, at the time when we declare PIV and RIV, you may not know whether any class will decide to inherit from both of them, and you shouldn't need to know this in order to declare them correctly. If you do not declare IV as a virtual base of PIV and RIV, a later designer of CIV may need to modify the declarations of PIV and RIV in order to use them effectively. Frequently, this is unacceptable, often because the declarations of IV, PIV and RIV are read-only. This would be the case if IV, PIV and RIV were in a library, for example, and CIV was being written by a library client.

On the other hand, if you do declare IV as a virtual base of PIV and CIV, you almost certainly impose an additional cost in both space and time on clients of those classes. That is because virtual base classes are typically implemented as pointers to objects, rather than as objects themselves. It goes without saying that the layout of objects in memory is highly compiler dependent, but the fact remains that the memory layout for an object of type CIV with IV as a nonvirtual base is typically a contiguous series of memory locations, whereas the memory layout for an object of type CIV with IV as a virtual base is typically a contiguous series of memory locations, one of which contains a pointer to an additional series of memory locations containing the data members of the virtual base class.

Passing constructors arguments to virtual base classes. Under nonvirtual inheritance, arguments for a base class constructor are specified in the member initialization lists of the classes immediately derived from the base class. Because single inheritance uses only nonvirtual bases, arguments are passed up the inheritance hierarchy in a very natural fashion: the classes at level n of the hierarchy pass arguments to the classes at level n-1. For constructors of a virtual base class, however, arguments are specified in the member initialization lists of the classes most derived from the base.

Dominance of virtual functions. Consider again the diamond shaped inheritance graph involving classes IV, PIV, RIV and CIV. Suppose that IV defines a virtual member function mf, and PIV redefines it; RIV and CIV, however, do not redefine mf. From our earlier discussion, you would expect this to be ambiguous:

```
CIV *ps = new CIV;
pd->mf();                  // IV :: mf() or PIV :: mf()
```

which mf should be called for CIV object, the one directly inherited from PIV or the one indirectly inherited (via RIV) from IV? The answer is that it depends on IV. In particular, if IV is a nonvirtual base of PIV and RIV, then the call is ambiguous, but if IV is a virtual base of both PIV and RIV, the redefinition of mf in PIV is said to dominate the original definition in IV, and the call to mf through pd will resolve (unambiguously) to PIV :: mf.

Casting restrictions. C++ explicitly prohibits you from casting down from a (pointer or reference to a) virtual base class to a derived class.

6.2.1 Multiple inheritance and constructors

When we have a virtual base class declaration, the combined_int_vector needs to call the constructors for any virtual base classes explicitly in its own constructors. It can't rely on the parent classes to do so, since the constructor would be called multiple times.

```
combined_int_vector :: combined_int_vector(int l, int h, int i, int e):
   int_vector(h - l + 1, i),
   protected_int_vector(h - l +1, i, e),
   range_int_vector(l, h, i)
{
      // Nothing here
}
```

6.2.2 Multiple inheritance and function invocation

What happens when we execute a[7] = 4 in the code below?

```
# include "comb_vec.h"

void main()
{
   combined_int_vector a(3, 11, 2, -1);
   a[7] = 4;
}
```

// main invokes operator [] (&a, 7)
// Both protected_int_vector (piv) and int_vector (iv) implement operator[]
// piv :: operator[] will be called

```
int & protected_int_vector :: operator[](int x)
{
   int x1 = get_offset(x);
   return(x1 >= 0 && x1< size()) ? int_vector :: operator[](x) : error_value;
}
```

// piv :: operator[] invokes get_offset(&a, 7)
// Both range_int_vector (riv) and int_vector implement get_offset
// riv :: get_offset will be called

```
int range_int_vector :: get_offset(int x)
{
   return (x - offset);
}
```

// riv :: get_offset returns 4 to piv :: operator[]
// piv :: operator[] calls iv :: operator[] (&a, 7)

```
int& int_vector :: operator[](int x)
{
   return ip[get_offset(x)];
}
```

// iv :: operator[] invokes get_offset(&a, 7) a virtual function
// Both riv and iv implement get_offset
// riv :: get_offset will be called

```
int range_int_vector :: get_offset(int x)
{
   return (x - offset);
}
```

// riv :: get_offset returns 4 to iv :: operator[]

```
int& int_vector :: operator[](int x)
{
   return ip[get_offset(x)];
}
```

// iv :: operator[] returns &ip + 4 to piv :: operator[]

```
int & protected_int_vector :: operator[](int x)
{
   int x1 = get_offset(x);
   return(x1 >= 0 && x1< size()) ? int_vector :: operator[](x) : error_value;
}
```

// piv :: operator[] returns &ip + 4 to main
// main executes *(&ip + 4) = 4

6.3 Exercise

Theory Questions

1. What problem does having a common ancestor in two different lines of inheritance raise? How can you solve this problem?
2. What is the difference between multiple and repeated inheritance?
3. How does multiple inheritance differ from using member classes as a way of incorporating two classes into a new class?
4. What does percolating functionality upwards mean? Is percolating upwards always a good thing? Why.
5. Why is switching on the runtime type of an object bad?
6. Why is casting bad?
7. Why are all the functions not declared as virtual?

Problems

1. Create a base class "mystr" (similar to the strings example of chapter 4). The class mystr has all the common data required by the derived classes; however an instance of it cannot be created. From the base class derive four classes "stcpy", "stcmp", "stcat" and "stlen" for the different string functions required. Using the concept of multiple inheritance derive a class called "stlib", which has all the features of its parent classes.

2. Create a base class "data" which has name and address (char *). From this class derive 2 other classes called "student" which has educational qualification (char*) data, and "employee" which has employee_id (int) and designation (char *) data. From these two classes derive 3 other classes called "research_scholar" (derived from student), "manager" (derived from student and employee) and "office_boy" (derived from employee). Read the data and output the same for each of the objects created.

3. Start with the publication, book and tape as specified in problem 3 of chapter 5. Add a base class 'sales' that holds an array of three doubles so that it can record the sales for a particular publication for the last three months. Include a getdata() function to get three sales amounts from the user and putdata() function to display the sales figures. Alter the book and tape classes so that they are derived from both publication and sales. An object of class 'book' or 'tape' should input and output sales data along with its other data. Write a main() function to create a book object and tape object and output their data.

4. An educational institution wishes to maintain records of its employees. The data is divided into a number of classes whose hierarchical relationships are shown in Fig. 6.5. The figure also shows the minimum information required for each class. Specify all the classes and define functions to create the data and retrieve individual information as and when required. Don't bother about file storage. Store information using either linked list or arrays.

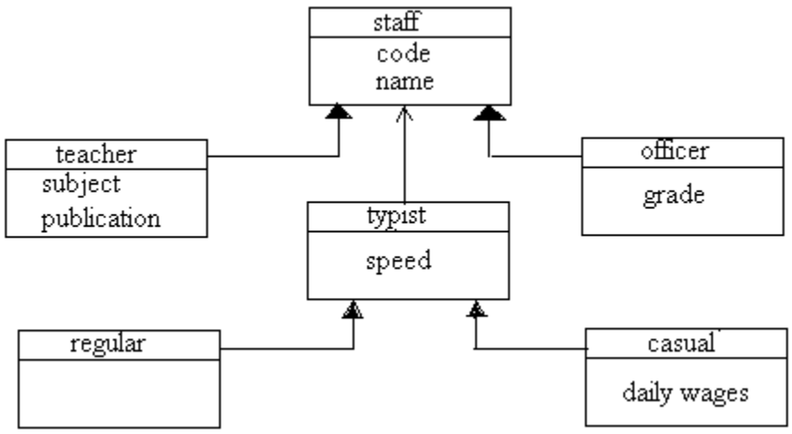

Fig. 6.5

5. In problem 4, the class teacher, officer, and typist are derived from the class staff. As we know, we can use container classes in place of inheritance in some situations. Redesign the program of problem 4 such that the class teacher, officer, and typist contain an object of staff.