C++ Input and Output

* the standard C++ library has a collection of classes that can be used for input and output
* most of these classes are based on a stream abstraction, the input or output device is viewed as a stream of characters
* so far we have used two basic output streams, cout and cerr, these streams are objects of the class ostream
* the ostream class has a << operator that is used to output a value onto the stream, the ostream class defines this operator for most of the standard C++ data types
* as we have seen before we can extend the range of this operator by defining our own << operator for any of the classes that we define
* for input we have one predefined object, cin, which is an object of the class istream
* this class uses the >> operator to input a value, it defines the >>operator for most of the standard C++ data types
* for each of the classes that we define, we can produce a version of the>> operator to input values of that class
* the >> operator normally returns the istream that it was called on, if it can't read an item of the expect type it returns zero
* this could happen if the end of file has been reached, or the next item on the input stream doesn't match the type of value that >> is expecting

* we can use this fact to write input loops, for example

```
int i;

while(TRUE) {
   if(cin >> i)
      cout << i;
   else break;
}
```

* this loop will copy integers from cin to cout until the end of file is reached or a non-integer appears on cin
* we could convert this into a template function that copies any type of value from an istream to an ostream

```
template void iocopy(T z, istream& is, ostream& os) {
   while(is >> z) os << z;
}
```

* even though we don't really need to pass z to this function, we need to have it as a parameter, otherwise the iocopy function won't compile
* the standard input operations skip white space the same way that the C input functions do, for most applications this is okay, we don't care about the white space between values
* but it we want to copy a text file exactly the way it is, including the whitespace characters this approach doesn't work

* the istream call defines a get function that handles this problem, there are two versions of this function
* the get(char&) version of this function reads a single character from the istream and places it in its parameter, this function returns NULL when the end of file is reached
* the other version of this function has the following declaration:

```
get(char* p, int n, char='\n')
```

* this procedure reads one or more characters into the buffer pointed to by p
* the parameter n gives the size of the buffer, get will always terminate the character string that is read by a 0, so there is room for at most n-1 characters in this buffer
* the get operation terminates when an end of file is encountered, n-1 characters have been read, or the character that is passed as the third parameter is encountered
* this terminating character isn't placed in p, and isn't read by the get operation, to determine whether get has overflowed the buffer you can check whether the next character in the stream is the terminating character

Stream States

* each stream has a state, this state is defined in the ios class that both istream and ostream inherit from
* there are many functions for manipulating the state of a stream, but there are only four that you really need to know about for most I/O programming:
    * eof() - returns true if the end of file has been seen
    * fail() - returns true if the next operation will fail
    * bad() - returns true if the stream has been corrupted
    * good() - returns true if the next operation might succeed

Formatted I/O

* the C printf and scanf functions provided formatted I/O, some of this is handled by the >> and << operators, but other parts, such as field width, aren't
* formatting is handled by the ios class, this class handles most of the interaction between the stream classes and their character buffers
* we will briefly look at some of the formatting functions that are available
* the flush() function is used to transfer all the characters in an output buffer to the device it is connected to

* this function is mainly used for streams connected to a display screen, so information is on the screen when the user needs to see it
* the tie() function is used to tie together an input and an output stream, in an interactive application we often send information to the screen, or requests and expect the user to respond to them, in this case the ostream needs to be flushed before each input operation
* the tie() function is called on an istream and it's parameter is the ostream that is to be flushed before each input operation
* the following call is automatically made for us:

```
cin.tie(&cout)
```

* the width of an output field can be controlled by calling the width() function, the parameter to this call is the minimum width of the output field
* the next output operation that requires formatting will produce an output field of at least this width, if the value needs more characters the field will be wider, otherwise it will be padded with a fill character
* the fill character is specified using the fill() procedure, the parameter to this procedure is a char giving the character to be used for padding, by default this character is the blank

```
cout << width(5) << i << width(4) << j << flush;
```

File Positioning

* both the istream and ostream classes have functions for positioning the character pointer within the stream, these functions only work if the stream is connected to a disk file
* for the ostream class we have the following functions:

```
ostream& seekp(streampos)  ostream& seekp(streamoff, seek_dir);
streampos tellp()
```

* the streampos type is an absolute position within the stream and the streamoff type is an offset from a reference position
* the possible reference positions (values of type seek_dir) are:

* beg - beginning of the file
* cur - current position in the file
* end- end of the file

* these functions work in essentially the same way as the stdio functions
* for istreams we have the following functions:

```
istream& seekg(streampos);  istream& seekg(streamoff, seek_dir);
streampos tellg();
```

* these functions operate in the same way as the corresponding ostream functions
* they have been assigned different names since a stream can be used for both input and output, therefore, a value of stream class could have an istream and ostream side and both could be active at the same time
* by having two sets of functions the position we are reading from could be different from the position we are writing to, even in the same file

The fstream Classes

* the fstream classes are designed for reading and writing files, there are three related classes:

  ifstream - input from a file
  ofstream - output to a file
  fstream - both input and output to a file

* when an ifstream file object is created it automatically assumes that the file will be opened for input, the first parameter to its constructor is the name of the file

* when an ofstream file object is created it automatically assumes that the file will be opened for output, the first parameter to its constructor is the name of the file
* the destructors for both of these objects will close their files, or you can explicitly close the file by calling the close() function on the object
* the constructors for these classes can take a second parameter that specifies how the file is opened, the values for these flags are specified in the ios class
* some of the possible values are:

- in - open for reading
- out - open for writing
- ate - open and seek to end of file
- app - append
- trunc - truncate file to zero length
- nocreate - fail if file does not exist
- noreplace - fail if the file exists

* if we wanted to open a file for both reading and writing we can do the following:

```
fstream index("database.index", ios::in | ios::out);
```

String Steams

* the istrstream and ostrstream classes can be used to read and write from character arrays, these classes operate in the same way as the sscanf and sprintf functions in stdio

* the first parameter to their constructors is a pointer to the character array and the second parameter is the size of the array, the array must be allocated before the constructor is called

* in the case of an ostrstream and end of file condition is signalled when the end of the string is reached, an ostrstream will not overwrite the end of the character array that it is passed

* the istrstream class expects a NULL terminated character string and extracts characters from this string as it reads, again an end of file will be signalled when the end of the string is reached

* all the istream and ostream functions can be performed on these classes

```
#include <strstream.h>
#include <fstream.h>
#include <iostream.h>
```

Template Classes and Functions

* Many of the functions and classes that we write are independent of the types of data that are manipulated
* we have already seen this with the list module, we really didn't care about the type of data that was stored in the list, the algorithms were the same
* in order to implement our C list module we had to rely on a trick, all pointers in C are the same size, so storing a pointer to the node value's works for any type of data
* the main problem with this approach is that we lose all type checking, we can't create a list of integers, and then have the compiler check that we only add integers to this list
* we would like to be able to write these generic classes, like lists, trees, arrays, etc, and still have type checking
* similarly a lot of the functions that we write have a similar structure, for example the max function we used to illustrate function overloading
* C++ had the notions of template classes and template functions that allows us to produce generic classes and functions
* we can parameterize the definition of a class or a function with one or more template parameters, these are parameters that are used to define a new type when it is declared

* for example if we wanted to have a generic list class, we would have the type of the list nodes as a parameter, from this generic class we could generate specific classes for different types of lists
* in C++ we can do this in the following way:

```
template < class T > class list {
    .
    .
    public:
        list();
        ~list();
        append(T a);
};
```

* with the list template class we can now create lists of different types:

```
list<int>  a;
list<char*>  b;
```

* in this example a would be a list of integers and b would be a list of character strings, we could only append integers to a, and strings to b, for example:

```
a.append(3);    // ok
b.append("a string");   //ok
b.append(6); // error, parameter must be of type char*
```

* when we create a class from a template class we must specify the class's parameters when we create it, in the case of our list example this parameter is the node type, we cannot just use list itself as a class
* a template class is somewhat like a macro, when we declare a class using it we substitute the actual parameters into the body of the class in order to generate the implementation of the class, this is done at compile time
* thus the entire definition of the template class must be in an include file, so the compiler can generate the functions, once it knows the types

Example

* to see how the idea of template classes works we can go back to our intarray example
* we produced an array that did range checking on its subscripts and then generalized this to an array that allowed arbitrary index ranges, we assumed that the elements of the arrays were integers

* none of the code in these classes really depended upon the type of the values stored in the array, we would use the exact same code for an array of doubles or an array of character strings
* we can make the type of array element a parameter to the class definition, this will give us a template class that can produce an array of any type
* there are two steps in this process 1. replace the int type by a parameter, for example T 2. combine the .h and .cc files into one file containing the complete class definition

Press here to see the example program

Warning

* the g++ implementation of templates still has some bugs in it, you may encounter some problems
* two of the problems that I encountered when doing the template classes for arrays are:

1. g++ can't tell the difference between a destructor and a constructor with no parameters in a template class, so you can't have constructors with no parameters
2. g++ occasionally gets confused in friend functions, this problem has been seen when trying to implement the << output operator for arrays, g++

couldn't seem to determine the type of the values data member inside of the friend function. Avoid the problem by not using friend functions and using the indexing operators in the << functions

Arrays of Objects

- our array class contains an array of objects, called values, this array can contain any type of object, so we need to be more careful than we were
- allocating the values array is fairly easy, we just need to call new with the number of items that we need
- the problem comes when we need to delete the array, all we have is a pointer to T, when we go to delete values, the c++ compiler doesn't know whether it is deleting a single object, pointed to by values, or an array of objects
- by default it assumes it is deleting a single object, which is reasonable, since it has been given a pointer
- in our case we have used a pointer since we didn't know how large the array would be until it was constructed, thus we need to do something different
- to delete an array we need to add [] after the delete keyword, this tells c++ that we are actually deleting an array of objects, and the destructor must be called for each object in the array

Templates and Derivation

* there are two ways in which templates can be used with derived classes
* one way was shown in the array example, range was derived from array, in this case the parameter for range was passed on to its base class, array
* when we do this type of derivation we must be sure to pass on the parameter to the base class, otherwise the compiler won't be able to construct the base class
* note that the base class could have a different set of parameters
* the other way is to derive a template class from a non-template class, that is we have a class with one or more parameters that is derived from a base class that has no parameters
* there is one main reason for trying to do this, efficiency
* each time we use a template class, all the procedures in that class are generated, that is when we provide parameter values, say in a declaration, the compiler must generate all the procedures in the class, if they haven't already been generated
* in the case of our array class we could have the following:

```
Array<int>  a;
```

```
Array<int>  b;
Array<double>  d;
Array<short>  s;
```

* these declarations will generate three sets of the array procedures, the first declaration will generate all the procedures for Array<int>, the second declaration won't need to generate these procedures, since the previous declaration had generated them
* the third declaration will generation the procedures for Array<double> and the last declaration will generate the procedures for Array<short>
* note: if there are two Array<int> declarations in different files, then the Array<int> procedures will be generated twice, once for each file
* there are two problems with this, it takes the compiler some time to generate all these procedures, so compile times will be longer
* these procedures take up space, this is particularly bad when multiple copies of the same procedure are generated
* in some cases we can't avoid generating all of these procedures, since they all depend upon the type of the parameter, but in other cases this can be avoided
* if a large number of the procedures in a template class don't depend upon the template parameters then there is no reason to generate them each time a new version of the class is generated

* these procedures can be placed in a non-template base class that is inherited by the template class, in this way only one copy of these procedures will be used by all the classes that are generated from the template
* our array example isn't a good example of this, since most of the functions in this class depend upon the parameter, but for more complex classes a significant number of the functions can be independent of the template parameters

Template Functions

* we can also produce functions that have template parameters, this allows us to write functions that can take a wide range of parameter types, the types of the parameters are a parameter to the function definition
* for example, we could write a simple swap procedure for any type of data:

```
template < class T > void swap(T& a, T& b) {
    T&  t;
    t = a;
    a = b;
    b = t;
```

```
};
```

* whenever we call swap the compiler will examine the types of its parameters and then generate the appropriate version of the swap procedure if it hasn't already been generated
* Examples:

```
int i, j;
double x, y;

swap(i,j);   // generate swap(int, int)
swap(x,y); // generate swap(double,double)
swap(i,x);   // error, both parameters must be of the same type
```

* when we define a function template at least one of the parameters to the function must be of the parameter type, otherwise the compiler won't know which version of the function to generate, all it has to go on are the types of the function parameters
* for example, the following declaration is incorrect

```
template < class T > T& generate(int number) {
    .
    .
```

```
    .
};
```

* you need to be careful about the interaction between template functions and overloading, since both of these mechanisms appear to be doing the same thing
* in the case of our swap procedure, the programmer could have produced a special purpose version of the swap procedure for character strings that understood about pointers to characters, etc, the programmer wants this version of the swap procedure to be called and not the generated version
* the rules for template procedures and overloading are:

1. Exact match on a non-template function
2. Exact match using a template function
3. Ordinary parameter resolution on a non-template function

* note we can force a particular version of the template function to be generated using an explicit declaration, and it will act like a non-template function in overloading

* Examples:

```
void swap(int,int);     // force the generation of swap(int,int);
```

```
int  i,j;
char  c;
double  x,y;

swap(x,y);    // generate the swap(double,double) function call
swap(i,j);    // use the previously built swap(int,int) function
swap(i,c);    // convert c to int and call swap(int,int)
```

Non-Type Parameters

* the template parameters to a class don't need to be types or class, they can be any type of value, for example integers
* why would we want an integer parameter? It can be used to specify the size of the object created from the class
* we could modify our array class template to have an integer parameter that specifies the size of the array, then the values data member can be declared as an array of T instead of a pointer to T
* Why is this useful??
* our current version of the array class allocates the values data member from the data segment, that is it must eventually call malloc or a similar function, which is an expensive operation
* most of our arrays will act like variables, they will be constructed when we enter the scope of a procedure and destroyed when we exit the

scope of the procedure, it would be much more efficient to store the values data member on the stack

* with an integer parameter we can do this
* when the class is generated, the size of the array is passed as a parameter, we know the size of the object when it is created so it can be allocated on the stack
* we now don't need a constructor or destructor for the array class, since there is no dynamic allocation
* since the values data member is allocated on the stack, the array class is much more efficient - malloc and free don't need to be called each time we declare an array variable
* this is just as easy for the programmer to use, for an array declaration we now have:

```
array<int,50>  a;
```

instead of:

```
array<int>  a(50);
```

Virtual Functions

* to see how virtual functions are used we will return to our university database example, we had the following class hierarchy [Image]
* we now want to be able to compute the payroll for the university, to do this we would like to iterate through all the staff members and compute their base pay and deductions
* we would like to do this with a simple for loop, assuming that we have a linked list of staff members
* we could use the following type of function to compute the pay

```
void compute_pay(staff* staff_list) {
    staff* s;
    for(s = staff_list; s != NULL; s = s->next)
        s->pay = s->gross() - s->deductions();
}
```

* this function looks like its doing the right thing, we have a pay data member for the staff class, this will be inherited by all the staff members, and we have functions gross() and deductions() that are defined in the staff class that compute the gross pay and deductions
* unfortunately the methods used to compute gross pay and deductions vary from one sub-class of staff to another, academic staff have a monthly salary, non-academic are paid hourly and graduate students have their tuition payments subtracted from their pay

* in order to compute pay correctly we need to know the type of object involved, but we only have a pointer to a staff object
* we could solve this problem by adding an extra field to the staff class called type, this field would have the values Academic, Non-Academic or Grad, then the gross() and deductions() functions can use this field to determine the correct computation to perform
* this solution is a major pain in the neck, since we will need to have a switch statement of the following form in each of these functions:

```
switch(type)  {
    case(Academic):                    ...
    case(Non-Academic):                    ...
    case(Grad):                    ...
}
```

* there is lots of room for mistakes here, and we need to remember to add this code to each function that treats the sub-classes differently
* C++ provides virtual functions to handle this problem for us
* a virtual function is declared in a base class, each of its sub-classes can redefine the virtual function, C++ provides the mechanism required to determine the type of object

* for example, we can use virtual functions for gross() and deductions(), they would be declared as virtual in the staff class, in the following way:

```
class staff {
    public:
    virtual double gross() = 0;
    virtual double deductions() = 0;

    protected:
    double pay;
};
```

* now when we call either gross() or deductions() from a staff pointer, C++ examines the type of object being pointed to and calls the correct function, we don't need to store a type in each object, C++ handles these details for us
* the virtual functions that we have declared in staff are called pure virtual functions, they have no definition and can't be called from a staff object
* an object that has pure virtual functions can't be used to create an object, it can only be used as the base class in a derivation of new classes, for example:

```
staff s;        // this is an error
staff* s;    // this is okay, it doesn't create a new object
academic fred;
staff &s = fred;   // this is okay, a reference to an existing object
```

* now consider the following examples:

```
academic fred;
non_academic george;
grad peter;
staff*  s;

s = &fred;
d = s->gross();    //  uses the academic class gross() function

    s = &peter;
    d = s->gross();    // uses the gross() function from the grad class

    s = &george;
    d = s->gross();   // uses the gross() function from the non-academic
    class
```

* when we define a virtual function in a derived class it must have the same name, return type and parameter types as the declaration in the base class, the keyword virtual doesn't need to be used when the function is declared in the derived class
* for example

```
Class academic : public staff {
    public:
      double gross();
      double deductions();
    protected:
      double salary;
};

double academic::gross() {
    return(salary);
}
```

* if the declaration in the derived class doesn't exactly match the declaration in the base class then the function won't be virtual, the virtual function from the base class will be inherited in this case
* how can you declare a virtual function in a base class without making it a pure virtual function, but still not provide a definition for it?

* the only way of doing this is to provide a definition for the function, and have the body of the function return some default value, for the staff class we can have:

```
class staff {
    public:
    virtual double gross() ;
    virtual double deductions() ;

    protected:
    double pay;
};

double staff::gross() {
    return(0.0);
}

    ...
```

Destructors

* can we have a virtual destructor? It looks as if this isn't possible, since a destructor must have the same name as the class, but C++ allows one

exception to this rule, a base class can define a virtual destructor, then all its derived classes can define their own destructors that are invoked through the virtual function mechanism

* Why would we want this? At the end of our program we might want to destroy all the people in our database, we could do this by maintaining a list of all the person (and its derived classes) objects and then just go through the list destroying the objects
* we could have created some temporary objects of various types, virtual destructors can then be used to destroy them in an organized way
* Note: virtual functions are more expensive than regular functions since C++ must determine at run-time the appropriate function to call - this is called dynamic binding, the function to call is determined at run-time, not at compile or load time

Access Levels

* what happens to access rights when a virtual function is used? whose access rights do we use?
* assume that the staff class declares gross() as a public function and the grad class declares gross() as a protected function, now what happens in the following

```
staff*  s;
```

```
grad peter;

s = &peter;

d = s->gross();     // this is legal
d = peter.gross();  // this will cause an error
```

* in the case of virtual functions, the access rights for the pointer are used, for staff gross() is public, therefore, s->gross() is legal even though it accesses a protected function in this case, if this wasn't the case virtual functions would be much more expensive, since the access rights would need to be checked on each call

Virtual Function Invocation

* virtual functions can only be invoked from a pointer or reference variable, they can't be invoked from an object variable
* for example:

```
academic fred;
staff*  s;
staff  t;
```

```
s = &fred;
d = s->gross();    //  calls the gross() function in academic

t =* s;   // not usually a good idea
d = t.gross();  //  calls the gross() function in the staff class, not virtual
```

- even through t contains (part of) an academic object, the staff object's gross function is called, since an object and not a pointer or reference is used
- the class used for the pointer must be the one that declares the function to be virtual, or a derived class of the base class that declares it to be virtual
- what happens if we have a pointer and we don't want to use a virtual function, we want to invoke a function defined in a particular class?
- the scope operator can be used for this purpose, for example

```
staff*  s;
grad  peter;

s = &peter;
d = s->staff::gross();  //  use gross() from the staff class
```

* in this case dynamic binding isn't used, the binding is done at compile time
* in a constructor or destructor, virtual functions are never used, the functions defined in the base class are used, since the object hasn't been constructed yet the virtual function may not be able to properly execute, the data may not be there yet
* again in this case dynamic binding isn't used, the binding to the function is done at compile time

Virtual Base Classes

* there is a problem with the class hierarchy for our university database example, the hierarchy is: [Image]
* the graduate student class inherits from both staff and student classes, and both of these classes inherit from the person class, therefore, the graduate class inherits two copies of the person class
* the graduate class will have two copies of all the data members in the person class, and won't be able to call any of the functions in the person class, the scope operator won't help since there are two person classes
* for example, we have the following declaration:

```
class graduate : public student, public staff { ....
```

* we have the following data in each graduate object: [Image]
* in most cases this is not what we want, if we inherit the same base class from two different branches of the tree we would like to have only one copy of it, we don't want to end up with multiple copies
* multiple copies wastes space and references to the common base class are ambiguous if we are not careful how we reference it, recall the person class:

```
class person {
    public:
    person(char*  name);
    person(char* name, char* address, sex gender, int age);
    char*  age();
    .
    .
    .
    protected:
    char*  Age;
    char*  Address;
    sex gender;
    int age;
};
```

* with the current derivation, each graduate object will have two Age data members, two Address data members, etc, we really only need to store this information once
* what happens if we want to call the person function age(), for example we have the following:

```
graduate*  g;
char*  s;

s = g->age();    //  won't work, two versions of age
s = g->person::age(); //  also won't work, two versions of person
```

* the second call correctly identifies the function, but not the object, there are two versions of the person object in a graduate object, one from staff and one from student, age() doesn't know which version to operate on, we must use something like this:

```
s = g->staff::age();   //  use the person data inherited from staff
s = g->student::age();  // use the person data inherited from student
```

* these two calls may not return the same value, since they are accessing different locations in memory

* these problems can be solved by using a virtual base class, when staff and student inherit from person they state that the inheritance is virtual, as follows:

```
class student: virtual public person {  ....  };
class staff : public virtual person {  ...  };
```

* now when we declare the graduate object we will only get one copy of the person object inherited:

```
class graduate : public student, public staff { ... } ;
```

* we can not reference all the data members of the person object without any problem, and we know that there will be only one copy of each data member
* also there is no problem with calling the member functions of the person object, we don't need to specify the object that we inherited the person from, since there is only one copy of the data, we can now do the following:

```
graduate*  g;
char*  s;
```

```
s = g->age();
```

* with the use of virtual base classes, the memory organization for a graduate object becomes: [Image]
* now the staff and student parts of the graduate object point to the same person data, they share the same data members from the person object

Initialization of Virtual Base Classes

* when is a virtual base class constructed and initialized? We need to be careful about this, the person class is inherited from both the staff and student classes, we don't want its constructors executed twice on the same data members
* virtual base classes are always constructed first, when a derived class is declared, for example

```
class graduate : public student, public staff { ...
```

* we list the classes its derived from, normally we would invoke the constructors in the order given by this list, in this case we would invoke the constructors in the student sub-tree followed by the ones in the staff sub-tree, and the classes at the root of the tree are constructed first

* when a virtual base class is used , the virtual base class is constructed first, no matter where it is placed in the inheritance tree, this ensures that we have a pointer to its data when we construct the other objects
* in a constructor function for a derived class we can call the constructors for the immediate base classes, so in the case of a graduate student constructor we can have:

```
graduate( ... ) : student( ... ), staff( ...) {
```

* what happens with a virtual base class?
* normally the constructors for student and staff would call the constructors for their base classes, which is person, in this case the person constructors would be called twice for the same object
* in the case of virtual base classes, the most derived class can call the constructor for the virtual class, in this way it will only be called once, if the most derived class doesn't call one of the virtual base class constructors, then the default constructor is called
* so in the case of graduate we would have:

```
graduate( ... ) : person ( ... ), student( ... ), staff( ... ) {
```

* we can only do this with virtual base classes

Dominance

* with a virtual base class there are several potential conflicts that can occur when the two branches of the tree treat the base class differently
* let's start by looking at access rights, the access rights for a virtual base class are handled in basically the same way as any other base class:

1. if we use a public derivation, the base class's public members remain public, etc
2. if we use a private derivation, then all the base class's members become private in the derived class

* now what happens if there is both a public and a private derivation of the common virtual base class?
* one derivation will state that all the base class's members are private, while the other will say that its public members are still public, which one do we use?
* in this case the public derivation wins, the public members of the virtual base class will remain public in the derived classes, even if they are private in the other branches of the tree
* we will use our university database example to illustrate this idea:

```
class person {
```

```
    public:
    char*  name();
    int age:

    protected:
    sex gender;
};

class student : public virtual person { ... };

class staff : private virtual person { ... };

class graduate : public student, public staff { ... };
```

* in staff, gender is a private data member, but in student gender is a protected data member, since gender comes from the common virtual class, it will be protected in the graduate class, the more public derivation always wins
* what would happen if the staff class defined its own version of the age member function, such as

```
class staff : private virtual person {
    public:
```

```
    int age();

        ...

};
```

* the graduate class now inherits two versions of age, one from staff and the other from student, which comes from the person class, normally this would cause an error when age() was referenced, but with a common virtual class this isn't a problem, any redefinition overrides the definition in the common virtual base class

```
graduate fred;
int a

a = fred.age();  // the version of age from staff
a = fred.student::age();  // the original version from person
```

* there is one last case left to consider, what happens when we have the following derivations:

```
class staff : virtual public person { .. };
```

```
class student : public person { ... };

class graduate : public staff, public student { ... };
```

* in the case of graduate, we inherit person as a virtual base class from staff and as a regular base class from student
* in this case there will be one set of person data members that are shared by all the virtual derivations, and for each non-virtual derivation there will be a separate set of person data (one for each derivation)
* it is highly unlikely that anyone would want to do this, but C++ doesn't produce an error or warning message when this occurs, so be careful