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