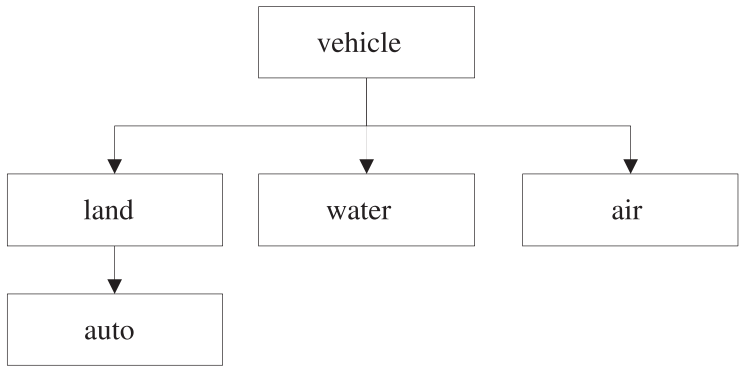
From chapters 11 of "C++ Annotation" version 4.3.1 by Frank B. Brokken and Karel Kubat.

Inheritance

Related classes :



Definition of the class vehicle :

```
class vehicle
{  public:
           //constructors
           vehicle();
           vehicle(int wt);
           //interface
           int getweight() const;
           void setweight(int wt);
   private:
           // data
           int weight;
};
```

A class `land` can be defined as follows :

```
class land
{ public:
         void setweight(int wt);
```

```
  private:
         vehicle v;
};
void land::setweight(int wt)
{
     v.setweight(wt);
}
```

There are 2 problems with this definition :

1. A semantic problem : the class `land` vehicle **contains** a vehicle. The correct relationship should be : a `land` vehicle is a **special case** of a `vehicle`.
2. A practical problem : needless code is introduced. `land::setweight` only calls `vehicle::setweight` - No extra functionality is added, so why the extra code ?

Inheritance solves both these problems :

```
class land: public vehicle
{ public:
         //constructors
         land();
         land(int wt, int sp);
         // interface
         void setspeed(int sp);
         int getspeed() const;
  private:
         //data
         int speed;
};
```

By post-fixing the class name `land` in its definition by `public vehicle`, the derivation is defined : The class `land` now contains all the functionality of its base class `vehicle plus` its own specific information.

Example of the use of the derived class :

```
land veh(1200,145);

int main()
{   cout << "Vehicle weight " << veh.getweight() << endl
         << "Speed is " << veh.getspeed() << endl;
    return(0);
}
```

This example shows two features of derivation :

1. `getweigth()` is no direct member of a `land`. Nevertheless it is used in `veh.getweight()` - This member function is an implicit, part of the class, inherited from its " parent" `vehicle`.
2. Although the derived class `land` contains the functionality of `vehicle`, the private fields of `vehicle` remain private : The only can be accessed by member functions of `vehicle` itself - `land` must use vehicle's `getweight` and `setweight` functions to address the weight field, just as any other code outside the `vehicle` class.

Similarly, we can create a class `auto` that is derived from `land` - This is called **nested** derivation.

Example:

```
class auto: public land
{  public:
           //constructors
           auto();
           auto(int wt, int sp, char const *nm);
           //interface
           char const *getname() const;
           void setname(char const *nm);
   private:
           // data
           char const *name;
};
```

auto contains the weight, speed and name of a car.

The Constructor of a Derived Class

Land's constructor could be defined as follow :

```
land::land(int wt, int sp)
{
     setweight(wt);
     setspeed(sp);
};
```

- Not efficient since `setweight` must call `vericle::setweight`.
Better way : call directly the constructor of `vehicle` :

```
land::land(int wt, int sp) : vehicle(wt)
{
     setspeed(sp);
};
```

Redefining Member Functions

The actions of all functions which are defined in a base class can be **redefined**.

Let trucks be represented in two parts : The front engine and the trailer
- Both the front engine and the trailer have their own weight, but the `getweight` function should return the combination weight.

Example :

```
class truck: public auto
{  public:
           //constructors
           truck();
           truck(int engine_wt, int sp, char const *nm, int trailer_wt);
           //interface : to set 2 weight fields
           void detweight(int engine_wt, int trailer_wt);
           //and returns combined weight
           int getweight() const;
   private:
```

```
           // data
           //the weight of the front part
           //is represented in class auto
           int trailer_weight;
};

//constructor
truck::truck(int engine_wt, int sp, char const *nm, int trailer_wt)
           :auto(engine_wt,sp,nm)
{

      trailer_weight = trailer_wt;
};
```

We redefine functions `setweight` and `getweight` as follow :

```
void truck::setweight(int engine_wt, int trailer_wt)
{
      trailer_weight = trailer_wt;
      setweight(engine_wt); //uses auto::setweight()
};

int truck::getweight() const
{
    return  (
auto::getweight() +    //sum of engine part plus
              trailer_weight);       //the trailer

};
```

Note that `auto::getweight()` must be `void` otherwise we could have infinite recursion.

Multiple Inheritance

It is possible for a class to be derived not from one but from several base classes - Such a class would inherit the functionality from more than one 'parent' at the same time.

Example : a class engine can store information about engine (serial number, power, type of fuel) can be defined.

```
class engine
{  public:
           ... //constructors and interface
   private:
           // data
           char const *serial_number, *fuel_type;
           int power;
};
```

In order to represent an `auto` together with the extra information contained in `engine`, we can define a new class, `motorcar`, derived from `auto` and from `engine` simultaneously:

```
class motorcar: public auto, public engine
{  public:
           //constructors
           motorcar();
           motorcar(int wt, int sp, char const *nm, char const *ser,
                    int pow, char const *fuel);
};

motorcar::motorcar(int wt, int sp, char const *nm, char const *ser,
                   int pow, char const *fuel);
{
      engine(ser,pow,fuel);
      auto(wt,sp,nm);
};
```

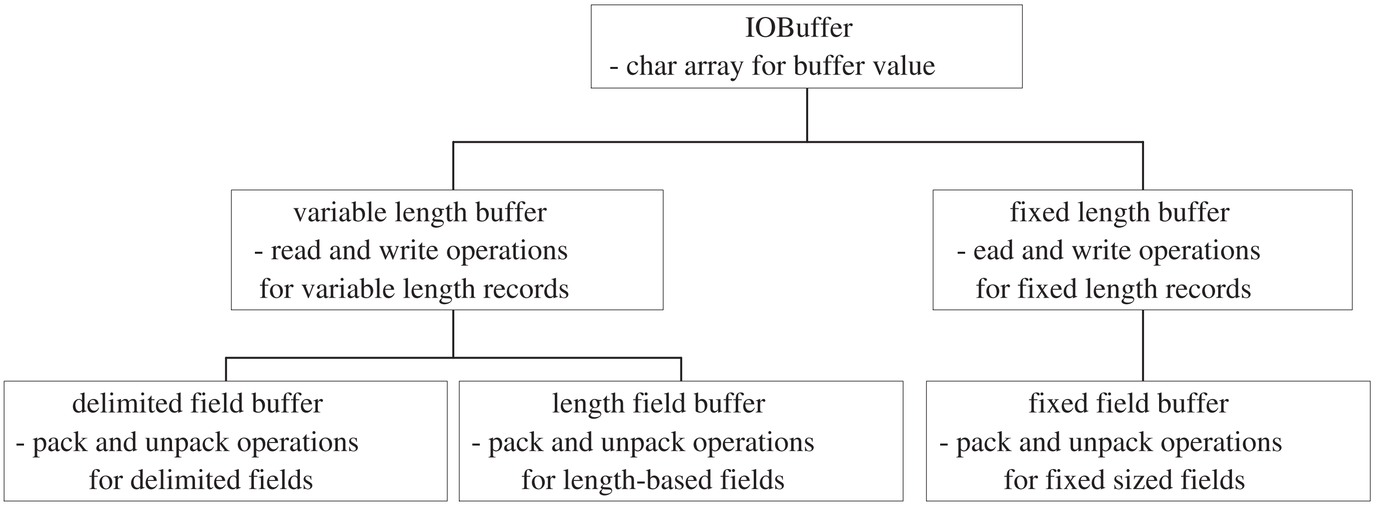
- Note : semantically, this definition is a bit odd since it suggests that a `motorcar` is both an `auto` and an `engine` rather than saying that a `motorcar` has an `engine` - However, if we expressed the relationship a `motorcar` has an `engine`, we would get duplicated code.

An example of the usefulness of inheritance for file structures (read section 4.2-5 of "File Structures : An Object-Oriented Approach with C++" by

Folk, Zoellick and Riccardi)

Prior to writing a record into a file, it might be useful to store it first into a **buffer** - This is particularly useful for variable length representation putting a length indicator at the beginning of each record : the size on the record can be obtained while packing the record in a buffer - This size is then written to the file followed by the content of the buffer.

There is a natural hierarchical organization of the different types of buffer that can be implemented using the inheritance tools provides by C++ - The hierarchy looks as follow :




```
class IOBuffer
{  public :
            IOBuffer(int maxBytes = 1000);
            virtual int read(istream &)=0;
            virtual int write(ostream &) const =0;
            virtual int pack(const void *field, int size = -1)=0;
            virtual int unpack(void *field, int maxbytes=-1)=0;
   protected :
            char * buffer;
            int BufferSize;
            int MaxBytes;
```

```
}
```

Notes :

- The methods with the `virtual` keyword are abstract methods - each class will define its own implemenetation.
- The keyword `virtual` is used in the case of multiple inheritance to make sure that classes common to 2 different ancestors are included only once in the ancestry of the inheriting class.
- The elements in the `protected` definition are visible to derived classes but not to other classes

Almost all the members and methods of all the buffer class are identical. The only differences are in the exact packing and unpacking and in the minor differences in read and write between the variable-length and fixed-length record structures.

- A lot of code will be shared when using inheritance :
Example : delimited and length-based variable length records can both use the same variable-length record `read` and `write` methods.