In C++ you can make one class an *extension* of another. This is called *inheritance*. The classes form an *is-a* hierarchy. For example, to implement a B-tree, one might want a class for the internal nodes and a class for the leaf nodes, but want pointers to be able to point to either type. One solution is to create a BTreeNode class with two derived classes.

The advantage of inheritance is that it avoids code duplication, promotes code reuse, and improves maintenance and extendibility. For example, one might have general code to handle graphics such as Shape’s, with specific code specialized to each graphics component such as Rectangle’s.

*Inheritance* allows one to create multiple *derived classes* from a *base class* without disturbing the implementation of the base class. Using public inheritance (the only version we study), the class is defined as follows:

```cpp
class Derived : public Base
{
    additional instance variables;
    new constructors;
    // inherited members and functions;
    overriding methods; // replacing those in Base
    additional methods;
};
```

The derived class automatically has the methods of the base class as member functions, unless declared as private in the base class. They may be declared as protected in the base class to allow direct access only to extensions. Similarly, the derived class can access the instance variables of the base class, provided not private.

16.2 Polymorphism, Static Types, and Dynamic Types

The derived class (subclass) is a new class that has some *type compatibility*, in that it can be substituted for the base class (superclass). A pointer has a static type determined by the compiler. An object has a dynamic type that is fixed at run-time creation. A pointer reference is polymorphic, since it can reference objects of different dynamic type. A pointer may reference objects of its declared type or any subtype of its declared type; subtype objects may be used whenever supertype objects
are expected. There are times an object may need to be *cast* back to its original type. Note that the actual dynamic type of an object is forever fixed.

Suppose class `Rectangle` extends class `Shape`. Then we could do the following assignments:

```cpp
Shape *X;
Rectangle *Y;
Y = new Rectangle();
X = Y;  // okay
X = new Rectangle();  // okay
Y = static_cast<Rectangle*>(X);  // cast needed
```

### 16.3 Overriding Functions

An important facet of inheritance is that the derived class can replace a general function of the base class with a tailored function. **Overriding** is providing a function with the same signature as the superclass but different body. The base class labels a function as *virtual* if it expects that function to be overridden by its derived classes. Note that a function declared as *virtual* in the base class is automatically *virtual* in the derived class.

But then a key question is: Which version of the function gets used? If we declare an object directly with code like `Rectangle R;` then the functions of the `Rectangle` class are used.

But if we reference an object (that is, use a pointer), then there are two options: if we declare the function as *virtual*, then which version of the function is used is determined at run-time by the object’s actual dynamic type. (In Java, all functions are implicitly virtual.) If we declare the function as *nonvirtual*, then which version is determined by the compiler at compile time, based on the static type of the *reference*.

```cpp
class Shape {
    virtual void foo(){ cout << "base foo"; }
    void bar() { cout << "base bar"; }
};
class Rectangle : public Shape {
    void foo() { cout << "derived foo"; }
    void bar() { cout << "derived bar"; }
};
Shape *X;
Rectangle *Y;
X = Y = new Rectangle();
Y -> foo();  // prints derived foo
```

51
Y -> bar(); // prints derived bar
X -> foo(); // prints derived foo
X -> bar(); // prints base bar

One can access in the Derived class the Base version of an overridden function by using the scope resolution operator: Base::fooBar().

16.4 Constructors in Derived Classes

The default constructor for Shape is automatically called before execution of the constructor for Rectangle, unless you specify otherwise. To specify otherwise, put the base class name as the first entry in the initializer list:

Derived(int alpha, string beta) : Base(alpha,beta) { }

16.5 Interfaces and Abstract Base Classes

In Java, an interface specifies the methods which a class should contain. A class can then implement an interface (actually it can implement more than one). In doing so, it must implement every method in the interface (it can have more). This is just a special case of inheritance: the base class specifies functions, but none is implemented.

C++ uses abstract base classes. An abstract base class is one where only some of the functions are implemented. A function is labeled as abstract by setting it equal to 0 when declaring; this tells the compiler that there will be no implementation of this function. It is called a pure function. An object with one or more pure functions cannot be instantiated.

Example: the abstract base class (interface)

class Number {
    public:
        virtual void increment()=0;
        virtual void add(Number &other)=0;
    ETC
};

the implementation:

class MyInteger : public Number {
    private:
        int x;
    public:
        virtual void increment(){x++;}
    ETC
};
the calling program (but then one can only execute Number's methods on ticket).

    Number *ticket = new MyInteger();
    ticket->increment();

Sample Code
A (somewhat artificial) example of using inheritance to create a 3-dimensional point
given code for a 2-dimensional point.

| TwoDPoint.h       |
| TwoDPoint.cpp     |
| ThreeDPoint.h     |
| ThreeDPoint.cpp   |
| TestPoint.cpp     |

Here is code for a primitive implementation of a B-tree.

| BTreeNode.h       |
| BTreeInternal.cpp |
| BTreeLeaf.cpp     |
| BTree.h           |
| BTree.cpp         |