# Classes Revisited: Templates and Inheritance

## 1 Introduction

The purpose of these notes is to introduce basic concepts of templates and inheritance, not to explain either topic in an exhaustive way. Templates are a complex subject and inheritance is even more complex. We begin with concepts about the different types of constructors and destructors, then move on to templates, and then move into an introduction to inheritance.

## 2 Constructors, Copy Constructors, and Destructors

This is an overview of the basics of constructors and destructors and things related to them. First, a refresher about constructors:

- Constructors have no return values and have the same name as the class. They cannot contain a return statement.
- Constructors can have arguments.
- A **default constructor** is a constructor that can be called with no arguments. If a class does not have a user-supplied default constructor, the compiler generates one at compile time. We call this a **compiler-generated constructor**.
- If a class must initialize its data members, then the class needs a user-defined constructor because the compiler-generated constructor will not be able to do this.
- A class should always have a default constructor, i.e., one that requires no arguments.

### 2.1 Copy Constructors

A **copy constructor** is a kind of constructor for a class. It is invoked in the following situations:

1. When an object needs to be constructed because of a declaration with initialization from another object of a type that can be converted to the object's class, as in

   ```
   ScreenData C;
   ScreenData B = C; // copy constructor called here
   ScreenData B(C);  // copy constructor called here
   ```

2. When an object is passed by value to a function. For example, if the function `redraw_screen` has the signature

   ```
   void redraw_screen( ScreenData S );
   ```

   and it is used in the following code snippet:

   ```
   ScreenData new_screen;
   // code to fill in new_screen
   redraw_screen(new_screen);
   ```

then `new_screen` will be copied into `S`.

3. When an object is returned by value from a function. If the function `get_old_screen()` has the signature

   ```
   ScreenData get_old_screen( ScreenData S);
   ```

   then with the call

   ```
   old_screen = get_old_screen(C);
   ```

   the `ScreenData` object returned by `get_old_screen()` will be copied into `old_screen` by the copy constructor.

4. *Note that it is not invoked here*:

   ```
   ScreenData screen2 = ScreenData(params);
   ```

   The ordinary constructor is used to create the object `screen2` directly. The right-hand side is not a call to a constructor. The compiler arranges for screen2 to be initialized from a constructor that is given the parameters `params`.

To illustrate, suppose that `Date` is the following class:

```
class Date
{
public:
    Date       ( int y = 1970, int m = 1, int d = 1 );
    Date       ( const Date &date );
    string get ( );
private:
    short year;
    short month;
    short day;
};
```

It has three private members that store the date as a day, month, and year, and three member functions. The first constructor serves as both a default constructor and a constructor that can be supplied values for the private members. The second is a copy constructor. A copy constructor has a single argument which is a reference to an object of the class. We would provide a definition of this copy constructor such as

```
Date::Date  ( const Date &date )
{
    year = date.year;
    month = date.month;
    day = date.day;
}
```

Because the parameter is not being changed, it is always safer to make it a `const` reference parameter.

## 2.2 Copy Assignment Operators

The copy constructor is called only when an object is being created for the first time, not when it already exists and is being assigned a new value. In this case the **copy assignment operator** is invoked. The *operator=* is the *copy assignment* operator. It is invoked when one object is assigned to an existing object.

Unlike constructors, the copy assignment operator must return a value, and not just any value: it must return the object on which it is invoked. For example, if we were to add a copy assignment operator to the `Date` class, it would be defined as follows:

```
Date Date::operator=  ( const Date &date )
{
    this->year = date.year;
    this->month = date.month;
    this->day = date.day;
    return *this;
}
```

This function returns a `Date` object by value, not reference. It first copies the values out of the argument passed to it, into the object on which it is called. It then returns this object. The value of any assignment operation is always the value assigned to the left-hand side of the assignment; returning the object is required to maintain this semantics.

## 2.3 Destructors

A **destructor** is called when an object goes out of scope or is deleted explicitly by a call to `delete()`. The compiler supplies destructors if the user does not. The reason to supply a destructor is to remove memory allocated by a call to `new()`, to close open files, and so on. If your object does not dynamically allocate memory, directly or indirectly, you do not have to write a destructor for the class; it is sufficient to let the compiler create one for you.

The form of a destructor definition is similar to that of a constructor, except that

- it has a leading tilde ('~'), and
- it has no parameters.

As with constructors, it has no return type.

Some, but not all, of the situations in which a destructor is called are

- when the program terminates, for objects with static storage duration (e.g. global variables or static variables of functions),
- when the block in which an object is created exits, for objects with automatic storage duration (i.e., for local variables of a function when the function terminates),
- when `delete()` is called on an object created with `new()`.

The following listing is a program that writes messages to standard output showing when the constructors and destructors are called. Running it is a good way to explore what the lifetimes of various types of objects are in C++. The class and main program are in a single file here, for simplicity.

```
#include <string>
#include <iostream>
using namespace std;

class MyClass
{
public:
   MyClass( int id=0, string mssge = "" );  //constructor
   ~MyClass();                               // destructor

private:
   int    m_id;       // stores object's unique id
   string m_comment;  // stores comment about object
};  // end of MyClass definition

// Constructor and destructor definitions
MyClass::MyClass( int id, string comment )
{
    m_id = id;
    m_comment = comment;

    cout << "    MyClass CONSTRUCTOR: id = " << m_id
         << ": " << m_comment << endl;
}

MyClass::~MyClass()
{
    cout << "    MyClass DESTRUCTOR:  id = " << m_id
         << ": " << m_comment << endl;
}

/****************************** main program file ***********************/

MyClass Object1(1, "global, static variable");

void foo();
void bar();

int main()
{
    cout << "main() started.\n";

    MyClass Object2(2, "local automatic in main()");
    static MyClass Object3( 3, "local static in main()");

    cout << "calling foo() ...\n";
    foo();
    cout << "foo() returned control to main() ...\n";
    cout << "main() is about to execute its return statement\n";
    return 0;
}

void foo()
{
```

```
    cout << "foo() started.\n";

    MyClass        Object5(5, "local automatic in foo()");
    static MyClass Object6(6, "local static in foo()");

    cout << "foo() is calling bar() ...\n";
    bar();
    cout << "bar() returned control to foo().\n";
    cout << "foo() ended.\n";
}

void bar()
{
    cout << "bar() started.\n";
    MyClass        Object7(7, "local automatic in bar()");
    static MyClass Object8(8, "local static in bar()");
    cout << "bar() ended.\n";
}
```

# 3 Templates

One difference between C and C++ is that C++ allows you to define templates for both classes and functions. It is easier to understand class templates if you first understand function templates, and so we start with these.

Suppose that in the course of writing many, many programs, you find that you need a swap function here and there. Sometimes you want a swap function that can swap two integers, sometimes you want one that can swap two doubles, and sometimes you need to swap objects of a class. In C, you have to write a swap function for each type of object, or you can reduce the work by writing something like this[1]:

```
typedef int elementType;

void swap ( elementType *x, elementType *y)
{
    elementType temp = *x;
    *x = *y;
    *y = temp;
}
```

and you would call this with a call such as

```
int a, b;
a = ... ; b = ... ;
swap(&a, &b);
```

In C, the parameters need to be pointers to the variables to be swapped, and their address must be passed. If you wanted to swap doubles, you would change the `typedef` by replacing the word "`int`" by "`double`."

In C++, you could do the same thing using reference parameters:

[1] There are other methods as well, but these are the two principal approaches.

```
typedef int elementType;

void swap ( elementType &x, elementType &y)
{
    elementType temp = x;
    x = y;
    y = temp;
}
```

and you could call this with code such as

```
int a, b;
a = ... ; b = ... ;
swap(a, b);
```

Although you do not have to write a separate swap function for each different element type, it is inconvenient. The C++ language introduced function templates as a way to avoid this.

## 3.1 Function Templates

A **function template** is a template for a function. It is not an actual function, but a template from which the compiler can create a function if and when it sees the need to create one. This will be clarifed shortly. A template for the swap function would look like this:

```
template <class elementType>
void swap ( elementType &x, elementType &y)
{
    elementType temp = x;
    x = y;
    y = temp;
}
```

The word "class" in the template syntax has nothing to do with classes in the usual sense. It is just a synonym for the word "type." All types in C++ are classes. The syntax of a (single-parameter) function template definition is

```
template <class type_parameter> function-definition
```

where function-definition is replaced by the body of the function, as `swap()` above demonstrates. The syntax for a (single-parameter) function template declaration (i.e., prototype) is

```
template <class type_parameter > function-declaration
```

You need to repeat the line

```
template <class type_parameter>
```

before both the declaration and the definition. For example:

```
// Declare the function template prototype
template <class T>
void swap( T & x, T & y );

int main()
{
     int n= 5;
     int m= 8;
     char ch1 = 'a', ch2 = 'b';
     // more stuff here
     swap(n,m);
     swap(ch1, ch2);
     // ...
}

// Define the function template declared above
template <class T>
void swap( T & x, T & y )
{
    T temp = x;
    x = y;
    y = temp;
}
```

You will often see just the letter "T" used as the type parameter in the template.

When the compiler compiles the main program above, it sees the first call to a function named `swap`. It is at that point that it has to create an instance of a function from the template. It infers from the types of its parameters that the type of the template's parameter is `int`, and it creates a function from the template, replacing the type parameter by `int`. When it sees the next call, it creates a second instance whose type is `char`.

Because function templates are not functions, but just templates from which the compiler can create functions, there is a bit of a problem with projects that are in multiple files. If you want to put the function prototype in a header file and the function definition in a separate `.cpp` file, the compiler will not be able to compile code for it in the usual way if you use that function in a program. To demonstrate, suppose that we create a header file with our swap function prototype, an implementation file with the definition, and a main program that calls the function.

This is `swap.h`:

```
#ifndef SWAP_H
#define SWAP_H

template <class T>
void swap( T &x, T &y);
#endif
```

and `swap.cpp`:

```
template <class T>
void swap( T &x, T &y)
{
    T temp = x;
    x = y;
```

```
    y = temp;
}
```

and `main.cpp`:

```
#include "swap.h"
int main ()
{
    int a = 10, b = 5;
    swap(a,b);
    return 0;
}
```

When we run the command

```
g++ -o demo swap.cpp main.cpp
```

we will see the error message

```
/tmp/ccriQBJX.o: In function ‘main’:
main.cpp:(.text+0x29): undefined reference to ‘void swap<int>(int&, int&)’
collect2: ld returned 1 exit status
```

This is because the function named `swap` does not really exist when `main` is compiled. It has a reference only to a function template. The solution is to put the function template implementation into the header file, as unsatisfying as that is because it breaks the wall that separates interface and implementation. This can be accomplished with an `#include` directive:

```
#ifndef SWAP_H
#define SWAP_H

template <class T>
void swap( T &x, T &y);

#include “swap.cpp”
#endif
```

The general rule then, is to put the function template prototypes into the header file, and at the bottom, include the implementation files using an `#include` directive. There will be no problem with multiply-defined symbols in this case when you compile the code.

*Note.* Function templates, and templates in general, can have multiple parameters, and they do not have to be classes, but that is a topic beyond the scope of this introduction. You may also see the word `typename` used in place of the word `class` as the type of the template parameter. For the most part, these are interchangeable, but it is better to use `class` until you know the subtle difference. The interested reader can refer to a good C++ book for the details.

## 3.2 Class Templates

Imagine that you want to implement a list class, such as the one we described in the introduction to this course. If you go back and look at the list ADT, you will find nothing in it that is specific to any particular type of data object, other than the ability to copy objects. For the sorted list ADT, the objects did have to

be comparable to each other in some linear ordering, but that was about it, in terms of specific properties. It stands to reason that you should be able to create a generic sort of list, one whose definition does not depend on the underlying element type. This is one reason that C++ allows you to create templates for classes as well. A class template is like a generic description of that class that can be instantiated with different underlying data types.

### Defining Class Templates

As with function templates, a C++ class template is not a class, but a *template for a class*. An example of a simple class template interface is

```
template <class T>
class Container
{
public:
    Container();
    Container( T initial_data);
    void set( T new_data);
    T get() const;
private:
    T mydata;
};
```

Notice that a class template begins with the `template` keyword and template parameter list, after which the class definition looks the same as an ordinary class definition. The only difference is that it uses the type parameter `T` from the template's parameter list. The syntax for the implementations of the class template member functions when they are outside of the interface is a bit more complex. The above functions would have to be defined as follows:

```
template <class T>
void Container<T>::set ( T new_data )
{
    mydata = new_data;
}

template <class T>
T Container<T>::get() const
{
    return mydata);
}
```

Notice the following:

1. Each member function is actually a function template definition.
2. All references to the class are to `Container<T>` and not just Container. Thus, the name of each member function must be preceded by `Container<T>::`.

In general the syntax for creating a class template is

```
template <class T>  class class_name {  class_definition  };
```

and a member function named `foo` would have a definition of the form

```
template <class T>
return_type class_name<T>::foo ( parameter_list ) { function definition }
```

### Declaring Objects

To declare an object of a class that has been defined by a template requires , in the simplest case, using a syntax of the form

```
class_name<typename> object_name;
```

as in

```
Container<int>    int_container;
Container<double> double_container;
```

If the `Container` class template had a constructor with a single parameters, the declarations would instead be something like

```
Container<int>    int_container(1);
Container<double> double_container(1.0);
```

The following is a complete listing of a very simple program that uses a class template.

Listing 1: A program using a simple class template.

```
#include <iostream>
using namespace std;

template <class T>
class MyClass
{
  public:
      MyClass( T initial_value);
      void set( T x) ;
      T get( )    ;

  private:
      T val;
};

template < class T >
MyClass < T >:: MyClass (T initial_value)
{
    val = initial_value;
}

template < class T >
void MyClass < T >:: set (T x)
{
    val = x;
}

template < class T >
T MyClass < T >::get ()
{
    return val;
}
```

```
int main ()
{
    MyClass<int>      intobj(0);
    MyClass<double>   floatobj(1.2);

    cout << "intobj value = " << intobj.get()
         << " and floatobj value = "  << floatobj.get() << endl;
    intobj.set(1000);
    floatobj.set (0.12345);
    cout << "intobj value = " << intobj.get()
         << " and floatobj value = "  << floatobj.get() << endl;

    return 0;
}
```

Again, remember that *a class template is not an actual definition of a class, but of a template for a class.* Therefore, if you put the implementation of the class member functions in a separate implementation file, which you should, then you must put an `#include` directive at the bottom of the header file of the class template, including that implementation file. In addition, make sure that you do not add the implementation file to the project or compile it together with the main program. For example, if `myclass.h`, `myclass.cpp`, and `main.cpp` comprise the program code, with `myclass.h` being of the form

```
#ifndef MYCLASS_H
#define MYCLASS_H

// stuff here

#include "myclass.cpp"
#endif // MYCLASS_H
```

and if `main.cpp` includes `myclass.h`, then the command to compile the program must be

```
g++ -o myprogram main.cpp
```

*not*

```
g++ -o myprogram myclass.cpp main.cpp
```

because the latter will cause errors like

```
myclass.cpp:4:6: error: redefinition of 'void MyClass<T>::set(T)'
myclass.cc :4:6: error: 'void MyClass<T>::set(T)' previously declared here
```

This is because the compiler will compile the `.cpp` file twice! This is not a problem with function templates, but it is with classes, because classes are turned into objects.

# 4 Inheritance

Inheritance is a feature that is present in many object-oriented languages such as C++, Eiffel, Java, Ruby, and Smalltalk, but each language implements it in its own way. This chapter explains the key concepts of the C++ implementation of inheritance.

## 4.1 Deriving Classes

Inheritance is a feature of an object-oriented language that allows classes or objects to be defined as extensions or specializations of other classes or objects. In C++, classes inherit from other classes. Inheritance is useful when a project contains many similar, but not identical, types of objects. In this case, the task of the programmer/software engineer is to find commonality in this set of similar objects, and create a class that can serve as an archetype for this set of classes.

### Examples

- Squares, triangles, circles, and hexagons are all 2D shapes; hence a `Shape` class could be an archetype.
- Faculty, administrators, office assistants, and technical support staff are all employees, so an `Employee` class could be an archetype.
- Cars, trucks, motorcycles, and buses are all vehicles, so a `Vehicle` class could be an archetype.

When this type of relationship exists among classes, it is more efficient to create a class hierarchy rather than replicating member functions and properties in each of the classes. Inheritance provides the mechanism for achieving this.

### Syntax

The syntax for creating a derived class is very simple. (You will wish everything else about it were so simple though.)

```
class A
{/* ... stuff here ... */};

class B: [access-specifier] A
{/* ... stuff here ... */};
```

in which an *access-specifier* can be one of the words, `public`, `protected`, or `private`, and the square brackets indicate that it is optional. If omitted, the inheritance is `private`.

In this example, `A` is called the **base class** and `B` is the **derived class**. Sometimes, the base class is called the **superclass** and the derived class is called a **subclass**.

### Five Important Points (regardless of access specifier):

1. The constructors and destructors of a base class are not inherited.
2. The assignment operator is not inherited.
3. The friend functions and friend classes of the base class are not inherited.
4. The derived class does not have access to the base class's private members.
5. The derived class has access to all public and protected members of the base class.

Public inheritance expresses an **is-a** relationship: a B *is a* particular type of an A, as a car *is a* type of vehicle, a manager *is a* type of employee, and a square *is a* type of shape.

Protected and private inheritance serve different purposes from public inheritance. Protected inheritance makes the public and protected members of the base class protected in the derived class. Private inheritance makes the public and protected members of the base class private in the derived class. These notes do not discuss protected and private inheritance.

**Example**

In this example, a `Shape` class is defined and then many different kinds of shapes are derived from it.

```
class Shape {
private:
    Point Centroid;
public:
    void Move(Point newCentroid); // move Shape to new centroid
    /* more stuff here */
};

class Square   : public Shape {/* stuff here */ };
class Triangle : public Shape {/* stuff here */ };
class Hexagon  : public Shape {/* stuff here */ };
/* and so forth */
```

## 4.2 Implicit Conversion of Classes

The C++ language allows certain assignments to be made even though the types of the left and right sides are not identical. For example, it will allow an integer to be assigned to a floating-point type without an explicit cast. However, it will not in general let a pointer of one type be assigned to a pointer of another type. One exception to this rule has to do with assigning pointers to classes. I will begin this section by stating its conclusion. If you do not want to understand why it is true or get a deeper understanding of the nature of inheritance, you can then just skip to the next section.

**Implicit Conversion of Classes**: *The address of an object of a derived class can be assigned to a pointer declared to the base class, but the base class pointer can be used only to invoke member functions of the base class.*

Because a `Square` is a specific type of `Shape`, a `Square` is a `Shape`. But because a `Square` has other attributes, a `Shape` is not necessarily a `Square`. But consider a `Shape*` pointer. A `Shape*` is a pointer to a `Shape`. A `Square*` is a pointer to a `Square`. A `Square*` is not the same thing as a `Shape*`, since one points to `Squares` and the other points to `Shapes`, and so they have no inherent commonality other than their "pointerness."

However, since a `Square` is also a `Shape`, a `Square*` can be used wherever a `Shape*` can be used. In other words, `Squares`, `Triangles`, and `Hexagons` are all `Shapes`, so whatever kinds of operations can be applied to `Shapes` can also be applied to `Squares`, `Triangles`, and `Hexagons`. Thus, it is reasonable to be able to invoke any operation that is a member function of the `Shape` class on any dereferenced `Shape*`, whether it is a `Square`, a `Triangle`, or a `Hexagon`. This argument explains why, in C++, the address of any object derived from a `Shape` can be assigned to a `Shape` pointer; e.g., a `Square*` can be assigned to a `Shape*`.

The converse is not true; a `Shape*` cannot be used wherever a `Square*` is used because a `Shape` is not necessarily a `Square`! Dereferencing a `Square*` gives access to a specialized set of operations that only work on `Squares` and cannot be applied to arbitrary shapes. If a `Square*` contained the address of a `Shape` object, then after dereferencing the `Square*`, you would be allowed to invoke a member function of a `Square` on a `Shape` that does not know what it is like to be a `Square`, and that would make no sense. So this cannot be allowed.

The need to make the preceding argument stems from the undecidability of the **Halting Problem** and the need for the compiler designer to make sensible design decisions. If you are not familiar with the Halting Problem, you can think of it as a statement that there are problems for which no algorithms exist. One consequence of the Halting Problem is that it is not possible for the compiler to know whether or not the address stored in a pointer is always going to be the address of any specific object. To illustrate this, consider the following code fragment, and assume that the `Square` class is derived from the `Shape` class.

```
1. Square* pSquare;
2. Shape* pShape;
3. void* ptr;
4. Shape someShape;
5. Square someSquare;
6. /* .. ..... */
7. if ( some condition that depends on user input )
8.     ptr = (void*) &someShape;
9. else
10.    ptr = (void*) &someSquare;
11. pShape = (Shape*) ptr;
```

The compiler cannot tell at compile time whether the true or false branch of the if-statement will always be taken, ever be taken, or never taken. If it could, we could solve the Halting Problem. Thus, at compile-time, it cannot know whether the assignment in line 11 will put the address of a `Square` or a `Shape` into the variable `pShape`. Another way to say this is that, at compile-time, the compiler cannot know how to bind an object to a pointer. The designer of C++ had to decide what behavior to give to this assignment statement. Should it be allowed? Should it be a compile time error? If it is allowed, what operations can then be performed on this pointer?

The only sensible and safe decision is to allow the assignment to take place, but to play it "safe" and allow the dereferenced `pShape` pointer to access only the member functions and members of the `Shape` class, not any derived class, because those operations are available to both types of objects.

## 4.3 Multiple Inheritance

Suppose that the shapes are not geometric abstractions but are instead windows in an art-deco building supply store. Then what they also have in common is the property of being a window, assuming they are all the same type of window (e.g., double hung, casement, sliding, etc.). Then geometric window shapes really inherit properties from different archetypes, i.e., the property of being a shape and the property of being a window. In this case we need multiple inheritance, which C++ provides:

**Example**

```
class Shape {
private:
    Point Centroid;
public:
    void Move(Point newCentroid); // move Shape to new centroid
    /* more stuff here about being a Shape */
};
class Window
{
   /* stuff here about being a Window */
};
class SquareWin   : public Shape, public Window {/* stuff here */ };
class TriangleWin : public Shape, public Window {/* stuff here */ };
class HexagonWin  : public Shape, public Window {/* stuff here */ };
/* and so forth */
```

Note the syntax. The derived class is followed by a single colon (:) and a comma-separated list of base classes, with the inheritance qualifier (`public`) attached to each base class. The set of classes created by the above code creates the hierarchy depicted in Figure 1.

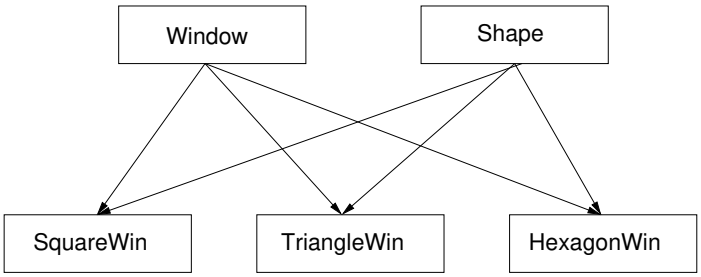



Figure 1: Multiple inheritance.

## 4.4 Extending Functionality of Derived Classes with Member Functions

Inheritance would be relatively useless if it did not allow the derived classes to make themselves different from the base class by the addition of new members, whether data or functions. The derived class can add members that are not in the base class, making it like a subcategory of the base class.

### Example

The `Rectangle` class below will add a member function not in the base class.

```
class Shape
{
private:
    Point Centroid;
public:
    void Move(Point newCentroid); // move Shape to new centroid
    Point getCentroid() const;
    /* more stuff here */
};

class Rectangle : public Shape
{
private:
    float length, width;
public:
    /* some stuff here too */
    float LengthDiagonal() const;// functionality not in Shape class
};
```

The `Rectangle` class can add a `LengthDiagonal` function that was not in the base class since it did not makes sense for it to be in the base class and only the `Rectangle` class has all diagonals of the same length. Remember though that the member functions of the derived class cannot access the private members of the base class, so the base class must either make protected or public accessor functions for its subclasses if they need access to the private members.

## 4.5 Redefining Member Functions in Derived Classes (Overriding)

Derived classes can redefine member functions that are declared in the base class. This allows subclasses to specialize the functionality of the base class member functions for their own particular needs. For example, the base class might have a function `print()` to display the fields of an employee record that are common to

all employees, and in a derived class, the `print()` function might display more information than the `print()` function in the base class.

*Note.* A function in a derived class ***overrides*** a function in the base class only if its signature is identical to that of the base class except for some minor differences in the return type. It must have the same parameters and same qualifiers. Thus,

```
void print();
void print() const;
```

are not the same and the compiler will treat them as different functions, whereas

```
void print() const;
void print() const;
```

are identical and one would override the other. Continuing with the `Shape` example, suppose that a `Shape` has the ability to print only its `Centroid` coordinates, but we want each derived class to print out different information. Consider the `Rectangle` class again, with a `print()` member function added to `Shape` and `Rectangle` classes.

```
class Shape
{
private:
    Point Centroid;
public:
    void Move(Point newCentroid); // move Shape to new centroid
    Point getCentroid() const;
    void print() const; // prints just Centroid coordinates
    /* more stuff here */
};

class Rectangle : public Shape
{
private:
    float length, width;
public:
    /* some stuff here too */
    float LengthDiagonal() const;// functionality not in Shape class
    void print() const
    { /* print stuff that Rectangle class has here */ }
};

/* .... */
Shape     myShape;
Rectangle myRect;
myRect.print();
myShape.print();
```

The call to `myRect.print()` will invoke the `print()` member function of the `Rectangle` class, since `myRect` is bound to a `Rectangle` at compile time. Similarly, the call to `myShape.print()` will invoke the `Shape` class's `print()` function. But what happens here:

```
Shape* pShape;
pShape = new Rectangle;
pShape->print();
```

In this case, the address of the dynamically allocated, anonymous `Rectangle` object is assigned to a `Shape` pointer, and referring back to Section 4.2 above, the dereferenced `pShape` pointer will point to the `print()` member function in the `Shape` class, since the compiler binds a pointer to the member functions of its own class. Even though it points to a `Rectangle` object, `pShape` cannot invoke the `Rectangle`'s `print()` function. This problem will be overcome below when virtual functions are introduced.

# 5 Revisiting Constructors and Destructors

## 5.1 Constructors

Let us begin by answering two questions.

***When does a class need a constructor?***

If a class must initialize its data members, then the class needs a user-defined constructor because the compiler-generated constructor will not be able to do this.

***If a class is derived from other classes, when does it need a user-defined constructor?***

To understand the answer, it is necessary to understand what happens at run time when an object is created. Class objects are always constructed from the bottom up, meaning that the lowest level base class object is constructed first, then any base class object immediately derived from that is constructed, and so on, until the constructor for the derived class itself is called. To make this concrete, suppose that four classes have been defined, and that they form the hierarchy depicted in Figure 2, in which `D` is derived from `C`, which is derived from `B`, which is derived from A.


Figure 2: Class hierarchy.

Then when an object of class `D` is created, the run time system will recursively descend the hierarchy until it reaches the lowest level base class (`A`), and construct `A`, then `B`, then `C`, and finally `D`.

From this discussion, it should be clear that a constructor is required for every class from which the class is derived. If a base class's constructors require arguments, then there must be a user-supplied constructor for that class, and any class derived from it must explicitly call the constructor of the base class, supplying the arguments to it that it needs. If the base class has at least one default constructor, the derived class does not need to call it explicitly, because the default constructor can be invoked implicitly as needed. In this case, the derived class may not need a user-defined constructor, because the compiler will arrange to have the run time system call the default constructors for each class in the correct order. But in any case, when a derived class object is created, a base class constructor must always be invoked, whether or not the derived class has a user-defined constructor, and whether or not it requires arguments. The short program that follows demonstrates the example described above.

**Example**

```
class A  {
public:
    A() {cout << "A constructor called\n";}
};

class B : public A  {
public:
    B() {cout << "B constructor called\n";}
};

class C : public B  {
public:
    C() {cout << "C constructor called\n";}
};

class D : public C  {};
void main() {
    D  d;
}
```

This program will display

```
A constructor called
B constructor called
C constructor called
```

because the construction of `d` implicitly requires that `C`'s constructor be executed beforehand, which in turn requires that `B`'s constructor be executed before `C`'s, which in turn requires that `A`'s constructor be executed before `B`'s. To make this explicit, you would do so in the initializer lists:

```
class A {
public:
    A() {}
};

class B : public A  {
public:
    B(): A() {}
};

class C : public B  {
public:
    C(): B() {}
};

class D : public C  {
public:
    D(): C() {}
};
```

This would explicitly invoke the `A()`, then `B()`, then `C()`, and finally `D()` constructors.

**Summary**

The important rules regarding constructors are:

- A base class constructor is ***ALWAYS*** called if an object of the derived class is constructed, even if the derived class does not have a user-defined constructor.
- If the base class does not have a default constructor, then the derived class must have a constructor that can invoke the appropriate base class constructor with arguments.
- The constructor of the base class is invoked before the constructor of the derived class.
- If the derived class has members in addition to the base class, these are constructed after those of the base class.

## 5.2 Destructors

Destructors are slightly more complicated than constructors. The major difference arises because destructors are rarely called explicitly. They are invoked for only one of two possible reasons:

1. The object was not created dynamically and execution of the program left the scope containing the definition of the class object, in which case the destructors of all objects created in that scope are implicitly invoked, or
2. The `delete` operator was invoked on a class object that was created dynamically, and the destructors for that object and all of its base classes are invoked.

**Notes**

- When a derived class object must be destroyed, for either of the two reasons above, it will always cause the base class's destructor to be invoked implicitly.
- Destructors are always invoked in the reverse of the order in which the constructors were invoked when the object was constructed. In other words, the derived class destructor is invoked before the base class destructor, recursively, until the lowest level base class destructor is called.
- If a class used the `new` operator to allocate dynamic memory, then the destructor should release dynamic memory by called the `delete` operator explicitly.
- From the preceding statements, it can be concluded that the derived class releases its dynamic memory before the classes from which it was derived.

To illustrate with an example, consider the following program. The derived class has neither a constructor nor a destructor, but the base class has a default constructor and a default destructor, each of which prints a short message on the standard output stream.

```
class A
{
public:
    A()  {cout << "base constructor called.\n";}
    ~A() {cout << "base destructor called.\n";}
};

class B : public A  // has no constructor or destructor
```

```
{ };

void main()
{
    B* pb;       // pb is a derived class pointer
    pb = new B;  // allocate an object of the derived class
    delete pb;   // delete the object
}
```

This program will display two lines of output:

```
base constructor called.
base destructor called.
```

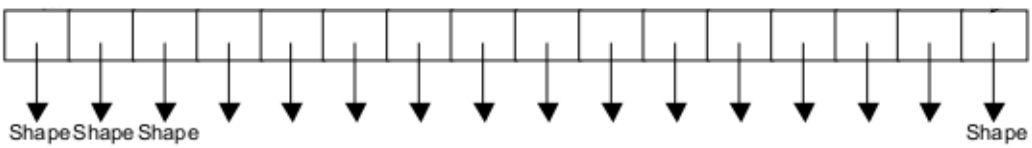
This confirms that the base class destructor and constructor are called implicitly.

# 6 Virtual Functions

I stressed above that the compiler always binds pointers to the member functions of the class to which they are declared to point to in their declarations (i.e., at compile time.) This is not a problem if derived objects are never created dynamically. In this case, inheritance is really only buying a savings in the amount of code needed in the program; it does not give the language polymorphism. ***Polymorphism*** exists when objects can alter their behavior dynamically. This is the reason for the virtual function specifier, "`virtual`."

### Example

Suppose that you have an application that draws different types of shapes on the screen. The number and type of shapes that may appear on the screen will vary over time, and you decide to create an array to store and process them. The array will therefore need to access `Shape` objects of all kinds. Since you do not know in advance which cells of the array will access which objects, the array must be able to change what it can hold dynamically, i.e., at run time. If the array element type is `Shape*`, then each array element can point to a `Shape` of a different class. However, because the pointers are of type `Shape*`, the program will only be able to access the member functions of the `Shape` class through this pointer, not the member functions of the derived classes. The following figure illustrates this idea.




What is needed is a way to allow the pointer to the base class to access the members of the derived class. This is the purpose of a **virtual function**. Declaring a function to be virtual in the base class means that if a function in a derived class overrides it and a base class pointer is dereferenced, that pointer will access the member function in the derived class. To demonstrate, consider the following program.

```
class CBase
{
public:
virtual void print()
{ cout<< "Base class print() called. " << endl; }
};

class CDerived : public CBase
{
public:
void print()
{ cout<< "Derived class print function called." << endl; }
};

void main()
{
CBase*  baseptr;
baseptr = new CDerived;
baseptr->print();
}
```

When this program is run, even though the type of `baseptr` is `CBase*`, the function invoked by the dereference "`baseptr->print()`" will be the `print()` function in the derived class, `CDerived`, because the run time environment bound `baseptr->print()` to the derived object when it was assigned a pointer of type `CDerived` and it knew that `print()` was virtual in the base class.

## 6.1 Virtual Destructors and Constructors

Constructors cannot be virtual. Each class must have its own constructor. Since the name of the constructor is the same as the name of the class, a derived class cannot override it. Furthermore, constructors are not inherited anyway, so it makes little sense.

On the other hand, destructors are rarely invoked explicitly and surprising things can happen in certain circumstances if a destructor is not virtual. Consider the following program.

```
class CBase {
public:
CBase()
{ cout << "Constructor for CBase called." << endl;}
~CBase()
{ cout << "Destructor for CBase called." << endl;}
};

class CDerived: public Cbase {
public:
CDerived()
{ cout << "Constructor for CDerived called." << endl;}
~CDerived()
{ cout << "Destructor for CDerived called." << endl;}
};

void main() {
CBase *ptr = new CDerived();
```

```
delete ptr;
}
```

When this is run, the output will be

```
Constructor for CBase called.
Constructor for CDerived called.
Destructor for CBase called.
```

The destructor for the `CDerived` class was not called because the destructor was not declared to be a virtual function, so the call "`delete ptr`" will invoke the destructor of the class of the pointer itself, i.e., the `CBase` destructor. This is exactly how non-virtual functions work. Now suppose that we make the destructor in the base class virtual. Even though we cannot actually override a destructor, we still need to use the virtual function specifier to force the pointer to be bound to the destructor of the most-derived type, in this case `CDerived`.

```
class CvirtualBase {
public:
CVirtualBase()
{ cout << "Constructor for CVirtualBase called." << endl; }
virtual ~CVirtualBase()  // THIS IS A VIRTUAL DESTRUCTOR!!!
{ cout << "Destructor for CVirtualBase called." << endl; }
};

class CDerived: public CvirtualBase {
public:
CDerived()
{ cout << "Constructor for CDerived called." << endl; }
~CDerived()
{ cout << "Destructor for CDerived called." << endl; }
};

void main() {
CVirtualBase *ptr = new CDerived();
delete ptr;
}
```

The output of this program will be:

```
Constructor for CVirtualBase called.
Constructor for CDerived called.
Destructor for CDerived called.
Destructor for CVirtualBase called.
```

This is the correct behavior.

In summary, a class `C` must have a virtual destructor if both of the following conditions are true:

- A pointer `p` of type `C*` may be used as the argument to a `delete` call, and
- It is possible that this pointer may point to an object of a derived class.

There are no other conditions that need to be met. You do not need a virtual destructor if a derived class destructor is called because it went out of scope at run time. You do not need it just because the base class has some virtual functions (which some people will tell you.)

### 6.2 Pure Virtual Functions

Suppose that we want to create an `Area()` member function in the `Shape` class. This is a reasonable function to include in this base class because every closed shape has area. Every class derived from the `Shape` class can override this member function with its own area function, designed to compute the area of that particular shape. However, the `Area()` function in the base class has no implementation because a `Shape` without any particular form cannot have a function that can compute its area. This is an example of a pure virtual function. A **pure virtual function** is one that has no possible implementation in its own class.

To declare that a virtual function is **pure**, use the following syntax:

```
virtual return-type function-name( parameter-list) = 0;
```

For example, in the `Shape` class, we can include a pure `Area()` function by writing

```
class Shape
{
private:
     Point Centroid;
public:
    void Move(Point newCentroid);
    Point getCentroid() const;
    virtual double Area() = 0; // pure Area() function
};
```

## 7 Abstract Classes

A class with at least one pure virtual function cannot have any objects that are instances of it, because at least one function has no implementation. Such a class is called an **abstract class**. In contrast, a class that can have objects is called a **concrete class**. An abstract class can serve as a class interface that can have multiple implementations, by deriving classes from it that do not add any more functionality but provide implementations of the pure virtual functions. It can also serve as an abstraction that is extended in functionality by deriving more specific classes from it.

### 7.1 Abstract Classes as Interfaces

An abstract class can act like a class interface, without divulging the "secret implementation." The following code demonstrates this idea.

```
class List // an abstract List  class
{
public:
   List();  // default constructor
   ~List(); // destructor
   virtual bool is_empty() const = 0;
   virtual int length()    const = 0;
   virtual void insert(int new_position,
           list_item_type new_item, bool& Success) = 0;
   void delete(int position, bool& Success) = 0;
   void retrieve(int position, list_item_type & DataItem,
                  bool& Success) const = 0;
};
```

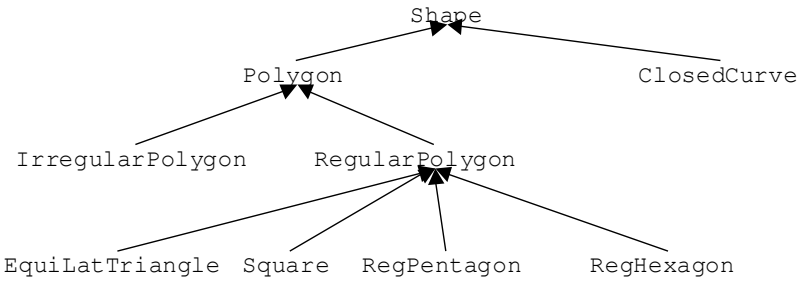



Figure 3: An abstract class hierarchy

We do not have to specify in this abstract class how the `List` is actually represented, such as whether an array stores the list, or a vector, or some linked representation. There is no private data in this class, and it has no implementation. We can derive a new class from it and let the derived class implement it. Any such derived class must conform to the abstract base class's interface, but it is free to add private members.

## 7.2 Abstract Class Hierarchies

When a class is derived from an abstract class and it does not redefine all of the pure virtual functions, it too is an abstract class, because it cannot have objects that represent it. This is often exactly what you want.

Using the `Shape` example again, imagine that we want to build an extensive collection of two-dimensional shapes with the `Shape` class at the root of a tree of derived classes. We can subdivide shapes into polygons and closed curves. Among the polygons we could further partition them into regular and irregular polygons, and among closed curves, we could have other shapes such as ellipses and circles. The class `Polygon` could be derived from the `Shape` class and yet be abstract, because just knowing that a shape is a polygon is still not enough information to implement an `Area` member function. In fact, the `Area` member function cannot be implemented until the shape is pretty much nailed down. This leads to a multi-level hierarchy that takes the shape of a general tree with the property that the interior nodes are abstract classes and the leaf nodes are concrete classes. A portion of that hierarchy is shown in Figure 3.