**Chapter 11: Virtual Functions**

Table of Contents

[Virtual Functions 2](#_Toc49595551)

[Abstract Classes and Pure Virtual Functions 4](#_Toc49595552)

[Virtual Destructors 5](#_Toc49595553)

[Virtual Base Classes 6](#_Toc49595554)

[friend Functions 7](#_Toc49595555)

[friend Classes 9](#_Toc49595556)

[The this Pointer 10](#_Toc49595557)

[Dynamic Type Information 12](#_Toc49595558)

[Dynamic Casting 12](#_Toc49595559)

[The typeid Operator 13](#_Toc49595560)

## Virtual Functions

Say we have two classes, People and Student, and Student inherits from People. If we create an array of pointers of the People class, then we can do this:

People \*ptr[3];  
ptr[0] = new People ("Sam");  
*//object of People class, using constructor*ptr[1] = new Student ("Henry");  
*//object of Student class using constructor*

C++

A pointer of the People class is being used to point to an object of the Student class. This is perfectly valid. A pointer of a parent class can be used to point to an object of a derived class. However, the pointer cannot be used to access any functions or data of the derived class, only those of the parent class. If both of the classes had their own separate display functions, then writing this

ptr[1]->display();

C++

would cause the display function of the People class to run, i.e. People::display()

We can however, allow access to the functions and data of the derived class. To do this, we must edit the display function in the parent class header file like this:

virtual void display();

C++

With this declaration, we have created a virtual function. Now if we write this:

ptr[1]->display();

C++

The display function of the derived class will run, i.e. Student::display().

When we do this, the compiler does not know which function to run until the program is actually running. While the program runs, it is discovered that the pointer points to a certain object and then it is decided that the function for that object should be used. This is called late binding or dynamic binding.

## Abstract Classes and Pure Virtual Functions

We can go one step further and edit the header file of the People class to this:

virtual void display() = 0;

C++

This is called a pure virtual function. If a class contains even one pure virtual function, it becomes an abstract class. An abstract class is not allowed to have any objects and cannot be instantiated. They only exist to provide a base to their derived classes. Any classes that inherit from an abstract class can use the functions of the abstract class, but the pure virtual functions must be overridden. Not overriding them will cause the derived classes themselves to become abstract classes, which means we cannot create objects of those classes either.

## Virtual Destructors

The destructor function for the base class should also be made virtual if virtual functions are being used. If this is not done, the destructor of the base class will run instead of that of the derived class, which may cause problems. If the destructor of the base class is virtual, then both the destructor function of the derived class and of the base class will run.

## Virtual Base Classes

We have previously mentioned a situation called the diamond problem. A class D inherits from two classes B and C, which in turn inherit from a single class A. Both classes B and C have a copy of the data and functions from class A. When class D tries to access data or functions from class A, an error occurs. This is because the compiler does not understand which copy of the data to use. This can be solved in this manner:

class Parent  
{  
public:  
 void display() {cout<<"Parent Class"<<endl;}  
};  
class Child1 : virtual public Parent  
{};  
class Child2 : virtual public Parent  
{};  
class GrandChild : public Child1, public Child2  
{};

C++

Using the keyword virtual while inheriting from Parent causes both Child1 and Child2 to share a single copy of the Parent data and functions. This means that the diamond problem does not occur, and the GrandChild class can use data and functions from the Parent class.

## friend Functions

The concepts of data hiding and encapsulation dictate that non-member functions should not be able to access an object’s private or protected data. However, there may be situations where this can be inconvenient. For example, if a function needs to operate on the private data of objects from two different classes. This is where friend functions come in.

class beta; *// need to declare early*class alpha  
{  
private:  
 int data;  
public:  
 alpha() : data(3) {} *// constructor* friend int someFunction (alpha, beta); *// declaring the function a friend*};  
  
class beta  
{  
private:  
 int data;  
public:  
 beta() : data(7) {} *// constructor* friend int someFunction (alpha, beta); *// can be private/public in both classes*};  
  
int someFunction (alpha a, beta b) *// normal function declaration*{  
 return (a.data + b.data);  
}

C++

Notice how the actual declaration that makes an otherwise normal function a friend function is inside the classes it deals with. This helps protect the privacy of the data, since someone who does not have access to the source code cannot create a friend function and access the private data.

It is strongly recommended to use friend functions sparingly, since it can quickly lead to spaghetti code if overused.

friend functions can also be very useful in situations where the syntax is non-intuitive.

sqft = dist.square() *// calls the square() funciton of the dist object; non-intuitive*sqft = square(dist) *// the square() function as a friend of the dist class; intuitive*

C++

## friend Classes

friend classes are entire classes that can access another class’s private data.

class alpha  
{  
private:  
 int data;  
public:  
 alpha() : data(99) {} *// constructor* friend class beta; *// declaring beta a friend*};  
  
class beta *// can access alpha's private data*{  
public:  
 void someFunction(alpha a) {cout<<a.data<<endl;}  
};

C++

Note that we had to write friend class beta; in the friend class declaration since beta was undeclared at this point. If beta had already been declared, the declaration inside alpha would have been friend beta;

## The this Pointer

The this pointer can be used by members functions of a class to point to the object using the function.

class what  
{  
private:  
 int alpha;  
public:  
 void tester()  
 {  
 this->alpha = 1;  
 }  
};

C++

The above example is a bit useless. A more practical example would be returning values from member functions and overloaded operators.

class alpha  
{  
private:  
 int data;  
public:  
 alpha& operator = (alpha& a) *// overloaded =; returns alpha object* {  
 data = a.data;  
 return \*this; *// returning this object itself* }  
};

C++

Note that the this pointer is not available in static member functions, since they are not associated with any specific object.

Also note that the this pointer, combined with the overloaded = operator, can be used to assign an object to itself. However, this can cause problems, especially if we are deleting the original object after assignment. Thus, we need to check if the same object is being assigned, and if it is, we must return it immediately so as to avoid a program crash.

if (this == &S) *// addresses match* return \*this;

C++

## Dynamic Type Information

It is possible to find out information about an object’s class and even change the class of an object at runtime. These are advanced capabilities, and are usually used in situations where a variety of classes are descended in complicated ways from a base class. One of the capabilities we will see, dynamic casting, relies on the base class being polymorphic, i.e. having at least one virtual function. Both dynamic casting and typeid, the other capability we will see, require the compiler to enable Run-Time Type Information (RTTI). Note that you need to include the typeinfo header file to use these capabilities.

### Dynamic Casting

Supposes when some program sends our program an object, we would want to check if the right type of object has been sent. The dynamic\_cast operator can be used to do this, but only if both object types are descended from the same common ancestor.

bool isDerv (Base\* pUnknown) *// unknown subclass of Base*{  
 Derv1\* pDerv1;  
 if (pDerv1 = dynamic\_cast<Derv1\*>(pUnknown)) return true;  
 else return false  
}

C++

We can also change pointer types using dynamic casting, allowing casts both upwards and downloads in an inheritance tree.

int main()  
{  
 Base\* pBase = new Base(10); *// base value* Derv\* pDerv = new Derv(21, 22); *// base value, child value  
  
 /\* upcasting \*/* pBase = dynamic\_cast<Base\*>(pDerv); *// child pointer converted to base pointer; child loses child value; retains base value  
  
 /\* downCasting \*/* pBase = new Derv(31, 32); *// new values* pDerv = dynamic\_cast<Derv\*>(pBase); *// base pointer converted to child pointer*}

C++

### The typeid Operator

Sometimes, you want more information, like the type of the object, its class name etc. This can be obtained using the typeid operator.

void displayName (Base\* pB)  
{  
 cout<<"Pointer to an object of "<<typeid(\*pB).name()<<endl; *// prints name of class*}

C++