After Mid Lab #1

Submitted by : Hasnain Abbas

Sap Id : 42917

Course : OOP

Section : BSCS – 3

Submitted to : Sir Shehzad Shameer

Report of Lab 1:

Task #1:

In the first task, I created two classes base and derived and the base class is inherit in derived class and created a function in their public and created two header files of cout and endl and then created a base pointer in main body and called its function to print the base class twice. After that, if I add virtual with the function of the base class, then the base class will be printed once and the derived class will also be printed once.

Programm :

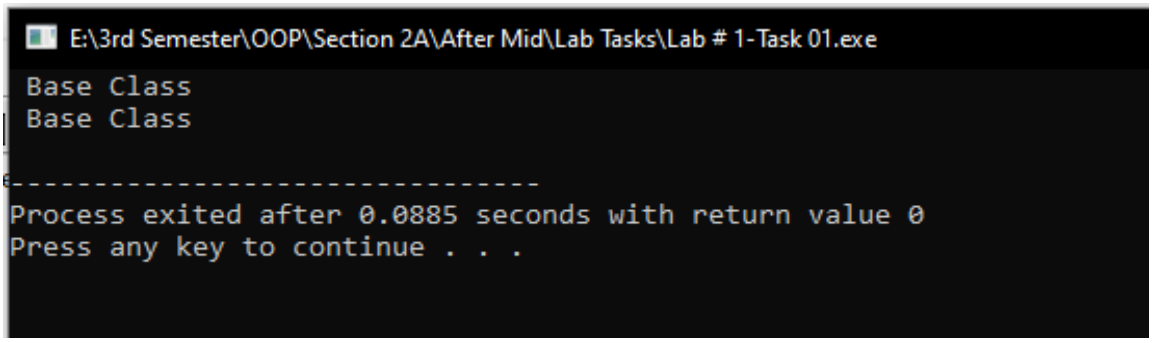
```
//Task 1.
//To gain a better understanding of polymorphic and virtual functions start
```

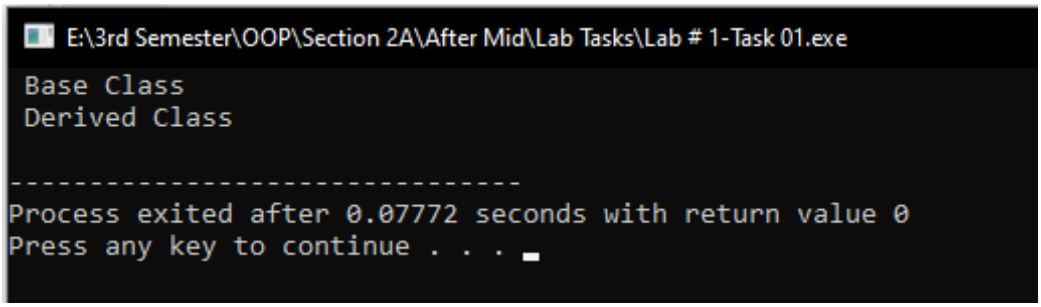
```
//with the following simple example. Notice we have not defined a virtual
//function yet.

//    Task1.h

#include <iostream>

using std::cout;
using std::endl;

class Base{
        public:
                //void testFunction ();
        virtual       void testFunction ();

};
class Derived : public Base{
        public:
```

```
            void testFunction ();

};

//   Task1.cpp

// #include "Task1.h"
void Base::testFunction(){
        cout<<" Base Class " <<endl;
}
void Derived::testFunction(){
        cout<<" Derived Class " <<endl;
}

//        main.cpp

// #include "Task1.h"

int main(void){
```

```
    Base* ptr = new Base;
    ptr -> testFunction();    // prints "Base class"

    delete  ptr;

    ptr =  new Derived;
    ptr -> testFunction();     // print "Base class" because the base
class function is not virtual

    delete ptr;

    return 0;
}
```

Output :





Task #2:

In the second task, create two classes of Memel and Dog. Created a constructor and destructor function in the public of Memel class and created two more virtual functions named move and speak and declared its age in protected. After that, the Memel class was inherited into the Dog class, and the Memel's constructor, move and speak functions were C-outed, and after that, a Dog pointer was created in the int main and its function was called.

Program :

```
//Task 2.

//You will first build two classes, Mammal and Dog. Dog will inherit from
//Mammal. Below is the Mammal class code. Once you have the Mammal class
//built, build a second class Dog that will inherit publicly from Mammal.

//    Mammal.h

#include <iostream>

using std::cout;
using std::endl;
```

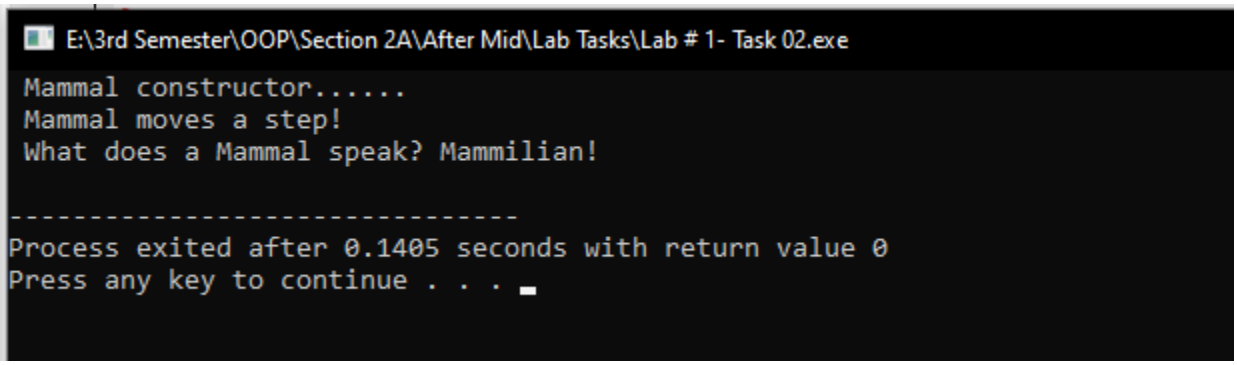
```
class Mammal{
        public:
                Mammal (void);
                ~Mammal (void);

                virtual void Move () const;
                virtual void speak () const;

                protected:

                  int   itsAge;
};
class Dog : public Mammal{

};

//      Mammal.cpp

//    #include "Mammal.h"
```

```
Mammal::Mammal(void):itsAge(1){

        cout<<" Mammal constructor..... " <<endl;
}

Mammal::~Mammal(void){

        cout<<" Mammal destructor..... " <<endl;
}

void Mammal::Move() const{

        cout<<" Mammal moves a step! " <<endl;
}

void Mammal::speak() const{

        cout<<" What does a Mammal speak? Mammilian! " <<endl;
```

```
}

//  #include "Mammal.h"
//  #include "Dog.h"


int main()
{
	Mammal *pDog = new Dog;

	pDog->Move();
	pDog->speak();

//	Dog *pDog2 = new Dog;
//
//	pDog2->Move();
//	pDog2->speak();

	return 0;
```

}

Output :



Task #3:

In the third task, create five classes of Memel Dog , Cat , Horse and Guineapig . Created a constructor and destructor function in the public of Memel class and created two more virtual functions named move and speak and declared its age in protected. After that, the Memel class was inherited into the Dog ,

Cat , Horse and Guineapig class.After that , the constructor , destructor , move and speak functions were C-outed. In main body , create an array of five element of memel pointer and create a for loop and C-outed the 1 Dog , 2 Cat , 3 Horse and 4 Guineapig and the choice from user . After that , create a switch case of choice condition , and the array of for loop is equal to pointer . After that the for loop is shown the array of speak and other for loop is shown the delete array .

Prograam :

```
//     Task 3.
//
//Develop additional classes for Cat, Horse, and GuineaPig overriding the
//move and speak methods. (If you do not know guinea pigs go “weep weep”)
//Next, test with the modified main:
```

```
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

class Mammal{
        public:
                Mammal (void);
                ~Mammal (void);

                virtual void Move () const;
                virtual void speak () const;

                protected:

                  int   itsAge;
};
```

```
class Dog : public Mammal{


};

class Cat : public Mammal{



};

class Horse : public Mammal{


};

class  Guineapig: public Mammal{
```

```
};

//      Mammal.cpp

//     #include "Mammal.h"

Mammal::Mammal(void):itsAge(1){

        cout<<" Mammal constructor..... " <<endl;
}

Mammal::~Mammal(void){

        cout<<" Mammal destructor..... " <<endl;
}

void Mammal::Move() const{
```

```
        cout<<" Mammal moves a step! " <<endl;
}

void Mammal::speak() const{

        cout<<" What does a Mammal speak? Mammilian! " <<endl;
}



int main()
{
        Mammal* theArray[5];
        Mammal* ptr;

        int choice, i;

        for(i=0; i<5; i++){
```

```
cout<<" (1) Dog (2) Cat (3) Horse (4) Guinea pig: " <<endl;
cin>>choice;

switch (choice) {

        case1 : ptr = new Dog;
        break;

        case2 : ptr = new Cat;
        break;

        case3 : ptr = new Horse;
        break;

        case4 : ptr = new Guineapig;
        break;

        default : ptr = new Mammal;
        break;
```

```
            }
              theArray[i]=ptr;

        }

        for(i=0; i<5; i++)

          theArray[i]->speak();

          for(i=0; i<5; i++) // Always free dynamically allocated objects

             delete theArray[i];

             return 0;

}
```

Output :

```
E:\3rd Semester\OOP\Section 2A\After Mid\Lab Tasks\Lab # 1-Task 03.exe
 (1) Dog (2) Cat (3) Horse (4) Guinea pig:
1
 Mammal constructor......
 (1) Dog (2) Cat (3) Horse (4) Guinea pig:
2
 Mammal constructor......
 (1) Dog (2) Cat (3) Horse (4) Guinea pig:
3
 Mammal constructor......
 (1) Dog (2) Cat (3) Horse (4) Guinea pig:
4
 Mammal constructor......
 (1) Dog (2) Cat (3) Horse (4) Guinea pig:
4
 Mammal constructor......
 What does a Mammal speak? Mammilian!
 What does a Mammal speak? Mammilian!
 What does a Mammal speak? Mammilian!
 What does a Mammal speak? Mammilian!
 What does a Mammal speak? Mammilian!
 Mammal destructor.....
 Mammal destructor.....
 Mammal destructor.....
 Mammal destructor.....
 Mammal destructor.....

--------------------------------
Process exited after 6.932 seconds with return value 0
Press any key to continue . . .
```

Questions

Q #1.

If, in the example above, Mammal overrides a function in Animal, which does Dog get, the original or the overridden function?

Answer :

If Mammal overrides a function in Animal and Dog inherits from Mammal, then Dog will get the overridden function from Mammal. This is because in the process of inheritance, the subclass (in this case, Mammal) can override the methods or functions of the superclass (in this case, Animal), and the subclass's implementation will take precedence over the superclass's implementation when called from the subclass or any of its derived classes. Therefore, since Dog is a subclass of Mammal, it will inherit the overridden function from Mammal rather than the original function from Animal.

Q #2.

Can a derived class make a public base function private?

Answer :

No, a derived class cannot make a public base function private. When a derived class inherits from a base class, it inherits all the members of the base class that are accessible to the derived

class, including public members. Once a member is declared as public in the base class, it remains public in the derived class and cannot be changed to a more restrictive access specifier, such as private.

However, a derived class can choose not to expose the base class's public member by not providing a public interface to it. In other words, the derived class can make the inherited public member effectively private by not making it accessible from outside the derived class. This can be achieved by redefining the function with the same name in the derived class, which effectively hides the base class's function from outside the derived class.

Note that this technique only affects the accessibility of the function outside the derived class. Within the derived class, the base class's function is still accessible and can be called directly using the scope resolution operator (::).

Q #3.

Why not make all class functions virtual?

Answer :

Making all class functions virtual has a performance cost because virtual functions are implemented using a mechanism called the virtual function table (vtable) or virtual method table (vtable), which adds a level of indirection to the function call. This means that calling a virtual function can be slower than calling a non-virtual function.

In addition, making a function virtual can also have implications for memory usage, as each object of the class that contains virtual functions will have a pointer to its corresponding vtable, which can increase the size of each object. This can be a concern when working with large numbers of objects or when memory usage is a critical factor.

Therefore, it's generally not a good idea to make all class functions virtual unless they need to be. Only functions that are intended to be overridden by derived classes or that need to be called dynamically based on the type of the object should be declared as virtual.

Declaring functions as virtual should be a conscious design decision that takes into account the trade-offs between performance, memory usage, and the flexibility of the class hierarchy. When used appropriately, virtual functions can provide a powerful mechanism

for implementing polymorphism and enabling runtime binding of functions based on the type of the object.

Q #4.

f a function (SomeFunc()) is virtual in a base class and is also overloaded, so as to take either an integer or two integers, and the derived class overrides the form taking one integer, what is called when a pointer to a derived object calls the two-integer form?

Answer:

If a function SomeFunc() is virtual in a base class and is also overloaded to take either an integer or two integers, and the derived class overrides the form taking one integer, then the two-integer form in the base class will be called when a pointer to a derived object calls the two-integer form.

This is because function overloading is resolved statically based on the number and types of arguments at compile time, while virtual function dispatch is resolved dynamically based on the type of the object at runtime. Since the two-integer form is not overridden in the derived class, it remains a member of the base class and will be called when a pointer to a derived object calls it.

Here's an example code snippet to illustrate the behavior:

```
#include <iostream>

class Base {
public:
    virtual void SomeFunc(int x) {
        std::cout << "Base::SomeFunc(int) called with x = " << x << std::endl;
    }
    virtual void SomeFunc(int x, int y) {
        std::cout << "Base::SomeFunc(int, int) called with x = " << x << " and y = " << y << std::endl;
    }
};

class Derived : public Base {
```

```
public:
    void SomeFunc(int x) override {
        std::cout << "Derived::SomeFunc(int) called with x = " << x << std::endl;
    }
};

int main() {
    Base* ptr = new Derived();
    ptr->SomeFunc(10); // calls Derived::SomeFunc(int)
    ptr->SomeFunc(20, 30); // calls Base::SomeFunc(int, int)
    delete ptr;
    return 0;
}
```

In this example, the derived class `Derived` overrides the one-integer form of `SomeFunc()`, and a pointer to a `Derived` object is assigned to a pointer of type `Base*`. When `SomeFunc(10)` is called through the `Base*` pointer, the overridden version in

`Derived` is called, whereas when `SomeFunc(20, 30)` is called, the version in `Base` is called. The output of the program will be:

```
Derived::SomeFunc(int) called with x = 10
Base::SomeFunc(int, int) called with x = 20 and y = 30
```

Here are some more questions:

1. What is a v-table?

A v-table (short for virtual table) is a data structure used by the C++ language to implement virtual functions and polymorphism. It is an array of function pointers that is created by the compiler for each class that has virtual functions, and is stored as part of the object's runtime state.

The v-table contains a pointer to each virtual function defined by the class, and each pointer points to the address of the corresponding

function implementation in the object's memory. When a virtual function is called through a pointer to a base class object, the compiler generates code that looks up the function pointer in the object's v-table and jumps to the corresponding function implementation.

The v-table is typically generated by the compiler at compile-time and is statically linked to the object code. Each object of a class that has virtual functions has its own v-table, which is initialized when the object is constructed and remains the same throughout the object's lifetime.

In summary, the v-table is a key mechanism that allows C++ to implement dynamic dispatch and runtime polymorphism using virtual functions. By using a table of function pointers, the language can resolve function calls dynamically based on the actual type of the object at runtime, rather than statically based on the type of the pointer at compile time.

2. What is a virtual destructor?

In C++, a virtual destructor is a special type of destructor that is declared as virtual in the base

class of a hierarchy. When a class has a virtual destructor, it ensures that the destructors of both the base class and any derived classes are called in the correct order when an object of a derived class is destroyed.

The need for a virtual destructor arises from the fact that C++ allows polymorphic objects to be created on the heap using a base class pointer. When such an object is destroyed, it is essential that the destructor for the actual derived class is called, as well as the destructor for the base class. If the base class destructor is not declared as virtual, then the behavior is undefined and may result in memory leaks and other issues.

Here is an example code snippet that illustrates the use of virtual destructors:

```
#include <iostream>

class Base {
public:
```

```
    virtual ~Base() {
        std::cout << "Base destructor called" << std::endl;
    }
};

class Derived : public Base {
public:
    ~Derived() override {
        std::cout << "Derived destructor called" << std::endl;
    }
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // calls Derived destructor, then Base destructor
    return 0;
}
```

In this example, the base class `Base` has a virtual destructor, and the derived class `Derived` overrides it. When an object of type `Derived` is created on the heap and assigned to a `Base*` pointer, the destructor for the derived class will be called first, followed by the destructor for the base class. This ensures that any resources allocated by the derived class are properly deallocated before the base class is destroyed.

In summary, a virtual destructor is an important feature in C++ that allows for proper destruction of polymorphic objects and avoids memory leaks and other issues that can arise when objects are not destroyed in the correct order.

3. How do you show the declaration of a virtual constructor?

In C++, virtual constructors are not allowed. This is because constructors are not inherited, so there is no need to override them. Furthermore, the new operator is responsible for creating objects, and it requires a concrete type to create an object, not an abstract type.

The concept of a virtual constructor is often confused with the concept of a virtual factory method. A virtual factory method is a

virtual function that creates and returns an object of a derived class. The factory method is declared in the base class and implemented in the derived classes, and it allows clients to create objects of the derived class without having to know their concrete type.

Here is an example of how to declare and use a virtual factory method:

```
#include <iostream>

class Base {
public:
    virtual ~Base() {}
    virtual Base* create() const {
        return new Base();
    }
};
```

```
class Derived : public Base {
public:
    Derived() {}
    virtual Derived* create() const override {
        return new Derived();
    }
};

int main() {
    Base* b = new Derived();
    Base* b2 = b->create();
    std::cout << typeid(*b2).name() << std::endl; // prints "Derived"
    delete b2;
    delete b;
    return 0;
}
```

In this example, the base class `Base` has a virtual factory method `create()` that returns a pointer to a `Base` object. The derived class `Derived` overrides the `create()` method to return a pointer to a `Derived` object. When `create()` is called on an object of type `Derived`, it returns a pointer to a `Derived` object, even though the pointer type is `Base*`.

In summary, virtual constructors are not allowed in C++, but virtual factory methods can be used to create objects of derived classes without knowing their concrete type.

4. How can you create a virtual copy constructor?

In C++, a copy constructor is a special constructor that creates a new object as a copy of an existing object. However, copy constructors cannot be declared as virtual functions, because constructors cannot be overridden like other member functions.

To create a virtual copy constructor in C++, you can use a combination of a virtual function and a static function. The virtual function will be used to create a new object of the same type as the

object being copied, while the static function will be used to invoke the virtual function.

Here is an example of how to create a virtual copy constructor using this technique:

```
#include <iostream>

class Base {
public:
    virtual ~Base() {}
    virtual Base* clone() const {
        return new Base(*this);
    }
};

class Derived : public Base {
public:
```

```
    Derived() {}
    Derived(const Derived& other) : Base(other) {}
    virtual Derived* clone() const override {
        return new Derived(*this);
    }
};

int main() {
    Base* b = new Derived();
    Base* b2 = b->clone();
    std::cout << typeid(*b2).name() << std::endl; // prints "Derived"
    delete b2;
    delete b;
    return 0;
}
```

In this example, the base class `Base` has a virtual function `clone()` that returns a pointer to a new object of the same type as

the object being copied. The derived class `Derived` overrides the `clone()` function to return a pointer to a new `Derived` object.

To create a copy of an object, you can call the virtual function `clone()` on the object and then use the returned pointer to create a new object of the same type. In the example, the `clone()` function is called on a `Base*` pointer that points to a `Derived` object. The function returns a `Derived*` pointer, which is used to create a new `Derived` object.

In summary, while copy constructors cannot be declared as virtual functions, you can use a combination of a virtual function and a static function to create a virtual copy constructor in C++.

5. How do you invoke a base member function from a derived class in which you've overridden that function?

In C++, when you override a member function of a base class in a derived class, you can still access the base class's implementation of the function using the scope resolution operator (::).

To invoke the base member function from the derived class, you can use the name of the base class followed by the scope resolution operator and the name of the member function. Here is an example:

```
#include <iostream>

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

class Derived : public Base {
public:
    virtual void print() override {
        Base::print();
        std::cout << "Derived::print() called" << std::endl;
```

```
    }
};

int main() {
    Derived d;
    d.print();
    return 0;
}
```

In this example, the `Base` class has a virtual function `print()`, and the `Derived` class overrides it. In the `print()` function of the `Derived` class, the base class's implementation of the function is called using the scope resolution operator `Base::print()`. This allows the derived class to execute the base class's implementation of the function before executing its own code.

When the program is run, it will output:

```
```

Base::print() called

Derived::print() called

```

This shows that the base class's implementation of the `print()` function was called before the derived class's implementation.

In summary, to invoke a base member function from a derived class in which you've overridden that function, you can use the scope resolution operator and the name of the base class followed by the name of the member function.

6. How do you invoke a base member function from a derived class in which you have not overridden that function?

In C++, when you have a derived class that does not override a member function of its base class, you can still access the base class's implementation of the function using the scope resolution operator (::).
```

To invoke the base member function from the derived class, you can use the name of the base class followed by the scope resolution operator and the name of the member function. Here is an example:

```
#include <iostream>

class Base {
public:
    void print() {
        std::cout << "Base::print() called" << std::endl;
    }
};

class Derived : public Base {
public:
    void printDerived() {
        Base::print();
    }
```

```
};

int main() {
    Derived d;
    d.printDerived();
    return 0;
}
```

In this example, the `Base` class has a member function `print()`, and the `Derived` class does not override it. Instead, it has a member function `printDerived()` that calls the base class's implementation of `print()` using the scope resolution operator `Base::print()`.

When the program is run, it will output:

```
Base::print() called
```

This shows that the base class's implementation of the `print()` function was called from the `printDerived()` function of the derived class.

In summary, to invoke a base member function from a derived class in which you have not overridden that function, you can use the scope resolution operator and the name of the base class followed by the name of the member function.

7. If a base class declares a function to be virtual, and a derived class does not use the term virtual when overriding that class, is it still virtual when inherited by a third-generation class?

Yes, if a base class declares a function to be virtual, and a derived class overrides it without using the `virtual` keyword, the function is still virtual when inherited by a third-generation class.

In C++, when a virtual function is declared in a base class, it remains virtual in all its derived classes unless explicitly marked as non-virtual by using the `final` keyword. When a derived class

overrides a virtual function, it can use the `virtual` keyword to explicitly mark it as virtual, but this is not necessary because the function is already virtual due to its declaration in the base class.

Therefore, if a derived class overrides a virtual function without using the `virtual` keyword, the function is still considered virtual in that class and in all its derived classes. When a third-generation class inherits from this derived class, it will also inherit the virtual function.

Here is an example:

```
#include <iostream>

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

```
};

class Derived : public Base {
public:
    void print() override {
        std::cout << "Derived::print() called" << std::endl;
    }
};

class Third : public Derived {
public:
    void print() override {
        std::cout << "Third::print() called" << std::endl;
    }
};

int main() {
    Third t;
    Base* b = &t;
```

```
    b->print();
    return 0;
}
```

In this example, the `Base` class has a virtual function `print()`, which is overridden in the `Derived` class without using the `virtual` keyword. The `Derived` class is then inherited by the `Third` class, which also overrides `print()`.

When the program is run, it will output:

```
Third::print() called
```

This shows that the `print()` function of the `Third` class was called, and that it inherited the virtual function from the `Base` class through the `Derived` class, even though the `virtual` keyword was not used when overriding the function in the `Derived` class.

In summary, when a base class declares a function to be virtual, and a derived class overrides it without using the `virtual` keyword, the function is still virtual and will be inherited as virtual by all its derived classes, including those further down the inheritance hierarchy.

8. What is the protected keyword used for?

In C++, the `protected` keyword is used as an access specifier for class members. It specifies that the member can be accessed by the class itself, its derived classes, and friend classes, but not by code outside the class or its derived classes.

More specifically, when a class member is declared as `protected`, it can be accessed from within the class, from within any class derived from that class, and from within any friend function or class of either the base class or the derived class.

Here is an example:

```
class Base {
protected:
    int x;
};

class Derived : public Base {
public:
    void foo() {
        x = 5; // x is accessible because it is protected
    }
};

int main() {
    Base b;
    b.x = 5; // error: x is not accessible because it is protected
    Derived d;
    d.x = 5; // error: x is not accessible because it is protected
    return 0;
```

}
```

In this example, the `Base` class has a protected member variable `x`. The `Derived` class inherits from `Base` and has a member function `foo()`, which can access the protected variable `x`. The `main()` function tries to access the protected variable `x` from both `Base` and `Derived`, but both attempts result in a compilation error because `x` is not accessible outside of the class and its derived classes.

In summary, the `protected` keyword in C++ is used to specify that a class member can be accessed by the class itself, its derived classes, and friend classes, but not by code outside the class or its derived classes.

## *Some more exercises:*

1. Show the declaration of a virtual function that takes an integer parameter and returns void.
```

Here is an example declaration of a virtual function that takes an integer parameter and returns void:

```
class MyBaseClass {
public:
    virtual void myVirtualFunction(int x) = 0;
};
```

In this example, the virtual function is called `myVirtualFunction`, it takes an integer parameter `x`, and it returns `void`. The `virtual` keyword indicates that this function is meant to be overridden in derived classes, and the `= 0` syntax at the end of the declaration makes this function a pure virtual function, which means that it has no implementation in the base class and must be implemented in any class that inherits from it.

Note that this example uses a pure virtual function, but a virtual function can also have an implementation in the base class. In that

case, the implementation in the derived class can override the behavior of the function.

2. Show the declaration of a class Square, which derives from Rectangle, which in turn derives from Shape.

Here is an example declaration of the `Square` class, which derives from `Rectangle`, which in turn derives from `Shape`:

```
class Shape {
public:
    virtual double area() = 0;
};

class Rectangle : public Shape {
protected:
    double width;
    double height;
```

```
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    virtual double area() { return width * height; }
};

class Square : public Rectangle {
public:
    Square(double side) : Rectangle(side, side) {}
};
```

In this example, `Shape` is the base class, which has a pure virtual function `area()` that returns a `double`. `Rectangle` is a derived class from `Shape` that has two protected member variables, `width` and `height`, and implements the `area()` function to return the area of the rectangle. `Square` is a derived class from `Rectangle` that takes a single `double` parameter `side` and passes it to the `Rectangle` constructor for both the width and height, effectively creating a square.

Note that in this example, the `Square` class does not need to implement its own `area()` function, because it inherits the implementation from the `Rectangle` class, which in turn inherits the pure virtual function from the `Shape` class.

3. If, in Exercise 2, Shape takes no parameters, Rectangle takes two (length and width), but Square takes only one (length), show the constructor initialization for Square.

Here is an example of how the constructor initialization for `Square` could be written based on the requirements specified:

```
class Shape {
public:
    virtual double area() = 0;
};

class Rectangle : public Shape {
protected:
```

```
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}
    virtual double area() { return length * width; }
};

class Square : public Rectangle {
public:
    Square(double length) : Rectangle(length, length) {}
};
```

In this example, the `Square` constructor takes a single `double` parameter `length`, which is passed to the `Rectangle` constructor using the `length` parameter for both the `length` and `width` member variables. This effectively creates a square with equal length and width. The `Rectangle` constructor is called from the `Square` constructor using the `:` syntax to initialize the base class `Rectangle`.

4. Write a virtual copy constructor for the class Square?

In C++, a copy constructor is a special constructor that creates a new object as a copy of an existing object. In order to create a virtual copy constructor for the `Square` class, we can use the virtual constructor idiom. Here is an example implementation:

```
class Shape {
public:
    virtual ~Shape() {}
    virtual Shape* clone() const = 0;
    virtual double area() const = 0;
};

class Rectangle : public Shape {
protected:
    double length;
    double width;
```

```
public:
    Rectangle(double l, double w) : length(l), width(w) {}
    virtual ~Rectangle() {}
    virtual Rectangle* clone() const { return new Rectangle(*this); }
    virtual double area() const { return length * width; }
};

class Square : public Rectangle {
public:
    Square(double side) : Rectangle(side, side) {}
    virtual ~Square() {}
    virtual Square* clone() const { return new Square(*this); }
};
```

In this example, the `Shape` class has a pure virtual function `clone()` that returns a pointer to a new copy of the object, and a pure virtual function `area()` that returns the area of the shape. The `Rectangle` class and `Square` class both inherit from `Shape`.

The `Rectangle` class has its own copy constructor `clone()` that creates a new `Rectangle` object with the same `length` and `width` as the original. The `Square` class overrides the `clone()` function to create a new `Square` object with the same `side` as the original.

Note that the `clone()` function returns a pointer to a new copy of the object, so the caller must take responsibility for deallocating the memory when it is no longer needed.

With this implementation, a virtual copy constructor for the `Square` class can be called as follows:

```
Shape* original = new Square(5.0);
Shape* copy = original->clone();
delete original;
delete copy;
```

This creates a new `Square` object with side length 5.0 using the virtual constructor, makes a copy of the original object using the virtual copy constructor, deletes the original object, and then deletes the copy.

5. BUG BUSTERS: What is wrong with this code snippet? void SomeFunction (Shape); Shape * pRect = new Rectangle; SomeFunction(*pRect);

In this code snippet, `SomeFunction` is defined to take a `Shape` object by value, not by reference or pointer. Therefore, when it is called with `SomeFunction(*pRect)`, the `Rectangle` object is sliced, meaning only the `Shape` part of the `Rectangle` object is passed to the function.

To fix this issue, `SomeFunction` should take a `Shape` object by reference or pointer instead of by value. Here is an example:

```
void SomeFunction(Shape&);
Shape* pRect = new Rectangle;
SomeFunction(*pRect);
```

```

In this revised code snippet, `SomeFunction` takes a `Shape` object by reference. Now, when it is called with `SomeFunction(*pRect)`, the function will receive a reference to the original `Rectangle` object, not a sliced copy of it.

6. BUG BUSTERS: What is wrong with this code snippet? class Shape() { public: Shape(); virtual ~Shape(); virtual Shape(const Shape &); };

The `Shape` class appears to be defining a virtual copy constructor, but constructors cannot be virtual in C++. Virtual functions can be declared in a constructor, but constructors themselves cannot be virtual.

To create a virtual copy constructor, you should define a pure virtual function in the base class and then override it in the derived class to provide the copy constructor implementation.

Here is an example of how you can modify the code snippet to implement a virtual copy constructor using a pure virtual function:
```

```
class Shape {
public:
  Shape();
  virtual ~Shape();
  virtual Shape* clone() const = 0;  // Pure virtual function for virtual copy constructor
};

class Rectangle : public Shape {
public:
  Rectangle();
  virtual ~Rectangle();
  virtual Rectangle* clone() const;  // Override the clone() function to implement the virtual copy constructor
};
```

In this revised code snippet, `Shape` defines a pure virtual function `clone()` which returns a pointer to a new `Shape` object that is a

copy of the current object. `Rectangle` overrides this function to implement the virtual copy constructor for `Rectangle` objects.

Thank You So Much