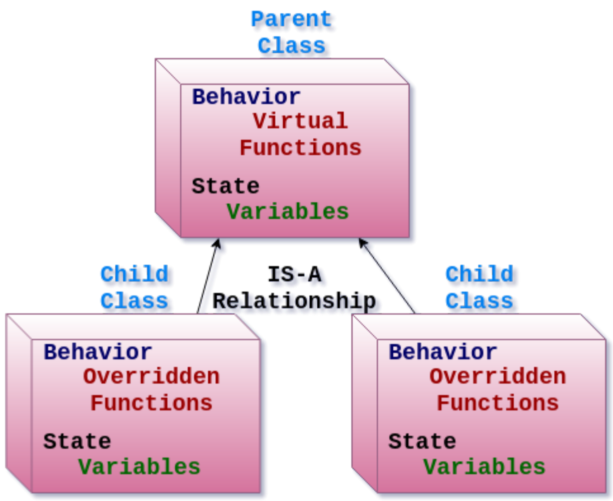
Inheritance

Inheritance Overview



Inheritance is a powerful Object-oriented technique where a general class is created and more specific classes are created, or derived, from the general class.

The general class is known by a few terms, such as **base class**, **parent class**, and **superclass.** The more specific classes are known by a few terms, such as **derived class**, **child class**, and **subclass**.

Child classes form an **IS-A** relationship with its **parent class**. For example, if you have a parent class called Fruit and a child class called Orange, you may read this as Orange **IS-A** Fruit. Here is an example on how to specify a child class:

```
class Parent
{
    //Class Functions & Variables
};

class Child : public Parent
{
    //Class Functions & Variables
};
```

After declaring the name of the child class, put a colon, the keyword **public**, and then the parent class' name (the keywords **protected** and **private** are allowed there, but aren't discussed here). This allows all **public** and **protected** member functions and member variables of the parent class to be used freely in the child classes. The parent class' **private** member functions are only valid via its member functions.

Inheritance allows the reuse of member functions and member variables from the parent class in its child classes.

You are able to put an object of a child class into an object of its parent class. This is possible due to the **IS-A** relationship. However, the inverse is not true. For example, an Orange **IS-A** Fruit, but a Fruit isn't always an Orange. Example:

```
Fruit FruitObject;
Orange OrangeObject;
FruitObject = OrangeObject; //Valid but slices data (Slicing Problem)
OrangeObject = FruitObject; //Illegal
```

Behavior of Compiler-Defined Constructors/Destructor/Assignment Operators

Please take note that these aspects of a class are **not** inherited. See **FruitInheritance** for an example on how to use user-defined ones.

Default Constructor: If you do not declare **any** type of constructor for a child class, the compiler will automatically provide a default constructor. The default constructor will:

1) Activate the default constructor of the parent class. This initializes member variables of the base class.
2) Then activate the default constructor of the child class. This initializes the member variables that the child class uses but the base class does not use.

Copy Constructor: If you do not define a copy constructor in your child class, the compiler will automatically provide a copy constructor. The copy constructor will:

1) Activate the copy constructor of the parent class. This copies the member variables of the base class.
2) Then activate the copy constructor of the child class. This copies the member variables that the child class uses but the base class does not use.

Assignment Operator: If you do not define an assignment operator in your child class, the compiler will automatically prove a default assignment operator. The assignment operator will:

1) Activate the assignment operator for the parent class. This copies the member variables of the base class.
2) Then activate the assignment operator of the child class. This copies the member variables that the child class uses but the base class does not use.

Destructor: If you do not define a destructor in your child class, the compiler will automatically provide a destructor. The destructor will:

1) Activate the destructor in the child class and delete any member variables the child class has but the base class does not have.
2) Then activate the destructor in the parent class and delete any member variables of the base class.

Please take note that the **destructor** works from child to parent.

Redefining Functions

The first concept of inheritance is the process known as **redefining** functions. Note that you **cannot** utilize polymorphism using this technique as in you cannot set an object of a child class to a pointer of the parent class and execute the redefined function in the child class (more on that in the **Virtual Functions** section). Redefining a function is the act of having a function in your parent class and the same function with the same formal parameters in your child classes. This may seem pointless, but the useful aspect of this method is that you are able to have a child class object and use the parent class' function if the child class does not have the function redefined.

Here is an example of a redefined function:

```
class Parent
{
    public:
        //Default function used in case child class doesn't have a
        //redefined function. Function used for a parent class object
        void display() { cout << "Parent Display Function \n"; }
};

class Child : public Parent
{
    Public:
        //Redefined function of the parent class' function
        void display() { cout << "Child Display Function \n"; }
};

int main()
{
    Parent PObj;
    PObj.display(); //Invoke Parent class function
    Child CObj;
    CObj.display(); //Invoke Child class function or Parent class
                    //function if not defined in the Child class
    Parent *Obj = new Child;
    Obj->display(); //Invoke Parent class function. NO polymorphism
}   //delete Obj;
```

Please note that you should not use this at it goes against polymorphism, which is one of the important aspects of Object-Oriented programming. However, it is doable and is here because it is in the CSI book. Virtual functions are able to invoke the same behavior as this method. However, it is important to know the difference between **redefining** functions and **overriding** functions (overriding is in the **Virtual Functions** section).

Pointers/Dynamic Aspect

Pointers: Objects of classes are able to be in the form of a pointer. This goes for normal pointers as well as dynamically allocated pointers. As noted on the first page of this section, you are able to put a child class object into a parent class object. The concept of the pointers is the same as any pointer/dynamic pointer you have seen. Don't forget to **delete** your dynamic pointer when finished. Examples:

```
Fruit *FruitPointer;  //Fruit = parent, Orange = child
Orange OrangeObject;
//Valid but pointer can't invoke non-virtual functions of Orange class
FruitPointer = &OrangeObject; //since object is now of type Fruit
Fruit *FruitObject2 = new Banana; //Valid
Banana *BananaObject = new Fruit;  //Illegal
```

Destructors: As noted before, **destructors** are invoked in the child class and then the parent class. This is necessary to prevent memory leaks, because if the parent class destructor were invoked first, the child class destructor would never be invoked leaving its data in the heap.

Virtual destructors work the same way as the compiler-defined destructor and should be implemented in order to dynamically allocate objects with the **new** operator. Don't forget to **delete** your dynamic data when you are finished with it (see below). Here is an example using dynamic memory:

```
class Parent
{
    public:
        virtual ~Parent() { cout << "Parent Destructor"; }
        //Other parts of the class implementation
};
class Child : public Parent
{
    public:
        virtual ~Child() { cout << "Child Destructor"; }
        //Other parts of the class implementation
};

int main()
{
    Child *Obj = new Child;
    delete Obj; //Invokes Child destructor then Parent destructor
    Parent *PObj = new Child;
    delete PObj; //Invokes Child destructor then Parent destructor
}
```

Slicing Problem

The slicing problem occurs when you put an object of a child class into an object of its parent class (this section is **not** about pointers and dynamic memory. See the **FruitInheritance** program for comments about that topic). The child class' data is lost in the process. After all, the parent class does not know about the contents of its child classes and the child class object is now of type parent class. Even if the functions are **virtual**, doing this will result in the parent function being invoked. Here is a short example:

```
class Parent
{ //Constructors and Rule of Three omitted for brevity
    public:
        virtual int get() { return x; }
    private:
        int x = 10;
};
class Child : public Parent
{
    public:
        virtual int get() override { return y; }
    private:
        int y = 20;
};
int main()
{
    Parent PObject;
    Child CObject;
    PObject = CObject; //Putting Child object into Parent object
    cout << PObject.get() << endl; //Displays 10
}
```

Protected Qualifier

You have seen the qualifiers **public** and **private**. The third qualifier is called **protected** and it is used during inheritance. When you use the **protected** qualifier, it has two distinct meanings:

1) Whatever is in the **protected** scope will act as if it were in the **private** scope for any class or function other than its child classes (cannot access it directly).
2) For any child class, you are able to access whatever is in the **protected** scope by name (access it directly in a function of the child class).

To see an example of inheritance and the **protected** qualifier, see **FruitInheritance** at the end of this section.