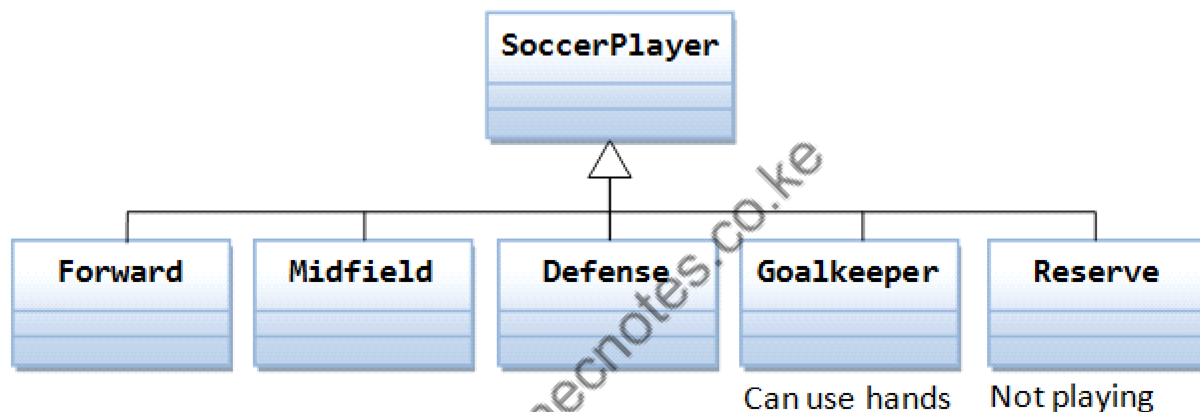
CHAPTER 5: INHERITANCE

Introduction and Rules in inheritance

In OOP, we often organize classes in *hierarchy* to *avoid duplication and reduce redundancy*. The classes in the lower hierarchy inherit all the variables (static attributes) and methods (dynamic behaviors) from the higher hierarchies. A class in the lower hierarchy is called a *subclass* (or *derived*, *child*, *extended class*). A class in the upper hierarchy is called a *superclass* (or *base*, *parent class*). By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, *redundancy* can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses. For example,



www.knecnotes.co.ke

Example

The following C++ code establishes an explicit inheritance relationship between classes **B** and **A**, where **B** is both a subclass and a subtype of **A**, and can be used as an **A** wherever a **B** is specified (via a reference, a pointer or the object itself).

```
class A
{ public:
   void DoSomethingALike() const {}
};

class B : public A
{ public:
   void DoSomethingBLike() const {}
};

void UseAnA(A const& some_A)
{
   some_A.DoSomethingALike();
}

void SomeFunc()
{
   B b;
   UseAnA(b); // b can be substituted for an A.
```

}

Types of inheritance

There are various types of inheritance, depending on paradigm and specific language.

- **Single Inheritance** : In the single inheritance, subclasses inherits the features of a single super class. A class acquire the property of another class.
- **Multiple Inheritance** : Multiple Inheritance allows a class to have more than one super class and to inherit features from all parent class.
- **Multilevel Inheritance** : In multilevel inheritance a subclass is inherited from another subclass.
- **Hierarchical Inheritance** : In hierarchical inheritance a single class serves as a superclass (base class) for more than one sub class.
- **Hybrid Inheritance** : It is a mixture of all the above types of inheritance.

www.knecnotes.co.ke

Importance of Inheritance

Inheritance is a good choice when:-

- Your inheritance hierarchy represents an "is-a" relationship and not a "has-a" relationship.
- You can reuse code from the base classes.
- You need to apply the same class and methods to different data types.
- The class hierarchy is reasonably shallow, and other developers are not likely to add many more levels.
- You want to make global changes to derived classes by changing a base class.

Inheritance Advantages and Disadvantages

Advantages:-

1. One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses. Where equivalent code exists in two related classes, the hierarchy can usually be refactored to move the common code up to a mutual superclass. This also tends to result in a better organization of code and smaller, simpler compilation units.
2. Inheritance can also make application code more flexible to change because classes that inherit from a common superclass can be used interchangeably. If the return type

of a method is superclass

3. Reusability -- facility to use public methods of base class without rewriting the same
4. Extensibility -- extending the base class logic as per business logic of the derived class
5. Data hiding -- base class can decide to keep some data private so that it cannot be altered by the derived class
6. Overriding--With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

Disadvantages:-

1. One of the main disadvantages of inheritance in Java (the same in other object-oriented languages) is the increased time/effort it takes the program to jump through all the levels of overloaded classes. If a given class has ten levels of abstraction above it, then it will essentially take ten jumps to run through a function defined in each of those classes
2. Main disadvantage of using inheritance is that the two classes (base and inherited class) get tightly coupled.
   This means one cannot be used independent of each other.
3. Also with time, during maintenance adding new features both base as well as derived classes are required to be changed. If a method signature is changed then we will be affected in both cases (inheritance & composition)
4. If a method is deleted in the "super class" or aggregate, then we will have to re-factor in case of using that method. Here things can get a bit complicated in case of inheritance because our programs will still compile, but the methods of the subclass will no longer be overriding superclass methods. These methods will become independent methods in their own right.

www.knechotes.co.ke

Implementation of Inheritance

Following are attributes observed in the implementation of inheritance

Base & Derived Classes:

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the

name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>

using namespace std;

// Base class
class Shape
{
   public:
      void setWidth(int w)
      {
         width = w;
      }
      void setHeight(int h)
      {
         height = h;
      }
   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape
{
   public:
      int getArea()
      {
         return (width * height);
      }
};

int main(void)
{
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

www.knecnotes.co.ke

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
```

Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived

classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

- **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Concepts in inheritance

Single Inheritance is method in which a derived class has only one base class.

Example:

```
#include <iostream.h> class Value
{
protected:
int val;
```

www.knecnotes.co.ke

```
public:
void set_values (int a)
{ val=a;}
}; class Cube: public Value
{
public:
int cube()
{ return (val*val*val); }
}; int main ()
{
Cube cub;
cub.set_values (5);
cout << "The Cube of 5 is::" << cub.cube() << endl;
return 0;
}
```

Result:
The Cube of 5 is:: 125

www.knecnotes.co.ke

Multiple inheritance is achieved whenever more than one class acts as base classes for other classes. This makes the members of the base classes accessible in the derived class, resulting in better integration and broader re-usability.

example:

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

class Cpolygon
{
	protected:
		int width, height;
	public:
		void input_values (int one, int two)
		{
			width=one;
			height=two;
		}
};

class Cprint
{
	public:
		void printing (int output);
};

void Cprint::printing (int output)
{
	cout << output << endl;
}
```

```
class Crectangle: public Cpolygon, public Cprint
{
        public:
                int area ()
                {
                        return (width * height);
                }
};

class Ctriangle: public Cpolygon, public Cprint
{
        public:
                int area ()
                {
                        return (width * height / 2);
                }
};

int main ()
{
        Crectangle rectangle;
        Ctriangle triangle;
        rectangle.input_values (2,2);
        triangle.input_values (2,2);
        rectangle.printing (rectangle.area());
        triangle.printing (triangle.area());
        return 0;
}
```

Note:the two public statements in the Crectangle class and Ctriangle class.

www.knecnotes.co.ke

Inheritance and friends

Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to *"friends"*.

Friends are functions or classes declared with the `friend` keyword.

A non-member function can access the private and protected members of a class if it is declared a *friend* of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword `friend`:

```
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
```

```
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

The `duplicate` function is a *friend* of class `Rectangle`. Therefore, function `duplicate` is able to access the members `width` and `height` (which are private) of different objects of type `Rectangle`.

www.knecnotes.co.ke

Friend classes

Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class:

```
// friend class
#include <iostream>
using namespace std;

class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
```

```
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

In this example, class `Rectangle` is a friend of class `Square` allowing `Rectangle`'s member functions to access private and protected members of `Square`. More concretely, `Rectangle` accesses the member variable `Square::side`, which describes the side of the square.

Pointers to objects

A variable that holds an address value is called a pointer variable or simply pointer.
Pointer can point to objects as well as to simple data types and arrays. sometimes we don't know, at the time that we write the program , how many objects we want to creat. when this is the case we can use new to create objects while the program is running. new returns a pointer to an unnamed objects. Let's see the example of student that will clear your idea about this topic

Pointer to Members in C++ Classes

Just like pointers to normal variables and functions, we can have pointers to class member functions and member variables.

Defining a pointer of class type

We can define pointer of class type, which can be used to point to class objects.

```
class Simple
{
 public:
 int a;
};

int main()
{
 Simple obj;
 Simple* ptr;   // Pointer of class type
 ptr = &obj;

 cout << obj.a;
 cout << ptr->a;  // Accessing member with pointer
}
```

Here you can see that we have declared a pointer of class type which points to class's object. We can access data members and member functions using pointer name with arrowsymbol.

Pointer to Data Members of class

We can use pointer to point to class's data members (Member variables).

Syntax for Declaration :

```
datatype class_name :: *pointer_name ;
```

Syntax for Assignment :

```
pointer_name = &class_name :: datamember_name ;
```

www.knecnotes.co.ke

Both declaration and assignment can be done in a single statement too.

```
datatype class_name::*pointer_name =
&class_name::datamember_name ;
```

Using with Objects

For accessing normal data members we use the dot "." operator with object and "->" with pointer to object. But when we have a pointer to data member, we have to dereference that pointer to get what its pointing to, hence it becomes,

```
Object.*pointerToMember
```

and with pointer to object, it can be accessed by writing,

```
ObjectPointer->*pointerToMember
```

Let's take an example, to understand the complete concept.

```
class Data
{
 public:
 int a;
 void print() { cout << "a is="<< a; }
};

int main()
{
 Data d, *dp;
 dp = &d;     // pointer to object

 int Data::*ptr=&Data::a;   // pointer to data member 'a'

 d.*ptr=10;
 d.print();

 dp->*ptr=20;
 dp->print();
}
```

```
Output : a is=10 a is=20
```

The syntax is very tough, hence they are only used under special circumstances.

Pointer to Member Functions

Pointers can be used to point to class's Member functions.

Syntax :

www.knecnotes.co.ke

```
return_type (class_name::*ptr_name) (argument_type) =
&class_name ::function_name ;
```

Below is an example to show how we use pointer to member functions.

```
class Data
{ public:
  int f (float) { return 1; }
};

int (Data::*fp1) (float) = &Data::f;   // Declaration and
assignment
int (Data::*fp2) (float);        // Only Declaration

int main(0
{
 fp2 = &Data::f;   // Assignment inside main()
}
```

Some Points to remember

1. You can change the value and behaviour of these pointers on runtime. That means, you can point it to other member function or member variable.

2. To have pointer to data member and member functions you need to make them public.

www.knecnotes.co.ke

Inheritance and constructors

Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deallocation of memory for objects.

Derived classes do not inherit constructors or destructors from their base classes, but they do call the constructor and destructor of base classes. Destructors can be declared with the keyword **virtual**.

Constructors are also called when local or temporary class objects are created, and destructors are called when local or temporary objects go out of scope.

Constructor and Destructor invoking sequence with inheritance

Example Program

```
class Base
{
  public:
```

```
  Base ( )
  {
    cout << "Inside Base constructor" << endl;
  }


  ~Base ( )
  {
    cout << "Inside Base destructor" << endl;
  }

};

class Derived : public Base
{

  public:

  Derived  ( )
  {
    cout << "Inside Derived constructor" << endl;
  }

  ~Derived ( )
  {
    cout << "Inside Derived destructor" << endl;
  }

};

void main( )
{
  Derived x;
}
```

www.knecnotes.co.ke

So, here is what the output of the code above would look like:

```
Inside Base constructor
Inside Derived constructor
Inside Derived destructor
Inside Base destructor
```

Base class constructors and derived class destructors are called first

In the code above, when the object "x" is created, first the Base class constructor is called, and after that the Derived class constructor is called. Because the Derived class inherits from the Base class, both the Base class and Derived class constructors will be called when a Derived class object is created.

When the main function is finished running, the object x's destructor will get called first, and after that the Base class destructor will be called.

Base class conversions

Upcasting is converting a derived-class reference or pointer to a base-class. In other words, upcasting allows us to treat a derived type as though it were its base type. It is always allowed for **public** inheritance, without an explicit type cast. This is a result of the **is-a** relationship between the base and derived classes.

Here is the code dealing with shapes. We created **Shape** class, and derived **Circle**, **Square**, and **Triangle** classes from the **Shape** class. Then, we made a member function that talks to the base class:

```
void play(Shape& s)
{
   s.draw();
   s.move();
   s.shrink();
   ....
}
```

The function speaks to any **Shape**, so it is independent of the specific type of object that it's drawing, moving, and shrinking. If in some other part of the program we use the **play( )** function like below:

```
Circle c;
Triangle t;
Square sq;
play(c);
play(t);
play(sq);
```

www.knecnotes.co.ke

Let's check what's happening here. A **Triangle** is being passed into a function that is expecting a **Shape**. Since a **Triangle** is a **Shape**, it can be treated as one by **play()**. That is, any message that **play()** can send to a **Shape** a **Triangle** can accept.

Upcasting allows us to treat a derived type as though it were its base type. That's how we decouple ourselves from knowing about the exact type we are dealing with.

Note that it doesn't say "If you're a **Triangle**, do this, if you're a **Circle**, do that, and so on." If we write that kind of code, which checks for all the possible types of a **Shape**, it will soon become a messy code, and we need to change it every time we add a new kind of **Shape**. Here, however, we just say "You're a **Shape**, I know you can move(), draw(), and shrink( ) yourself, do it, and take care of the details correctly."

The compiler and runtime linker handle the details. If a member function is virtual, then when we send a message to an object, the object will do the right thing, even when upcasting is involved.
Note that the most important aspect of inheritance is not that it provides member functions for the new class, however. It's the **relationship** expressed between the new class and the base class. This relationship can be summarized by saying, **"The new class is a type of the existing class."**

```
class Parent {
public:
  void sleep() {}
```

```
};

class Child: public Parent {
public:
  void gotoSchool(){}
};

int main( )
{
  Parent parent;
  Child child;

  // upcast - implicit type cast allowed
  Parent *pParent = &child;

  // downcast - explicit type case required
  Child *pChild =  (Child *) &parent;

  pParent -> sleep();
  pChild -> gotoSchool();

  return 0;
}
```

A **Child** object is a **Parent** object in that it inherits all the data members and member functions of a **Parent** object. So, anything that we can do to a **Parent** object, we can do to a **Child** object. Therefore, a function designed to handle a **Parent** pointer (reference) can perform the same acts on a **Child** object without any problems. The same idea applies if we pass a pointer to an object as a function argument. Upcasting is **transitive**: if we derive a **Child** class from **Parent**, then **Parent** pointer (reference) can refer to a **Parent** or a **Child** object.

www.knecnotes.co.ke

Upcasting can cause object slicing when a derived class object is passed by value as a base class object, as in **foo(Base derived_obj)**.

Downcasting

The opposite process, converting a base-class pointer (reference) to a derived-class pointer (reference) is called **downcasting**. Downcasting is not allowed without an explicit type cast. The reason for this restriction is that the **is-a** relationship is not, in most of the cases, symmetric. A derived class could add new data members, and the class member functions that used these data members wouldn't apply to the base class.

As in the example, we derived **Child** class from a **Parent** class, adding a member function, **gotoSchool()**. It wouldn't make sense to apply the **gotoSchool()** method to a **Parent** object. However, if implicit downcasting were allowed, we could accidentally assign the address of a **Parent** object to a pointer-to-**Child**

```
Child *pChild =  &parent; // actually this won't compile
        // error: cannot convert from 'Parent *' to 'Child *'
```

and use the pointer to invoke the **gotoSchool()** method as in the following line.

```
pChild -> gotoSchool();
```

Because a **Parent** isn't a **Child** (a **Parent** need not have a **gotoSchool()** method), the

downcasting in the above line can lead to an **unsafe** operation.

C++ provides a special explicit cast called **dynamic_cast** that performs this conversion. Downcasting is the opposite of the basic object-oriented rule, which states objects of a derived class, can always be assigned to variables of a base class.

One more thing about the upcasting:
Because **implicit upcasting** makes it possible for a base-class pointer (reference) to refer to a base-class object or a derived-class object, there is the need for **dynamic binding**. That's why we have **virtual** member functions.

- **Pointer (Reference)** type: known at **compile** time.
- **Object** type: not known until **run** time.

Dynamic Casting

The **dynamic_cast** operator answers the question of whether we can **safely** assign the address of an object to a pointer of a particular type.

www.knecnotes.co.ke

Here is a similar example to the previous one.

```
#include <string>

class Parent {
public:
  void sleep() {
  }
};

class Child: public Parent {
private:
  std::string classes[10];
public:
  void gotoSchool(){}
};

int main( )
{
  Parent *pParent = new Parent;
  Parent *pChild = new Child;

  Child *p1 = (Child *) pParent;  // #1
  Parent *p2 = (Child *) pChild;  // #2
  return 0;
}
```

Let look at the lines where we do type cast.

```
Child *p1 = (Child *) pParent;  // #1
Parent *p2 = (Child *) pChild;  // #2
```

Which of the type cast is safe?
The only one guaranteed to be safe is the ones in which the pointer is the same type as the object or else a base type for the object.

Type cast #1 is not safe because it assigns the address of a base-class object (**Parent**) to a derived class (**Child**) pointer. So, the code would expect the base-class object to have derived class properties such as **gotoSchool()** method, and that is false. Also, **Child** object, for example, has a member **classes** that a **Parent** object is lacking.

Type case #2, however, is safe because it assigns the address of a derived-class object to a base-class pointer. In other words, public derivation promises that a **Child** object is also a **Parent** object.

The question of whether a type conversion is safe is more useful than the question of what kind of object is pointed to. The usual reason for wanting to know the type is so that we can know if it's safe to invoke a particular method.

Here is the syntax of **dynamic_cast**.

```
Child *p = dynamic_cast<Child *>(pParent)
```

This code is asking whether the pointer **pParent** can be type cast safely to the type **Child ***.

- It returns the address of the object, if it can.
- It returns **0**, otherwise.

How do we use the **dynamic_cast**?

```
void f(Parent* p) {
  Child *ptr = dynamic_cast<Child*>(p);
   if(ptr) {
    // we can safely use ptr
  }
}
```

www.knecnotes.co.ke

In the code, if **(ptr)** is of the type **Child** or else derived directly or indirectly from the type **Child**, the **dynamic_cast** converts the pointer **p** to a pointer of type **Child**. Otherwise, the expression evaluates to **0**, the null pointer.

In other words, we want to check if we can use the passed in pointer **p** before we do some operation on a child class object even though it's a pointer to base class.

"The need for **dynamic_cast** generally arises because we want perform **derived class operation** on a **derived class object**, but we have only a pointer-or reference-to-**base**."
-Scott Meyers

Notes

- *downcast* - A downcast is a cast from a base class to a class derived from that base class.

- *cross-cast* - A cross-cast is a cast between unrelated types (user-defined conversion)

Class Scope under Inheritance

Each class defines its own scope within which its members are defined. Under inheritance, the scope of a derived class is nested inside the scope of its base classes. If a name is unresolved within the scope of the derived class, the enclosing base-class scopes are searched for a definition of that name.

The fact that the scope of a derived class nests inside the scope of its base classes can be surprising. After all, the base and derived classes are defined in separate parts of our program's text. However, it is this hierarchical nesting of class scopes that allows the members ...

Overloading with Inheritance

Overloading doesn't work for derived class in C++ programming language. There is no overload resolution between Base and Derived. The compiler looks into the scope of Derived, finds the single function "double f(double)" and calls it. It never disturbs with the (enclosing) scope of Base. In C++, there is no overloading across scopes – derived class scopes are not an exception to this general rule.

www.knecnotes.co.ke

Inheritance relationship

Subclasses and superclasses can be understood in terms of the **is a** relationship. A subclass **is a** more specific instance of a superclass. For example, an orange **is a** citrus fruit, which **is a** fruit. A shepherd **is a** dog, which **is an** animal.
If the **is a** relationship does not exist between a subclass and superclass, you should not use inheritance.

Note: inheritance only reuses implementation and establishes a syntactic relationship, not necessarily a semantic relationship (inheritance does not ensure **behavioral subtyping**). To distinguish these concepts, subtyping is also known as *interface inheritance,* while inheritance as defined here is known as **implementation inheritance**.