2021/2022

University of Technology - IRAQ
Computer Sciences Department

Object-Oriented Programming in C++

2nd Level
2nd Course

Asst. Lect. Saif Bashar

Lecture notes about the basic concepts of object-oriented programming with C++ for more information and resources visit my website at: s4ifbn.com

updated @ October 2021

Chapter 14

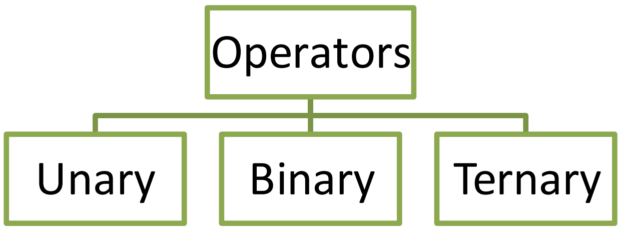
Operators Overloading

14.1 Introduction

We came across the term **overloading** when we discussed function overloading in Chapter 5 and constructor overloading in Chapter 11, the word overloading means giving more than one job to something.

Function overloading is when we have the same function perform different tasks the intended function is called depending on its argument list. The same definition applied to constructor overloading.

This chapter present a new OOP feature that apply overloading to the C++ language operators, to make the operator perform additional tasks beside the built-in task it assigned to. Before we discuss operators overloading, we have seen and worked in the previous chapters of this book with various types of operators like arithmetic, comparison, logical, bitwise, addressing and other operators, now let us organize these operators depending on their operands.



If the operator takes only one operand it is called unary operator, if the operator takes two operands it is called binary operator, and if it takes three operands it is called ternary operator.

For example:

++A the increment operator **++** is considered unary operator because it takes only one operand **A**.

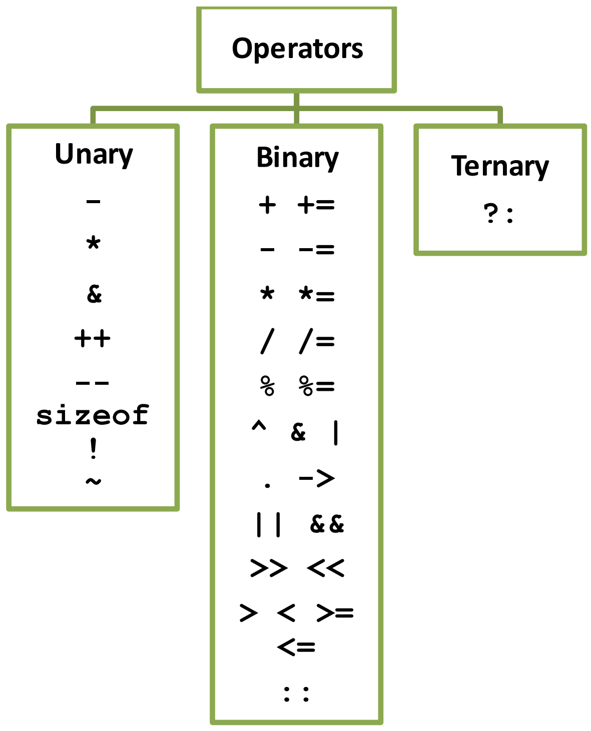
A + B the addition operator **+** is considered a binary operator because it takes two operands **A** and **B**.

A > B ? true : false; the conditional operator **?:** is considered a ternary operator because it takes three operands: **A>B**, **true** and **false**.

Note that some operators can beave as unary and binary in defferent functionality, for example the minus sign **–** is considered unary operator while the subtraction sign **–** is binary operator.

```
-A;           //  minus sign unary operator
A-B;          //  subtraction sign unary operator
```

The most common operators are:



Warning

All the operators mentioned above can be overloaded except:

```
::      the scope resolution operator
?:      the conditional operator
.       the member access operator
sizeof size of operator
```

The idea behind operators overloading is to make the operators that works on simple data types also can work on objects, for example the increment operator **++** is defined in the language to work on numbers, we can overload this operator by adding additional functionality to it to make it work on objects also as we will see in the upcoming section.

14.2 Overloading Unary Operators

As we stated above unary operators take only one operand, in order to overload an operator to make it operate one objects we need to include a special member function named with the operator symbol preceded with the keyword **`operator`**.

For example, if we have the following student class, and we wanted to increment the student age by one, we need to implement a member function to do that. See Program 14.1

The member function simply increments the age data member by one when called in the main function Line 36.

We can overload the increment operator and make it work with objects so we can do this:

```
++s;
```

Instead of this:

```
s.incAge();
```

Program 14.1

```
// Student Class
#include<iostream>
using namespace std;
class student {
private:
string Name;
int ID, Age, Deg1, Deg2, Deg3;
public:
  student(){
  cout<<"Enter student ID: "; cin>>ID;
  cout<<"Enter student Name: "; cin>>Name;
  cout<<"Enter student Age: "; cin>>Age;
  cout<<"Enter student Deg1: "; cin>>Deg1;
  cout<<"Enter student Deg2: "; cin>>Deg2;
  cout<<"Enter student Deg3: "; cin>>Deg3;
  }

void incAge(){
   Age++;
}

void print(){
  cout<<"Student Name "<<Name<<endl;
  cout<<"Student ID: "<<ID<<endl;
  cout<<"Student Age: "<<Age<<endl;
  cout<<"Student Deg1: "<<Deg1<<endl;
  cout<<"Student Deg2: "<<Deg2<<endl;
  cout<<"Student Deg3: "<<Deg3<<endl;
}
};

int main(){
  student s;

  s.incAge();
  s.print();

  return 0;
}
```

Now to make the following statement work:

```
++s;
```

We need to add a special function inside the call we will name the function **++** preceded with the keyword **operator,** this function will be added:

```
void operator ++(){

  ++Age;

}
```

We can rewrite Program 14.1 adding the unary ++ overloaded operator, see Program 14.2, note the operator overloading function does not return a value to the calling line so we need to give it `void` as a return type.

Also note that if we write in postfix notation:

```
s++;
```

We need to implement another function to overload the postfix increment operator

```
void operator ++(int){

  Age++;

}
```

Note that the int added to the argument for the compiler to not confuse it with the prefix increment operator, the argument list must be unique for each function with the same name, remember this was a condition for function overloading, so here we have overloaded function for operator overloading, see the Program 14.3 both functions are added.

Video lecture: Operators Overloading (Unary Operators)
https://youtu.be/07x82MzCHDo

Program 14.2

```
// Unary Overloading
#include<iostream>
using namespace std;
class student {
private:
string Name;
int ID, Age, Deg1, Deg2, Deg3;
public:
   student(){
   cout<<"Enter student ID: "; cin>>ID;
   cout<<"Enter student Name: "; cin>>Name;
   cout<<"Enter student Age: "; cin>>Age;
   cout<<"Enter student Deg1: "; cin>>Deg1;
   cout<<"Enter student Deg2: "; cin>>Deg2;
   cout<<"Enter student Deg3: "; cin>>Deg3;
   }

void print(){
   cout<<"Student Name "<<Name<<endl;
   cout<<"Student ID: "<<ID<<endl;
   cout<<"Student Age: "<<Age<<endl;
   cout<<"Student Deg1: "<<Deg1<<endl;
   cout<<"Student Deg2: "<<Deg2<<endl;
   cout<<"Student Deg3: "<<Deg3<<endl; }

void operator ++(){
   ++Age; }
};

int main(){
   student s;
   ++s;
   s.print();

   return 0;
}
```

We can apply the same steps above to overload the decrement operator in both notations the prefix and the postfix.

Program 14.3

```
// Unary Overloading
#include<iostream>
using namespace std;
class student {
private:
string Name;
int ID, Age, Deg1, Deg2, Deg3;
public:
   student(){
   cout<<"Enter student ID: "; cin>>ID;
   cout<<"Enter student Name: "; cin>>Name;
   cout<<"Enter student Age: "; cin>>Age;
   cout<<"Enter student Deg1: "; cin>>Deg1;
   cout<<"Enter student Deg2: "; cin>>Deg2;
   cout<<"Enter student Deg3: "; cin>>Deg3;
   }
void print(){
   cout<<"Student Name "<<Name<<endl;
   cout<<"Student ID: "<<ID<<endl;
   cout<<"Student Age: "<<Age<<endl;
   cout<<"Student Deg1: "<<Deg1<<endl;
   cout<<"Student Deg2: "<<Deg2<<endl;
   cout<<"Student Deg3: "<<Deg3<<endl; }

void operator ++(){
   ++Age; }

void operator ++(int){
   Age++; }

};
int main(){
   student s;
   ++s;        // s++ also do the same thing
   s.print();

   return 0;
}
```

14.3 Overloading Binary Operators

The largest group of operators are binary operator, we can overload binary operators like we did with unary operators with some differences. We know that binary operators take two operands and should return a value. Since we are working with objects, overloaded binary operators work on objects and return a new object as a result to the intended operation.

Let's take an example on adding two vectors, for simplicity let's suppose a vector has only a starting point with two coordinates so in order to add two vectors we intend to add the corresponding coordinates to produce a new vector that represent the summation of two vectors, so we will overload the + operator.

Program 14.4

```cpp
// Binary Overloading
#include<iostream>
using namespace std;
class vector {
private:
int x, y;
public:
   vector(int a, int b){
   x=a;
   y=b;
   }

void print(){
   cout<<"X = "<<x<<" Y = "<<y<<endl;
}

vector operator +(vector v2){
   vector v3(x + v2.x, y + v2.y);
   return v3; }

};

int main(){
   vector v1(1,2), v2(3,4);

   v1.print();
   v2.print();

   vector v3 = v1 + v2;

   v3.print();

   return 0;
}
```

Video lecture: Operators Overloading (Binary Operators)
https://youtu.be/vVfDnyQsQt0

Exercises 14

Q1: Write an object-oriented program in C++ that overload the + operator to concatenate two strings.

Video lecture: Solution for Q1 - Exercises 14
https://youtu.be/CE9s0MVMR2w

Q2: Write an object-oriented program in C++ that overload the subscript **[ ]** operator.

Video lecture: Solution for Q2 - Exercises 14
https://youtu.be/AMMahVZEpZU

Chapter 15

Inheritance

15.1 Introduction

Another fundamental concept in object-oriented programming is the concept of **inheritance**, it is considered the third pillar of OOP in addition to encapsulation and polymorphism.

Inheritance also called code reuse, code reuse means that when we build a class for a particular concept, we can use this class and inherit its functionality solving some problem, this mechanism will save huge amount of time since we can reuse classes already had been built without the need to rebuild them.

This chapter present a new OOP feature which is inheritance and illustrate the basic types of inheritance and how to implement them in C++ language, but before we start, we need to know some basic definitions.

Base class: it is the class that we want to reuse its code (data members, member functions), also called parent class or superclass in java language.

Derived class: it is the class that is using the data and/or functionality of the base class through inheritance, also called child class or sub class in java language.

There are many types of inheritance depending on the connection between the base class (classes) and the derived class (classes), **Figure 15.1** illustrate the basic types.

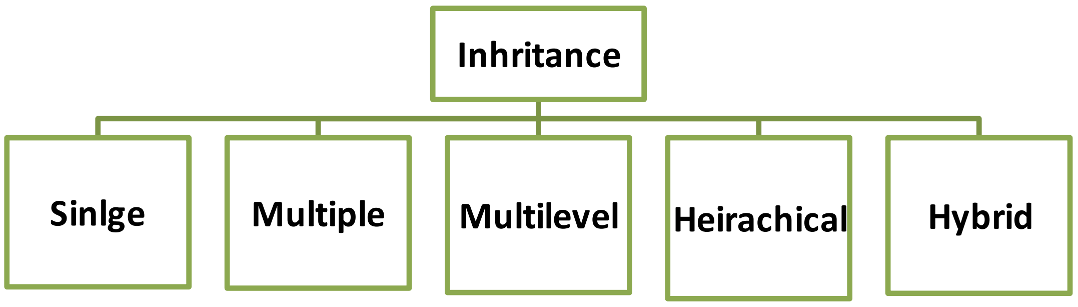


Figure 15.1, Inheritance Types

15.2 Single Inheritance

The simplest type it has a single base class and a single derived class, the data members and member function are inherited by the derived class directly.

As we have seen in the previous chapters, classes are meant to represent the real world we can represent anything in the world in classes, in inheritance the base class can be thought of the more general class and the derived class is the more specific class, to illustrate this idea let's take an example from the real world. If we wanted to represent a **person** in class and a **student** in another class, we can represent them in separate classes each of which have its own data members and member functions, but there are many data member that are common in both classes, every person has a name, age, length, weight, etc. also every student has a name, age, length, weight, etc. so instead of redundantly writing these data in both classes we can write then once in the base class and reuse them in the derived class.

Here we have the class person is the base class because it is the more general concept, and the student class is considered the derived class because it is the more specific concept. We can say every student is a person but not every person is a student.

Program 15.1 implement both classes, not the base class **person** is written first then the derived class **student** came second.

Program 15.1

```
#include<iostream>
using namespace std;
class person {
private:
string name;
int age, length, weight;
public:
   person(){
   cout<<"Enter Name: "; cin>>name;
   cout<<"Enter Age: "; cin>>age;
   cout<<"Enter Length: "; cin>>length;
   cout<<"Enter Weight: "; cin>>weight; }

void print(){
   cout<<"Name: "<<name<<endl;
   cout<<"Age: "<<age<<endl;
   cout<<"Length: "<<length<<endl;
   cout<<"Weight: "<<weight<<endl; }
};

class student : public person {
private:
int level;
float avg;
public:
   student(){
   cout<<"Enter Average: "; cin>>avg;
   cout<<"Enter Level: "; cin>>level; }

void print(){
   person::print();
   cout<<"Average: "<<avg<<endl;
   cout<<"Level: "<<level<<endl; }
};

int main(){
   student s;

   s.print();

   return 0; }
```

Program 15.1 Line 21

```
class student : public person
```

This line mean connect the class student with the class person using the colon operator **(:)** here the person class is the base class and the student class is the derived class. The inheritance type is public more on this later.

We can visualize the single inheritance with Figure 15.2

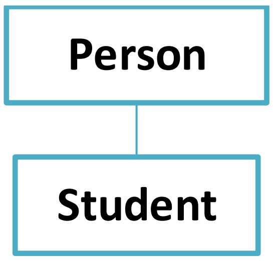


Figure 15.2, Single Inheritance

Note that creating an object from the derived class will automatically trigger the base class constructor then the derived class constructor.

If we want to send values in the derived class object's definition take a look at Program 15.2

Lines 26 and 27 the head of the student constructor will receive all the data from the main function and pass the inherited data to the parent constructor the person constructor, and take the values of level and avg to the student's data member

Program 15.2

```
#include<iostream>
using namespace std;
class person {
private:
string name;
int age, length, weight;
public:
  person(string n, int a, int len, int w){
  name=n;
  age=a;
  length=len;
  weight=w; }

void print(){
  cout<<"Name: "<<name<<endl;
  cout<<"Age: "<<age<<endl;
  cout<<"Length: "<<length<<endl;
  cout<<"Weight: "<<weight<<endl; }
};

class student : public person {
private:
int level;
float avg;
public:
  student(string n, int a, int len, int w, float
v, int lev): person(n, a, len, w){
 avg=v;
 level=lev; }

void print(){
  person::print();
  cout<<"Average: "<<avg<<endl;
  cout<<"Level: "<<level<<endl; }
};

int main(){
  student s ("ali",20,170,75,95,2);
  s.print();

  return 0; }
```

Important note: the inherited data from the base class cannot be accessed directly in the derived class due to encapsulation, if we want to access these data directly, we need to use the access specifier `protected` in the base class, protected means the data will be private in both the base and the derived classes.

Video lecture: Inheritance (Single Inheritance)
https://youtu.be/1nhlK34BEyY

Video lecture: Inheritance (Single Inheritance with constructors)
https://youtu.be/FOh-jjmVFt8

15.3 Multiple Inheritance

This type of inheritance when we have more than one base class to a derived class as illustrated in the Figure 15.3

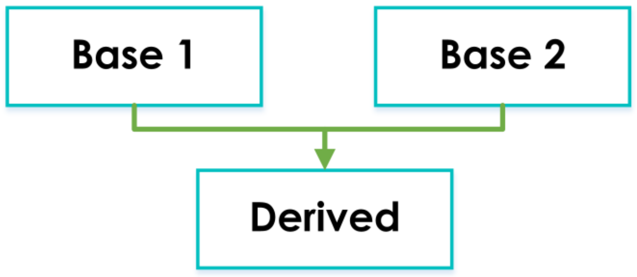


Figure 15.3, Multiple Inheritance

The derived class will inherit the data members and member functions of both base classes, let carry on the same previous example by adding another class **gradStudent** that represent a graduate student, that will inherit from the **person** and the **student** classes, see Program 15.3

Program 15.3

```
#include<iostream>
using namespace std;
class person {
private:
string name;
int age;
public:
  person(string n, int a){
  name=n;
  age=a; }

void print(){
  cout<<"Name: "<<name<<endl;
  cout<<"Age: "<<age<<endl;}
};

class student {
private:
int level;
float avg;
public:
 student(float v, int lev){
  avg=v;
  level=lev; }

void print(){
  cout<<"Average: "<<avg<<endl;
  cout<<"Level: "<<level<<endl; }
};

class gradStudent: public person, public student
{
private:
string research;
public:
  gradStudent(string n, int a, float v, int lev,
string r):person(n,a),student(v,lev)
{
 research=r; }
```

Lines 31 and 32 specify the multiple inheritance by specifying both base classes person and student.

Lines 36 and 37 the derived class constructor invoke the parent classes constructors and sending the appropriate data.

Program 15.3 (cont.)

```
41
42      void print(){
43       person::print();
44       student::print();
45        cout<<"Research: "<<research<<endl; }
46      };
47
48      int main(){
49       gradStudent s ("Sara",20,75,2,"AI");
50       s.print();
51
52       return 0; }
```

Video lecture: Multiple Inheritance
https://youtu.be/-CsX5AAU9U0

15.4 Hierarchical Inheritance

Hierarchical inheritance is the opposite of the multiple inheritance, here we have one base class and multiple derived classes, as illustrated in Figure 15.3

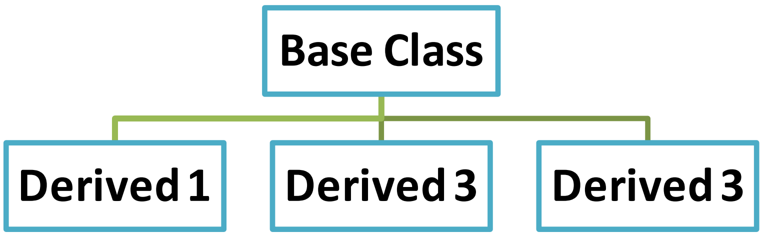


Figure 15.3, Hierarchical Inheritance

For example, id we have an **Employee** class with the data members (**name**, **age**, **salary**) and we derive a class of **Manager** that will inherit all the data from the employee class and has the data member (**title** as string), another derived class **Scientist** also will inherit from the employee class with a data member of its own (**publication** as string), the third class is **Worker** will inherit from the employee class with no data of its own, the three classes are illustrated in Figure 15.4

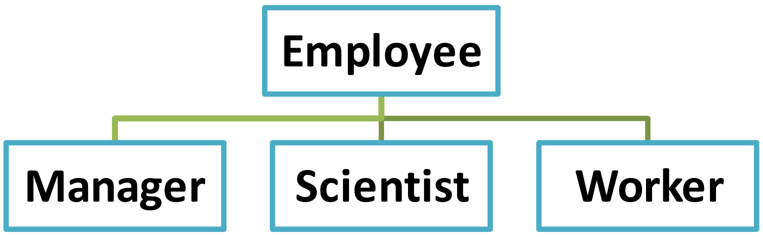


Figure 15.4, Hierarchical Inheritance

Program 15.4 implement the four classes with the appropriate inheritance connections, and constructors' callings.

Line 19 connects the Manager class with the Emp class

Line 32 connects the Scientist class with the Emp class

Line 47 connects the Worker class with the Emp class

The appropriate data are assigned to the classes data members and the base class data are sent to the base class constructor from the derived classes constructors.

Video lecture: Hierarchical Inheritance
https://youtu.be/UYvHqAug-rE

Program 15.4

```
#include<iostream>
using namespace std;
class Emp {
private:
string name;
int age, salary;
public:
  Emp(string n, int a, int s){
  name=n;
  age=a;
  salary=s; }

void print(){
  cout<<"Name: "<<name<<endl;
  cout<<"Age: "<<age<<endl;
  cout<<"Salary: "<<salary<<endl;}
};

class Manager : public Emp{
private:
string title;
public:
 Manager(string n, int a, int s, string t):
Emp(n, a, s){
  title=t; }

void print(){
  Emp::print();
  cout<<"Title: "<<title<<endl; }
};

class Scientist: public Emp {
private:
string publication;
public:
  Scientist(string n, int a, int s, string p):
Emp(n,a,s){
 publication=p; }
```

Program 15.4 (cont.)

```

    void print(){
     Emp::print();
       cout<<"Publication: "<<publication<<endl;  }
    };

    class Worker: public Emp {
    public:
       Worker(string n, int a, int s): Emp(n,a,s){ }

    void print(){
     Emp::print();}
    };

    int main(){

    Manager *M;
    M = new Manager("Ali",54,7000,"HR Manager");
    M->print();

    Scientist *S;
    S = new Scientist("Ahmed",43,5000,"AI Tech.");
    S->print();


    Worker *W;
    W = new Worker("Hasan",22,2000);
    W->print();

    delete M;
    delete S;
    delete W;

    return 0;

    }
```

15.5 Multilevel Inheritance

Multilevel inheritance is when we have more than one level of inheritance, it is the type when a derived class become a base class to another class, as illustrated in Figure 15.5

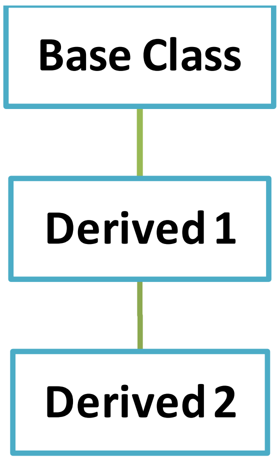
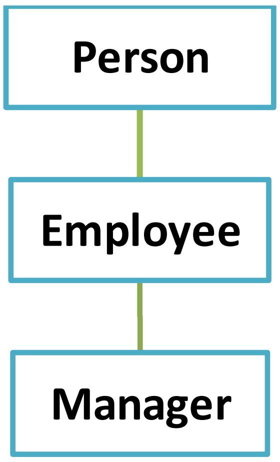


Figure 15.5, Multilevel Inheritance

The derived 1 class will be a derived class from the base class and a base class for the derived 2 class. The most general concept is considered a first level base class, the next specific class is considered the base class for the more specific class and so on. This process can be extended to any number of levels, to illustrate the idea let's take the following example:

Since every manager is an employee and every employee is a person we can organize the three classes in the following multilevel inheritance, as the following figure explain, and program 15.5 as the implementation of these classes.



Program 15.5

```
#include<iostream>
using namespace std;
class Person {
private:
string name;
int age;
public:
   Person(string n, int a){
   name=n;
   age=a; }

void print(){
   cout<<"Name: "<<name<<endl;
   cout<<"Age: "<<age<<endl; }
};

class Emp : public Person{
private:
int salary;
public:
   Emp(string n, int a, int s):Person(n,a){
   salary=s; }

void print(){
   Person::print();
   cout<<"Salary: "<<salary<<endl; }
};

class Manager : public Emp{
private:
string title;
public:
 Manager(string n, int a, int s, string t):
Emp(n, a, s){
   title=t; }

void print(){
   Emp::print();
   cout<<"Title: "<<title<<endl; }
};
```

Program 15.5 (cont.)

```

 int main(){

   Manager M("Ali",54,7000,"HR Manager");
   M.print();

   Emp A("Ahmed",43,5000);
   A.print();


   Person S("Saif",34);
   S.print();

   return 0; }
```

Video lecture: Multilevel Inheritance
https://youtu.be/YhiGXZdQuI8

The **Hybrid Inheritance** is in fact the combination of two or more types of inheritance with the same methodologies used in connecting the classes and constructors. For example, combining the multilevel inheritance with hierarchical inheritance we may have the following figure

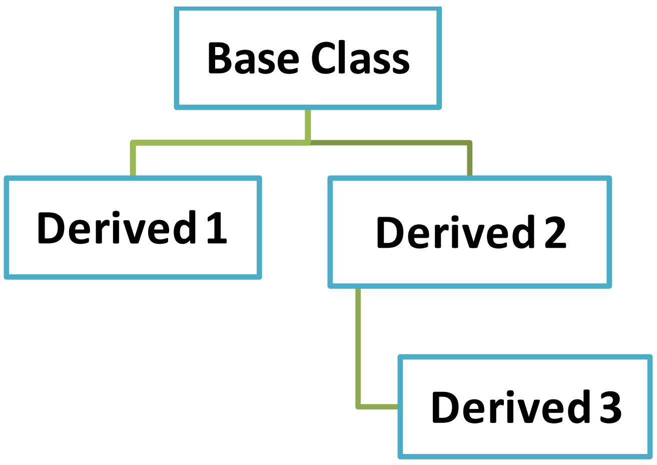


Figure 15.6, Hybrid Inheritance

15.6 Private, Public and Protected Inheritance

There are access specifiers for the inheritance they can be specified by the following table:

Base Class Data	Derived Class		
	Public Inheritance	**Private** Inheritance	**Protected** Inheritance
• Private • Protected • Public	• No Access • Protected • Public	• No Access • Private • Private	• No Access • Protected • Protected

For example, if the base class has public data and methods and the inheritance specifier was private:

```
class baseClass : private derivedClass
```

this means that the inherited data and methods will be private in the derivedClass.

Video lecture: Public, Protected & Private Inheritance
https://youtu.be/fz2tjOqr9ts

15.7 Overriding Methods

The concept of Methods overriding simply means that if you have a method in the derived class and an **inherited** method with the **same name** in the base class and you created an object from the derived class and called this method. Which one will be executed? The method in the derived class or the inherited method?

The answer is the method in the derived class will be executed, we say the method in the derived class will override the inherited method that has the same name. for example, see Program 15.6

Program 15.6

```cpp
#include<iostream>
using namespace std;
class A {
public:
   void test(){
   cout<<"this is a test from A class"<<endl;
}
};

class B : public A{
public:
   void test(){
   cout<<"this is a test from B class"<<endl;
}
};


int main(){

   A Obj1;
   Obj1.test();

   B Obj2;
   Obj2.test();

   Obj2.A::test();

   return 0;
}
```

The output will be:

```
this is a test from A class
this is a test from B class
this is a test from A class
```

Program 15.6 Line 20, declare an object from the base class. **Line 21** call the member function of the base class (no inheritance here).

Line 23, declare an object from the derived class (which now has two test methods). **Line 24** call the test method of the derived class; the test method of the derived class will **override** the test method in the base class.

Line 25, call the test method of the base class canceling the overriding feature.

The benefit of the override feature is that we can customize the methods in the base class without changing their code.

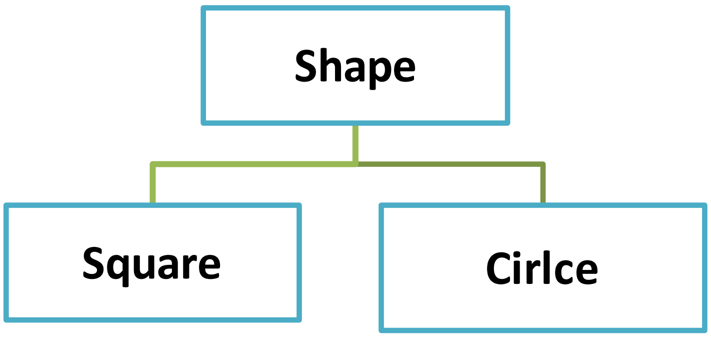
Video lecture: Methods Overriding
https://youtu.be/SLkCGDIlhuA

15.8 Virtual Functions

There is a problem in inheritance that will appear if we have the following:

- Inheritance of any type
- Overridable method (method with the same name in base and derived classes)
- Pointer to object in the main function

For example, if we have the following classes all of them have the **`area ()`** method: (Program 15.7)



Program 15.7

```
#include<iostream>
using namespace std;
class Shape {
public:
   void area(){
   cout<<"this is a shape area"<<endl;
}
};

class Square : public Shape{
public:
   void area(){
   cout<<"this is a square area"<<endl;
}
};

class Circle : public Shape{
public:
   void area(){
   cout<<"this is a circle area"<<endl;
}
};

int main(){

   Square S;
   Circle C;

   Shape *Sh1 = &S;
   Sh1->area();

   Shape *Sh2 = &C;
   Sh2->area();

   return 0;
}
```

Lines 26, define an object from the derived square class, Line 27 define an object from the derived class circle.

Line 29, define a pointer to object of the base class that points to the square object defined earlier, Line 30 calling the area method for this pointer to object will invoke the area of the base class, the override feature now is broken.

Line 32, define a pointer to object of the base class that points to the circle object defined earlier, Line 33 calling the area method for this pointer to object will invoke the area of the base class, the override feature now is broken.

We use the virtual function in this situation to restore the override feature, so the area method of the base class should be like this:

```
virtual void area(){
  cout<<"this is a shape area"<<endl;
}
```

By adding the keyword **`virtual`** the appropriate method will be called at runtime this is called dynamic dispatching or (late binding) and it is a type of polymorphism.

Virtual function*: is an inheritable and overridable function or method for which dynamic dispatch is facilitated. This concept is an important part of the (runtime) polymorphism a virtual function defines a target function to be executed, but the target might not be known at compile time.*

Note that the virtual function if it was empty it is called pure virtual function written like this

```
virtual void area()=0;
```

Video lecture: Virtual Functions
https://youtu.be/ZMYkxtVXcq8

Chapter 16

Templates

16.1 Introduction

Templates are considered a useful technique in programming with C++, it allows us to build programs that will work on multiple data types.

For example, if we have a program for the stack data structure (or any data structure) we know that it has an array that will hold the items that we want to push inside this stack, we have to define this array and specify a data type to this array.

By specifying a fixed data type for this array, we can work only with a stack that holds integers, if we wanted to work with a stack that holds characters, we need to build another stack program for that.

Fortunately, there is another way that enable us to work with the stack with any data type that we want by implementing only **one** program by using a technique called **templates**.

16.2 Function Templates

We can write templates for functions and classes, let's talk first about function templates.

Suppose you want to build a function that return the max number from two numbers, you may code it like the following simple program 16.1:

Program 16.1

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

int maximum(int n1, int n2) {
  if(n1 > n2)
    return n1;
  else
    return n2;
}

int main(){

  int number1, number2;
  cout<<"Enter Two Numbers: ";
  cin>>number1>>number2;

  cout<<maximum(number1, number2) <<endl;

  return 0;
}
```

Now what if you want to test two float numbers, or two doubles or even two characters, you have to build a function for every data type.

There is a way by building only **one** function and it will work for any data type that we give it using the template technique, see the following program 16.2

Program 16.2, Finding the maximum using template function

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

template <class T>

T maximum(T n1, T n2) {
  if(n1 > n2)
    return n1;
  else
    return n2;
}

int main(){

  int number1, number2;
  cout<<"Enter Two Numbers: ";
  cin>>number1>>number2;

  cout<< maximum(number1, number2)<<endl;

  char char1, char2;
  cout<<"Enter Two Characters: ";
  cin>> char1>> char2;

  cout<< maximum(char1, char2)<<endl;

  float f1, f2;
  cout<<"Enter Two Floats: ";
  cin>> f1>> f2;

  cout<< maximum(f1, f2)<<endl;

  return 0;
}
```

Note that in program 16.2 we built one function that can handle three different data types, in **Line 19** we sent two integers, in **Line 25** we sent two characters, in **Line 31** we sent two floats.

The template statement in **Line 4** declares **T** as a class template also called **template argument**, that will take any data type we specify, simply **T** will be replaced by the type of the passed parameters at runtime.

We say that the function maximum is a template function that works on any data type the we pass to it. The compiler will generate the function code for the passed argument when the function is called.

Let's take another example, for finding an element and return its index see program 16.3

Program 16.3 illustrate the idea of function template by using a single function to search arrays of different types.

Line 17 we passed an array of integers and an integer value to find its location in the array

Line 20 we passed an array of characters and a character to find its location in the array

Line 23 we passed an array of floats and a float number to find its location in the array

Program 16.3, Finding the location of an element using template function

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

const int size = 5;
template <class S>

int find(S array[size], S element) {

   for(int i=0; i<size; i++)
     if(array[i]== element)
       return i;
}

int main(){

   int a[size] = {1, 2, 3, 4, 5};
   cout<< find(a, 4)<<endl;

   char c[size] = {'a', 'b', 'c', 'd', 'e'};
   cout<< find(c, 'e')<<endl;

   float f[size] = {1.3, 3.4, 6.5, 7.9, 8.5};
   cout<< find(f, 1.3)<<endl;

   return 0;
}
```

16.3 Class Templates

The idea of template is the same in classes we can build classes that deal with different data types using a single class, rather than building a class for each data type.

For example, the stack class, program 16.4

Program 16.4, implementing a stack class template

```
#include<iostream>
using namespace std;
template <class S>
class stack
{
    S item[100];
    int top;

public:
    stack(){
        top=-1; }

    void push(S value){
        item[++top]=value; }

    S pop(){
        return(item[top--]); }
};

int main() {
    stack<int> s1;

    s1.push(10);
    cout<<s1.pop()<<endl;

    stack<char> s2;
    s2.push('A');
    cout<<s2.pop()<<endl;

    stack<string> s3;
    s3.push("Saif");
    cout<<s3.pop()<<endl;
}
```

Note that we specified the type of the items array as a template variable also the value in the push method's argument and the return type of the pop method.

We do not need to make the top variable as a template variable because the top is always integer pointing at the top of the stack.

Lines 21 we specified the stack will work on integer values.

Lines 26 we specified the stack will work on characters values.

Lines 30 we specified the stack will work on string values.

And so on…

Video lecture: Templates
https://youtu.be/tXBEQl8U3sY