# Objects III

**T2** Chapter **11**

## Inheritance & Polymorphism

**Inheritance** and **Polymorphism**, together with **Encapsulation** and **Data-Hiding** are the four pillars of object-oriented programming. In Objects II, we looked at how *methods* and *member variables* through access qualifiers like *public* and *private* provide us with the ability to encapsulate and hide internal details of how an object provides its functionality. Through these features it becomes possible for us to write flexible code that can be updated with relative ease when the requirements of the application change, as they inevitably do. Often times, we want to be able to work with several objects through the same interface. The ability for an object to present a common interface while being different objects is called *polymorphism* and in object-oriented programming and C++ in specific, this is traditionally achieved through the mechanism of *inheritance*.

Consider the following example program illustrating inheritance,

#include <iostream>
  
#include <string\_view>
  
#include <cstdint>
  
  
/\*\*
  
 \* @brief An Account is associated with the customer's name and account number.
  
 \* @note An Account cannot be constructed directly. Use SavingsAccount or CurrentAccount
  
 \* instead.
  
 \*/
  
class Account
  
{
  
 protected:
  
 std::string\_view name;
  
 uint64\_t number;
  
 double balance;
  
  
 protected:
  
 Account(std::string\_view name, uint64\_t number, double balance) :
  
 name{name},
  
 number{number},
  
 balance{balance}
  
 {}
  
  
 public:
  
 virtual auto display\_balance() const -> double
  
 {
  
 return balance;
  
 }
  
};
  
  
/\*\*
  
 \* @brief SavingsAccount is an Account that accrues a fixed interest on the current balance.
  
 \*/
  
class SavingsAccount final : public Account
  
{
  
 double interest;
  
  
 public:
  
 SavingsAccount(std::string\_view name, uint64\_t number, double interest, double starting\_balance) :
  
 Account{name, number, starting\_balance},
  
 interest{interest}
  
 {}
  
  
 public:
  
 auto display\_balance() const -> double override
  
 {
  
 return balance + (interest \* balance);
  
 }
  
};
  
  
/\*\*
  
 \* @brief CurrentAccount is an Account without an accruing interest rate.
  
 \*/
  
class CurrentAccount final : public Account
  
{
  
 public:
  
 CurrentAccount(std::string\_view name, uint64\_t number, double starting\_balance) :
  
 Account{name, number, starting\_balance}
  
 {}
  
};
  
  
auto print\_account\_balance(const Account& acc) -> void
  
{
  
 const double b = acc.display\_balance();
  
 printf("balance: %lf\n", b);
  
}
  
  
  
int main()
  
{
  
 SavingsAccount sa {"Ravi", 123456, 0.05, 100'000};
  
 print\_account\_balance(sa);
  
  
 CurrentAccount ca {"Ravi", 123456, 100'000};
  
 print\_account\_balance(ca);
  
}

We have defined three classes Account, SavingsAccount and CurrentAccount, the latter two of which, as the naming would suggest, are intended to be used as an Account irrespective of whether an object is SavingsAccount or CurrentAccount. In other words, we intend to use SavingsAccount and CurrentAccount polymorphically with Account. We want to establish an ***is-a*** relationship between SavingsAccount and CurrentAccount with Account. Specifically, SavingsAccount ***is-a*** Account, and CurrentAccount ***is-a*** Account.

There is a new terminology introduced inside the Account class - **protected**, which states that the member variables and methods marked protected are going to be accessible **only** by an inheriting or ***derived*** class. To the outside world, protected is exactly the same as private. It's only in the context of inheritance that the protected access qualifier becomes meaningful.

In the above program, Account is defined to hold some member variables and a constructor as protected, and expose one of its methods as public. Here too, there is a new terminology - **virtual** in front of a method, marking the method as overridable. In other words, a method marked as virtual can be overridden by a derived class - like SavingsAccount override-ing the display\_balance method. Notice that CurrentAccount though it inherits from Account does not override the display\_balance method, and consequently, inherits the method as defined by its parent.

Finally, we define a free function print\_account\_balance which accepts a const Account& to print the balance to the console. By virtue of being a reference to Account, a SavingsAccount or CurrentAccount or any other object of a class that inherits from Account can be given to the function. It is this uniformity in the public interface that an object presents that contributes towards better maintenance and reuse of code.

### Exercises

1. Define a Shape class implementing the computation of volume and area as virtual methods. Use inheritance and polymorphism to compute the volume and area of a rectangle, circle, square, ellipse and triangle. (M)

### Homework

**T2** Chapter **19** - Containers