Chapter 4

Object-oriented programming

At the end of the day computers just manipulate 0s and 1s, but binaries are very hard to humans understand. To write a program we might want to use higher abstract levels (Figure 4.1). High-level languages are simpler to write and understand and have more library support. Low-level languages are closer to hardware, can be more efficient (and more dangerous too).

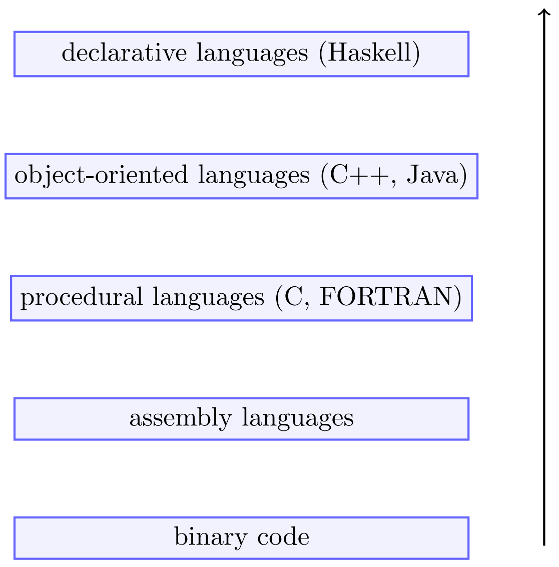
declarative languages (Haskell)
object-oriented languages (C++, Java)
procedural languages (C, FORTRAN)
assembly languages
binary code

Figure 4.1: Towards a higher level of abstraction.

Object-oriented programming refers to the paradigm where we try to see whole world in the form of objects. There are five object-oriented design(OOD) principles that might help programmers to develop software that are easy to maintain and extend:

- **Single-responsiblity** principle: a class should have one and only one job.
- **Open-closed** principle: objects or entities should be open for extension, but closed for

modification.

- **Liskov substitution** principle: let $q(x)$ be a property provable about objects of x of type T. Then $q(y)$ should be provable for objects y of type S where S is a subtype of T.
- **Interface segregation** principle: a client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.
- **Dependency Inversion** principle: entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

C++ is a higher level language that facilitates the building and use of objects. There is a set of principles and concepts that form the foundation of object-oriented programming in C++:

- **Object**: is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object;
- **Class**: it is an instance of a class;
- **Abstraction**: refers to, providing only essential information about a class to the outside world and hiding its background details
- **Encapsulation**: placing the data and the functions that work on that data in the same place
- **Inheritance**: the process of forming a new class from an existing class. The existing class is called base class, the new class formed is called as derived class;
- **Polymorphism**: the ability to use an operator or function in different ways
- **Overloading**: is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

4.1 Abstraction

C++ classes provides great level of data abstraction, because they provide mechanisms to expose public methods to the outside world to play with without actually knowing how class has been implemented internally. A class may contain zero or more access labels:

- Members defined with a *public* label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.

- Members defined with a *private* label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

Data abstraction provides two important advantages: class internals are protected from inadvertent user-level errors, which might corrupt the state of the object. The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

A good strategy is to have class members private by default unless we really need to expose them. That's just good **encapsulation**.

4.2 Inheritance

Any class type (whether declared with class-key class or struct) may be declared as derived from one or more base classes which, in turn, may be derived from their own base classes, forming an inheritance hierarchy. It prevents the programmer from unnecessary work and it is the inheritance concept that does this job. Inheritance gives us the facility to an object to inherit only those qualities that make it unique within its class.

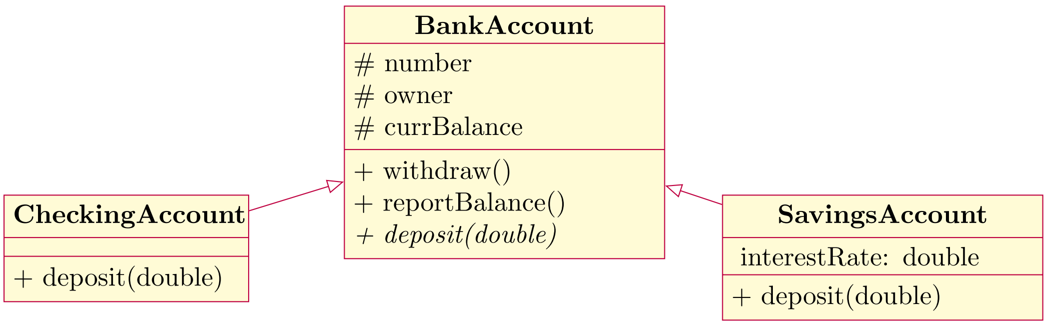

Figure 4.2: Inheritance example.

In the diagram shown in 4.2 `savings account` (child) is derived from `account` (parent). The concept of parent and child class was developed to manage generalization and specialization in object-oriented programming and it is represented by a *is-a* relationship. The generalization means that an object encapsulates common state behavior for a category of objects. The concept of inheritance makes code readable and avoids repetition.

When deriving a class from a base class, the base class may be inherited through *public*, *protected* or *private* inheritance. We hardly use protected or private inheritance, but public inheritance is commonly used. Table 4.1 summarizes the access control modifiers. See Program 25 for an example of inheritance.

Program 25 Inheritance example.

```
class Base
{
public:
  Base(int mprot, int mpriv):
    m_protected(mprot),
    m_private(mpriv)
  { std::cout << "calling Base class constructor\n"; }

  void method()
  { std::cout << "calling Base class method\n";  }

  void describe()
  {
    std::cout << "I am a base object. My members are: "
    << m_protected << " and " << m_private << endl;
  }
protected:
  int m_protected;
private:
  int m_private;
};

class Derived: public Base
{
public:
  Derived(int mprot, int mprivb, int mprivd):
    Base(mprot,mprivb),
    m_private(mprivd){};

  void method()
  { std::cout << "calling Derived class method\n"; }

  void describe()
  {
    std::cout << "I am a derived object. My members are: "
    << m_protected << " and " << m_private << endl;
  }

private:
  double m_private;
};

int main()
{
  Base b(1,1);
  b.method();
  b.describe();

  cout << endl;
  Derived d(1,1,2);
  d.method();
  d.describe();
  return 0;
}
```

Table 4.1: Access control and inheritance

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

4.3 Polymorphism

The word *polymorphism* means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Polymorphism is implemented using *virtual member functions* whose behavior can be overridden in derived classes. As opposed to non-virtual functions, the overridden behavior is preserved even if there is no compile-time information about the actual type of the class. If a derived class is handled using pointer or reference to the base class, a call to an overridden virtual function would invoke the behavior defined in the derived class. Program 26 shows an example.

The use of inheritance, run-time polymorphism and encapsulation is the most common definition of *object-oriented programming.*

4.4 Abstract classes

An *abstract class* is a class that can be used only as a base class. They represent abstract concepts, i.e. we use abstract classes for concepts that are generalization of common characteristics of related entities. For example, consider "vehicle" as an abstract concept (e.g., you can?t build a ?vehicle? unless you know what kind of vehicle to build). In C++, class `Vehicle` would be an ABC - Abstract Class, with `Bicycle`, `SpaceShuttle`, etc, being derived classes. In real-world OO, ABCs show up all over the place.

In Program 27 we made the class `Pet` abstract, meaning that all pets can talk, is a common characteristic of pets. But only "concrete" pets know how talk.

The notation `=0` says that the virtual function `Pet::run()` is pure; that is it must be overridden in some derived class. Since `Vehicle` has pure virtual functions it is not possible to create an object of class `Vehicle`. Classes with virtual functions tend to be pure interfaces; that is, they tend to have no data members (the data members will be in the derived classes).

Note that whenever you want to implement a new derived class you have to provide an im-

Program 26 Polymorphism example.

```cpp
class Pet
{
  public:
  Pet() {}

  virtual void talk()
  {    cout << "????\n"; }
};

class Cat: public Pet
{
  public:
  Cat(){}

  virtual void talk()
  {   cout << "Miau!\n"; }
};

class Dog: public Pet
{
  public:
  Dog(){}

  virtual void talk()
  {   cout << "Au! Au!\n"; }
};

int main()
{
  cout << "Type 1 for a Dog or 2 for a Cat: ";
  int opt;
  cin >> opt;

  Pet* p=0;
  if (opt == 1)
    p = new Dog();
  else if (opt == 2)
    p = new Cat();
  else
    p = new Pet();
  p->talk();
  return 0;
}
```

Program 27 Polymorphism example.

```
class Vehicle
{
public:
  Vehicle() {}
  virtual void run() = 0;
};

class Car: public Vehicle
{
  Car();
  void run();
}
```

plementation for every virtual member defined in the base class. It means reinforcing a behavior of the concept. Also, when you write code depending on the interface, it means that you can add new derived classes without modifying existing code.

4.5 Composition and Aggregation

Inheritance is used to express a *is-a* relationship among objects. For example "a dog is a pet", whenever you expect a `Pet` you can pass a `Dog`.

Another useful relationship is **composition**. It is used for objects that have a *has-a* relationship to each other. A car has-a metal frame, has-an engine, and has-a transmission. A personal computer has-a CPU, a motherboard, and other components. In composition the composite object has the ownership of the components. This means the composite is responsible for the creation and destruction of the component parts.

Aggregation differs from ordinary composition in that it does not imply ownership. In composition, when the owning object is destroyed, so are the contained objects. In aggregation, this is not necessarily true. For example, the data members of `Band` could consist of objects from the `Guitarist`, `Drummer`, and `Vocalist` classes. These objects are data members of the `Band` class, but not descendants or parent classes. They are related by composition, not by inheritance. When the band is destroyed its members can still exist and play in other bands.

IMPORTANT fact to make good inheritance decisions in OO design/programming: recognize that the derived class objects **must be substitutable** for the base class objects. That means objects of the derived class must behave in a manner consistent with the promises made in the base class' contract.

The ideal for software is not ot build a single program that does everything. The ideal is to build lot of classes that closely reflect our concepts and that work together to build our applications elegantly, with minimal effort (relative to the complexity of our task), with adequate performance, and with confidence that results produced are adequate. Such programs are comprehensible and maintainable in a wa that code that simply thrown together to get a particular job done is quickly as possible is not. Classes, encapsulation (as supported by `private` and `protected`), inheritance (as supported by class derivation), and run-time polymorphism (as supported by class derivation) are among our most powerful tools to structure systems.