

# Object Oriented Programming

Christoph Neumann

March 19, 2004

## 1 Introduction

Computers are more than overgrown calculators. This might be obvious today, but at the dawn of computing not all people saw computers as more than very fast calculating machines. As computers have become faster and cheaper, software has become more interactive and complex. Computers are now applied to a wider variety of problems than before. The principle cost for solving most problems is no longer the time spent running the machine, but the time spent developing the software. Because of this change, creating software in an efficient, human-oriented manner is very important. Object oriented programming seeks to ease the challenges of programming by allowing a programmer to create software in a way that more closely matches his way of thinking and to build new software using components he has previously created.

Fundamentally, the computer acts as a fast processing unit that retrieves numbers from memory, performs calculations on them, and stuffs the results back into memory. Many early programming languages, such as Fortran, were modeled in a manner that closely resembles the way computers work. This form of programming, known as imperative programming, performs computation by organizing sections of memory into logical units, called data structures, and operating on those units.

An imperative program, though easy to translate to machine code, is not a natural representation for problem solving. When using the imperative approach, one must first think of how the information will be stored in the machine and then how that information will be manipulated by the program. Unfortunately people usually have a fuzzy idea of what they want to accomplish; the clear picture only emerges as a person begins solving the problem. Object oriented programming addresses this design mismatch by allowing people to think about a program in general terms first while deferring details until later.

Instead of focusing on computation, object oriented programming (OOP) allows the programmer to focus on broad ideas like responsibility and behavior. OOP achieves this by modeling the problem as a community of interacting agents. Computation is performed through the interaction of the agents. This turns out to be a very natural way for humans to think about problems.

Suppose there is a room of people and one wants to find the age of the oldest person. One could tell everyone to stand up, find one other person standing,

exchange ages with that person, and have the younger of the two sit down. If this process is repeated until only one person is standing, that person will be the oldest in the room. No single entity calculated the maximum age in the room. The maximum age was found through the peoples' interaction. OOP takes this approach.

An imperative approach might involve creating a list of everyone's name and age, skimming down the list one-by-one while remembering the name and age of the oldest person seen so far. Notice how recording the name and age is an additional concern than simply finding the maximum age, and this concern is pivotal to solving the problem. The forced need to focus on detail is precisely what object oriented programming seeks to avoid.

In the example description, many details were left out that were not important for understanding the problem. No mention is made of how people are to find each other, if people communicate through hand signals or talking, or where people sit when they are done. If any of those details proves to be important, they can be elaborated on later. Throughout the solution to the problem, the focus remains on each persons' responsibility in solving the problem and what specific behavior they must employ to fulfill their responsibility. Consequently, OOP is naturally geared toward behavior-oriented design.

## 2 Classes, Objects, and Methods

Object oriented programming allows one to divide up the problem into smaller, self-contained units called objects. These objects interact with each other to perform the desired computation. Of course, there are many different kinds of objects and each kind has a different behavior. A kind, or type, of an object is termed a "class", and a specific object is an "instance" of a class. For example, a "person" is a type of thing, and you are a specific instance of a person. In OOP, objects are not designed, classes are; an object is then created from a class definition. By developing classes and specifying how instances of the classes interact with each other, large, difficult problems can be decomposed into understandable, behavior-oriented units.

Information hiding is the key facility for problem decomposition. The object keeps the details of operation private. No other object must know anything other than how to communicate with the given object. Whatever information the given object needs to implement its behavior must not be made available outside of the object. Information hiding allows an object to be a self-contained part of the solution. Connecting the objects forms the complete solution. By examining how the objects are connected, one can understand the solution to the problem without knowing all the details of how the solution is implemented.

Objects interact by sending messages to each other. Each message has a name and, optionally, arguments. In OOP languages, a message is implemented through a "method". Methods are like functions contained within a class. Unlike functions, methods have a receiver. In C++, we can define a function like this:

```
int printNum (int value) {
    cout << "The number is " << value << endl;
}
```

And we can define a class with a method like this:

```
class MyClass {
    public:
    int printNum (int value) {
        cout << "The number is " << value << endl;
    }
};
```

Note the similarity between the function and the method, but there are a few differences. The method is contained within the class and the `public` keyword allows the method to be accessed outside of the class.<sup>1</sup>

The following illustrates the difference between calling the function and the method:

```
printNum(42);           // function call
MyClass mine;          // create an object of type MyClass
mine.printNum(42);     // send printNum message to the object
```

Notice how the receiver `mine` is specified for the method call whereas no receiver is specified for the function call. The `printNum` message is passed to the `mine` object with 42 as an argument.

Together, the method name and arguments constitute the signature. Because each class may have its own methods, the same method signature may appear in more than one class. This mechanism allows different kinds of objects to behave differently given the same message. For example, we can define a different class with the `printNum` method.

```
class MyOtherClass {
    public:
    int printNum (int value) {
        cout << "My favorite number is " << 2 * value << endl;
    }
};
```

When an object of type `MyOtherClass` receives the `printNum` message, it behaves differently than an object of `MyClass`. Specifying different behaviors for identical messages is an important feature of OOP. With it, each class can be tailored to respond to a message in an appropriate way for the class's responsibilities.

---

<sup>1</sup>C++ also has a `private` keyword that ensures the method may only be accessed inside the class. Together, the `public` and `private` keywords are used to help the compiler enforce information hiding. See the section on "access modifiers" for more information.

### 3 Statefulness

Suppose we have a `Person` class implemented like this:

```
class Person {
    public:
    int getAge(void) {
        return 12;
    }
};
```

We can create as many instances of this class as we like, but each instance will always return the same age. We can vary the instances' behavior by storing information within each object called "data members". A data member is a variable defined in the class that each object has its own instance of. Methods can manipulate the data members and behave differently based on the data members' values. By using the `private` keyword, the data members are kept internal to the class. After modifying the previous class definition to add a data member, we get this:

```
class Person {
    public:
    int getAge(void) {
        return age;
    }
    void setAge(int years) {
        age = years;
    }

    private:
    int age;
};
```

Now, each object may contain a different value for `age`. The value is set and retrieved using the methods.

```
Person p1, p2;
p1.setAge(12);
p2.setAge(87);
p1.getAge();    // returns 12
p2.getAge();    // returns 87
```

Unfortunately, if we forget to call `setAge`, `getAge` returns junk. We need to initialize the data members to some default value when the object is first created. Using a "constructor", we can initialize the data members. The constructor is a method that is automatically called when the object is first created. In C++, the constructor has the same name as the parent. Our revised class definition is:

```

class Person {
    public:
    Person (void) {
        age = 0;
    }
    int getAge(void) {
        return age;
    }
    void setAge(int years) {
        age = years;
    }

    private:
    int age;
};

```

Now, when a `Person` object is created, the age will be initialized to 0.

Since methods embody the behavior of the class and they can act on data members, using data members allows objects of the same class to vary in behavior. Data members create flexible classes, and flexible classes promote code reuse. We can define one class for `People` and manipulate the data member values to customize different `People` objects.

## 4 Interfaces, Substitutability and Inheritance

Together, all of the method signatures constitute a class's interface. The interface specifies what messages the class can receive. Suppose class B has the same interface as class A, then class B, in some sense, should be able to serve as a substitute for class A. If we were to define an `Architect` class, we would expect to be able to send `Architect` objects the same messages as `Person` objects. After all, an architect is a person. However, if both the `Person` and `Architect` classes contain a `getCredentials` method, it would be appropriate for the `Architect` to provide different credentials than just any `Person`. By changing the methods but maintaining the same interface, we can make a more specialized version of a class. And, in some cases, it would be useful to substitute a more specialized object for a more general one.

It is also reasonable for the `Architect` class to respond to more messages than the `Person` class. After all, an architect knows how to do more than the average person. Without altering the original interface, the `Architect` class could expand the interface to accept messages like `designStructure` or `calculateSquareFootage`. This new `Architect` class should be able to act as a `Person` wherever needed as well as take on additional responsibility as an `Architect`.

This example illustrates the OOP notions of “substitution” and the “is-a” relationship. An architect “is-a” person, so an `Architect` object should be a suitable substitute for a `Person` object any place a `Person` object is used. And

since an architect is a person, it would be convenient to base the `Architect` class on the `Person` class so we only program the difference between the classes. OOP languages provide a construct called “inheritance” to provide for code reuse associated with “is-a” relationships. Inheritance establishes the is-a relationship and creates the new class based on the old one. In C++, we can reuse the definition for `Person` and perform inheritance like this:

```
class Architect : public Person {
    public:
        double calculateSquareFootage(double length, double width) {
            return length * width;
        }
};
```

The “:” after `Architect` means “inherits from”. So the first line reads, “The class called ‘Architect’ inherits from the ‘Person’ class.” The “public” means that all the public methods inherited from `Person` will also be defined as public in `Architect`.

Now, if we create an instance of `Architect`, we can call all of the `Person` methods as well as the additional methods defined within `Architect`.

```
Architect a;
a.setAge(41);
a.getAge();
a.calculateSquareFootage(10, 21.2); // the additional method
```

With inheritance, familial vocabulary is often used to describe relationships. In our example, `Person` is the “parent” class and `Architect` is the “child” class. Two different classes that share the same parent are termed “siblings”. Also, “descendant”, “grandchild”, “grandparent”, and “cousin” are often used.

Using inheritance, one can create a new class based on an existing class. This accomplishes two ends: common program code can be shared between the classes since only new behavior must be defined; and objects of the child class can serve as substitutes for objects of the parent class.

## 5 Access Modifiers

As mentioned before, C++ provides the keywords `public` and `private` to control access to the methods and data members of a class. These keywords establish the visibility of things inside a class for code outside a class. If something is declared public, it can be accessed both inside or outside the class. If something is declared private, it can be accessed only within code for the class. C++ also defines the keyword `protected` to establish special visibility for descendants. Methods and data members declared `protected` are only visible within the class and the descendants of the class. With the previous definition for `Architect`, the `age` data member is not accessible in the child. By changing

the `age` data member to `protected`, the `Architect` can modify the age directly. The `Person` class is abbreviated for clarity.

```
class Person {
    ...
    protected:
    int age;
};
```

Now, we define the `getOlder` method that acts on `age`.

```
class Architect : public Person {
    public:
    double calculateSquareFootage(double length, double width) {
        return length * width;
    }

    void getOlder(void) {
        age = age + 1;
    }
};
```

When `getOlder` is called, `age` will be modified even though `age` is defined in the `Person` class.

Access modifiers allow one to specify what is visible to the class, the descendant classes, and all other classes. Using access modifiers helps one apply the object oriented principle of information hiding.

## 6 Dynamic and Static Typing

Recall that OOP allows objects with the same interface to serve as substitutes for each other. This principle is generally true, but different languages have different restrictions. Statically typed languages associate a class (a “type”) with a variable at compile time. Variables may only contain instances of the specified class or classes descending from the specified class. Dynamically typed languages associate a class with the current value of a variable during runtime. Variables may contain any value. With static typing, an object may only be a substitute for another object if it was created as a product of inheritance. With dynamic typing, simply having the same interface is sufficient. The inheritance requirement is stronger because it mandates a relationship in addition to guaranteeing the same interface.<sup>2</sup>

If dynamically typed languages are so flexible, why would someone use a statically typed language? Statically typed languages can verify if a program is type-correct at compile time. Dynamically typed languages place the responsibility of type verification on the programmer. If the programmer makes a

---

<sup>2</sup>Modern statically typed OOP languages such as Java and C# address the need for unrelated classes to serve as substitutes by introducing a construct called the “interface”.

mistake, a dynamically typed program can have type errors during runtime. With that said, dynamic typing can easily solve certain problems that are difficult with static typing. One such problem is implementing lists that may contain values of any type.

## 7 Polymorphism

Variables in OOP languages are said to be “polymorphic” because they may contain values of different types. Polymorphism simply means “many forms”. The same variable could contain a value of type `Person` and then later a value of type `Architect`. As noted above, the restrictions on values are language specific, but in general, every OOP language allows the programmer to create a variable that may be assigned values of different types.

In addition to polymorphic variables, other forms of polymorphism exist within OOP. The most notable are “overloading” (ad hoc polymorphism) and “overriding” (inclusion polymorphism).

Overloading occurs when a new method is defined with the same name as an existing method but with a different signature. The method is said to be “overloaded” because the name of the method has more than one meaning. For example, in C++, we can overload the `printNumber` method like so:

```
class Person {
public:
    void printNumber(int num) {
        cout << "My favorite number is " << num << endl;
    }
    void printNumber(int num1, int num2) {
        cout << "My favorite numbers are " << num1
            << " and " << num2 << endl;
    }
};
```

We then call the methods like this:

```
Person p;
p.printNumber(12);
p.printNumber(16, 42);
```

Notice how the message `printNumber` is being passed both times, but the appropriate method is chosen based on matching the signature. Overloading allows an additional mechanism for message arguments to vary the behavior of a class.

Overriding occurs when behavior defined in the parent is changed in the child. C++, Java, and many other OOP languages use replacement semantics: the child replaces the behavior of the parent. Some other languages, such as Beta, use “refinement semantics”: the child adds behavior to the parent. To override a method, the method signature in the child must match the method signature in the parent. In C++ we can override a method like so:

```

class Parent {
public:
    virtual void printGreeting(void) {
        cout << "Hello, how are you?" << endl;
    }
};

class Child {
public:
    virtual void printGreeting(void) {
        cout << "Don't bother me! I'm eating." << endl;
    }
};

```

The “virtual” keyword indicates that we still want the method overriding to occur even when we only have a reference to an object. We can execute the methods by doing this:

```

Parent p;
p.printGreeting();
Child c;
c.printGreeting();

```

Which outputs:

```

Hello, how are you?
Don't bother me! I'm eating.

```

With method overriding, a child can replace the behavior defined by the parent with a new behavior. This allows behavior to vary based on where the class is located in the chain of inheritance. One use for overriding is to allow a child to perform a more specific behavior than the parent. For example, with the `Architect` class, we may want the `setAge` method to guarantee the provided value is at least 18 since it would not be reasonable for an architect to be a minor.

Polymorphism is the general idea that means that something may take on many different forms. A variable may take on different forms by containing values of different types. A message can take on different forms by varying its signature or where it is located in the chain of inheritance.

## 8 Summary

Object oriented programming languages were created to allow people to reuse components they have already built and to express solutions to complex problems in a natural manner. By focusing on behavior, OOP allows the programmer to design a general solution before concerning himself with the details. By using the technique of information hiding, OOP to allows a programmer to divide a

large problem into discreet, self-contained units known as classes. Instances of these classes communicate via messages (methods) to perform computation.

Data members allow the behavior of objects of the same class to vary. Substitutability allows the object used in a particular context to vary. Inheritance allows an existing class to be varied while preserving an interface. Access modifiers allow the visibility of data members and methods to vary. Polymorphism allows variables and methods to be varied without changing names.

The flexible nature of object oriented programming gives programmers a wealth of options for solving a problem. Practice has shown the principles of object oriented programming have made significant strides in the efficiency and efficacy of the programmer.

## 9 References

The material in this essay was drawn from the excellent object oriented programming book by Timothy Budd.

Timothy Budd. *An introduction to object-oriented programming*. Addison-Wesley, third edition, 2002.