Methodology First and Language Second:
A Way to Teach Object-Oriented Programming

Haibin Zhu

Department of Computer Science and Mathematics,
Nipissing University, 100 College Drive, North Bay,
Ontario, P1B 8L7, Canada
1-705-474-3461 ext. 4434

haibinz@nipissingu.ca

MengChu Zhou

Electrical and Computer Engineering Department,
New Jersey Institute of Technology, University
Heights, Newark, NJ 07102, USA
1-973-596-6282

zhou@njit.edu

ABSTRACT
C++ is a very successful object-oriented language. It is a required language for more and more students. It takes great effort and practice for these students to learn how to program in C++ and how to make object-oriented programs. One potential failure is that they have learned programming in C++ but do not know how to program in an object-oriented (OO) style. To avoid such failures, this paper proposes that first an object-oriented methodology is taught, and then the language itself. A six-step approach to teach the OO methodology is presented, followed by some innovative ways to teach different mechanisms in C++. In this way, students can master both object-oriented programming and C++ programming. The proposed teaching method is applicable to teaching other languages like Java and C#.

Categories and Subject Descriptors
D.1.5 [**Object-Oriented Programming**]; D.2.10 [**Software Engineering**]: Design -- *Methodologies*; D.3.2 [**Language Classifications**]: Object-oriented languages; D.3.3 [**Language Constructs and Features**].

General Terms
Design, Languages, and Theory

Keywords
Teaching Methodology, Object-Oriented Programming, C++

1. INTRODUCTION
Because of its efficiency, rich functions, fully supported object-oriented programming style, high flexibility and total compatibility with C programs, C++ is one of the world's most successful object-oriented languages. Most universities in the world teach C++ for students as a fundamental programming course. Thus, to enable them to master such a programming language, the study of teaching methods for teaching object oriented programming becomes an important topic. New teaching methods, e.g., icon-based method [2] and project-based one [11], have been proposed to help students gain programming knowledge and skills.

A basic problem facing them, as well as some other novice programmers and software engineers, is that they tend to code C++ programs in a non-object-oriented style, although the language is an object-oriented programming (OOP) language. For example, many programmers use C++ to write C programs, because C++ language keeps all the things in C. Even a faculty member may encounter such problems. Many faculty members transfer to teaching C++ from teaching other procedural languages. The transition to the object paradigm has been ugly because it is driven by a language choice not by an understanding of problem solving with an object orientation [13]. From another viewpoint, if people really master the object-oriented programming, they may even program an object-oriented program with a traditional language such as C. The latter may be encouraged while the former must be avoided. This is why many professors insist that the paradigm teaching be more important than the language teaching [9, 10]. A number of authors have discussed many aspects of teaching OOP with C++ [6, 14, 15], but only a few have discussed the relationships between teaching the methodology and teaching the language.

Hence, students are asked to pay more attention to the methodology and ideas when they learn to program in C++. With the object-oriented methodology in mind, software engineers can produce the highest-quality object-oriented program with C++. On the other hand, they may produce a low-quality program with C++ when they do not adhere to the principles of object-orientation. Low quality refers to "low readability, maintainability, and reusability."

This paper discusses the thoughts and suggestions obtained from the authors' multiple-year experience with application of object-orientation [21], teaching and practicing object-oriented programming with Smalltalk, C++ and Java [22, 23]. It is hoped that this paper is helpful to instructors who teach and students who learn the object-oriented programming. It is the most important for instructors to teach the object-oriented methodology first and C++ second in a C++ programming course. To facilitate in achieving this, Section 2 introduces six steps to teach object-oriented concepts. Section 3 discusses the way to teach inheritance in C++. Section 4 makes useful suggestions and comments on other important C++ concepts. Section 5 concludes this paper.

2. SIX STEPS TO INTRODUCE THE OBJECT-ORIENTED PROGRAMMING
Generally speaking, object-orientation is not only a programming style but also a methodology that deduces from general concepts to the special and induces from the special to the general. These methods are similar to human's natural thinking style. Therefore,

except for teaching the basic concepts of abstraction, information hiding, encapsulation, and modularity, instructors must show the students that it is a very powerful methodology for both thinking and programming. To introduce this fundamentally important methodology, the following six steps are proposed:

1) Discuss fundamental principles of object-orientation with respect to conventional thinking;
2) Introduce an object concept by observing the real world;
3) Acquire the class concept by abstraction of many common objects;
4) Introduce instantiation after the class concept is learned;
5) Illustrate subclasses by adding more details to an existing class and superclasses by finding common things among several classes;
6) (Optional) Discuss metaclasses to master completely the class and object concepts.

2.1 Discuss Fundamental Principles of Object-Orientation with Respect to Conventional Thinking

Among the difficulties that instructors face is the requirement to stimulate students' active thinking [11]. Therefore, new concepts and principles should be introduced with the help of examples in daily life.

In ordinary life, one uses thinking to know, understand, or reason something intuitively. The most used methodologies are induction and deduction. Induction means an abstraction process from the special to the general, e.g., one can compose a term "dog" after one knows many different real dogs. Deduction means a process from the general to the special. For example, in school, one learns the word "mammal" first, and then he or she can know if a dog is a mammal or not by testing if a dog has the properties a mammal should possess.

In a problem-solving field exist two basic design methodologies. One is functional decomposition, or called top-down design. Another is functional composition or bottom-up design. In functional decomposition, a whole system is characterized by a single function, and then the function is decomposed into a set of sub-functions in a process of stepwise refinement. At last, one can design or implement each small function and then complete the entire system design. In functional composition, one can have different components of functions such as those from a function library at first, then one can compose them into a module with a more significant function. At last, one can compose many modules into one large system.

Hence, in the introduction to the object-oriented programming, the general concept of object-orientation is shown from a methodology viewpoint [23].

Everything in the world is an object [4].

Every system is composed of objects (certainly a system is also an object).

The evolution and development of a system is caused by the interactions among the objects inside or outside the system.

The first statement says that everything in the world is an object. In the real world, flowers, trees, and animals are objects; students and professors are objects; desks, chairs, classrooms, and buildings are objects; universities, cities, and countries are objects; even the world and the universe are objects. In the abstract world, a subject, such as

electrical engineering, computer engineering, cybernetics, mathematics, and history, is also an object.

The second statement states that every system is composed of objects. For example, a cultural system includes history, language, food, costumes, relationships, and people etc. that are all objects; an educational system includes schools, students, professors, administrators, etc. that are also objects; an economic system includes objects like economic regulations, services, customers, and currency, etc.; a control system includes a plant to be controlled, a controller, sensors, actuators, and so on; a computer system includes monitor, keyboard, motherboard, CPU, memory, I/O devices, operating systems, and application software that are all objects.

The third statement describes the development of a system which is caused by the interactions. For example, the New Jersey Institute of Technology (NJIT) is a system. Its development is caused by the interactions among students, professors, staffs, the New Jersey state officers and even federal government officers of the country who are outside NJIT.

Object-orientation supports both induction and deduction methodologies. In the analysis phase, many objects can be used to form a class. This is an induction. On the other hand, when there are already some classes, more objects or called instances can be created from these classes. This is a process of deduction. In the latter, software engineers should have knowledge about the class library.

From the design viewpoint, object-orientation supports both top-down and bottom-up designs. Given a superclass, another class with more details may be designed. Here one uses the top-down design method. When a superclass is constructed based on a number of classes sharing many common properties, a bottom-up design method is being used.

In the top-down design view, one may have an object of a class that has many attributes that are objects of other classes $\{A_i\}$, and these objects may also have attributes that are instances of other classes $\{B_j\}$; … ; and this process is also called decomposition.

In the bottom-up design view, one may have many objects of different classes $\{X_i\}$; a new object may be constructed with these objects; by this new object one can introduce a new class; as a result, one can have many new classes $\{Y_i\}$ whose instances in turn can be constructed into additional objects; … ; and this process is also called composition.

Before other concepts are introduced, students can be asked to describe themselves by writing paragraph on a piece of paper.

2.2 Introduce an Object Concept by Observing the Real World

Among the difficulties that instructors face is the need to stimulate students' active thinking in a classroom [11]. Therefore, examples in daily-life should be used to introduce new concepts and principles.

Some professors argue that "Object-first" is not a good pedagogy for teaching object-orientation [8]. They think that the objects are derived from classes in the language viewpoint. The proposed method introduces the object concept before the class concept, because the class concept arises from the abstraction of ordinary objects. After classes are discussed, the understanding of the object concept can be reinforced based on the class concept.

From the view that everything in the world is an object, an object [12, 23] must be

- "uniquely named", i.e., any object should carry a unique name or identification;
- "created or destroyed"; and
- "communicative", i.e., an object can exchange messages with other objects;

and may be

- "nested", i.e., a complex object has other objects as its components (which in turn may have object components);
- "active and autonomous", i.e., an object is not controlled directly by people; and
- "collaborative", i.e., collaborative relationships between objects arise.

In daily life, "communicative" might be implicit. For example, a cart can be moved only when one pushes it. This situation can be seen in the following way. It is the cart that accepts a message "go" from someone, and it goes by responding to this message. Hence, the "communicative" property of a cart is implicit and "pushing" is one kind of message passing or communicating style. The objects that possess all the properties discussed above are also named agents [3].

Considering its general meaning and above properties, this paper proposes an object to be expressed by a quadruple.

Object $::= < \mathcal{N}, s, \mathcal{M}, \mathcal{X} >$, where

- $\mathcal{N}$ is an identification or name of an object;
- s is a state or a body represented by a set of attributes;
- $\mathcal{M}$ is a set of methods (also called services or operations) the object can perform; and
- $\mathcal{X}$ is an interface that is a subset of the specifications of all the methods of the object.

One can express every object by specifying these four elements.

At this step, students can be asked to describe who they are by using this formula, and to compare this description with what they described before.

2.3 Acquire the Class Concept by Abstraction of Many Common Objects

By induction, one can make an abstract concept by observing many concrete things in common. From the object-oriented viewpoint, one forms a class by this abstraction process. By analyzing the properties of abstraction results, one can define a class expressed by C of objects in a quadruple.

$C ::= < \mathcal{N}, \mathcal{D}, \mathcal{M}, \mathcal{X} >$, where

- $\mathcal{N}$ is an identification or a name of the class;
- $\mathcal{D}$ is a space description for memory;
- $\mathcal{M}$ is a set of the method definitions or implementations; and
- $\mathcal{X}$ is a unified interface of all the objects of this class.

It is easily observed that this formula is very similar to that of Object. Class is actually an abstraction of Object. It is used to express a group of objects with similar properties. The difference is that $\mathcal{D}$ in class C has no values and is just a space description or a specification, while s in Object is the real state expressed by the values stored in its private space. A class in a programming language is designed exactly the same as described in this definition.

At this step, students can be asked to describe what class they belong to and give the descriptions of each element of this class. At the same time, students are shown with the real class declaration in C++. By this illustration, they will be able to divide a C++ class into the four elements of this definition.

2.4 Introduce Instantiation after the Class Concept is Learned

Class is not only an abstract result of objects but also a template to create objects or called instances. At this step, because the class concept is already taught, the object concept can be redefined based on the class concept. For the instances created from a class, also from the implementation viewpoint, an object expressed by O can thus be redefined as a triple. That is to say, in a C++ or Java complier or in the Smalltalk system, an object is constructed as $O ::= < \mathcal{N}, C, S >$, where

- $\mathcal{N}$ is an identification or name of the object;
- C is the object's class identified by the class identification or name; and
- S is a body or an actual space whose values are called attributes, properties, or state.

Here, by creating objects from a class in C++, it can be illustrated that an object-oriented programming process with C++ is really the same as this definition describes.

At this very step, students are asked to describe who they are by designing a class at first, and then creating an instance.

2.5 Illustrate Subclasses by Adding More Details to an Existing Class and Superclasses by Finding Common Things among Several Classes

For a subclass, the classification idea should be emphasized. That is to say, even though the superclasses and subclasses are in different categories, a subclass expressed by C_s definitely inherits all the methods ($\mathcal{M}$), space description ($\mathcal{D}$) and interface ($\mathcal{X}$) of their superclasses. With this view, the following expression is reached.

$C_s ::= < \mathcal{N}, \{C\}, \mathcal{D}, \mathcal{M}, \mathcal{X} >$, where

- $\{C\}$ denotes a set of superclasses identified by their identifications or names, and
- $\mathcal{N}, \mathcal{D}, \mathcal{M},$ and $\mathcal{X}$ have the same meanings as those of C.

In this definition, $\{C\}$ is used to show the possibility of multiple inheritance. Please note that subclasses are called derived classes and superclasses are called base classes in C++.

At this step, students can be asked to describe who they are by designing a superclass, forming a subclass derived from the superclass, and creating instances at last. After they have completed the descriptions, they can compare the differences among the descriptions they have made and acquire the basic concepts of object-oriented programming.

Again, instructors can show that the C++ definition of a derived class can be completely described by the above expression.

By now, students have learned and built an object-oriented programming framework in mind. Instructors can then begin to introduce other OO and non-OO concepts together with C++.

2.6 (Optional) Discuss Metaclasses to Master Completely the Class and Object Concepts

This step is optional because there is no metaclass concept in C++. But the understanding of metaclass can really help students

understand the basic principle "Everything in the world is an object". The metaclass concept was introduced in Smalltalk to state its class hierarchy consistently [4]. A metaclass is actually a class of a class when this class is taken as an object. One thing can be sometimes a class, and an object at other times. In Figure 1, a rectangle expresses a class and an ellipse an object; a directed dotted arc means that a class can be taken as an object, and a directed bold arc means that an object belongs to a class.

From the authors' experience, even though this concept is difficult to understand, many students would like instructors to discuss it in detail and hope to understand the meanings as shown in Figure 1. Note that Metaclass and Class are two specific classes defined in the Smalltalk system.

After these steps, instructors can start the language teaching, and should be confident that the students can learn the language with correct thinking and with object-oriented methodology in mind.

One can use the following steps to understand the relationships among an object, a class and a metaclass in Smalltalk.
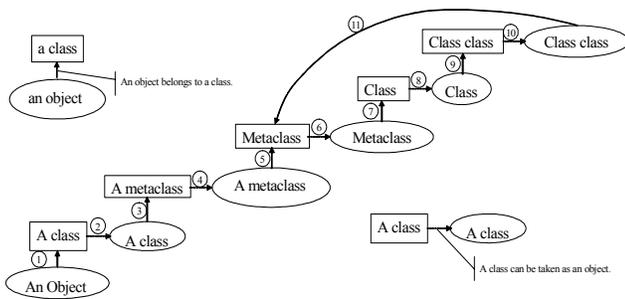
Figure 1. The Metaclass Concept in Smalltalk.

1) For any object, there is a class it belongs to.
2) A class can be taken as an object.
3) If one takes a class as an object, its class is a metaclass.
4) A metaclass can be taken as an object.
5) If one takes a metaclass as an object, its class is Metaclass.
6) Metaclass can be taken as an object.
7) If one takes Metaclass as an object, its class is Class.
8) Class can be taken as an object.
9) If one takes Class as an object, its class is Class class.
10) Class class can be taken as an object.
11) If one takes Class class as an object, its class is Metaclass, because Class class is a metaclass.

3. TEACHING INHERITANCE IN C++

3.1 Classification-Based View of Inheritance

In object-oriented technology, two basic activities are abstraction and classification. Classification is a central activity in object-oriented programming and distinguishes it from procedural programming [16].

In the process of classification, the inheritance concept is introduced to refine general ideas, and form subclasses and hierarchy of classes. In this sense, inheritance becomes an approach or a clue for abstraction (or specialization and generalization). Certainly it can be used to reuse code to some extent, but this is not its main function. Applying inheritance, software engineers can program their software when they have only elementary concepts about the classes used. Then, they can continuously update the software by adding subclasses till all classes and subclasses are completed. At last, some classes can be used for object instantiation.

In object-oriented programming, if some properties cannot be defined clearly, they should not be included in the class being defined; rather, they should be formed in some subclasses or other classes.

With a clear understanding of classification, one is free of the argument among the different kinds of inheritances, because the whole and simple inheritance is the natural meaning for inheritance under consistent classification. By this understanding, inheritance has no other meanings that may introduce inconsistency. This methodology is beneficial to ordinary management systems that generally have much classification [22].

3.2 Code Reuse-Based View of Inheritance

However, there exists another view of inheritance. It takes inheritance as a tool for code reuse [22]. From this viewpoint, classes become the objects to which inheritance is applied. In this sense, codes of classes are reused and classes are just a programming entity carrying no classification meaning. The major implication of inheritance is to reuse some classes' abilities and attributes. Therefore, the application of several kinds of inheritance becomes required. One subclass may inherit from several superclasses and multiple inheritance must be arranged. One subclass may use a superclass to form its own attributes and operations. If there are in the superclass some attributes and operations the subclass does not contain, partial inheritance must be used.

The different kinds of inheritance bring about the flexibility to reuse codes in classes, and make many previously designed codes easily usable in new software. Therefore, this methodology is welcome by many users who have already designed much code. With this understanding, a user may produce new software from old software by "cut and paste".

However, inconsistency can result from this methodology. Because users can reuse everything in the old software, they may think little about the resulting software's performance and architecture. Therefore, their software is often difficult to understand and modify. For example, when demonstrating the classification of animals, users may take "can fly" as the characteristic of a bird class at first, but after some time, they find that an ostrich cannot fly. They may form a new subclass by partial inheritance to avoid the attribute "can fly". In fact, it is not appropriate to take "can fly" as the attribute of class Bird. With a classification-based methodology, they should certainly exclude "can fly" from the attributes of class Bird [22].

Therefore, the suggestion to teach inheritance in C++ is to teach classification first, and reusability second.

3.3 Multiple Inheritance

The concept of multiple inheritance is introduced with the aim to increase the code reusability. Hence, as discussed above, it affects the consistency of classification if programmers pay insufficient attention to its usage. The following problems should be considered carefully.

- Multiple inheritance brings about replicated inheritances, and programmers must use a "virtual" specifier to make it clear.
- Multiple inheritance may be taken as a tool to reuse other classes. It is possible to implement a composition relation

by multiple inheritance. Such kind of inheritance examples can be found in early C++ textbooks [17, 18]. Even though many new textbooks teach multiple inheritance with better and more relevant examples [5, 7], instructors should still emphasize this possible misleading thinking. The implementation may be like this. The classes Head, Engine, Wing, and Tail are created at first, and then multiple inheritance is used to form a Plane class. In fact, a plane is composed of head, engines, wings, and tail. Some authors stated that aggregation is an inherent property of inheritance just like this [20], and that is why classification should be emphasized first. The class Plane should be implemented like the following with a composition to avoid implementing an aggregation via multiple inheritance.

```
class Plane
{
    private:
        Head head;
        Engine engines[4];   //4 engines
        Wing wings[2];              //2 wings
        Tail tail;
    public:
        … …
}
```

3.4 Inheritance and Overloading

Overloading in object-oriented programming is used when we hope to use the same name to express different functions that is kind of polymorphism. Overloading is generally resolved by a function's signature that is defined by the name, the parameter list of the function. However, in C++, we must pay more attention to overloading.

For example, in the following sample program (line 1, 2, and 3), we hope to get the results by binding b->display(a1) to function b_class::display(a1), b->play(a1) to b_class::play(a1), and b->play(a2) to b_class::play(a2). By the inheritance concept, d_class should inherit the functions b_class::display(A) and b_class::play(A) because they are different functions from d_class::display(A) and d_class::play(A). But the sample program cannot be compiled because C++ does not comply this regulation for inheritance. C++ accepts a scope concept and insists that no overloading across different scopes. "There is no overloading across scopes" and "A base class and its derived class are separate scopes", said the creator of C++ Bjarne Stroustrup.

```
#include <iostream>
using namespace std;
class A
{
        public:
        A(){x = 10, y =20;};
        int x,y;
};
class B: public A
{
    public:
        B(){i = 10, j =20;};
        int i,j;
};
class b_class
{
```

```
    public:  virtual void display(A  a)
        { cout<<"b_class, a= "<<a.x<<endl;
        }
    void play(A  b)
        { cout<<"b_class, b = "<<b.y<<endl;
        }
};
class d_class : public b_class
{
    public: void display(B f )
        { cout<<"d_class, f = "<<f.i<<endl;
        }
    void play(B g )
        { cout<<"d_class, g = "<<g.j<<endl;
        }
};
void main()
{
    d_class *b;
        a = new b_class();
        b = new d_class();
        A a1,a2;
        b->display(a1);             //(1)
        b->play(a1);                //(2)
        b->play(a2);                //(3)
}
```

4. TEACHING OTHER CONCEPTS IN C++

Bjarne Stroustrup, the creator of C++, said that C++ was not just an object-oriented language [19]. That is why it is suggested to instructors that teaching C++ should teach the methodology first. Otherwise, students may know much about the language but little about the object-oriented methodology.

But as a complete C++ course, instructors must teach the mechanisms such as friends, templates, operator overloading and exception handling that do not really adhere to the principles of the object-oriented methodology, in our opinions. Hence, they should teach students clearly that they are not object-oriented mechanisms, but they may help making an efficient object-oriented program.

4.1 Why Constructors and Destructors Cannot be Inherited

In C++, a derived class inherits all the things defined in its base class in general. Then, students often ask why constructors and destructors cannot be inherited.

From the viewpoint of object creation, a constructor should not belong to any relevant class. It actually belongs to the metaclass of this class that is a concept from Smalltalk. Hence, a metaclass is a fundamental concept for students to understand why constructors and destructors cannot be inherited.

Following this, another question may arise: "why can the destructors of a base class not be defined without a 'virtual' specifier? " A general answer is that there is an error report from the system for not releasing some spaces defined for a derived class if a "virtual" specifier is not given.

It would be understood more easily to illustrate that although the destructor names are different, they are taken as a single polymorphic function (every class has only one destructor) for a C++ compiler. "Virtual" specifiers are certainly needed to tell the

compiler that there are derived classes to be responsible for managing the spaces by themselves.

Polymorphism generally means that a single name has different meanings. The pointer mechanism is really polymorphic in C++ language, and it can be used to refer to any object. The pointers in C++ are very similar to the variables of Smalltalk that have no types. Without pointers, there would be no polymorphism in C++. Therefore, students are taught the casting between the pointers of base classes and those of the derived classes; they are told an analogy that "an apple can be put in a bag of fruit, but a fruit cannot be put in a bag of apple." The fruit bag is an analog of a pointer to a base class Fruit, and the apple bag is an analog of a pointer to a derived class Apple. This makes students understand the concepts easily and clearly.

4.2 Polymorphism and Pointers

4.2.1 About "this"

From the object-orientation viewpoint, each C++ function call should be invoked by a message. That is to say, there should be an object as a receiver for a message in a statement of a program. But in C++, there are many function calls in the class implementation in which one cannot find the message receiver, i.e., the object.

The following code is common.

```
class clsthis{
    int i;
    void add(int x) {i = i + x;};
  public:
        void init(int val) {i = val;};
        int get(){    add(50);
                      return i;};
}
```

Actually, within a member function of a class, "this" is an implicit pointer to the current object that receives the message relevant to this function. The above code can be rewritten as follows by explicitly showing the message receiver.

```
class clsthis{
    int i;
    void add(int x) { this->i = this->i +x;};
  public:
    void init(int val) { this->i = val;};
    int get(){    this ->add(50);
                  return this->i;};
}
```

Therefore, "this" is really a polymorphic key word that can be used to refer to any object in different context in C++.

4.3 Templates

A template is a mechanism to define a frame for different parameterized classes. This makes the strong-typed language more flexible. But instructors should teach it with clear concepts, because its syntax is similar to that of the class concept.

- Class templates are not classes, but parameterized classes from a class template can be used as ordinary classes;
- Class templates are not classes, but they can be defined by inheriting an ordinary class; and
- Class templates are not the same as the Smalltalk metaclasses.

4.4 Friends

Friend is a mechanism for C++ classes or functions to access directly the private members of a class. But it is really a killer to the encapsulation of objects and classes. Friends can destroy the information hiding mechanism defined by a private modifier. Thus, students are taught to "use friends when thought is very clear and they can control everything relevant."

Of course, if students want to understand the concept completely, instructors must illustrate this concept in the following clear statements.

- Friend has no transitivity. This means that if class A is a friend of class B and Class C is a friend of class A, class B is not granted a friend of class C. This reflects that even though you are my friend, but your friend is not definitely my friend and vise versa;
- Friend cannot be inherited. This means that if A (a class or a function) is a friend of class B and class C is a derived class from class B, A is not granted a friend of class C. This reflects the situation that the friends of your father are not definitely your friends; and
- Friend has no reflexivity. This means that if class A is a friend of class B, class B is not granted a friend of class A. This can be compared to the fact that you can take somebody as a friend of yours, but you should not take it for granted that he or she will take you as a friend. Only when he or she declares you a friend of him or her, can you be his or her friend.

We can suppose the reason to design the friend mechanism is to support the tight coupling among some classes. The above statements are really useful for students to understand the friend concept in C++. These properties may restrict the shortcomings of the friend mechanism and would not let it spread too broadly and too fast.

4.5 Operator Overloading

In C++, the overloaded operators can be used with the styles they are used to in an ordinary sense. This is helpful to experienced C programmers. It is actually another way to name a function. An example is given as follows. Suppose that, in a class Time, there is an overloaded operator "+=".

```
const Time & Time::operator +=( unsigned n )
{//...
}
void main()
{Time a, b, c;
 a+=b;                //(1)
 a.operator +=(b);       //(2)
}
```

Students are taught that two statements (1) and (2) actually have the same results and same meanings. Hence, it is easily understood that the overloaded operator is another method to name a function.

Operator overloading is indeed a powerful tool to use general operators such as "+","-","*", "/", etc. Yet it is merely another way to name a function conceptually.

The side effect of operator overloading is that it would make the programs less readable for beginners, because the same operators have likely different meanings for different classes. It is better for beginners to write a function with a more understandable name. This

method for naming should be in accordance with the messages relevant to the function's name.

4.6 Exception Handling

In C++, exception is defined as an error generated by the program at run-time that would normally terminate the execution of the program. Such exceptions include memory exhaustion, subscript range errors, or division by zero. Exception handling is a mechanism that allows a program to detect and possibly recover from errors during execution. This concept is not so easy for students to accept and understand.

When this mechanism is discussed, students are advised to regard any exception as an instance of a class. They can define an exception just like defining a class.

Let students conceive of this situation. There is a pool in a C++ run-time system. Any time when the program tries (TRY) some message and encounters a defined exception, it throws (THROW) the exception object to the pool. After that, the program searches the pool and tries to catch (CATCH) an exception with its class identification the same as what is declared in the parameter list of the CATCH statement. If it catches such kind of exception, it does some exception handling operations. If not, it just follows the normal procedure.

5. CONCLUSION

Object-oriented programming with C++ is very important in today's engineering education curriculum. C++ is a very good programming language with much flexibility. However, instructors must pay more attention to their teaching methods and let students master the principles of object-orientation first. The proposed six-step approach covers the three most important concepts of object-orientation, i.e., class, object, and inheritance [1]. Based on this, the C++ mechanisms are taught to help students master these mechanisms useful for coding. Only in this way will the students really obtain the whole picture about object-oriented programming with C++.

It would be a total failure if students who learnt object-oriented programming in C++ could make only programs in C++ in the C style or non-object-oriented style.

Finally the following suggestions are given to C++ programmers, software engineers, and instructors:

- Use as many protected members as possible. The protected members are really the members consistent with the concepts of objects and classes. This kind of variables can be accessed directly by the subclasses and cannot be accessed directly by the instances.
- Use as little multiple inheritance as possible. If simple inheritance is enough, do not use multiple inheritance.
- Use virtual inheritance when you apply multiple inheritance.
- Use as little overloaded operators as possible.
- Do not use "Friend", if it is not absolutely needed.
- Use a Static variable for the variables shared by all the instances of the class being defined.

We attach three projects for a course "object-oriented programming with C++" (See Appendix).

REFERENCES

[1] Arif, E. M., A Methodology for Teaching Object-Oriented Programming Concepts In an Advanced Programming Course, *SIGCSE Bulletin*, Vol 32. No. 2, June 2000, 30-34.

[2] Bagert, D. J. and Calloni, B.A., Teaching programming concepts using an icon-based software design tool", *IEEE Transactions on Education,* Vol 42, No 4, Nov. 1999, 365-378.

[3] Bresciani, P., et al, A knowledge level software engineering methodology for agent oriented programming", in *Proceedings of the fifth international conference on Autonomous Agents* May 2001, 648-655.

[4] Goldberg , A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Reading Mass. 1983.

[5] Horstmann, C. S., *Object-Oriented Development in C++ and Java*, John Weley & Sons, Inc. 1997.

[6] Kumar, A. N., Learning the interaction Between Pointers and Scope in C++, in *Proceedings of the 6th on Innovation and Teaching in Computer Science Education (ITiCSE)*, June 2001, 45-48.

[7] Lau, Y. T., *The Art of Object:Object-Oriented Design and Architecture*, Addison-Wesley, 2001

[8] Lewis, J. Myths about Object-Orientation and its Pedagogy, *ACM SIGCSE Bulletin*, Vol 32 No. 1, March 2000, 245-249.

[9] Luker, P. A., There's more to OOP than syntax! *ACM SIGCSE Bulletin*, Vol 26 No. 1, March 1994, 56-60.

[10] Luker, P. A. Never mind the language, what about the paradigm? *ACM SIGCSE Bulletin*, Vol 21 No 1, February 1989, 252-256 .

[11] McKim, J. C. Jr., Using a Multi-term Project to teach Object Oriented Programming and Design, TOOLS'98, 1998, 469-476.

[12] Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, 1988

[13] Mitchell, W., A Paradigm Shift to OOP Has Occurred… Implementation to Follow, *Journal of Computing in Small College*, Vol 16, No 2, May 2001, 95-106.

[14] Pattis, R. E., Teaching OOP in C++ Using an Artificial Life Framework, *SIGCSE Bulletin*, Vol 32. No. 2, June 1997, 39-43.

[15] Pullen, J. M. and Norris, E., Teaching C++ in a Multi-User Virtual Environment, *SIGCSE Bulletin*, Vol 32. No. 2, June 2000, 60-64.

[16] Raside, D. and Campbell, G. T. Aristotle and Object-Oriented Programming: Why Modern Students Need Traditional Logic, *ACM SIGCSE Bulletin*, Vol 32 No. 1, March 2000, 237-245.

[17] Shammas, N. C., *Moving from Turbo Pascal to Turbo C++*, SAMA Publishing, 1993.

[18] Sklyarov, V., *The revolutionary Guide to Turbo C++*, Wrox Press Ltd, 1993.

[19] Stroustrup, B., Why isn't C++ Just an Object-Oriented Programming Language, *OOPSLA'95, OOPS Messenge*r, Oct. 1995, 1-13.

[20] Taivalsaari, A., On the Notion of Inheritance", *ACM Computer Survey*, Vol 28 No 3, Sept. 1996, 438-479.

[21] Venkatesh , K. and Zhou, M. C. Object-oriented design of FMS Control Software based on Object Modeling Technique Diagrams and Petri nets, *J. of Manufacturing Systems*, Vol 17, No 2,1998, 118-136.

[22] Zhu, H. and Hu, S., The Chicken-and-Egg Problem in the Object-Oriented Technology, *Proceedings of 1994 Kunming International CASE Symposium(KICS'94)*, Kunming, China, Dec., 1994, 122-127.

[23] Zhu, H., Yang, G., and Liu, Z., *Object Oriented Principle and its Applications*, Publishing House of Changsha Institute of Technology, Sept. 1998.

APPENDIX

Project 1: A Class and its Application

This project is assigned to students when the concepts about classes are taught.

This project requires us to make a card guess game program. The requirements are the following.

Basic facts: There are a deck of playing cards. They are 2-10, J, Q, K, and A of spade, heart, diamond and club. 2-10, J, Q, K, and A of spade are larger than 2-10, J, Q, K, and A of heart that are larger than 2-10, J, Q, K, and A of diamond that are larger than 2-10, J, Q, K, and A of club. That is to say, if we compare two cards, the suit is at the first preference, and any card in suit Space is larger than any card in suit Heart. For example, 2 of Diamond is larger than K of Club, 3 of Spade is larger than Q of Heart, and so on. As a result, there are no equal cards in one set.

Basic functions of the program include:
1) Simulating the random pick of two cards;
2) Allowing users to guess which card is larger;
3) Displaying which one is larger (e.g., "The first one is larger.");
4) Displaying the suit and card names of the two cards ("Q of Heart" or "2 of Spade");
5) Displaying if the user's guess is correct;
6) Asking users if they want to quit the game;
7) If the input is "N" or "n", go to the first step; else, end the program.

Project 2: An EZ Booth Simulation

This project is assigned when the concepts of inheritance are taught.

This project is to make a program used in an EZ booth to deal with tolls for different kinds of vehicles (such as cars, buses and trucks). As a module, this program should simulate passing vehicles by random numbers for self-testing.

1) The input is a file named "vehicle.txt" containing registration of a group of vehicles. In the file, each line includes vehicle type, plate number, make, model, color, year, weight for a truck, largest passenger volume for a bus, and number of doors for a car as follows.
Car: JVR23K, TOYOTA, CAMERY, WHITE, 1996, 4
Truck: RTD45Q, MITSUBISHI, PETRO, RED, 1998, 50
Bus: UYT45M, XXXXX, YYYY, YELLOW, 1992, 40

…
2) The program loads the file contents into an array of vehicle object.
3) A user clicks a key to pretend that a vehicle is passing. The program shows it on the screen in a format as follows.
Car: JVR33K, TOYOTA, CAMERY, WHITE, 1993, 4
or
Bus: UYT65M, XXXXX, YYYY, YELLOW, 1998, 40
or
Truck: RTD46Q, MITSUBISHI, PETRO, RED, 1992, 50
4) A report is shown on the screen after 10 vehicles have passed.
5) A user enters a special letter to end the program.

Project 3: A Self-Test Program

This project is assigned when the concepts of polymorphism and overloading are taught.

This project is a self-test program used by a student for a course.
1) There are files containing questions and relevant answers named "yesno.txt", "mutiple.txt", and "number.txt" that include different type of questions in specific formats.
2) The program should read in all the questions and answers and form a question bank.
3) It should ask a user to enter a number N to express how many questions the test paper has.
4) It should create a test including N questions randomly selected from the question bank.
5) It should show the questions and gets the user answers.
6) It should tells if each answer is correct or not, show the correct answer for each.
7) It should count 20 points for each question.
8) It should create a report on the screen. For example,
You have taken totally 5(the number N of questions) questions.
You answered correctly 3 questions.
You total points are 60.out of 100 (20*N).
9) It should let the user select if he/she wants to take another test.
10) If yes, go to 3), if no, exit.