Chapter 1. Introduction to Objects

C++ Object Oriented Programming

Pei-yih Ting

NTOUCS

Contents

OOP vs. Procedural Programming

- ✧ Hardware improvements mark the revolution of computer technology in the past 40^+ years.
- ✧ Procedural programming:

  you program tends to control directly the underlying machine, which is mostly built with the von Neuman architecture

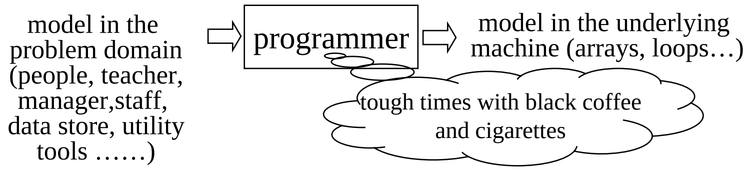
OOP vs. Procedural (cont’d)

- ✧ It is now changing gradually: tools are beginning to look less like machines and more like parts of our minds or daily lives --- much more accessible to general public (ex. Desktop metaphor, Document center…)
- ✧ Object-Oriented Programming (OOP) is part of this movement toward using the computer as an expressive medium.

Progress of Abstraction

- All (programming) languages provide abstractions.
  - abstraction: you transform your physical problem into a description in some other media
  - problem statement, machine language, assembly language, imperative languages (Fortran, BASIC, C, PASCAL, COBOL…), functional languages (LISP, ML, Scheme…), declarative languages (PROLOG, GPSS…), object-oriented languages (SIMULA, Smalltalk, Eiffel, C++, Java…)
- Assembly and Imperative Language:
  - ***abstraction*** requires you think in terms of the structure of the computer (solution space) rather than the structure of the problem (problem space)

Progress of Abstraction (cont'd)


model in the problem domain (people, teacher, manager,staff, data store, utility tools ……)
programmer
model in the underlying machine (arrays, loops…)
tough times with black coffee and cigarettes

- Good programmers do the mapping faster, with less errors, and produce better structured codes.
- Programmers need to know exactly the operating mechanisms of the underlying machines.
- Often:
  - the mapping mechanism are too complex to be documented well or even thrown away completely
  - result programs are expensive to maintain

Progress of Abstraction (cont'd)

- Functional, Declarative, and 4-th Gen. languages:
  - Each programming languages choose particular views of the world
    - "all problems are ultimately lists" in LISP
    - "all problems are algorithmic" in APL
    - "all problems are logical production rules" in PROLOG
    - "all problems are math equations" in Mathematica
    - "all problems are descriptive words" in WORD
    - "all problems can be manipulated with graphical symbols" in some visual languages
  - Each of these approaches is a good solution to the particular class of problem it is designed to solve.
  - When you step outside of that domain, it becomes awkward.

Progress of Abstraction (cont'd)

- Object-Oriented Programming (Analysis/Design)
  - Programmer represents elements in the problem space directly.
  - OOP minimizes transformations a programmer need to do on the original problem in order to let the computer solve it.
  - OOPL is not constrained to any particular type of problem.
  - Describe all elements in the problem space as "objects" (although a programmer still has some auxiliary abstract objects that have no correspondence in the physical problem domain).
  - When you read the codes describing the solution, you are reading words that also express the problem. You don't need those "transformation guidelines invented by any programmer". The code is not a bunch of CPU/Memory/IO operations.

Progress of Abstraction (cont'd)

- Objects …. The fundamental ingredients in OOP.
  - attributes/properties/states/data
  - ***behaviors*** : answering requests (messages)
- Five basic characteristics of Smalltalk -- a pure OOPL
  1. Everything is an object
  2. A program is a bunch of objects telling each other what to do by sending messages
  3. Each object has its own memory (probably made up of other objects)
  4. Every object has a type (instance/class) …. classification
  5. All objects of a particular type can receive the same messages also, an object of type "circle" is also an object of type "shape" ***substitutability*** is one of the most powerful concepts in OOP.

Algorithmic vs. OO Decomposition

- In summary:
  - Procedural programming: algorithmic decomposition or functional decomposition of the problem, the software is viewed as a process
  - Object Oriented programming: decompose the problem into a set of well-defined objects, functional decomposition is addressed after the system has been decomposed into objects (i.e. on top of objects)
    - decomposition is more intuitive
    - encourage the reuse of objects
    - emphasize the encapsulation at each level

Overview of OOP features

- An object has an interface
- The hidden implementation (Encapsulation)
- Reusing the implementation: hierarchy
- Reusing the interface: inheritance
- Is-a vs. is-like-a relationships
- Interchangeable objects with polymorphism
- Creating and destroying objects
- Exception handling: dealing with errors efficiently

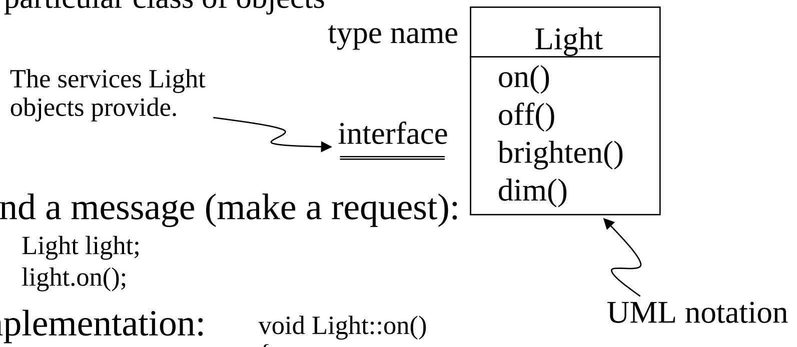
Class and Object

- **class**: objects that have *common* characteristics and behaviors … types, abstract data types
  - user defined, works almost exactly like built-in data types
  - ex.
    - Bank account: balance/deposit/withdraw...
    - Alarm clock
    - TV set
    - ATM

    $\left\{\begin{array}{l}\text{characteristics/attributes/properties/data}\\ \text{behaviors/functionality}\end{array}\right.$
- **objects/variables/instances**
  - creation: using the class as a template
  - manipulation: sending messages or requests
- a programmer defines a class to fit a problem rather than being forced to use existing data types

Object has Interface

- Interface:
  - the interface establishes requests that you can make for a particular class of objects



- Send a message (make a request):

```
Light light;
light.on();
```

- Implementation:

```
void Light::on()
{
        // do something
}
```

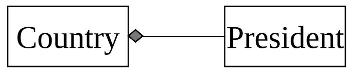
Encapsulation

- <u>class creator</u>: those who create/maintain new data types
  <u>client programmer</u>: the class/object consumers who use the new data types in their applications
- Implementation details are intended to be hidden from all client programmers except the class creator.
  - Integrity of the internal state of an object can be maintained.
  - Class creator can change the hidden portion at will without breaking the *contract* – the interface.
  - Reduce program bugs: client programmer tends to bypass the official contract promised by the interface and make very good use of every aspect of an existing code implementation.
- Enforced through access control: **public/protected/private**

Reusing the Implementation

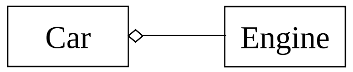
- Code reuse is one of the greatest advantages that OOPL provide.
- A class can have multiple instances.
- Build up hierarchies: avoid reinventing wheels
  - composition

```
class Country
{
 private:
     President m_president;
};
```



  - aggregation

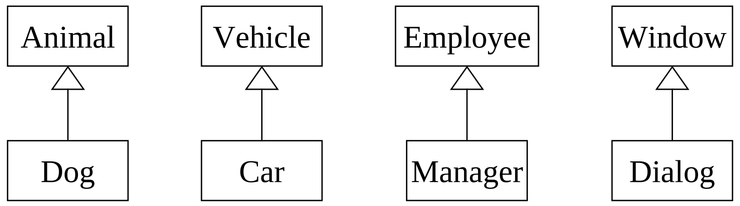
```
class Car
{
 private:
     Engine *m_engine;
};
```



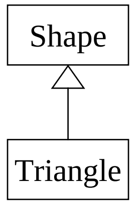
Note: hierarchical structure in this way is flexible,
can be dynamically changed at run time

Inheritance

- Inheritance: take an existing class, clone it, and then make modifications or additions to the clone.
- Base class / super class / parent class vs.
  Derived class / sub class / child class / inherited class
- Three things are inherited by the derived class : interfaces, implementations (data and functionality), and relationships



Reusing the Implementation



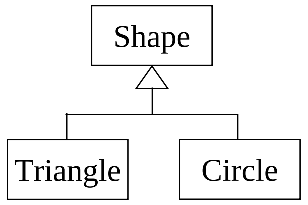
✶ inheritance

```
class Shape            class Triangle : public Shape
{                      {
    ….                     ….
};                     };
```

Note: 1. codes and data can be reused
2. static reuse, determined at compile time

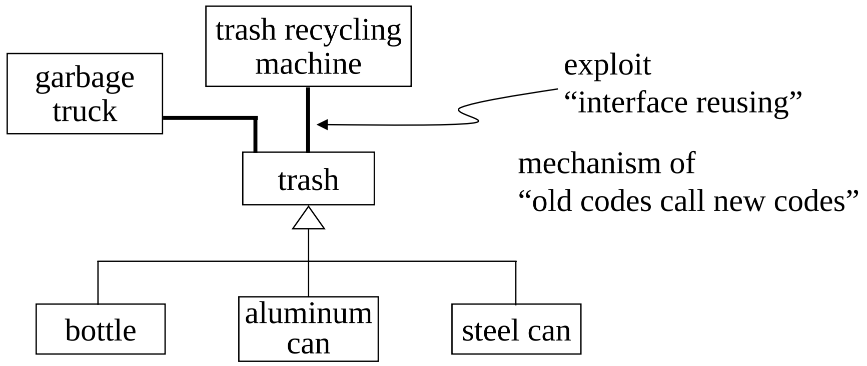
CAVEAT: Because inheritance is important in object-oriented programming, it is often over-emphasized. New OOP programmers can get the incorrect idea that inheritance should be used everywhere. This is not true. Always consider composition first when creating classes, since it is simpler and more flexible.

Reusing the Interface



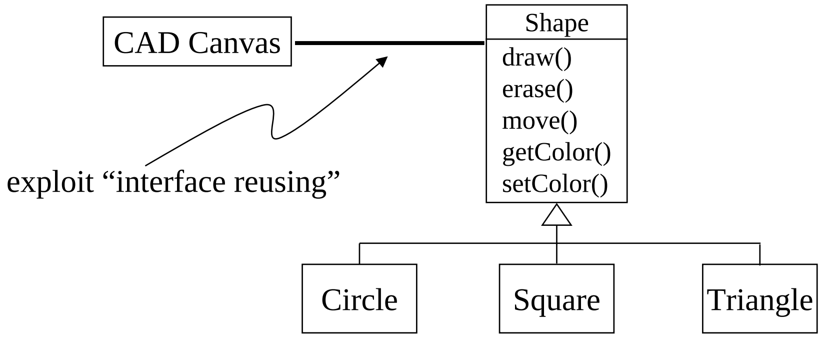
- ✧ You use inheritance if you want your objects of the Triangle class be handled by client programs exactly the same as if they were objects of the Shape class.
  (From the point of view of client programmers, these two types of objects are indistinguishable. They are completely substitutable in the sense that they all provide the same *Shape* interface.)
- ✧ Why do people like to ignore the differences between a triangle object and a circle object and call them Shape objects? **in order to simplify the handling mechanism**

Reusing the Interface (cont'd)



- ✧ Trash recycling machine can process all three classes (bottle, aluminum can, and steel can) in one unified way.
- ✧ Does this model capture the essence of real world model?

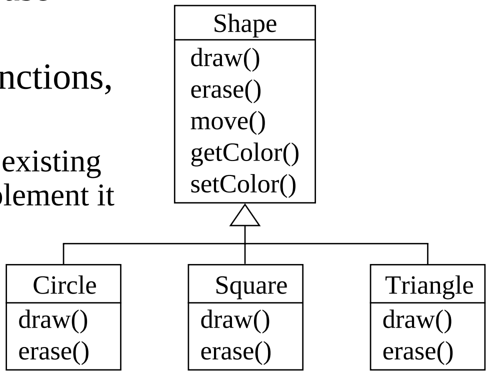
Reusing the interface



- ✧ All the interfaces (messages) are inherited by child classes. All the code implementations (message handling mechanisms) are also inherited by default if you do not modify them.

Is-a Relationship

- ✧ Reuse the interface of the base class through inheritance
- ✧ only ***override*** base-class functions, not adding new ones
  - ✶ changing the behavior of an existing base-class function by reimplement it in the derived class
- ✧ A derived-class object "***is a***" base-class object.
  - ✶ It can be used through out the client program as a replacement for a base-class object.
  - ✶ the "substitution principle"



Is-like-a Relationships

- ✧ Can you add brand new functionality to a derived class?

Yes. You add them when you discover them. This process of discovery and iteration of your design happen regularly in the design of your OOP process. However, these added functions will not be exploited by all existing client codes. This sort of design is not the main purpose for the existence of inheritance.

- A triangle "is like a" shape.

Is-like-a Relationships (cont'd)

- ✧ A heat pump "is like a" cooling system in the aspect that it provides the cooling function to a thermostat.

Polymorphism

- ✧ reuse the interface in a class hierarchy
- ✧ treat an object NOT as the specific type that it is but instead as its **base type** (the abstract properties).
- ✧ Client programmers can write code that doesn't depend on specific types (everybody wants easier life)
  - ✶ ex. In the shape example, from the point of view of the canvas, circles, squares, and triangles can be drawn, erased, and moved by just sending a message to a shape object
  - ✶ ex. When you write with a pen, most of the time you don't need to know the brand of that pen before you can take a memo.
  - ✶ New types can therefore be added to the class hierarchy without modifying the client programs –

old codes (client) call new codes (server)

Polymorphism (cont'd)

Polymorphism in C++

- Dynamic binding (late binding) through function pointers (called virtual function table)
  - When you send a message to an object, the code being called is not determined until runtime.
  - The compiler does ensure that the function exists and performs type checking on the arguments and the return value.
- Virtual function

```
class Shape
{
   ...
public:
   virtual void erase();
   virtual void draw();
   …
};
```

- Polymorphic pointer/reference

```
void doStuff(Shape& s)          Circle c;
{                               Triangle t;
   s.erase();                   doStuff(c);
   …                            doStuff(t);
   s.draw();
}
```

talk to the object according to the interface defined by its base class (message sending)

Polymorphism in C++ (cont'd)

- **Upcasting**
  - treating a derived type object as though it were its base type object
- **Downcasting**
  - cast in the reverse direction through dynamic_cast<>()
  - usually appears with the object selection codes
  - dangerous, breaking the checking enforced by the type system
  - often signals some defects in the design of the class hierarchy

Creating and Destroying Objects

- C/C++:
  - control of efficiency is the most important issue

    speed $\xleftrightarrow{\text{tradeoff}}$ memory
  - give the programmer choices of full control on an object's life cycle
    - global data segment
    - register
    - stack … automatic variable — compiler does the control (fast initialization, but little flexibility)
    - heap — programmer takes full control (longer initialization period, but greater flexibility)
      new/delete   new []/delete []

  Note: Java instead treat platform independence and ease of programming the most important goal… It handles memory automatically with a garbage collector module.

Exception Handling

- A very difficult issue, many programming languages simply ignore the issue.
- A major problem with most schemes: rely on programmer vigilance in following an agreed-upon convention that is not enforced by the language
  - ex. C *return value* and *errno*
- C++: exception handling (implicitly in the language)
  - *exceptions* are '*thrown*' from the site of the error and can be '*caught*' by an appropriate '*exception handler*'
  - different, parallel path of execution, automatically handle resource releasing codes (unlike notorious exit() in stdlib)
  - does not interfere with the normal flow of execution control
  - exception can not be ignored

Analysis and Design Methodologies

- Method (methodology): a formal set of processes and heuristics to analyze, design and build a software system
  - most OOAD methodologies are Spiral/Iterative, RAD
    - in the process, you might build codes in one stage and modify them or throw away them in another stage
  - the goal of an OOAD methodology is to discover
    - What are the objects? (How do you partition your system?)
    - What are their interfaces? (What messages are sent and handled?)
- Phase 0: Make a plan, set up the 'mission statement'
  - ex. In an air-traffic control system: you are building "The tower program keeps track of the aircraft"
- Phase 1: What are we making?
  - Requirements analysis and system specification
  - Use cases

Use Cases

- Use cases
  - identify key features in the system that will reveal some of the fundamental classes.
  - answers to questions like
    - Who will use this system?
    - What can those actors do with the system?
    - How does this actor do that with this system?
    - How else might this work if someone else were doing this, or if the same actor had a different objective? (to reveal variations)
    - What problems might happen while doing this with the system? (to reveal exceptions)

Use Cases (cont'd)

- ex. auto-teller

  what the auto-teller does in every possible situations (scenarios)

  a use case --- a collection of scenarios (flow of events)

  a sample scenario:

  "What does the auto-teller do if a customer has just deposited a check within 24 hours and there's not enough in the account to provide the desired withdrawal"
- A use case does not need to be terribly complex, even if the underlying system is complex. It is only intended to show the system as it appears to the user.
- You don't need to find the complete and perfect set of use cases for you system at the very start of the analysis stage. Many things will reveal themselves in time.

Analysis and Design (cont'd)

- Phase 2: How will we build it? (design of objects)
  - Class-Responsibility-Collaboration (CRC card):
    - name of the class
    - responsibilities of the class: what it should do, what it should response
    - collaborations of the classes: what other classes does it interact with?

    Have a group of people ready. Each person takes responsibility for several classes. Then you can run a live simulation by solving one scenario at a time, deciding which messages are sent to the various objects to satisfy each scenario
  - Five stages of object design:
    - Object discovery
    - Object assembly
    - System construction
    - System extension
    - Object reuse
  - Guidelines for object development

Analysis and Design (cont'd)

- Phase 3: Build the core
- Phase 4: Iterate the use cases
  - add a feature set during one iteration, the basis for one iteration is a single use case
- Phase 5: Evolution/ maintenance
  - tasks:
    - make all features work according the original requirements
    - fixing bugs
    - adding features that the customer forgot to mention
    - adding new features as the need arises
  - make your program go from good to great
  - OOPL are particularly adept at supporting this kind of continuing modifications (Interface/Object Structure and Boundary/Encapsulation/Class hierarchy)

Extreme Programming (XP) Features

- Kent Beck, "Extreme programming explained: embrace change"
- XP is both a <u>philosophy about programming</u> and <u>guidelines</u> to do it.
- <u>Write tests first</u>:basis for refactoring (test driven)

  traditionally low priority task, just for sure that everything works
  - 'test codes' have equal or even greater priority than the 'normal codes' in XP, write the test before you start coding the function
  - CPPUnit, JUnit for the unit test and functional test, running the tests every time you do a build (you have some modifications to your codes). Your test codes will always catch any problem that you have already tested and reintroduced in this modification.
- <u>Pair programming</u>
  - one coding, the other thinking and verifying (not resting)
  - one gets stuck, the other just takes over

Why C++ Succeeds

- A better C: stricter type system, namespace, reference...
- You're already on the learning curve: based on C
- Efficiency: same low level controls as in C
- Systems are easier to express and understand
- Maximal leverage with libraries: classes, encapsulation
- Source-code reuse with templates: generic programming
- Error handling: exception handling
- Programming in the large: Object Oriented Analysis and Design + namespaces + encapsulation/interfaces (reduce code coupling)

Definitions by Horowitz

- Object: an object is an entity that performs computations and has a local state.
- Object-oriented programming: is a method of implementation in which
  - Objects are the fundamental building blocks
  - Each object is an instance of some type (or class)
  - Classes are related to each other by inheritance and other relationships
- Object-oriented language:
  - It supports objects.
  - It require objects to belong a class.
  - It supports inheritance.

Some Terms

- Data Encapsulation or Information Hiding: is the concealing of the implementation details of a data object from the outside world.
- Data Abstraction: is the separation between the specification of a data object and its implementation.
- Data type: is a collection of objects and a set of operations that act on those objects.
- Abstract data type (ADT): is a data type that is organized in such a way that the *specification* of the objects and the specification of the operations on the objects is separated from the *representation* of the objects and the *implementation* of the operations

Paradoxial Description

- Rob Pike –

  "*OO languages conceptually provide little extra over judicious use of function pointers in C*"
- From the implementation level, this is true indeed. But this should not discourage you from using OO languages. After all, all languages are sequences of machine instructions.
- From the conceptual design level, this description is certainly incorrect because it neglects the modeling capability provided by an OOPL.