C++ Programming Language

Object-Oriented Programming (OOP) in C++

**1.  Why OOP?**

Suppose that you want to assemble your own PC, you go to a hardware store and pick up a motherboard, a processor, some RAMs, a hard disk, a casing, a power supply, and put them together.  You turn on the power, and the PC runs.  You need not worry whether the motherboard is a 4-layer or 6-layer board, whether the hard disk has 4 or 6 plates; 3 inches or 5 inches in diameter, whether the RAM is made in Japan or Korea, and so on. You simply put the hardware *components* together and expect the machine to run.  Of course, you have to make sure that you have the correct *interfaces*, i.e., you pick an IDE hard disk rather than a SCSI hard disk, if your motherboard supports only IDE; you have to select RAMs with the correct speed rating, and so on.  Nevertheless, it is not difficult to set up a machine from hardware *components*.

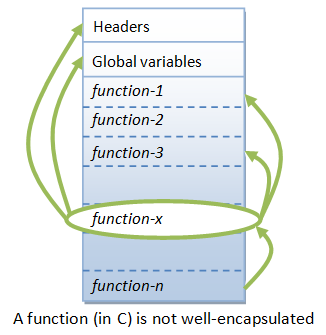
Similarly, a car is assembled from parts and components, such as chassis, doors, engine, wheels, brake, and transmission. The components are reusable, e.g., a wheel can be used in many cars (of the same specifications).

Hardware, such as computers and cars, are assembled from parts, which are reusable components.

How about software?  Can you "assemble" a software application by picking a routine here, a routine there, and expect the program to run?  The answer is obviously no!  Unlike hardware, it is very difficult to "assemble" an application from *software components*.  Since the advent of computer 60 years ago, we have written tons and tons of programs.  However, for each new application, we have to re-invent the wheels and write the program from scratch.

Why re-invent the wheels?

**1.1  Traditional Procedural-Oriented languages**



Can we do this in traditional procedural-oriented programming language such as C, Fortran, Cobol, or Pascal?

Traditional procedural-oriented languages (such as C and Pascal) suffer some notable drawbacks in creating reusable software components:

1. The programs are made up of functions. Functions are often not *reusable*. It is very difficult to copy a function from one program and reuse in another program because the the function is likely to reference the headers, global variables and other functions. In other words, functions are not well-encapsulated as a self-contained *reusable unit*.
2. The procedural languages are not suitable of *high-level abstraction* for solving real life problems. For example, C programs uses constructs such as if-else, for-loop, array, function, pointer, which are low-level and hard to abstract real problems such as a Customer Relationship Management (CRM) system or a computer soccer game. (Imagine using assembly codes, which is a very low level code, to write a computer soccer game. C is better but no much better.)

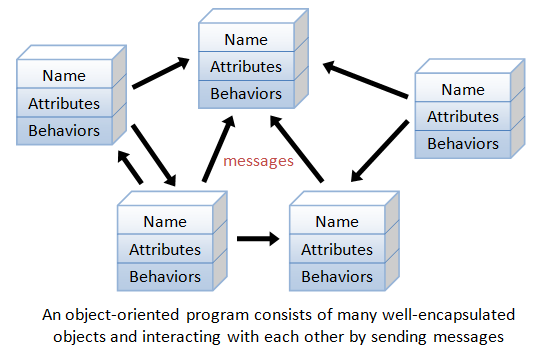
In brief, the traditional procedural-languages *separate* the data structures and algorithms of the software entities.

In the early 1970s, the US Department of Defense (DoD) commissioned a task force to investigate why its IT budget always went out of control; but without much to show for. The findings are:

1. 80% of the budget went to the software (while the remaining 20% to the hardware).
2. More than 80% of the software budget went to maintenance (only the remaining 20% for new software development).
3. Hardware components could be applied to various products, and their integrity normally did not affect other products. (Hardware can share and reuse! Hardware faults are isolated!)
4. Software procedures were often non-sharable and not reusable. Software faults could affect other programs running in computers.

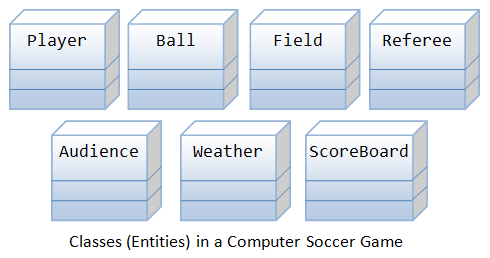
The task force proposed to make software behave like hardware OBJECT. Subsequently, DoD replaces over 450 computer languages, which were then used to build DoD systems, with an object-oriented language called Ada.

**1.2  Object-Oriented Programming Languages**



Object-oriented programming (OOP) languages are designed to overcome these problems.

1. The basic unit of OOP is a *class*, which encapsulates both the *static attributes* and *dynamic behaviors* within a "box", and specifies the public interface for using these boxes. Since the class is well-encapsulated (compared with the function), it is easier to reuse these classes. In other words, OOP combines the data structures and algorithms of a software entity inside the same box.
2. OOP languages permit *higher level of abstraction* for solving real-life problems. The traditional procedural language (such as C and Pascal) forces you to think in terms of the structure of the computer (e.g. memory bits and bytes, array, decision, loop) rather than thinking in terms of the problem you are trying to solve. The OOP languages (such as Java, C++, C#) let you think in the problem space, and use software objects to represent and abstract entities of the problem space to solve the problem.



As an example, suppose you wish to write a computer soccer games (which I consider as a complex application). It is quite difficult to model the game in procedural-oriented languages. But using OOP languages, you can easily model the program accordingly to the "real things" appear in the soccer games.

* Player: attributes include name, number, location in the field, and etc; operations include run, jump, kick-the-ball, and etc.
* Ball:
* Reference:
* Field:
* Audience:
* Weather:

Most importantly, some of these classes (such as Ball and Audience) can be reused in another application, e.g., computer basketball game, with little or no modification.

**1.3  Benefits of OOP**

The procedural-oriented languages focus on procedures, with function as the basic unit. You need to first figure out all the functions and then think about how to represent data.

The object-oriented languages focus on components that the user perceives, with objects as the basic unit. You figure out all the objects by putting all the data and operations that describe the user's interaction with the data.

Object-Oriented technology has many benefits:

* *Ease in software design* as you could think in the problem space rather than the machine's bits and bytes. You are dealing with high-level concepts and abstractions. Ease in design leads to more productive software development.
* *Ease in software maintenance*: object-oriented software are easier to understand, therefore easier to test, debug, and maintain.
* *Reusable software*: you don't need to keep re-inventing the wheels and re-write the same functions for different situations. The fastest and safest way of developing a new application is to reuse existing codes - fully tested and proven codes.

**2.  OOP Basics**

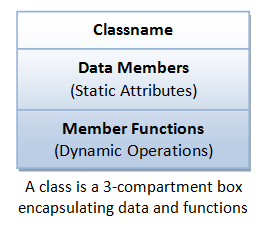
**2.1  Classes & Instances**

**Class**: A*class is a definition of objects of the same kind*. In other words, a *class* is a blueprint, template, or prototype that defines and describes the *static attributes* and *dynamic behaviors* common to all objects of the same kind.

**Instance**: An *instance* is *a realization of a particular item of a class*. In other words, an instance is an *instantiation* of a class. All the instances of a class have similar properties, as described in the class definition. For example, you can define a class called "Student" and create three instances of the class "Student" for "Peter", "Paul" and "Pauline".

The term "*object*" usually refers to *instance*. But it is often used quite loosely, which may refer to a class or an instance.

**2.2  A Class is a 3-Compartment Box encapsulating Data and Functions**



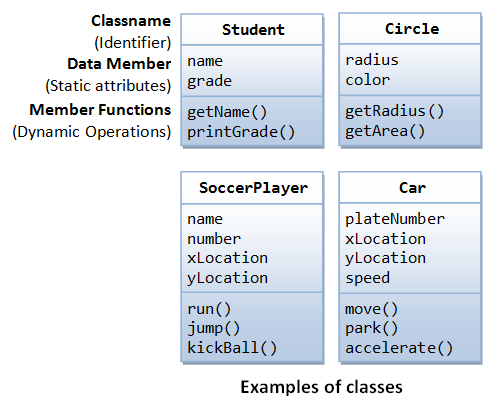
A class can be visualized as a three-compartment box, as illustrated:

1. ***Classname*** (or identifier): identifies the class.
2. ***Data Members*** or ***Variables*** (or *attributes*, *states*, *fields*): contains the *static attributes* of the class.
3. ***Member Functions*** (or *methods*, *behaviors*, *operations*): contains the *dynamic operations* of the class.

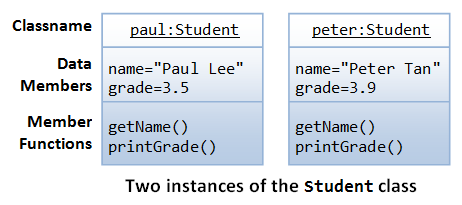
In other words, a class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.

**Class Members**: The *data members* and *member functions* are collectively called *class members*.

The followings figure shows a few examples of classes:



The following figure shows two instances of the class Student, identified as "paul" and "peter".



**Unified Modeling Language (UML) Class and Instance Diagrams:** The above class diagrams are drawn according to the UML notations. A class is represented as a 3-compartment box, containing name, data members (variables), and member functions, respectively. classname is shown in bold and centralized. An instance (object) is also represented as a 3-compartment box, with instance name shown as instanceName:Classname and underlined.

**Brief Summary**

1. A *class* is a programmer-defined, abstract, self-contained, reusable software entity that mimics a real-world thing.
2. A class is a 3-compartment box containing the name, data members (variables) and the member functions.
3. A class encapsulates the data structures (in data members) and algorithms (member functions). The values of the data members constitute its *state*. The member functions constitute its *behaviors*.
4. An *instance* is an instantiation (or realization) of a particular item of a class.

**2.3  Class Definition**

In C++, we use the keyword class to define a class. There are two sections in the class declaration: private and public, which will be explained later. For examples,

**class** **Circle** { // classname

private:

double radius; // Data members (variables)

string color;

public:

double getRadius(); // Member functions

double getArea();

}

**class** **SoccerPlayer** { // classname

private:

int number; // Data members (variables)

string name;

int x, y;

public:

void run(); // Member functions

void kickBall();

}

**Class Naming Convention:** A classname shall be a noun or a noun phrase made up of several words. All the words shall be initial-capitalized (camel-case). Use a *singular* noun for classname. Choose a meaningful and self-descriptive classname. For examples, SoccerPlayer, HttpProxyServer, FileInputStream, PrintStream and SocketFactory.

**2.4  Creating Instances of a Class**

To create *an instance of a class*, you have to:

1. Declare an instance identifier (name) of a particular class.
2. Invoke a constructor to construct the instance (i.e., allocate storage for the instance and initialize the variables).

For examples, suppose that we have a class called Circle, we can create instances of Circle as follows:

// Construct 3 instances of the class Circle: c1, c2, and c3

Circle c1(1.2, "red"); // radius, color

Circle c2(3.4); // radius, default color

Circle c3; // default radius and color

Alternatively, you can invoke the constructor explicitly using the following syntax:

Circle c1 = Circle(1.2, "red"); // radius, color

Circle c2 = Circle(3.4); // radius, default color

Circle c3 = Circle(); // default radius and color

**2.5  Dot (.) Operator**

To reference a *member of a object* (data member or member function), you must:

1. First identify the instance you are interested in, and then
2. Use the *dot operator* (.) to reference the member, in the form of *instanceName.memberName*.

For example, suppose that we have a class called Circle, with two data members (radius and color) and two functions (getRadius() and getArea()). We have created three instances of the class Circle, namely, c1, c2 and c3. To invoke the function getArea(), you must first identity the instance of interest, says c2, then use the *dot operator*, in the form of c2.getArea(), to invoke the getArea() function of instance c2.

For example,

// Declare and construct instances c1 and c2 of the class Circle

Circle c1(1.2, "blue");

Circle c2(3.4, "green");

// Invoke member function via dot operator

cout << c1.getArea() << endl;

cout << c2.getArea() << endl;

// Reference data members via dot operator

c1.radius = 5.5;

c2.radius = 6.6;

Calling getArea() without identifying the instance is meaningless, as the radius is unknown (there could be many instances of Circle - each maintaining its own radius).

In general, suppose there is a class called *AClass* with a data member called *aData* and a member function called *aFunction*(). An instance called *anInstance* is constructed for *AClass*. You use *anInstance.aData* and *anInstance.aFunction*().

**2.6  Data Members (Variables)**

A *data member* (*variable*) has a *name* (or *identifier*) and a *type*; and holds a *value* of that particular type (as descried in the earlier chapter). A data member can also be an instance of a certain class (to be discussed later).

**Data Member Naming Convention:** A data member name shall be a noun or a noun phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case), e.g., fontSize, roomNumber, xMax, yMin and xTopLeft. Take note that variable name begins with an lowercase, while classname begins with an uppercase.

**2.7  Member Functions**

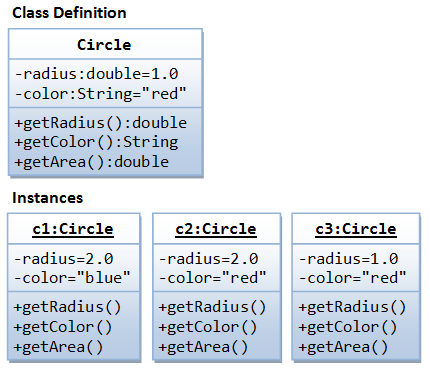
A member function (as described in the earlier chapter):

1. receives parameters from the caller,
2. performs the operations defined in the function body, and
3. returns a piece of result (or void) to the caller.

**Member Function Naming Convention:** A function name shall be a verb, or a verb phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case). For example, getRadius(), getParameterValues().

Take note that data member name is a noun (denoting a static attribute), while function name is a verb (denoting an action). They have the same naming convention. Nevertheless, you can easily distinguish them from the context. Functions take arguments in parentheses (possibly zero argument with empty parentheses), but variables do not. In this writing, functions are denoted with a pair of parentheses, e.g., println(), getArea() for clarity.

**2.8  Putting them Together: An OOP Example**



A class called Circle is to be defined as illustrated in the class diagram. It contains two data members: radius (of type double) and color (of type String); and three member functions: getRadius(), getColor(), and getArea().

Three instances of Circles called c1, c2, and c3 shall then be constructed with their respective data members, as shown in the instance diagrams.

In this example, we shall keep all the codes in a single source file called CircleAIO.cpp.

**CircleAIO.cpp**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | /\* The Circle class (All source codes in one file) (CircleAIO.cpp) \*/  #include <iostream> // using IO functions  #include <string> // using string  using namespace std;    class Circle {  private:  double radius; // Data member (Variable)  string color; // Data member (Variable)    public:  // Constructor with default values for data members  Circle(double r = 1.0, string c = "red") {  radius = r;  color = c;  }    double getRadius() { // Member function (Getter)  return radius;  }    string getColor() { // Member function (Getter)  return color;  }    double getArea() { // Member function  return radius\*radius\*3.1416;  }  }; // need to end the class declaration with a semi-colon    // Test driver function  int main() {  // Construct a Circle instance  Circle c1(1.2, "blue");  cout << "Radius=" << c1.getRadius() << " Area=" << c1.getArea()  << " Color=" << c1.getColor() << endl;    // Construct another Circle instance  Circle c2(3.4); // default color  cout << "Radius=" << c2.getRadius() << " Area=" << c2.getArea()  << " Color=" << c2.getColor() << endl;    // Construct a Circle instance using default no-arg constructor  Circle c3; // default radius and color  cout << "Radius=" << c3.getRadius() << " Area=" << c3.getArea()  << " Color=" << c3.getColor() << endl;  return 0;  } |

To compile and run the program (with GNU GCC under Windows):

> **g++ -o CircleAIO.exe CircleAIO.cpp**

// -o specifies the output file name

> **CircleAIO**

Radius=1.2 Area=4.5239 Color=blue

Radius=3.4 Area=36.3169 Color=red

Radius=1 Area=3.1416 Color=red

**2.9  Constructors**

A *constructor* is a special function that has the *function name same as the classname*. In the above Circle class, we define a constructor as follows:

// Constructor has the same name as the class

Circle(double r = 1.0, string c = "red") {

radius = r;

color = c;

}

A constructor is used to construct and *initialize all the data members*. To create a new instance of a class, you need to declare the name of the instance and invoke the constructor. For example,

Circle c1(1.2, "blue");

Circle c2(3.4); // default color

Circle c3; // default radius and color

// Take note that there is no empty bracket ()

A constructor function is different from an ordinary function in the following aspects:

* The name of the constructor is the same as the classname.
* Constructor has no return type (or implicitly returns void). Hence, no return statement is allowed inside the constructor's body.
* Constructor can only be invoked *once* to initialize the instance constructed. You cannot call the constructor afterwards in your program.
* Constructors are not inherited (to be explained later).

**2.10  Default Arguments for Functions**

In C++, you can specify the default value for the trailing arguments of a function (including constructor) in the function header. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | /\* Test function default arguments (TestFnDefault.cpp) \*/  #include <iostream>  using namespace std;    // Function prototype  int sum(int n1, int n2, int n3 = 0, int n4 = 0, int n5 = 0);    int main() {  cout << sum(1, 1, 1, 1, 1) << endl; // 5  cout << sum(1, 1, 1, 1) << endl; // 4  cout << sum(1, 1, 1) << endl; // 3  cout << sum(1, 1) << endl; // 2  // cout << sum(1) << endl; // error: too few arguments  }    // Function definition  // The default values shall be specified in function prototype,  // not the function implementation  int sum(int n1, int n2, int n3, int n4, int n5) {  return n1 + n2 + n3 + n4 + n5;  } |

**2.11  "public" vs. "private" Access Control Modifiers**

An *access control modifier* can be used to control the visibility of a data member or a member function within a class. We begin with the following two access control modifiers:

1. **public**: The member (data or function) is accessible and available to *all* in the system.
2. **private**: The member (data or function) is accessible and available *within this class only*.

For example, in the above Circle definition, the data member radius is declared private. As the result, radius is accessible inside the Circle class, but NOT outside the class. In other words, you cannot use "c1.radius" to refer to c1's radius in main(). Try inserting the statement "cout << c1.radius;" in main() and observe the error message:

CircleAIO.cpp:8:11: error: 'double Circle::radius' is private

Try moving radius to the public section, and re-run the statement.

On the other hand, the function getRadius() is declared public in the Circle class. Hence, it can be invoked in the main().

**UML Notation:** In UML notation, public members are denoted with a "+", while private members with a "-" in the class diagram.

#### 2.12  Information Hiding and Encapsulation

A class encapsulates the static attributes and the dynamic behaviors into a "3-compartment box". Once a class is defined, you can seal up the "box" and put the "box" on the shelve for others to use and reuse. Anyone can pick up the "box" and use it in their application. This cannot be done in the traditional procedural-oriented language like C, as the static attributes (or variables) are scattered over the entire program and header files. You cannot "cut" out a portion of C program, plug into another program and expect the program to run without extensive changes.

Data member of a class are typically hidden from the outside word, with private access control modifier. Access to the private data members are provided via public assessor functions, e.g., getRadius() and getColor().

This follows the principle of information hiding. That is, objects communicate with each others using well-defined interfaces (public functions). Objects are not allowed to know the implementation details of others. The implementation details are hidden or encapsulated within the class. Information hiding facilitates reuse of the class.

**Rule of Thumb:** Do not make any data member public, unless you have a good reason.

#### 2.13  Getters and Setters

To allow other to read the value of a private data member says xxx, you shall provide a get function (or getter or accessor function) called getXxx(). A getter need not expose the data in raw format. It can process the data and limit the view of the data others will see. Getters shall not modify the data member.

To allow other classes to modify the value of a private data member says xxx, you shall provide a set function (or setter or mutator function) called setXxx(). A setter could provide data validation (such as range checking), and transform the raw data into the internal representation.

For example, in our Circle class, the data members radius and color are declared private. That is to say, they are only available within the Circle class and not visible outside the Circle class - including main(). You cannot access the private data members radius and color from the main() directly - via says c1.radius or c1.color. The Circle class provides two public accessor functions, namely, getRadius() and getColor(). These functions are declared public. The main() can invoke these public accessor functions to retrieve the radius and color of a Circle object, via says c1.getRadius() and c1.getColor().

There is no way you can change the radius or color of a Circle object, after it is constructed in main(). You cannot issue statements such as c1.radius = 5.0 to change the radius of instance c1, as radius is declared as private in the Circle class and is not visible to other includingmain().

If the designer of the Circle class permits the change the radius and color after a Circle object is constructed, he has to provide the appropriate setter, e.g.,

// Setter for color

void setColor(string c) {

color = c;

}

// Setter for radius

void setRadius(double r) {

radius = r;

}

With proper implementation of information hiding, the designer of a class has full control of what the user of the class can and cannot do.