OOP Basics

Classes & Instances

Class: Definition of objects of the same kind.
A class is a template, or prototype that defines and describes

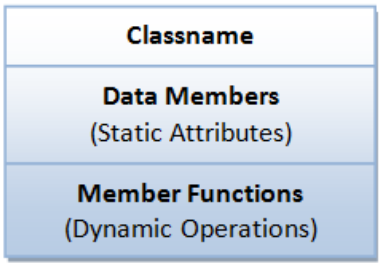
- the static attributes
- dynamic behaviors

common to all objects of the same kind.

Instantce: An instance is a realization (instantiation) of a **particular item** of a class. All the instances of a class have similar properties, as described in the class definition.

Example

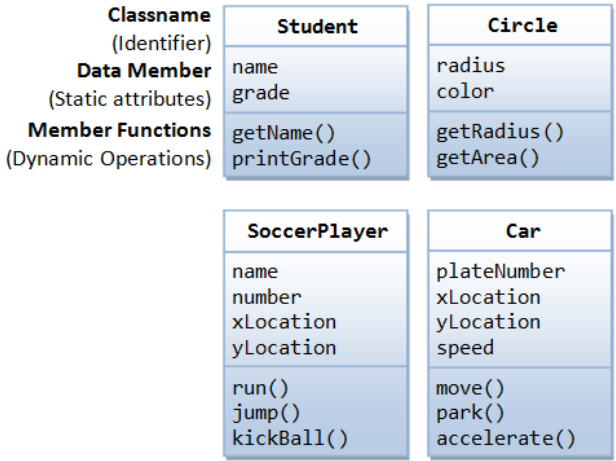
You can define a class called "Student" and create three instances of the class "Student" for "Peter", "Paul" and "Pauline".



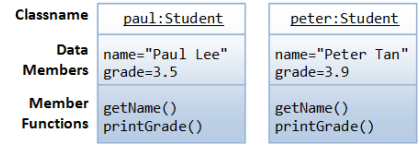
A class is a 3-compartment box encapsulating data and functions

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

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



Examples of classes



Two instances of the **Student** class

Class Naming Convention:

- Should be a noun or a noun phrase made up of several words.
- All words should be initial capitalized (camel-case)
- Singular noun
- Meaningful and Self-descripted

Example

SoccerPlayer, HttpProxyServer, FileInputStream, PrintStream, SocketFactory.

Class Definition

```
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();
};
```

Creating instances

```
// 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:

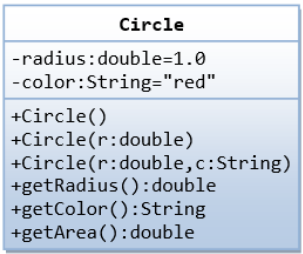
```
Circle c1 = Circle(1.2, "red");  // radius, color
Circle c2 = Circle(3.4);         // radius, default color
Circle c3 = Circle();            // default radius and color
```

Accessing member of instances wit dot operator (.):

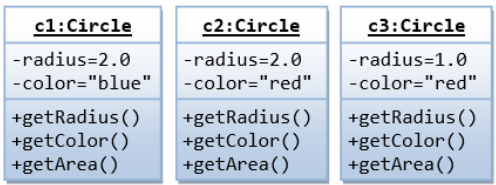
```
// Declare and construct instances c1 and c2 of the class Circle
Circle c1(1.2, "blue");
Circle c2(3.4, "green");
// Invoke member function via dot operator
cout << c1.getArea() << endl;
cout << c2.getArea() << endl;
// Reference data members via dot operator
c1.radius = 5.5;
c2.radius = 6.6;
```

OOP Example

Class Definition



Instances



`Circle` class contains two data members: `radius` (of type double) and `color` (of type String);

three member functions: `getRadius()`, `getColor()`, and `getArea()`.

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

→ `CircleAIO.cpp`

To run the cpp program:

```
g++ -o CircleAIO.exe CircleAIO.cpp
// -o specifies the output file name
```

Constructors

Constructor - Special function that has the *function name same as the classname*. A constructor is used to construct and *initialize all the data members*. For the `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;
}
```

To create a new instance of a class, you need to declare the name of the instance and invoke the constructor:

```
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 ()
```

N

- Constructor name is same as the class name.
- No return type. Hence no return statement is allowed.
- Can only be invoked once to initialize the instance.
- Constructors are not inherited.

Default arguments for functions

```
/* 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;
}
```

An access control modifier can be used to control the visibility of a data member or a member function within a class.

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.

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

Data members are hidden from outside world with `private` access control modifier. Access to the private data members are provide via `public` functions, e.g., `getRadius()` and `getColor()`

Information hiding

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.

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

Getters & Setters

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.

Example

```
// Setter for color
void setColor(string c) {
   color = c;
}

// Setter for radius
void setRadius(double r) {
   radius = r;
}
```

this

You can use keyword `this` to refer to **this instance** inside a class definition.

One of the main usage of keyword `this` is to resolve ambiguity between the names of data member and function parameter.

Example

```
class Circle {
private:
   double radius;                  // Member variable called "radius"
   ......
public:
   void setRadius(double radius) { // Function's argument also called "
       radius"
      this->radius = radius;
        // "this.radius" refers to this instance's member variable
        // "radius" resolved to the function's argument.
   }
   ......
}
```

Default constructor

A default constructor is a constructor with no parameters, or having default values for all the parameters. For example, the above `Circle`'s constructor can be served as default constructor with all the parameters default.

```
1 Circle c1;   // Declare c1 as an instance of Circle, and invoke the default
      constructor
2 Circle c1(); // Error!
3              // (This declares c1 as a function that takes no parameter
4              // and returns a Circle instance)
```

N

> if you did not provide ANY constructor, the compiler automatically provides a default constructor that does nothing

```
1 ClassName::ClassName(){} // Take no argument and do nothing
```

N

> Compiler will not provide a default constructor if you define any constructor(s).

If all the constructors you defined require arguments, invoking no-argument default constructor results in error. This is to allow class designer to make it impossible to create an uninitialized instance, by NOT providing an explicit default constructor.

Constructor's member initializer list

```
1 Circle(double r = 1.0, string c = "red") : radius(r), color(c) { }
```

Member initializer list:

- placed after the constructor's header
- separated by a colon (`:`)
- each initializer is in the form of `data_member_name(parameter_name)`

For fundamental type, it is equivalent to `data_member_name = parameter_name` .
For object, the constructor will be invoked to construct the object. The constructor's body (empty in this case) will be run after the completion of member initializer list.

N

> It is recommended to use member initializer list to initialize all the data members, as it is often more efficient than doing assignment inside the constructor's body.

Destructor

A *destructor*, similar to constructor, is a special function that has the same name as the classname, with a prefix `~`, e.g., `Circle()`. Destructor is called implicitly when an object is destroyed. If you do not define a destructor, the compiler provides a default, which does nothing.

```
class MyClass {
public:
   // The default destructor that does nothing
   ~MyClass() { }
......
}
```

N

> If your class contains data member which is dynamically allocated (via `new` or `new[]` operator), you need to free the storage via `delete` or `delete[]`.

Copy Constructor

A *copy constructor* constructs a new object by copying an existing object of the same type.

If you do not define a copy constructor, the compiler provides a default which copies all the data members of the given object.

```
Circle c4(7.8, "blue");
cout << "Radius=" << c4.getRadius() << " Area=" << c4.getArea()
     << " Color=" << c4.getColor() << endl;
                // Radius=7.8 Area=191.135 Color=blue

// Construct a new object by copying an existing object
// via the so-called default copy constructor
Circle c5(c4); //copy constructor takes an argument, which is an object of the
    same class.
cout << "Radius=" << c5.getRadius() << " Area=" << c5.getArea()
     << " Color=" << c5.getColor() << endl;
                // Radius=7.8 Area=191.135 Color=blue
```

When an object is *passed by value*, the copy constructor will make a clone of the argument.

Advanced Notes

- Pass-by-value for object means calling the copy constructor. To avoid the overhead of creating a clone copy, it is usually better to *pass-by-reference-to-const*, which will not have side effect on modifying the caller's object.
- The copy constructor has the following signature:

```
class MyClass {
private:
   T1 member1;
   T2 member2;
public:
   // The default copy constructor which constructs an object via memberwise
       copy
   MyClass(const MyClass & rhs) {
      member1 = rhs.member1;
      member2 = rhs.member2;
   }
......
}
```

- The default copy constructor performs *shadow copy*. It does not copy the dynamically allocated data members created via `new` or `new[]` operator.

Separating Header and Implementation

- **Header file (.h):** defines the public interface of the class
  - class declaration
  - public / private members
  - function prototypes
- **Implementation file (.cpp):** defines how the functions actually work
  - function bodies
  - private helper functions

A single class is commonly split into 3 files:

- `ClassName.h`
  Public interface (class declaration)
- `ClassName.cpp`
  Implementation of member functions
- `TestClassName.cpp`
  Test driver / demo program

Example (Circle):

- `Circle.h` - interface of `Circle`
- `Circle.cpp` - implementation of `Circle`
- `TestCircle.cpp` - test driver for `Circle`

Header guards (`#ifndef` / `#define`)

- Problem: the same header might be included multiple times
  - can lead to duplicate declaration errors
- Solution: wrap the header file with a preprocessor guard:

Example

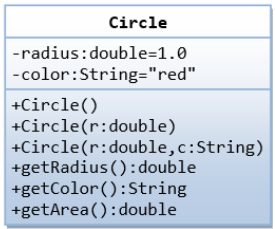
```
#ifndef TIME_H
#define TIME_H

// class Time { ... };

#endif  // TIME_H
```

- First inclusion: `TIME_H` is not defined -> body is included and `TIME_H` is defined.
- Later inclusions: `TIME_H` is already defined -> body is skipped.

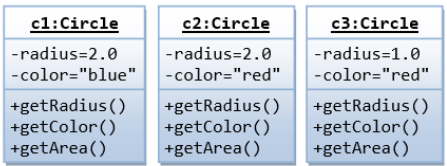
OOP Examples

Circle Class Interface

Class Definition



Instances



`Circle.h` declares:

- Private data members:
  - `double radius;`
  - `string color;`
- Public member functions:
  - constructor with default parameters
  - getters / setters for `radius`, `color`
  - `getArea()` as a `const` function

Compiling the modular Circle example

Files:

- `Circle.h`
- `Circle.cpp`
- `TestCircle.cpp`

Compile and link (one-shot):

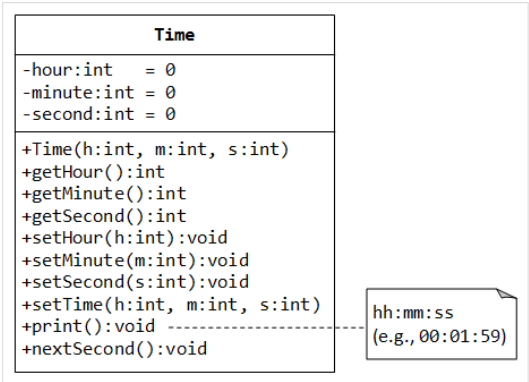
```
g++ -o TestCircle.exe TestCircle.cpp Circle.cpp
```

Or using object files:

```
g++ -c Circle.cpp          // Circle.o
g++ -c TestCircle.cpp      // TestCircle.o
g++ -o TestCircle.exe TestCircle.o Circle.o
```

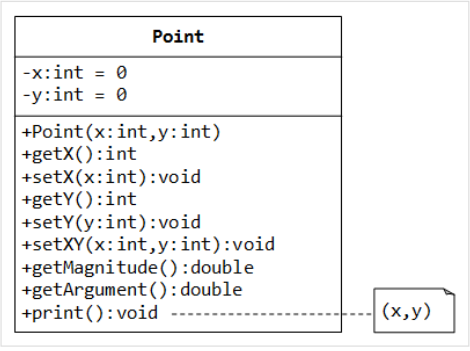
All implementation details are hidden behind the interface `Circle.h`.

Time Class



- Models a time of day in `hh:mm:ss` format
- Typical private data members:
  - `int hour;` (0-23)
  - `int minute;` (0-59)
  - `int second;` (0-59)
- Public operations (in `Time.h`):
  - constructor to initialize hour, minute, second
  - getters / setters for each field
  - `setTime(h, m, s)`
  - `print()` – prints `"hh:mm:ss"`
  - `nextSecond()` – advances time by 1 second

Point Class – Modeling 2D points

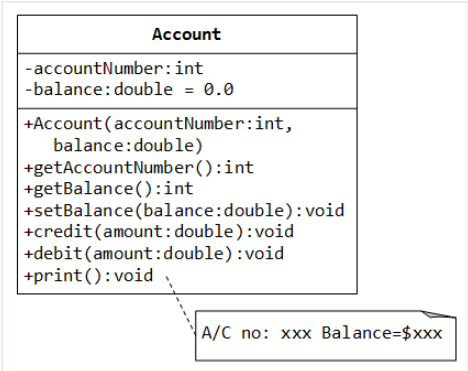


The `Point` class models 2D points with integer coordinates.

- Private data:
  - `int x`, `int y` (default 0)
- Behavior:
  - constructor, getters and setters
  - `setXY(x, y)`
  - `getMagnitude()` – returns $\sqrt{x^2 + y^2}$
  - `getArgument()` – returns $\tan^{-1}(y/x)$ via `atan2`
  - `print()` – prints `"(x,y)"`

Big code: **`Point.h`**, **`Point.cpp`**, **`TestPoint.cpp`**.

Account Class – Bank account model

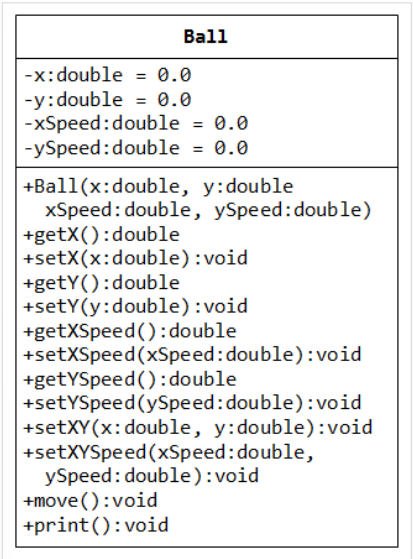


The `Account` class models a simple bank account.

- Private data:
  - `int accountNumber;`
  - `double balance;`
- Public operations:
  - constructor with `accountNumber` and optional `balance`
  - `credit(amount)` – adds `amount` to `balance`
  - `debit(amount)` – subtracts `amount`, or prints `"Amount withdrawn exceeds the current balance!"`
  - `print()` – prints `"A/C no: xxx Balance=$xx.xx"`

Big code: **`Account.h`**, **`Account.cpp`**, **`TestAccount.cpp`**.

Ball Class – Moving object

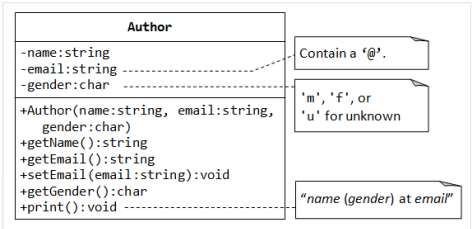


A `Ball` class models a moving ball in 2D.

- Private data:
  - `double x, y;` – position
  - `double xSpeed, ySpeed;` – velocity
- Public operations:
  - constructor with default position and speed
  - getters / setters for position and speed
  - `setXY(x, y)`, `setXYSpeed(xs, ys)`
  - `move()` – updates `x` and `y` by current speeds
  - `print()` – prints `"Ball @ (x,y) with speed (xSpeed,ySpeed)"`

Big code: **`Ball.h`**, **`Ball.cpp`**, **`TestBall.cpp`**

Author Class – Person who writes books

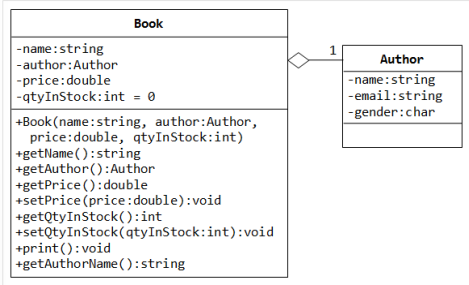


`Author` represents a single author.

- Private data:
  - `string name;`
  - `string email;`
  - `char gender;` (`'m'`, `'f'`, or `'u'`)
- Behavior:
  - constructor with `name`, `email`, `gender`
  - getters for all fields
  - setter for `email` only
  - `print()` – prints `"name (gender) at email"`

Big code: **`Author.h`**, **`Author.cpp`**, **`TestAuthor.cpp`**.

Book Class – Has-a Author



Book models a book written by exactly one author.

- Private data:
  - `string name;`
  - `Author author;` (object data member)
  - `double price;` (must be > 0)
  - `int qtyInStock;` (default 0, must be ≥ 0)
- Public operations:
  - constructor taking an `Author` object
  - getters / setters (including `getAuthor()`)
  - `print()` –
    `"'book-name' by author-name (gender) @ email"`
  - `getAuthorName()` – convenience getter

Big code: **`Book.h`**, **`Book.cpp`**, **`TestBook.cpp`**.

Aggregation: `Book` has-an `Author`

- In the UML diagram, the *hollow diamond* near `Book` means **aggregation** (a *has-a* relationship).
- In code:
  - `Book` has a data member `Author author;`
  - `Book` includes `"Author.h"` in `Book.h`
- Conceptually:
  - A `Book` object is composed of an `Author` object.
  - We can delegate author-related queries to the embedded `Author`.

Example

This demonstrates how classes can be built from other classes,
not only from primitive types.

Recap of the OOP examples

- **Circle, Time, Point, Account, Ball, Author, Book**
  - All follow the same pattern:
  - clear private data + public interface
  - constructors, getters/setters, utility methods
- **Header / implementation separation**
  - `ClassName.h` – interface
  - `ClassName.cpp` – implementation
  - `TestClassName.cpp` – test driver
- **Design lessons:**
  - encapsulate data, expose behavior
  - keep interfaces small and clear
  - build bigger abstractions (e.g., `Book`) out of smaller ones (e.g., `Author`)

To be Continued