Chapter 12 | Inheritance

CS185

Class Inheritance

Object-Oriented Programming

Inheritance is one of the three pillars of Object Oriented Programming:

1. Encapsulation (data abstraction/hiding; the **class**)
2. Inheritance (Is-a relationship, extending a **class**)
3. Polymorphism (dynamic binding, **virtual** methods)

You've already seen encapsulation (classes). Now we will look at *extending* a class via inheritance.

- Non OOP typically uses top-down design or structured design decomposing the problem into modules.
- These programs are collections of interacting functions.
- Before we used classes, we programmed top-down.
- Top-down doesn't scale up well for large programs.
- It is generally difficult to reuse code from one program to the next since the functions work directly on the data.
- Object-oriented languages must provide 3 facilities:
    - data abstraction
    - inheritance
    - polymorphism (dynamic binding)
- Programs written in an OO language are collections of interacting objects.
- In C++, classes provide data abstraction; a class is essentially an **A**bstract **D**ata **T**ype (ADT).
- Client programs don't work directly on the data in an object, they "ask" the object to manipulate its own data via public member functions (methods).
- OO refers to this "asking" as "sending a message" to the object.

Other OO languages use different terminology than C++. Here are some equivalents:

OOP	C++
Object	Class object or instance
Instance variable	Private data member
Method	Public member function
Message passing	Calling a public member function

Within a class, all of the data and functions are related. Within a program, classes can be related in various ways.

1. Two classes are independent of each other and have nothing in common
2. Two classes are related by *inheritance*
3. Two classes are related by *composition*, also called *aggregation* or *containment*

Inheritance

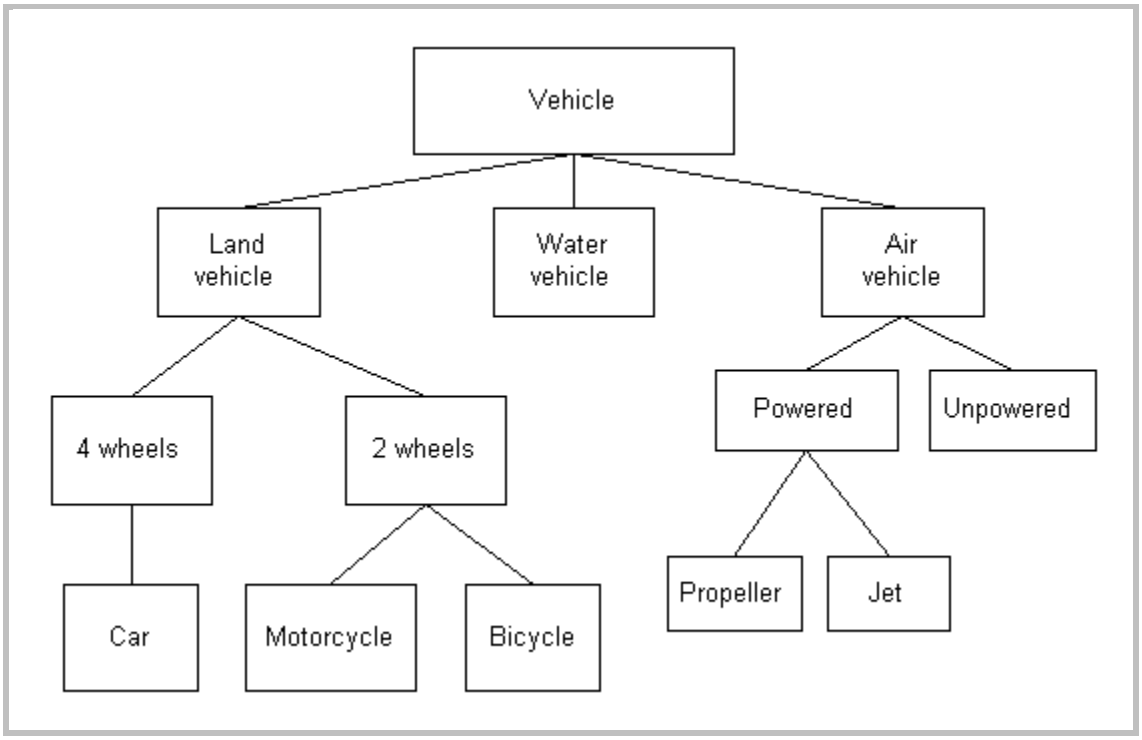
- Relation is an *is-a* relationship. (Also *is-a-kind-of* relationship)
- A car *is a* vehicle, A dog *is an* animal. (A car *is a kind of* vehicle, A dog *is a kind of* animal.)
- Generally, not reversible. All cars are vehicles but not all vehicles are cars.

Composition

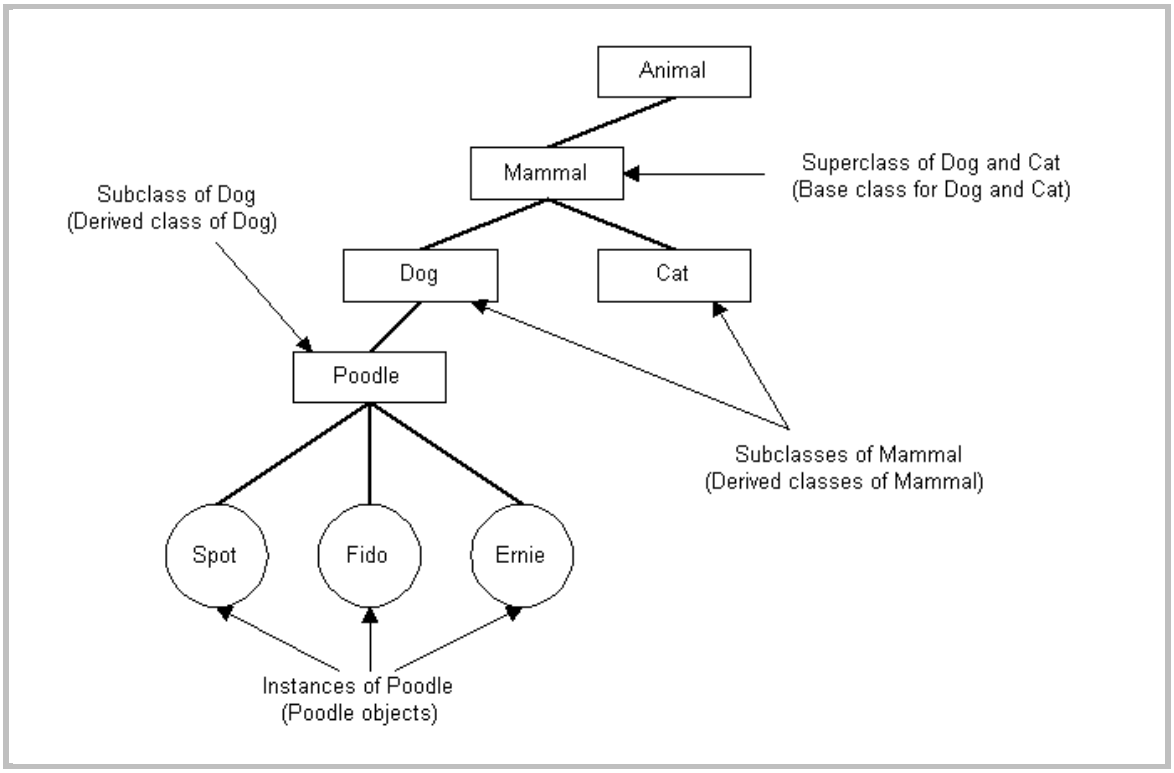
- Relation is a *has-a* relationship.
- Also called aggregation or containment.
- One class is composed of another (maybe several)
- A car *has a* motor (and a steering wheel, and 4 tires, etc.)

An inheritance relationship can be represented by a hierarchy.

A partial vehicle hierarchy:



A partial animal hierarchy:



A Simple Example

Structures to represent 2D and 3D points:

```
struct Point2D
{
    double x;
    double y;
};
```

```
struct Point3D
{
    double x;
    double y;
    double z;
};
```

Another way to define the 3D struct so that we can reuse the `Point2D` struct:

```
struct Point3D_composite
{
    // Struct contains a Point2D object
    Point2D xy;
    double z;
};
```

The memory layout of `Point3D` and `Point3D_composite` is identical and is obviously compatible with C, as there is nothing "C++" about them yet.

Accessing the members:

```
void PrintXY(const Point2D &pt_)
{
    std::cout << pt_.x << ", " << pt_.y;
}

void PrintXYZ(const Point3D &pt_)
{
    std::cout << pt_.x << ", " << pt_.y << ", " << pt_.z;
}

void PrintXYZ(const Point3D_composite &pt_)
{
    std::cout << pt_.xy.x << ", " << pt_.xy.y;
    std::cout << ", " << pt_.z;
}
```

Of course, the last function can be modified to reuse `PrintXY`:

```
void PrintXYZ(const Point3D_composite &pt_)
{
    PrintXY(pt_.xy); // delegate for X,Y
    std::cout << ", " << pt_.z;
}
```

Another way to do define the 3D point is to use *inheritance*.

```
// Struct inherits a Point2D object
struct Point3D_inherit : public Point2D
{
    double z;
};
```

This new struct has the exact same physical structure as the previous two 3D point structs:

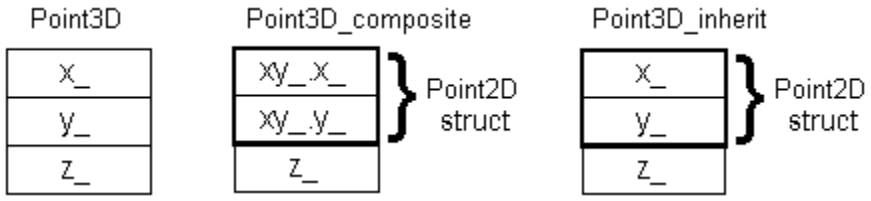
```
struct Point3D
{
    double x;
    double y;
    double z;
};
```

```
struct Point3D_composite
{
    // Struct contains a Point2D object
    Point2D xy;
    double z;
};
```

Another overloaded print function:

```
void PrintXYZ(const Point3D_inherit &pt)
{
    std::cout << pt.x << ", " << pt.y << ", " << pt.z;
}
```

Visually:



Sample Usage:

Code

```
int main(void)
{
	Point3D pt3;
	Point3D_composite ptc;
	Point3D_inherit pti;

	// Assign to Point3D members
	pt3.x = 1;
	pt3.y = 2;
	pt3.z = 3;
	PrintXYZ(pt3);
	std::cout << std::endl;

	// Assign to Point3D_composite members
	ptc.xy.x = 4;
	ptc.xy.y = 5;
	ptc.z = 6;
	PrintXYZ(ptc);
	std::cout << std::endl;

	// Assign to Point3D_inherit members
	pti.x = 7;
	pti.y = 8;
	pti.z = 9;
	PrintXYZ(pti);
	std::cout << std::endl;

	return 0;
}
```

Output

```
1, 2, 3
4, 5, 6
7, 8, 9
```

Notes about this syntax:

```
struct Point3D_inherit : public Point2D
```

- `Point2D` is the *base class* for `Point3D_inherit`.
- `Point3D_inherit` is the *derived class*.
- The public keyword specifies that the public methods of the base class remain public in the derived class. This is known as *public inheritance* and it is the most common.
- There is also **private** inheritance, but it is used much less. Unfortunately, this is the default if you do not specify it.
  - Technically, the default is whatever the base's default access is. (private for class, public for struct).
  - Tip: Always specify private or public when inheriting so everyone knows what you're intentions are.

Adding methods to the structs to make it more like C++:

```
struct Point2D
{
    double x;
    double y;

    void print(void)
    {
        std::cout << x << ", " << y;
    }
};
```

```
struct Point3D
{
    double x;
    double y;
    double z;

    void print(void)
    {
        std::cout << x << ", " << y << ", " << z;
    }
};
```

Composite vs. inheritance (everything is **public**):

```
struct Point3D_composite
{
	Point2D xy;
	double z;

	void print(void)
	{
		std::cout << xy.x << ", " << xy.y;
		std::cout << ", " << z;
	}
};
```

```
struct Point3D_inherit : public Point2D
{
	double z;

	void print(void)
	{
		// 2D members are public
		std::cout << x << ", " << y;
		std::cout << ", " << z;
	}
};
```

And in `main` we would have something that looks like this:

```
int main(void)
{
	Point3D pt3;
	Point3D_composite ptc;
	Point3D_inherit pti;

	// setup points

	pt3.print();
	ptc.print();
	pti.print(); // Is this legal? Ambiguous? Which method is called?

	return 0;
}
```

Let's make it more C++-like with **private** members and **public** methods and we'll use the **class** keyword instead of **struct**:

```
// This class is a stand-alone 2D point
class Point2D
{
public:
	Point2D(double x_, double y_) : x(x_), y(y_)
	{
	}

	void print(void)
	{
		std::cout << x << ", " << y;
	}
private:
	double x;
	double y;
};

// This class is a stand-alone 3D point
class Point3D
{
public:
	Point3D(double x_, double y_, double z_) : x(x_), y(y_), z(z_)
	{
	}

	void print(void)
	{
		std::cout << x << ", " << y << ", " << z;
	}
private:
	double x;
	double y;
	double z;
};
```

With composition, we must initialize the contained `Point2D` object in the initializer list:

```
// This class contains a Point2D object
struct Point3D_composite
{
public:
	Point3D_composite(double x_, double y_, double z_) : xy(x_, y_), z(z_)
	{
	}

	void print(void)
	{
		xy.print(); // 2D members are private
		std::cout << ", " << z;
	}
private:
	Point2D xy;
	double z;
};
```

With inheritance, we must initialize the `Point2D` *subobject* in the initializer list:

```
// This class inherits a Point2D object
struct Point3D_inherit : public Point2D
{
public:
	Point3D_inherit(double x_, double y_, double z_) : Point2D(x_, y_), z(z_)
	{
	}

	void print(void)
	{
		Point2D::print(); // 2D members are private
		std::cout << ", " << z;
	}
private:
	double z;
};
```

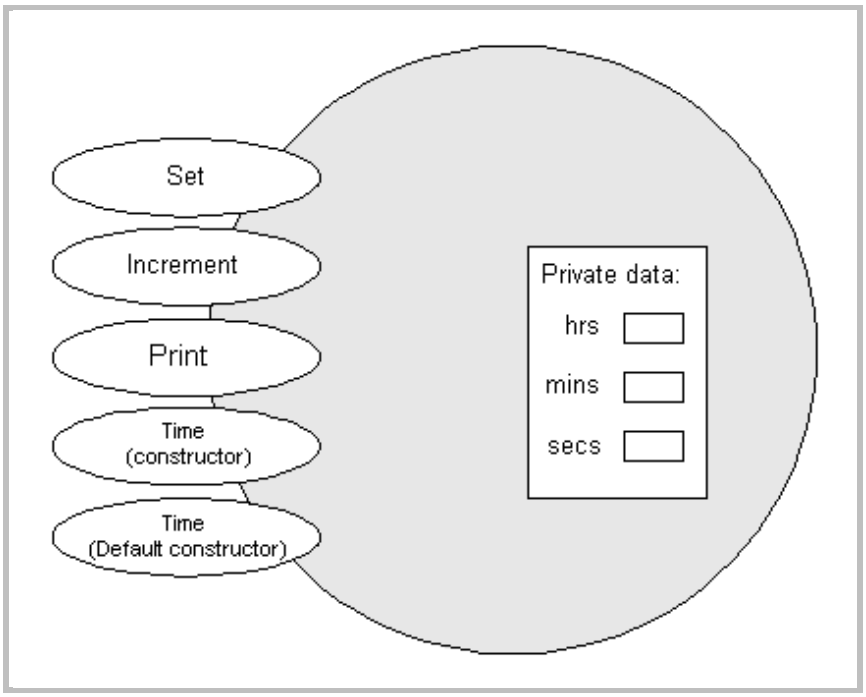
Sample Code

```
// Create Point3D
Point3D pt3(1, 2, 3);
pt3.print();
std::cout << std::endl;

// Create Point3D_composite
Point3D_composite ptc(4, 5, 6);
ptc.print();
std::cout << std::endl;

// Create Point3D_inherit
Point3D_inherit pti(7, 8, 9);
pti.print();
std::cout << std::endl;
```

A Larger Example
The Base Class

```
class Time
{
public:
	Time(int hrs_, int mins_, int secs_);
	Time(void);
	void Set(int hrs_, int mins_, int secs_);
	void Print(void) const;
	void Increment(void);

private:
	int hrs;
	int mins;
	int secs;
};
```

A diagram of the Time class:



Note that **`sizeof`**`(Time)` is 12 bytes.

Partial implementation from *Time.cpp*: (Notice the code reuse even in this simple example.)

```
Time::Time()
{
	Set(0, 0, 0);
}

Time::Time(int hrs_, int mins_, int secs_)
{
	Set(hrs_, mins_, secs_);
}

void Time::Set(int hrs_, int mins_, int secs_)
{
	hrs = hrs_;
	mins = mins_;
	secs = secs_;
}
```

Extending The *Time* Class

Now we decide that we'd like the Time class to include a Time Zone:

```
enum TimeZone {EST, CST, MST, PST, EDT, CDT, MDT, PDT};
```

We have several choices at this point:

1. Modify the Time class to include a TimeZone.
2. Create a new class by copying and pasting the code for the existing Time class and adding the TimeZone.
3. Create a new class by *inheriting* from the Time class.

What are the pros and cons of each of the choices above?

1. Easy to do. Affects (breaks) existing code, which may be what we want (bug fix).
2. Easy, can't affect old code (and vice versa). Bugs will need to be fixed in both places.
3. Easy (if you know what to do), maximum code reuse, straight-forward for simple classes.

Deriving *ExtTime* from *Time*:

```
enum TimeZone { EST, CST, MST, PST, EDT, CDT, MDT, PDT };

class ExtTime : public Time
{
public:
	ExtTime(void);
	ExtTime(int hrs_, int mins_, int secs_, TimeZone zone_);
	void Set(int hrs_, int mins_, int secs_, TimeZone zone_);
	void Print(void) const;

private:
	TimeZone zone;
};
```

What is `sizeof(ExtTime)`? How might it be laid out in memory?

Some implementations of the *ExtTime* constructors:

1. The derived class default constructor: (the base class default constructor is **implicitly** called)

```
ExtTime::ExtTime(void)
{
	zone = EST; // arbitrary default
}
```

2. The derived class non-default constructor: (the base class default constructor is **implicitly** called)

```
ExtTime::ExtTime(int hrs_, int mins_, int secs_, TimeZone zone_)
{
	zone = zone_;
	// what do we do with h, m, and s?
}
```

3. Calling a non-default base class constructor explicitly:

```
ExtTime::ExtTime(int hrs_, int mins_, int secs_, TimeZone zone_) : Time(hrs_,
mins_, secs_)
{
	zone = zone_;
}
```

4. Same as above using initializer list for derived member initialization:

```
ExtTime::ExtTime(int hrs_, int mins_, int secs_, TimeZone zone_) : Time(hrs_,
mins_, secs_), zone(zone_)
{
}
```

Notes:

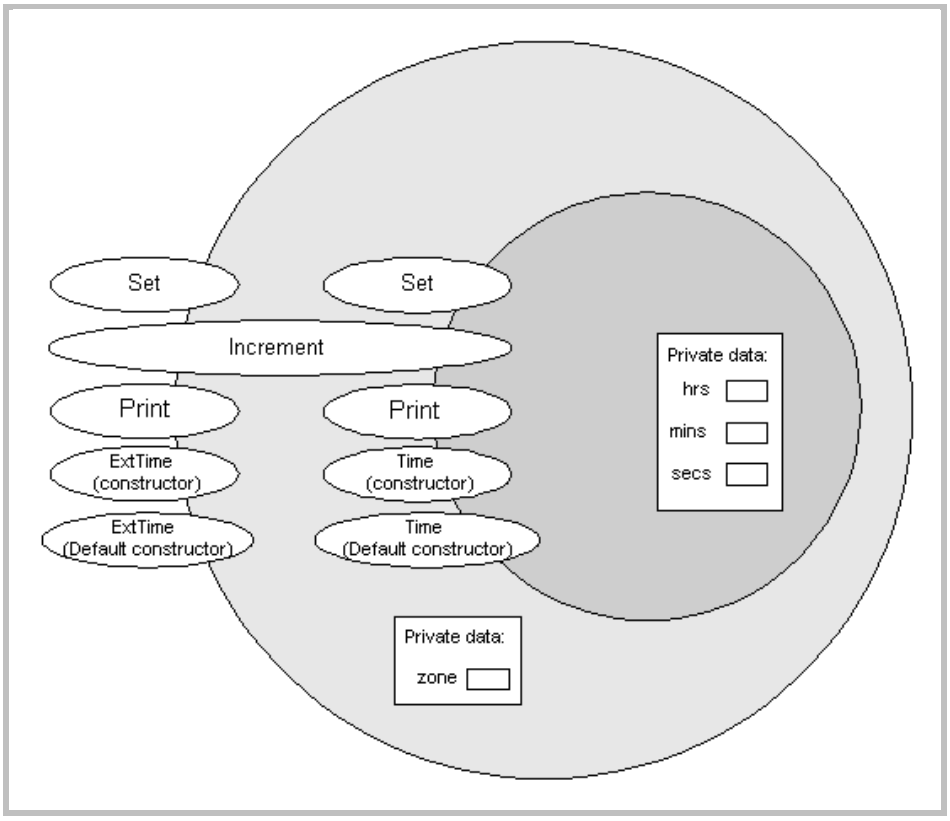
- The derived constructor calls the default base constructor if you don't call it explicitly.
- You can call any base constructor explicitly.
- A base constructor must be called from a derived constructor using the initializer list syntax. This is incorrect:

```
ExtTime::ExtTime(int hrs_, int mins_, int secs_, TimeZone zone_)
{
    // Can't call a base constructor explicitly.
    Time(hrs_, mins_, secs_); // (What is this statement actually doing?)
    zone_ = zone_;
}
```

Key Points:

- **A base constructor *must* be called, either implicitly or explicitly.**
- **If the base class has no default constructor, you *must* call another one explicitly. (If you don't, the compiler will generate an error.)**

The relationship between the Time and ExtTime classes:



In the *ExtTime* class:

- We *override* the Set and Print methods of the base class. (Override, not overload!)
- We *inherit* the Increment method of the base class.
- It's easy to see the relationship of the base class with its derived class in the diagram above.
- Because an ExtTime object is a Time object, an ExtTime object is valid anywhere in a program that a Time object is valid. (Note that the converse is not true.)
- The diagram also makes it clear how `sizeof` works in this case.
- Note that derived classes do not inherit these methods: (the signatures are different for each class)
    - Constructors (including copy constructors)
    - Destructors

Given our classes:

```
class Time
{
public:
	Time(int hrs_, int mins_, int secs_);
	Time(void);
	void Set(int hrs_, int mins_, int secs_);
	void Print(void) const;
	void Increment(void);

private:
	int hrs;
	int mins;
	int secs;
};
```

```
class ExtTime : public Time
{
public:
	ExtTime(void);
	ExtTime(int hrs_, int mins_, int secs_, TimeZone zone_);
	void Set(int hrs_, int mins_, int secs_, TimeZone zone_);
	void Print(void) const;

private:
	TimeZone zone;
};
```

What is the result of the code below? (What is the type of time?)

```
ExtTime time;
time.Set(9, 30, 0); // ???
time.Print();
```

Time Class

```
Time::Time()
{
	Set(0, 0, 0);
}

Time::Time(int hrs_, int mins_, int secs_)
{
	Set(hrs_, mins_, secs_);
}

void Time::Set(int hrs_, int mins_, int secs_)
{
	hrs = hrs_;
	mins = mins_;
	secs = secs_;
}
```

```
void Time::Print(void) const
{
	std::cout.fill('0');
	std::cout << std::cout.width(2) << hrs << ':';
	std::cout << std::cout.width(2) << mins << ':';
	std::cout << std::cout.width(2) << secs;
}

void Time::Increment(void)
{
	++secs;
	if (secs == 60)
	{
		++mins;
		secs = 0;
	}

	if (mins == 60)
	{
		++hrs;
		mins = 0;
	}
}
```

ExTime Class

```
ExtTime::ExtTime(void)
{
	zone = EST; // arbitrary default
}

ExtTime::ExtTime(int hrs_, int mins_, int secs_, TimeZone zone_) :
Time(hrs_, mins_, secs_)
{
	zone = zone_;
}

void ExtTime::Set(int hrs_, int mins_, int secs_, TimeZone zone_)
{
	Time::Set(hrs_, mins_, secs_); // Call base class Set.
	zone_ = zone;
}

void ExtTime::Print(void) const
{
	static const char *TZ[] = { "EST", "CST", "MST", "PST",
		"EDT", "CDT", "MDT", "PDT" };

	Time::Print(); // Call base class Print
	std::cout << " " << TZ[zone];
}
```

Additional notes:

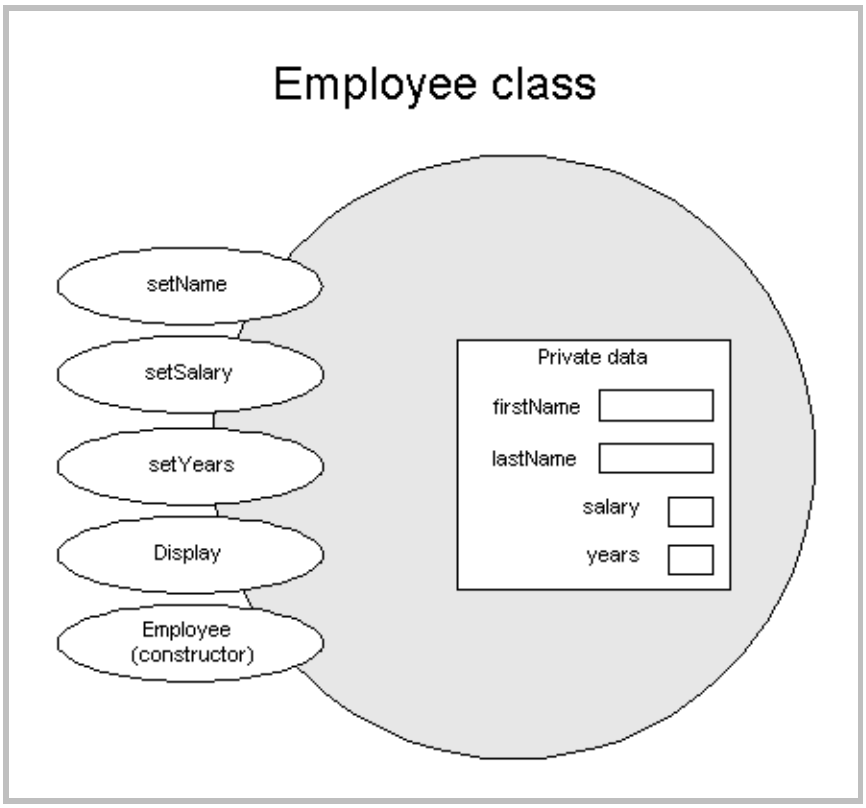
- You cannot overload functions across classes.
- The Set method in *ExtTime* hides the Set method in *Time*.
- Another way to say it: The Set method in the derived class *overrides* the Set method in the base class.

Another Example of Inheritance

The specification (**Employee.h**) for an Employee class:

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>

class Employee
{
private:
	std::string firstName;
	std::string lastName;
	float salary;
	int years;

public:
	Employee(const std::string& firstName_, const std::string& lastName_,
		 float salary_, int years_);
	void setName(const std::string& firstName_, const std::string& lastName_);
	void setSalary(float salary_);
	void setYears(int years_);
	void Display(void) const;
};
#endif
```

A diagram of the Employee class:



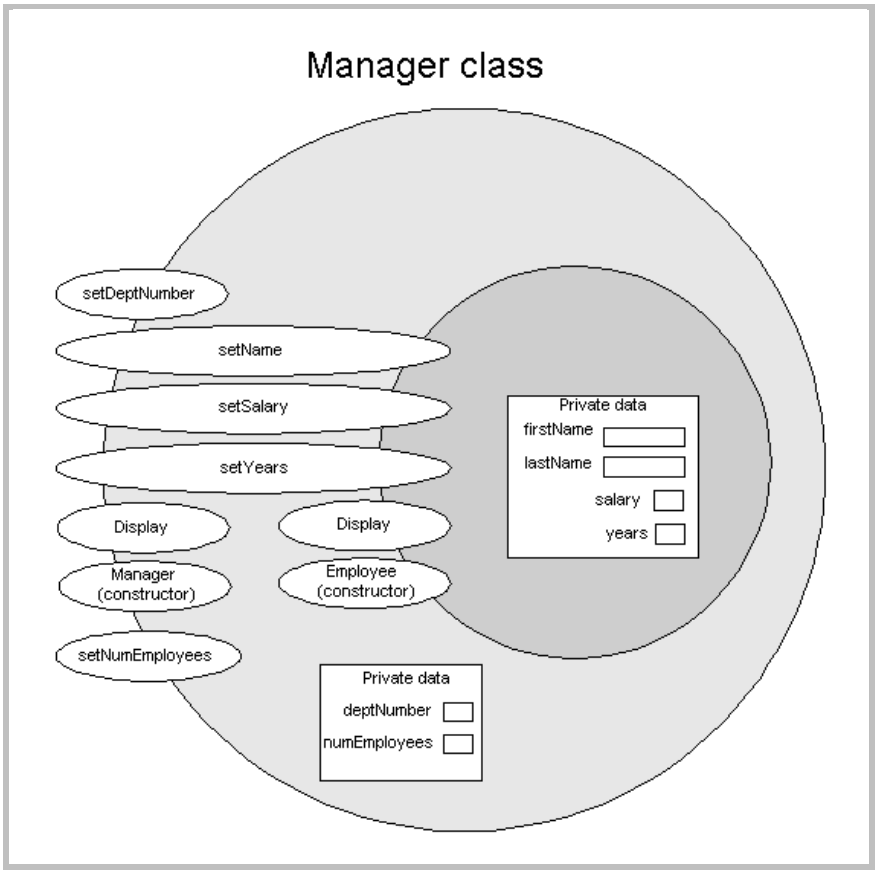
An implementation (**Employee.cpp**) for the Employee class:

```
#include <iostream>
#include <iomanip>
#include "Employee.h"

Employee::Employee(const std::string& firstName_, const std::string& lastName_,
float salary_, int years_) : firstName(firstName_), lastName(lastName_)
{
	salary = salary_;
	years = years_;
}

void Employee::setName(const std::string& firstName_, const std::string&
lastName_)
{
	firstName = firstName_;
	lastName = lastName_;
}

void Employee::setSalary(float salary_)
{
	salary = salary_;
}

void Employee::setYears(int years_)
{
	years = years_;
}

void Employee::Display(void) const
{
	std::cout << "  Name: " << lastName;
	std::cout << ", " << firstName << std::endl;
	std::cout << std::setprecision(2);
	std::cout.setf(std::ios::fixed);
	std::cout << "Salary: $" << salary << std::endl;
	std::cout << " Years: " << years << std::endl;
}
```

The specification (**Manager.h**) for the Manager class:

```
#ifndef MANAGER_H
#define MANAGER_H
#include "Employee.h"

class Manager : public Employee
{
public:
	Manager(const std::string& firstName_, const std::string& lastName_,
		float salary_, int years_, int deptNumber_, int numEmployees_);
	void setDeptNumber(int deptNumber_);
	void setNumEmployees(int numEmployees_);
	void Display(void) const;

private:
	int deptNumber;    // department managed
	int numEmployees;  // employees in department
};
#endif
```

A diagram of the Manager class:



An implementation (**Manager.cpp**) for the Manager class:

```
#include <iostream>
#include "Manager.h"

Manager::Manager(const std::string& firstName_, const std::string& lastName_,
                 float salary_, int years_, int deptNumber_, int numEmployees_) :
                 Employee(firstName_, lastName_, salary_, years_)
{
       deptNumber = deptNumber_;
       numEmployees = numEmployees_;
}

void Manager::Display(void) const
{
       Employee::Display();
       std::cout << "  Dept: " << deptNumber << std::endl;
       std::cout << "  Emps: " << numEmployees << std::endl;
}

void Manager::setDeptNumber(int deptNumber_)
{
       deptNumber = deptNumber_;
}

void Manager::setNumEmployees(int numEmployees_)
{
       numEmployees = numEmployees_;
}
```

Trace the execution of the following program through the class hierarchy. What is the output?

Code

```
#include "employee.h"
#include "manager.h"
#include <iostream>
using std::cout;
using std::endl;

int main(void)
{
       // Create an Employee and a Manager
       Employee emp1("John", "Doe", 30000, 2);
       Manager mgr1("Mary", "Smith", 50000, 10, 5, 8);

       // Display them
       emp1.Display();
       cout << endl;
       mgr1.Display();
       cout << endl;

       // Change the manager's last name
       mgr1.setName("Mary", "Jones");
       mgr1.Display();
       cout << endl;
```

```
        // add two employees and give a raise
        mgr1.setNumEmployees(10);
        mgr1.setSalary(80000);
        mgr1.Display();
        cout << endl;

        system("pause");
        return 0;
}
```

Output

```
  Name: Doe, John
Salary: $30000.00
 Years: 2

  Name: Smith, Mary
Salary: $50000.00
 Years: 10
  Dept: 5
  Emps: 8

  Name: Jones, Mary
Salary: $50000.00
 Years: 10
  Dept: 5
  Emps: 8

  Name: Jones, Mary
Salary: $80000.00
 Years: 10
  Dept: 5
  Emps: 10
```

Self-check

Given these two classes:

```
class A
{
  public:
    A(int x = 0) { a_ = x; }

    void f1()
    {
      std::cout << "A1";
    }

    void f2()
    {
      std::cout << "A2";
    }

    void f3(int)
    {
      std::cout << "A3";
    }

  private:
    int a_;
};
```

```
class B : public A
{
  public:
    B(int x) { a_ = x; }

    void f1(int)
    {
      std::cout << "B1";
    }

    void f3()
    {
      std::cout << "B3";
    }

    void f4()
    {
      std::cout << "B4";
    }

  private:
    int a_;
};
```

Determine if the statement compiles. If it does compile, what is the output?

```
A a;
B b(5);

a.f1();   1. __________

b.f1();   2. __________

a.f2();   3. __________

b.f2();   4. __________

a.f3();   5. __________

b.f3();   6. __________

b.f1(5);  7. __________

b.f3(5);  8. __________
```