Class Inheritance Part 1

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)

The topic in **bold** is what we will be discussing now.

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 more difficult to reuse code from one program to the next since the functions work directly on the data.
- Object-oriented languages generally 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 may use slightly 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*
    - The *Student* class contained (*is composed of*) a *String* object (`login_`).

Inheritance

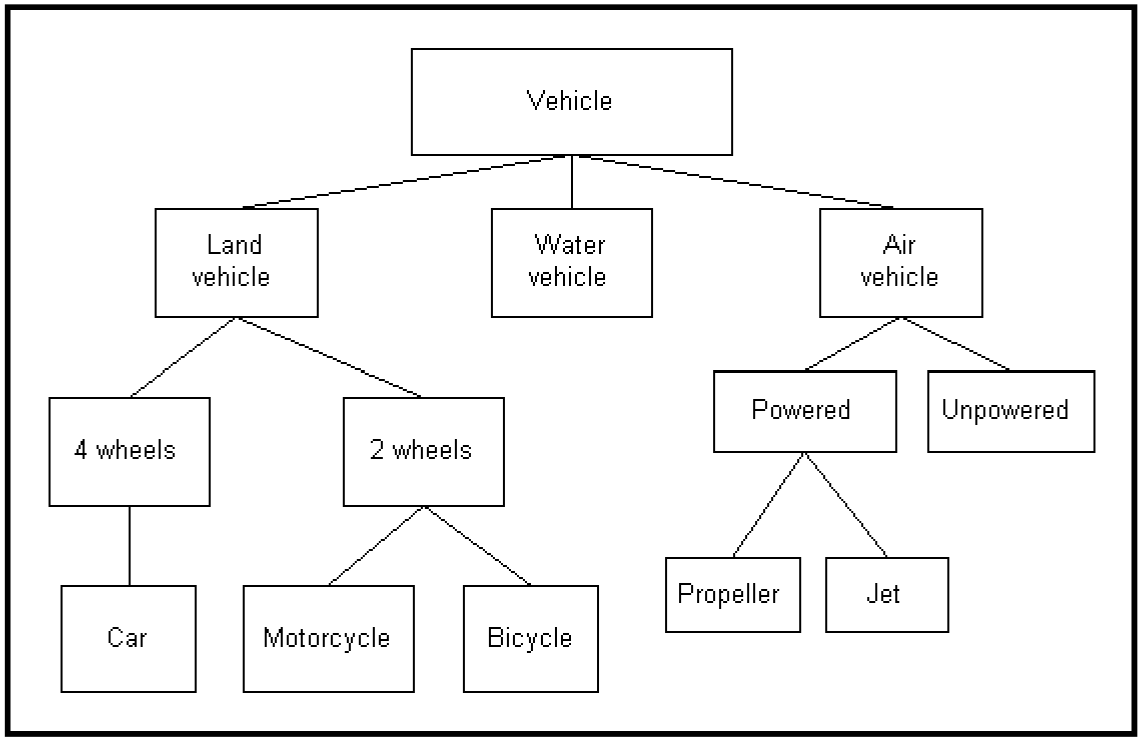
- Relation is an *is-a* relationship. (Also *is-a-kind-of* relationship)
- A car *is a* vehicle, A dog *is an* animal. (A car *is a kind of* vehicle, A dog *is a kind of* animal.)
- Generally, not reversible:
    - All cars are vehicles but not all vehicles are cars.

Composition

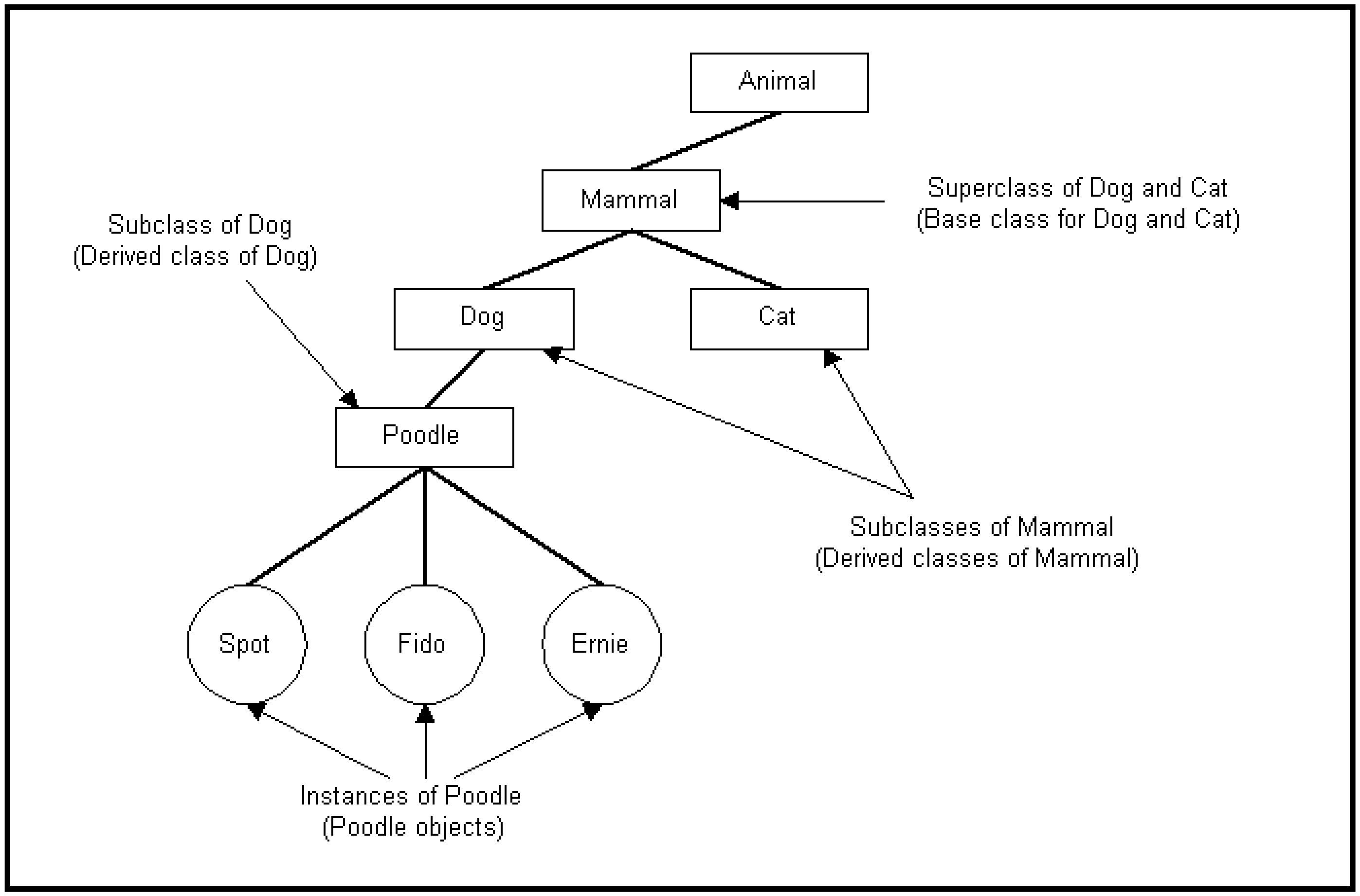
- Relation is a *has-a* relationship.
- Also called aggregation or containment.
- One class is composed of another (maybe several)
- A car *has a* motor (and a steering wheel, and 4 tires, etc.)
    - We don't say a car *is* a motor

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              struct Point3D
{                           {
  double x_;                  double x_;
  double y_;                  double y_;
};                            double z_;
                            };
```

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

```
struct Point3D_composite
{
  Point2D xy_; // Struct contains a Point2D object
  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* to extend the 2D point class:

```
  // 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 and is still compatible with C:

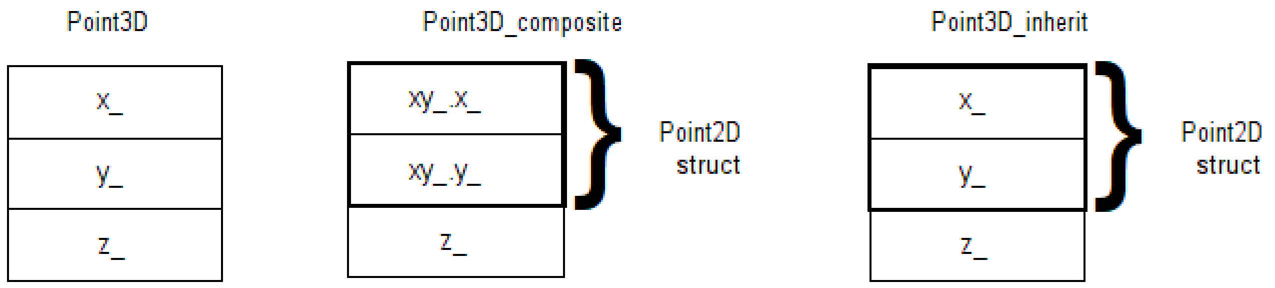
```
struct Point3D                struct Point3D_composite
{                             {
  double x_;                    Point2D xy_; // Struct contains a Point2D object
```

```
    double y_;                  double z_;
    double z_;                };
  };
```

Another overloaded print function:

```
void PrintXYZ(const Point3D_inherit &pt)
{
  std::cout << pt.x_ << ", " << pt.y_ << ", " << pt.z_;
}
```

Visually:



- All of these classes/objects have the same size: 24 bytes.
- There is a subtle difference between the composite and inherited version:
  - The composite object has an explicit, named `Point2D` subobject.
  - The inherited object has an implicit, anonymous `Point2D` subobject.

Sample usage:

```
int main()
{
  Point3D pt3;           // 24 bytes
  Point3D_composite ptc; // 24 bytes
  Point3D_inherit pti;   // 24 bytes

  char buffer[100]; // scratch buffer

    // Displays: pt3: x=0012FF68, y=0012FF70, z=0012FF78
  sprintf(buffer, "pt3: x=%p, y=%p, z=%p\n", &pt3.x_, &pt3.y_, &pt3.z_);
  std::cout << buffer;

    // Displays: ptc: x=0012FF50, y=0012FF58, z=0012FF60
  sprintf(buffer, "ptc: x=%p, y=%p, z=%p\n", &ptc.xy_.x_, &ptc.xy_.y_, &ptc.z_);
  std::cout << buffer;
```

```
    // Displays: pti: x=0012FF38, y=0012FF40, z=0012FF48
  sprintf(buffer, "pti: x=%p, y=%p, z=%p\n", &pti.x_, &pti.y_, &pti.z_);
  std::cout << buffer;

    // Assign to Point3D members
  pt3.x_ = 1;
  pt3.y_ = 2;
  pt3.z_ = 3;
  PrintXYZ(pt3);
  std::cout << std::endl;

    // Assign to Point3D_composite members (explicit subobject)
    // To access the 2D variables, you must use xy_ by name
  ptc.xy_.x_ = 4;
  ptc.xy_.y_ = 5;
  ptc.z_ = 6;
  PrintXYZ(ptc);
  std::cout << std::endl;

    // Assign to Point3D_inherit members (implicit subobject)
    // You can access the 2D variables directly (they are public)
  pti.x_ = 7;
  pti.y_ = 8;
  pti.z_ = 9;
  PrintXYZ(pti);
  std::cout << std::endl;

  return 0;
}
```

Output:

```
pt3: x=0012FF68, y=0012FF70, z=0012FF78
ptc: x=0012FF50, y=0012FF58, z=0012FF60
pti: x=0012FF38, y=0012FF40, z=0012FF48
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 by far 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. Most students don't know the difference, and even some professional C++ programmers don't either!

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

```
struct Point2D
{
  double x_;
  double y_;
  void print()
  {
    std::cout << x_ << ", " << y_;
  }

};
```

```
struct Point3D
{
  double x_;
  double y_;
  double z_;
  void print()
  {
    std::cout << x_ << ", " << y_ << ", " << z_;
  }
};
```

Composition vs. inheritance (currently, everything is **public**):

Composition

```
struct Point3D_composite
{
  Point2D xy_;
  double z_;
  void print()
  {
      // Point2D members are public as well
    std::cout << xy_.x_ << ", " << xy_.y_;
    std::cout << ", " << z_;
  }
};
```

Inheritance

```
struct Point3D_inherit : public Point2D
{
  double z_;
  void print()
  {
      // Point2D members are public as well
    std::cout << x_ << ", " << y_;
    std::cout << ", " << z_;
  }
};
```

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

```
Point3D pt3;
Point3D_composite ptc;
Point3D_inherit pti;

// setup points
```

```
pt3.print();
ptc.print();
pti.print();  // Is this legal? Ambiguous? Which print method is called? (Hint: Think like a compiler!)
```

Now that we have the syntax down, let's make it even more C++-like with constructors, **`private`** members, and **`public`** methods. We'll also 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()
    {
      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()
    {
      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. We've seen this before here with the *Student* class that contains a *String* object.

```
// 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()
    {
      xy_.print(); // 2D members are now private
      std::cout << ", " << z_;
    }
  private:
    Point2D xy_;
    double z_;
};
```

With inheritance, we also must initialize the `Point2D` *subobject* in the initializer list. This is new:

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

We've now come across yet-another situation that requires the use of the *member initializer list* when initializing members of a class: Base class initialization. This brings the count now to 4: 1) constants, 2) references, 3) user-defined types (composition), and 4) base classes (inheritance).

Sample usage:

```
int main()
{
    // 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;
```

```
    return 0;
  }
```

Output:

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

Notes:

- The *print* method in `Point3D_inherit` "hides" (overrides) the *print* method in `Point2D`.
  - You can't overload functions across classes.
- If you need to access the *print* method in the base class (`Point2D`) from outside of the derived class, you have to be explicit:

  ```
  pti.Point2D::print();
  ```

  This works because we're using **public** inheritance. The base class is accessible in the derived class.

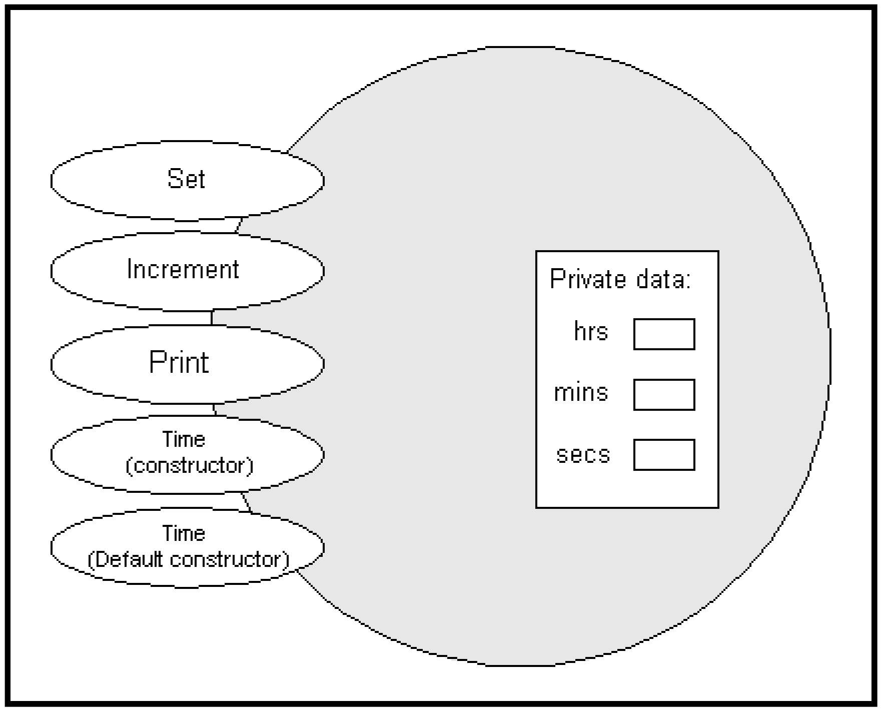
A Larger Example

The Base Class

```
class Time
{
  public:
    Time();
    Time(int h, int m, int s);
    void Set(int h, int m, int s);
    void Print() const;
    void Increment();

  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 h, int m, int s)
{
  Set(h, m, s);
}

void Time::Set(int h, int m, int s)
{
  hrs_ = h;
  mins_ = m;
  secs_ = s;
}
```

Note: To keep the examples simple, I'm omitting all of the data validation in the `Time` class. Normally, you would want to make sure that the values given for hours, minutes, and seconds make sense.

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. Need to maintain two similar sets of code.
3. Easy (if you know what to do), maximum code reuse, no code duplication, relatively straight-forward.

Deriving *ExtTime* from *Time*:

Base class	Derived class

```
class Time
{
  public:
    Time();
    Time(int h, int m, int s);
    void Set(int h, int m, int s);
    void Print() const;
    void Increment();

  private:
    int hrs_;
    int mins_;
    int secs_;
};
```
	```
	class ExtTime : public Time
	{
	  public:
	    ExtTime();
	    ExtTime(int h, int m, int s, TimeZone z);
	    void Set(int h, int m, int s, TimeZone z);
	    void Print() 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()
   {
     zone_ = EST; // arbitrary default
   }
   ```

2. The derived class non-default constructor: (the base class default constructor is **implicitly** called, although this is incorrect behavior)

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

3. Initializing the base class explicitly using the non-default constructor:

   ```
   ExtTime::ExtTime(int h, int m, int s, TimeZone z) : Time(h, m, s)
   {
     zone_ = z;
   }
   ```

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

   ```
   ExtTime::ExtTime(int h, int m, int s, TimeZone z) : Time(h, m, s), zone_(z)
   {
   }
   ```

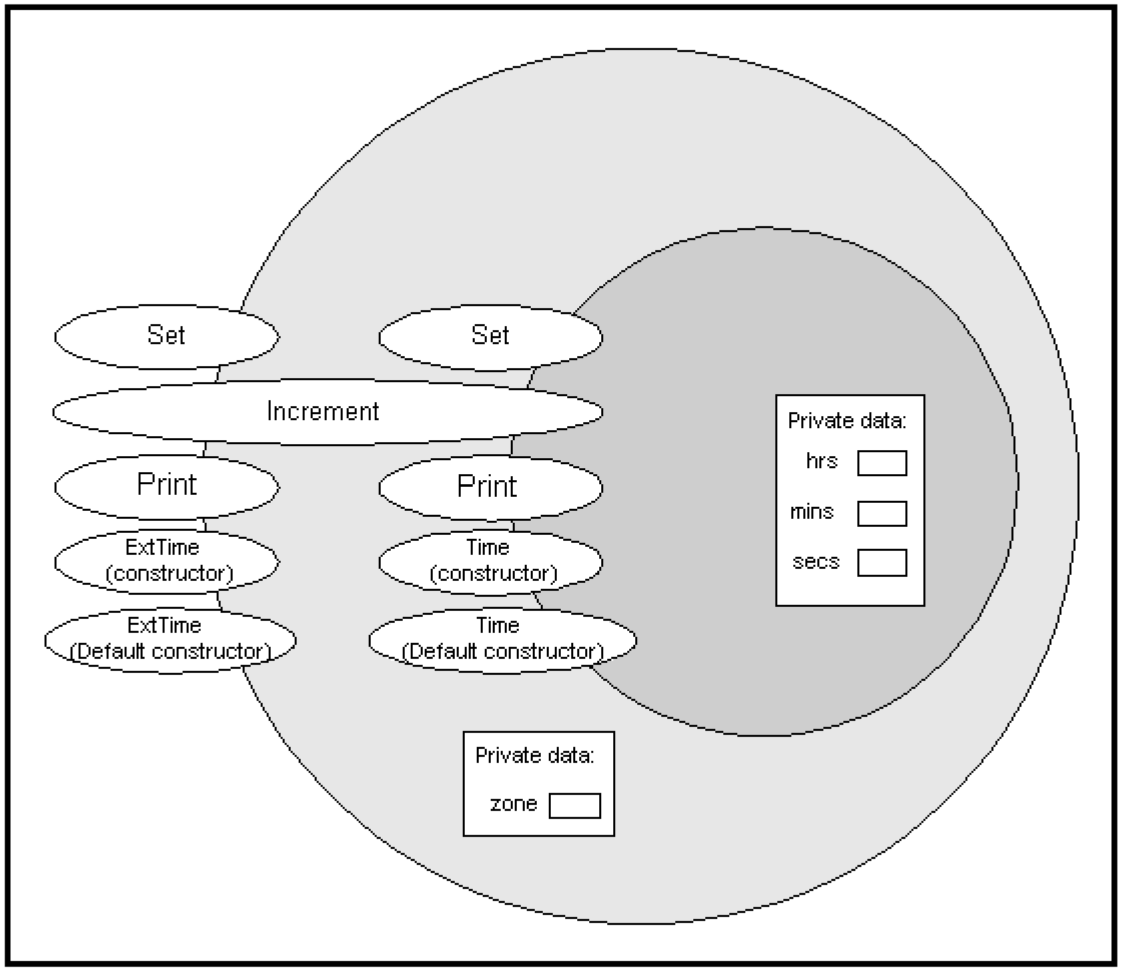
Notes:

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

  ```
  ExtTime::ExtTime(int h, int m, int s, TimeZone z)
  {
    Time(h, m, s); // Can't call a base constructor explicitly (What is this statement actually doing?)
    zone_ = z;
  }
  ```

Key Point: 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 *inherit* the Increment method of the base class.
- We *override* the Set and Print methods of the base class. (Override, not overload!)
    - You cannot overload across classes. (Can't overload a function from the base class in a derived 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) (**New in advanced C++11: inheriting constructors**)

        - Inheriting constructors is nothing special, it's just syntactic sugar for convenience.
    - Destructors
    - Assignment operators

Given our classes:

```
class Time
{
  public:
    Time(int h, int m, int s);
    Time();
    void Set(int h, int m, int s);
    void Print() const;
    void Increment();

  private:
    int hrs_;
    int mins_;
    int secs_;
};
```

```
class ExtTime : public Time
{
  public:
    ExtTime();
    ExtTime(int h, int m, int s, TimeZone z);
    void Set(int h, int m, int s, TimeZone z);
    void Print() const;

  private:
    TimeZone zone_;
};
```

What is the result of the code below? (What is the type of *time*? Remember, think like a compiler.)

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

Time Implementation

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

Time::Time(int h, int m, int s)
{
  Set(h, m, s);
}

void Time::Set(int h, int m, int s)
{
  hrs_ = h;
  mins_ = m;
  secs_ = s;
}
```

ExtTime Implementation

```
ExtTime::ExtTime()
{
  zone_ = EST;
}

ExtTime::ExtTime(int h, int m, int s, TimeZone z) : Time(h, m, s)
{
  zone_ = z;
}

void ExtTime::Set(int h, int m, int s, TimeZone z)
{
  Time::Set(h, m, s); // Call base class Set. h,m,s are private
  zone_ = z;
}
```

```
void Time::Print() const
{
  cout.fill('0');
  cout << setw(2) << hrs_ << ':';
  cout << setw(2) << mins_ << ':';
  cout << setw(2) << secs_;
}

void Time::Increment()
{
  secs_++;
}
```

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

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

If you really want to call the *Set* method of the base class, you'd have to do something like this:

```
ExtTime ext;
ext.Time::Set(1, 2, 3);
```

Of course, what happens to the time zone now? In general, you don't want to do this.

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.
- If the base class has overloaded functions and you create a function with the same name in the derived class, *all* of the overloaded functions with the same name are hidden in the base class.

Another Example of Inheritance

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

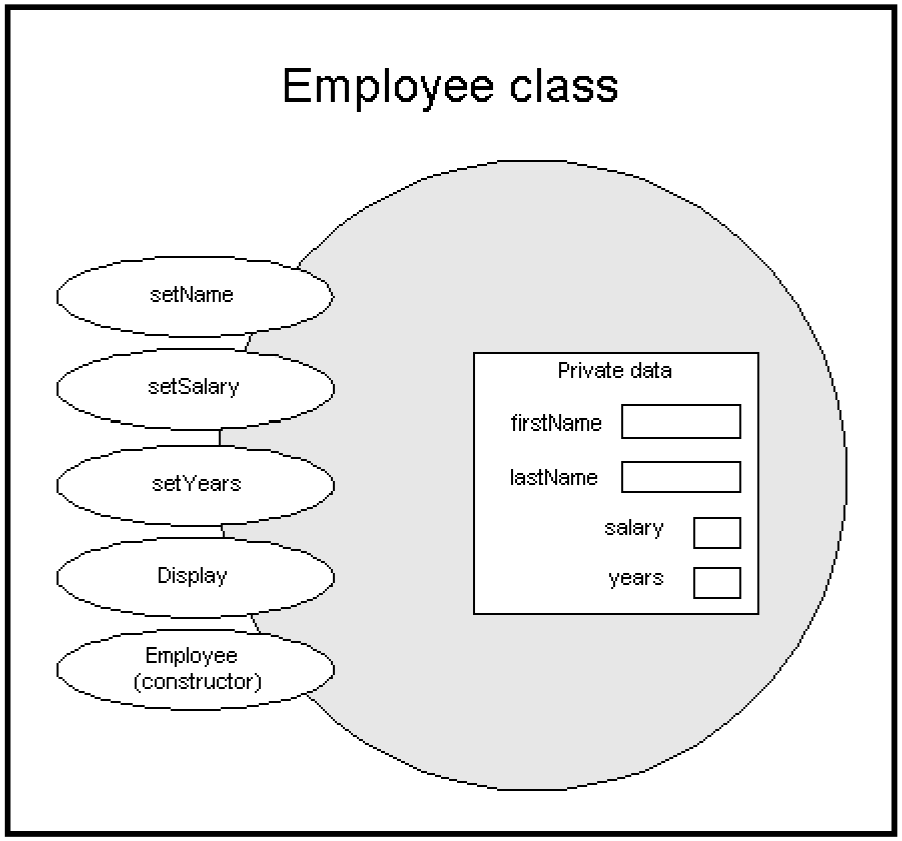
```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>

class Employee
{
  public:
    Employee(const std::string& first, const std::string& last, float sal, int yrs);
    void setName(const std::string& first, const std::string& last);
    void setSalary(float newSalary);
    void setYears(int numYears);
    void Display() const;

  private:
    std::string firstName_;
    std::string lastName_;
    float salary_;
    int years_;
};
#endif
```

A diagram of the Employee class:



What is **sizeof**(Employee)?
What is **sizeof**(std::string)? (Depends on the implementation)

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

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

Employee::Employee(const std::string& first, const std::string& last, float sal, int yrs) : firstName_(first), lastName_(last)
{
  salary_ = sal;
  years_ = yrs;
}

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

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

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

void Employee::Display() 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;
}
```

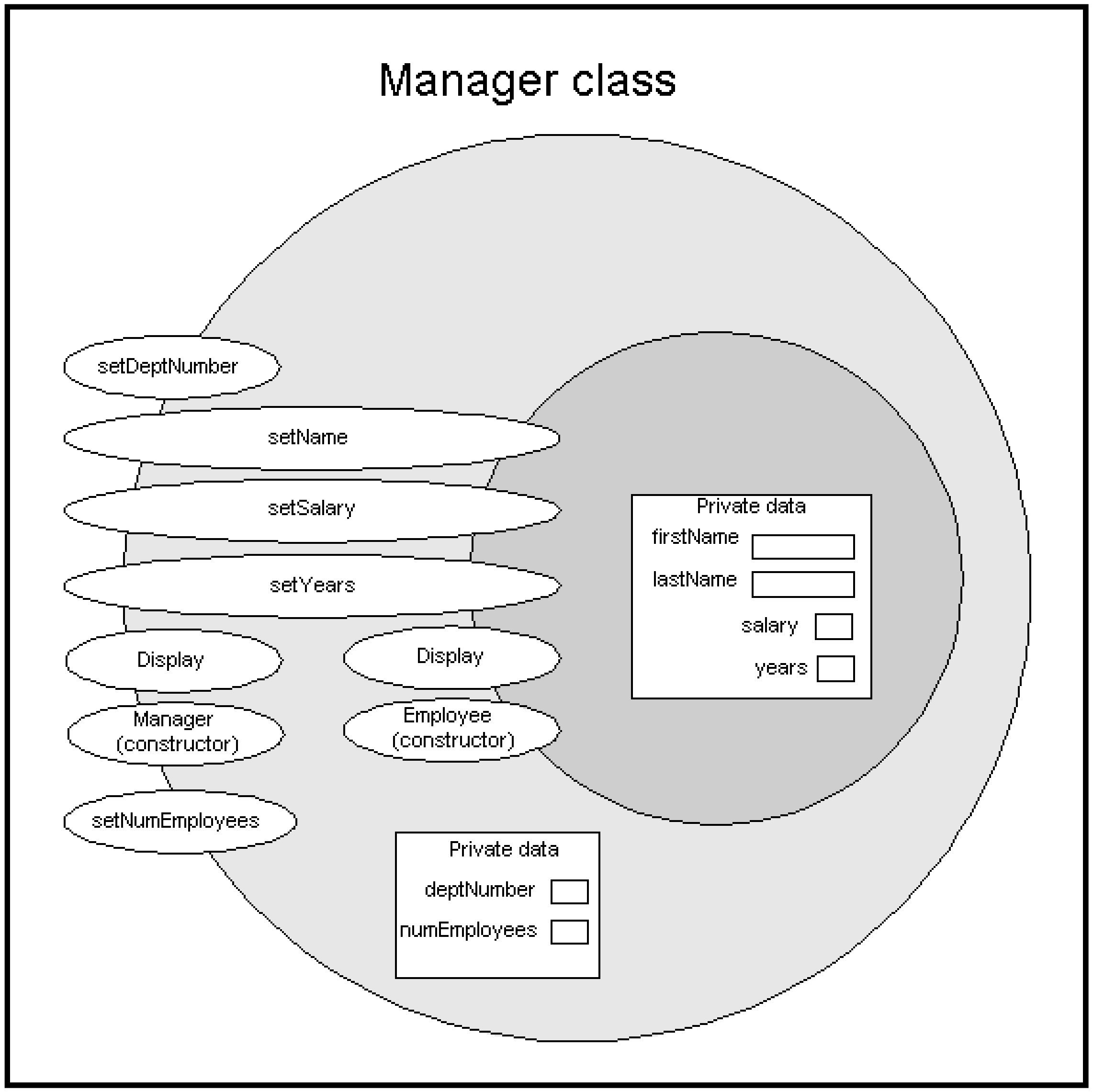
The *specification* (**Manager.h**) for the `Manager` class:

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

class Manager : public Employee
{
  public:
    Manager(const std::string& first, const std::string& last, float sal, int yrs, int dept, int emps);
    void setDeptNumber(int dept);
    void setNumEmployees(int emps);
    void Display() const;

  private:
    int deptNumber_;    // department managed
    int numEmployees_;  // employees in department
};
#endif
```

A diagram of the Manager class:



What is `sizeof(Manager)`?

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

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

Manager::Manager(const std::string& first, const std::string& last, float salary,
                 int years, int dept, int emps) : Employee(first, last, salary, years)
```

```
{
  deptNumber_ = dept;
  numEmployees_ = emps;
}

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

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

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

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

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

int main()
{
    // 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;
```

```
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
```

```
    // Add two employees and give a raise
  mgr1.setNumEmployees(10);
  mgr1.setSalary(80000);
  mgr1.Display();
  cout << endl;
  return 0;
}
```

Self-check

Given these two classes:

```
class A                                 class B : public A
{                                       {
  public:                                 public:
    A(int x = 0) { a_ = x; }                B(int x) { a_ = x; }
    void f1()                               void f1(int)
    {                                       {
      std::cout << "A1";                      std::cout << "B1";
    }                                       }
    void f2()                               void f3()
    {                                       {
      std::cout << "A2";                      std::cout << "B3";
    }                                       }
    void f3(int)                            void f4()
    {                                       {
      std::cout << "A3";                      std::cout << "B4";
    }                                       }
  private:                                private:
    int a_;                                 int a_;
};                                      };
```

Determine if the statement compiles. If it does compile, what is the ouput? If it doesn't compile, give a brief reason why it doesn't.

```
int main()
{
  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. __________

    return 0;
  }
```

Another Example

When learning about the containers and other good stuff in the STL, many students and beginning programmers get caught by this:

```
int main()
{
    // Create empty list of strings
  std::list<std::string> cont1;

    // Add 3 strings
  cont1.push_back("one");    // add to end
  cont1.push_back("two");    // add to end
  cont1.push_back("three");  // add to end

    // Print out the elements using subscripts like with vector
  for (unsigned int i = 0; i < cont1.size(); ++i)
    std::cout << cont1[i] << std::endl;

  return 0;
}
```

but we would quickly be met with this error message: (or something similar)

```
main.cpp: In function `int main()':
main.cpp: error: no match for 'operator[]' in 'cont1[i]'
```

There is no random access (no **`operator[]`**) so we need another way to iterate over the list. And we now know what the correct way is: use an iterator instead of subscripting.

However, suppose that you *really* wanted to add a subscript operator to the standard list class (NOT RECOMMENDED). You can do that easily with inheritance. BTW, you could do it at least 2 other ways (as mentioned at the top of this page). What are they?

```
#include <list>      // std::list (base class)
#include <iostream>  // cout, endl

  // MyList IS-A std::list
class MyList : public std::list<int>
{
  public:
    int& operator[](int index); // for L-values

    // All other public methods from std::list are inherited ...
};

int& MyList::operator[](int index)
{
    // Start at the "head"
  MyList::iterator it = begin();

    // Skip over items until we reach the one
  for (int i = 0; i < index; i++)
    ++it;

    // Return the item "pointed to" by it
  return *it;
}

int main()
{
  MyList list;

  list.push_back(8);
  list.push_back(3);
  list.push_back(5);
  list.push_back(2);
  list.push_back(1);
  list.push_back(9);

    // Prints out: 8 3 5 2 1 9
  for (int i = 0; i < list.size(); i++)
    std::cout << list[i] << " ";
  std::cout << std::endl;

  return 0;
}
```

The new *MyList* class *is-a std::list*, meaning, anything you can do with a *std::list*, you can do with a *MyList*. This was just a demonstration of "extending" an existing class. You would NEVER want to do this for real because the performance is abysmal. With a class like *vector*, which has a subscript operator that works in constant time, the *MyList* subscript operator is linear. Linear is much, much slower than constant. In fact, the loop above that prints out the items is actually N^2!!