# Introduction to Inheritance

## Object-Oriented Programming

The three pillars of Object Oriented Programming are:

1. Encapsulation [data abstraction/hiding implemented via a `class`]
2. Inheritance [Is-a relationship that extends a `class`]
3. Polymorphism [dynamic binding that is implemented via `virtual` methods]

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

- Non object-oriented programming typically uses top-down design or structured design where the problem is decomposed into modules.
- Such 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
  - dynamic polymorphism [or dynamic binding]
- Object-oriented programs are collections of interacting objects.
- In C++, classes provide data abstraction; a class is essentially an **A**bstract **D**ata **T**ype.
- Client programs don't work directly on the data in an object; instead they "ask" the object to manipulate its own data via its public interface consisting of member functions [called methods in other object-oriented languages such as Java].
- 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 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 ***composition***, also called ***aggregation*** or ***containment***
3. Two classes are related by ***inheritance***

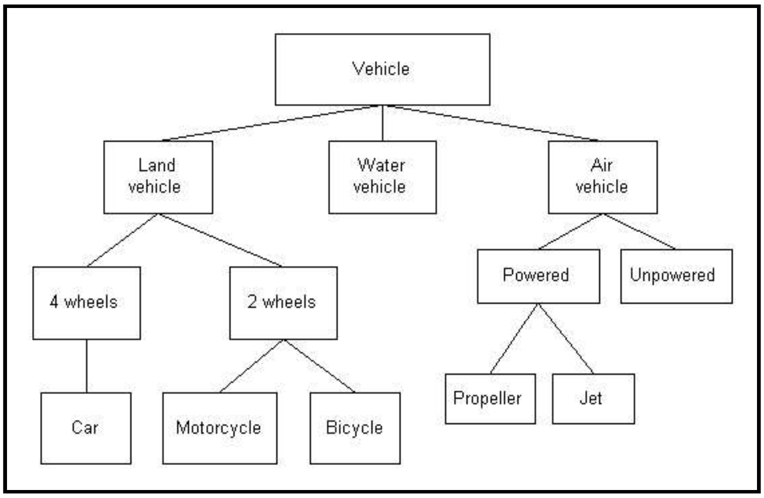
**Composition**

- 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.)

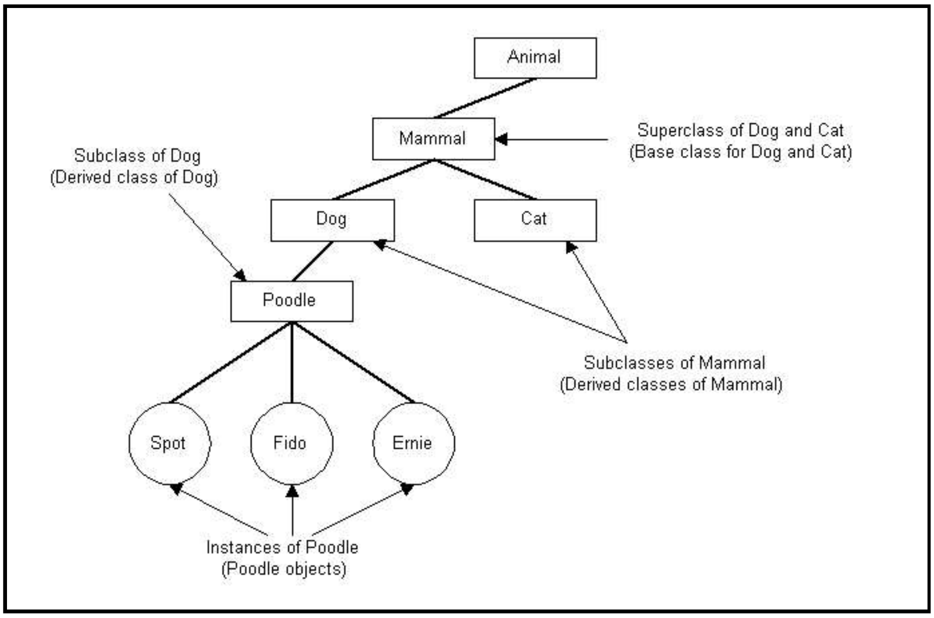
**Inheritance**

- 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.

An inheritance relationship can be represented by a hierarchy. A partial vehicle hierarchy is illustrated:




A partial animal hierarchy:




## A Simple Example

2D and 3D points can be represented as objects of structure types:

```
struct Point2D {
  double x_;
  double y_;
};
```

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

Another way to represent 3D points is by reusing type `Point2D`:

```
struct Point3D_composite {
  Point2D xy_; // Struct contains a Point2D object
  double z_;
};
```

The memory layout for objects of type `Point3D` and `Point3D_composite` is identical and is obviously compatible with C, as there is nothing "C++" about them yet. The data members of these two types are accessed like this:

```
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_;
};
```

Type `Point3D_inherit` has the exact same physical structure as types `Point3D` and `Point3D_composite` and continues to be compatible with C types:

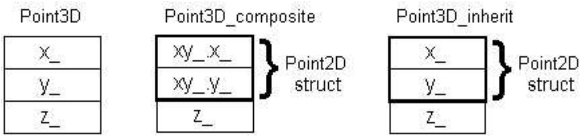
```
struct Point3D {
  double x_;
  double y_;
  double z_;
};
```

```
struct Point3D_composite {
  Point2D xy_; // Struct contains a Point2D object
  double z_;
};
```

We overload `print()` to deal with objects of type `Point3D_inherit`:

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

The memory layout of the data members of the three types can be visually represented like this:




and we use the three types like this:

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

  char buffer[100];

    // 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
  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;
}
```

The program has the following 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
```

The syntax:

```
struct Point3D_inherit : public Point2D
```

can be understood like this:

- **`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 type of inheritance.
- There is also **private** inheritance, but it is used much less.

We add methods to the types like this:

```
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_;
  }
};
<
```

The types with composition and inheritance are defined like this:

**Composite**

```
struct Point3D_composite {
  Point2D xy_;
  double z_;
  void print() {
      // 2D members are public
```

**Inheritance**

```
struct Point3D_inherit : public Point2D {
  double z_;
  void print() {
      // 2D members are public
    std::cout << x_ << ", " << y_;
```

```
    std::cout << xy_.x_ << ", " << xy_.y_;
    std::cout << ", " << z_;
  }
};
```

```
    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 method is called?
```

---

Let's make it more C++-like with `private` data members and `public` member functions:

```
// 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:

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

Sample usage of these classes looks like this:

```
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;
}
```

---

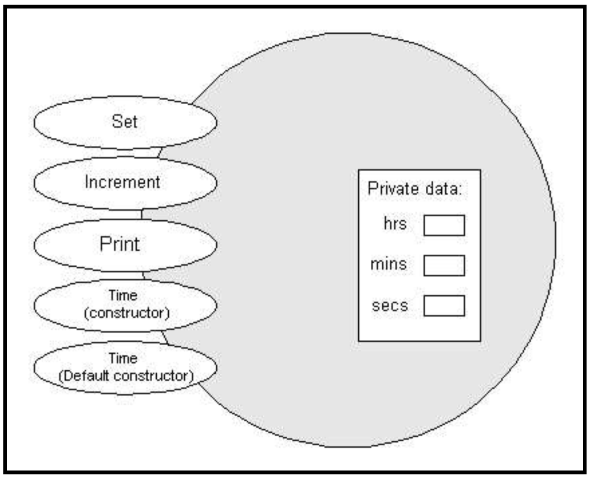
## A Larger Example

### The Base Class

```
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_;
};
```

Objects of type class **Time** have the following memory layout:




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

Notice the code reuse even in the definitions of member functions of class **Time** in *Time.cpp*:

```
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;
}
```

**Class Design Tip:** When a class has more than one constructor, move the common initialization code into a separate method [usually private] whenever possible.

## Extending class `Time`

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

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

We have several choices at this point:

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

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

Deriving **ExtTime** from **Time**:

```
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 **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]:

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

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

```
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) {
}
```

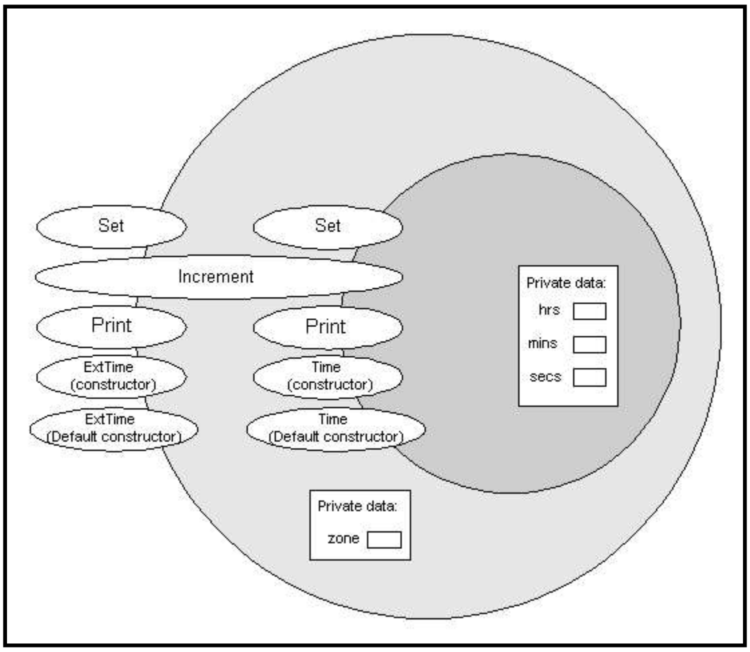
Notes:

- The derived constructor calls the default base constructor if you don't call it explicitly.
- You can call any base constructor explicitly.
- 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 classes **Time** and **ExtTime** is illustrated:




In class **ExtTime**:

- We *override* member functions **Set()** and **Print()** of the base class.
- We *inherit* member function **Increment()** 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 object of type **ExtTime** *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
  - 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_;
```

```
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_;
};
```

```
    int mins_;
    int secs_;
};
```

What is the result of the code below?

```
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;
}

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

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

**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 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_];
}
```

## Another Example of Inheritance

The specification for class **Employee** in file ***Employee.h*** looks like this:

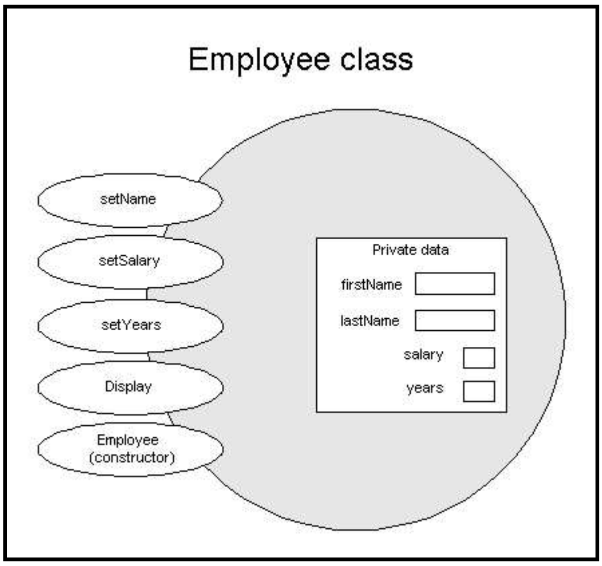
```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>

class Employee {
  private:
    std::string firstName_;
    std::string lastName_;
    double salary_;
    int years_;

  public:
    Employee(const std::string& first, const std::string& last, double sal, int yrs);
    void setName(const std::string& first, const std::string& last);
    void setSalary(double newSalary);
    void setYears(int numYears);
    void Display() const;
};
#endif
```

A diagram of class **Employee** looks like this:




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

An implementation of class **Employee** in file ***Employee.cpp*** looks like this:

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

Employee::Employee(const std::string& first, const std::string& last, double 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(double 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;
}
```

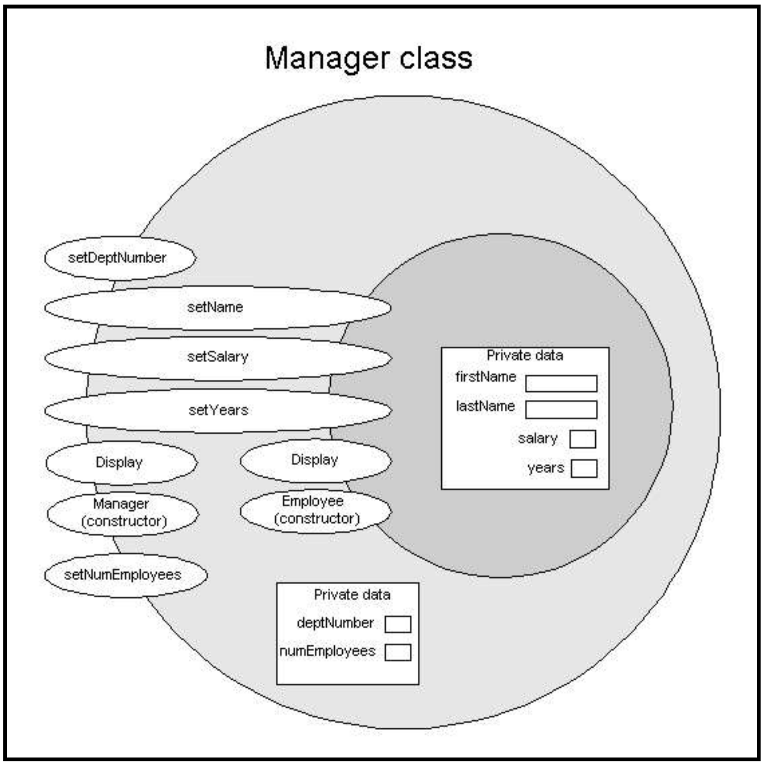
The *specification* for class **Manager** in file ***Manager.h*** looks like this:

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

class Manager : public Employee {
  public:
    Manager(const std::string& first, const std::string& last, double 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 class **Manager** looks like this:




What is **sizeof(Manager)** evaluate to?

An implementation for class **Manager** in file ***Manager.cpp*** looks like this:

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

Manager::Manager(const std::string& first, const std::string& last, double 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;

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

---

## Protected vs. Private Data

In a nutshell:

- All classes we've seen so far have **private** and **public** members.
- There is a third type of access control: **protected**, which is kind of like a hybrid of **private** and **public**
- Any base class member marked as **protected** can be accessed directly by derived classes (no need to go through a **public** member function)
- To the "rest of the world", the **protected** members appear to be **private**, but to derived classes they appear to be **public**.

Let's extend class **Employee** slightly by including accessor methods for the private data. These methods would most likely be necessary in any real-world class but were intentionally left out to keep the example and diagrams very simple. These methods simply retrieve the private data for clients.

**Interface:**

```
const char *getFirstName() const;
const char *getLastName() const;
double getSalary() const;
int getYears() const;
```

**Implementation:**

```
const char *Employee::getFirstName() const {
  return firstName_;
}

const char *Employee::getLastName() const {
  return lastName_;
}
```

```
double Employee::getSalary() const {
  return salary_;
}

int Employee::getYears() const {
  return years_;
}
```

We'll also add a new method to class **Manager** that simply reports its internal state. This type of operation was not required when class **Employee** was designed and implemented.

```
void Manager::LogActivity() const {
  char buffer[105];
  const char *fn = getFirstName();  // use public accessor method
  const char *ln = getLastName();   // use public accessor method

  sprintf(buffer, "%s, %s", ln, fn); // Format lastname, firstname

  cout << "===========================" << endl;
  cout << "Manager data:" << endl;
  cout << "  Dept: " << deptNumber_ << endl;
  cout << "  Emps: " << numEmployees_ << endl;
  cout << "Employee data:" << endl;
```

```
    cout << "  Name: " << buffer << endl;
    cout << "  Salary: " << getSalary() << endl;
    cout << "  Years: " << getYears() << endl;
    cout << "============================" << endl;
  }
```

Some sample client code:

```
#include "Manager.h"

int main() {
  Manager m1("Ian", "Faith", 5, 80000, 7, 25);
  m1.LogActivity();
}
```

**Output:**

```
============================
Manager data:
  Dept: 7
  Emps: 25
Employee data:
  Name: Faith, Ian
  Salary: 5
  Years: 80000
============================
```

---

## Modification #1

Modifying class **Employee** so that the **private** data is now **protected**:

```
class Employee            {
  protected:
    char firstName_[MAX_LENGTH];
    char lastName_[MAX_LENGTH];
    double salary_;
    int years_;

  public:

    // Public methods...

};
```

How will this affect existing code, both derived classes and "global" code?

This will allow us to change the implementation of class **Manager** to access the fields directly:

```
void Manager::LogActivity() const {
  char buffer[105];
  const char *fn = firstName_;  // direct access
  const char *ln = lastName_;   // direct access

  sprintf(buffer, "%s, %s", ln, fn);

  cout << "============================" << endl;
  cout << "Manager data:" << endl;
  cout << "  Dept: " << deptNumber_ << endl;
  cout << "  Emps: " << numEmployees_ << endl;
  cout << "Employee data:" << endl;
  cout << "  Name: " << buffer << endl;
  cout << "  Salary: " << salary_ << endl;
  cout << "  Years: " << years_ << endl;
  cout << "============================" << endl;
}
</pre>
```

Note that "regular" clients still must go through the public member functions:

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

int main() {
  Manager m1("Ian", "Faith", 5, 80000, 7, 25);
  m1.LogActivity();  // now using protected data directly; main is unaware

    // Error: lastName_ is a protected member (still inaccessible here)
  const char *last1 = m1.lastName_;

    // OK: getLastName() is a public member
  const char *last2 = m1.getLastName();
}
```

Notes:

- The **protected** data of class **Employee** is still hidden to the "general public" and is, therefore, safe.
- The **protected** data is now directly available to class **Manager** [and any other class that we might derive from **Employee**].
- Note that no changes to class **Manager** were *required*. We made them because class **Employee** relaxed its access to the data.
- Should we just make the data **protected** so that in case we extend a class via inheritance the derived class can access "its" data without going through cumbersome public methods?
- As always, the answer is *it depends*, but in general the safe answer is *"No"*.

---

## Modification #2

Modifying the implementation of class **Employee**:

```
#include "String.h"

class Employee {
  protected:
    String firstName_;  // String is a user-defined class
```

```
    String lastName_;   // String is a user-defined class
    double salary_;
    int years_;

  public:

    // Same public methods...
};
```

How will this affect existing code, both derived classes and "global" code?
Now what is **sizeof(Employee)**?

Won't affect global, breaks derived. Objects of type **String** will require an additional 4 bytes.

- Interface for class **String**:

```
class String {
  private:
    char *data_;       // the "real" C string

  public:
    String();                   // default constructor
    String(const String &str); // copy constructor
    String(const char *str);   // conversion constructor
    ~String();                  // destructor

    String & operator=(const String &str); // assigning another String
    String & operator=(const char *str);   // assigning a char *
    const char *c_str(void) const;         // get raw C string (data_)

      // Other public methods...
};
```

- We will have to modify the internal implementation of these two public methods. We may have to modify other methods as well, but these are sufficient to support our public interface.

```
const char *Employee::getFirstName() const {
  return firstName_.c_str(); // return as a C string
}

const char *Employee::getLastName() const {
  return lastName_.c_str(); // return as a C string
}
```

- This code in the derived class is now broken:

```
void Manager::LogActivity() const {
  char buffer[105];

  const char *fn = firstName_; // direct access
  const char *ln = lastName_;  // direct access

    // Other code...
}
```

  because **firstName_** and **lastName_** are no longer C strings.

- Interestingly, global code is unaffected:

```
int main() {
  Manager m1("Ian", "Faith", 5, 80000, 7, 25);

    // This is still OK because getLastName() was modified as well
  const char *last2 = m1.getLastName();
}
```

Notes:

- Making data **protected** essentially makes it public in derived classes.
- This is definitely a double-edged sword:
  1. Derived classes can access the data directly without going through public methods.
  2. Changes to the base class data may affect *all* derived classes that rely on a particular implementation.
- Any kind of direct access promotes *tight-coupling*, which is generally an undesirable coding practice. Code to an interface, not an implementation.
- Are you making data protected to keep the syntax simpler? This is generally not a good reason to use protected.
- With inline member functions, there is no penalty for making a function call.
- If you don't want all code to access the private data with public methods, consider making some of the methods protected for derived classes.