# Introduction to Inheritance in C++

1. **Introduction:**  
   Modern object-oriented (OO) languages provide 3 capabilities:  
   * encapsulation
   * inheritance
   * polymorphism

which can improve the design, structure and reusability of code.Here, we'll explore how the object-oriented (OO) programming capability known as *inheritance* can be used in C++.All code examples are available for [download](https://www.cs.bu.edu/teaching/cpp/inheritance/intro/download/).

1. ***Employee* example:**  
   Real-world entities, like *employees*, are naturally described by **both** data and functionality.  
   We will represent different types of employees:  
   * a *generic employee*
   * a *manager*
   * a *supervisor*

For these employees, we'll store *data*, like their:

* + name
  + pay rate

And...we'll require some *functionality*, like being able to:

* + initialize the employee
  + get the employee's fields (e.g., name)
  + calculate the employee's pay

**Note:** We don't care what the pay period for an employee is. They might receive pay weekly, bi-weekly, monthly, etc. It is not important in this example.

1. **Employee class:**  
   Object-oriented languages typically provide a natural way to treat data and functionality as a single entity. In C++, we do so by creating a *class*.  
   Here is a class definition for a generic Employee: class Employee {  
   public:  
    Employee(string theName, float thePayRate);  
     
    string getName() const;  
    float getPayRate() const;  
     
    float pay(float hoursWorked) const;  
     
   protected:  
    string name;  
    float payRate;  
   };  
     
   **Note:** For now, just think of the "protected" keyword as being like "private".  
   The class consists of:  
   * A constructor to initialize fields of the class.
   * Methods to "get" the fields.
   * A method to calculate the employee's pay (given the number of hours worked).

Definitions for each of the methods follow:Employee::Employee(string theName, float thePayRate)  
{  
 name = theName;  
 payRate = thePayRate;  
}  
  
string Employee::getName() const  
{  
 return name;  
}  
  
float Employee::getPayRate() const  
{  
 return payRate;  
}  
  
float Employee::pay(float hoursWorked) const  
{  
 return hoursWorked \* payRate;  
}  
Note that the payRate is used as an hourly wage.The class would be used something like:#include "employee.h"  
  
...  
  
 Employee empl("John Burke", 25.0);  
  
 // Print out name and pay (based on 40 hours work).  
 cout << "Name: " << empl.getName() << endl;  
 cout << "Pay: " << empl.pay(40.0) << endl;

1. **Manager class:**  
   In the real world, we don't view everything as unique; we often view something as being *like* something else but with *differences* or *additions*.  
   Managers are *like* regular employees; however, there might be differences. For example, they might be paid by a *salary*.  
   **Note:** Employees paid by a *salary* (i.e., those that are *salaried*) get a fixed amount of money each pay period (e.g., week, 2 weeks, month) regardless of how many hours they work.  
   Our first attempt to write a class for a manager gives the following class definition:class Manager {  
   public:  
    Manager(string theName,  
    float thePayRate,  
    bool isSalaried);  
     
    string getName() const;  
    float getPayRate() const;  
    bool getSalaried() const;  
     
    float pay(float hoursWorked) const;  
     
   protected:  
    string name;  
    float payRate;  
    bool salaried;  
   };  
   It mainly differs from Employee in that it has an additional field (salaried) and method (getSalaried()).  
   The method definitions for class Manager do not differ much from Employee either:Manager::Manager(string theName,  
    float thePayRate,  
    bool isSalaried)  
   {  
    name = theName;  
    payRate = thePayRate;  
    salaried = isSalaried;  
   }  
     
   string Manager::getName() const  
   {  
    return name;  
   }  
     
   float Manager::getPayRate() const  
   {  
    return payRate;  
   }  
     
   bool Manager::getSalaried() const  
   {  
    return salaried;  
   }  
     
   float Manager::pay(float hoursWorked) const  
   {  
    if (salaried)  
    return payRate;  
    /\* else \*/  
    return hoursWorked \* payRate;  
   }  
     
   They add very little new code to what was written in Employee.  
   Compared to Employee, in Manager...  
   * The methods getName() and getPayRate() are identical to those in Employee.
   * Method getSalaried() is new.
   * The constructor and pay() method work differently. Nonetheless, they do some of the same work as their counterparts in the Employee class.

Finally, the payRate has 2 possible uses in the Manager class...float Manager::pay(float hoursWorked) const  
{  
 if (salaried)  
 return payRate;  
 /\* else \*/  
 return hoursWorked \* payRate;  
}  
If the manager is salaried, payRate is the fixed rate for the pay period; otherwise, it represents an hourly rate, just like it does for a regular employee.Such a Manager can be used in a similar manner to an Employee:#include "manager0.h"  
  
...  
  
 Manager mgr("Jan Kovacs", 1200.0, true);  
  
 // Print out name and pay (based on 40 hours work).  
 cout << "Name: " << mgr.getName() << endl;  
 cout << "Pay: " << mgr.pay(40.0) << endl;

1. **Reuse:**  
   We have done unnecessary work to create Manager, which is similar to (and really is a "kind of") Employee.  
   We can fix this using the OO concept of *inheritance*. If we let a manager inherit from an employee, then it will get all the data and functionality of an employee. We can then add any new data and methods needed for a manager and *redefine* any methods that differ for a manager.  
   Here, we show a new implementation of Manager that *inherits* from Employee:#include "employee.h"  
     
   class Manager : public Employee {  
   public:  
    Manager(string theName,  
    float thePayRate,  
    bool isSalaried);  
     
    bool getSalaried() const;  
     
    float pay(float hoursWorked) const;  
     
   protected:  
    bool salaried;  
   };  
     
   The line:class Manager : public Employee {  
   causes Manager to inherit all the data and methods of Employee.  
   **Note:** Although other access specifiers (besides "public") can be used with inheritance, we will only discuss public inheritance here.  
   The only things included in the class definition are:  
   * a constructor,
   * the new field salaried,
   * a way to access it with the method getSalaried(),
   * and a declaration for pay() (which is redefined in Manager).

Like this new class definition, the method definitions are also simplified:Manager::Manager(string theName,  
 float thePayRate,  
 bool isSalaried)  
 : Employee(theName, thePayRate)  
{  
 salaried = isSalaried;  
}  
  
bool Manager::getSalaried() const  
{  
 return salaried;  
}  
  
float Manager::pay(float hoursWorked) const  
{  
 if (salaried)  
 return payRate;  
 /\* else \*/  
 return Employee::pay(hoursWorked);  
}  
There are some things to note about these method definitions...**Member initialization list**For constructors that require arguments, you must write a new constructor for each class.**Note:** Classes don't explicitly inherit constructors.For the Manager class, we needed a constructor:Manager::Manager(string theName,  
 float thePayRate,  
 bool isSalaried)  
 : Employee(theName, thePayRate)  
{  
 salaried = isSalaried;  
}  
that does some of the same work as the Employee constructor. To do so, we *reused* Employee's constructor.The only way to pass values to Employee's constructor in this context is via a *member initialization list*.A member initialization list follows a constructor's parameter list. It consists of a colon (:) and a comma-separated list of inherited class names (and values to be passed to their constructors).**Note:** The *member initialization list* can also be used to pass values to constructors of data members. For example,class SomeClass {  
public:  
 SomeClass();  
  
private:  
 const int SIZE;  
 AnotherClass data;  
};  
  
SomeClass::SomeClass() : SIZE(10), data("foo")  
{  
 // more initialization code  
}  
Without doing so, SIZE could not be initialized (because its constant) and data's default constructor (if it has one) would be used.**The protected access specifier**Methods of Manager have access to payRate because it was [declared in Employee](https://www.cs.bu.edu/teaching/cpp/inheritance/intro/#Employee-class-def) as "protected":float Manager::pay(float hoursWorked) const  
{  
 if (salaried)  
 return payRate; // Yeah, I can use!  
 ...  
}  
I.e., classes that inherit a "protected" field or method can access them.For those *using* an object (versus those *defining* a class), "protected" works like the "private" access specifier:Manager mgr;  
mgr.payRate; // Doesn't work!  
I.e., the "protected" fields remain inaccessible just as they were in Employee:Employee empl;  
empl.payRate; // Doesn't work!  
**Calling inherited methods**The pay() method of Manager uses a different calculation if the manager is *salaried*. Otherwise, it makes the same calculation as a regular Employee:float Manager::pay(float hoursWorked) const  
{  
 if (salaried)  
 return payRate;  
 /\* else \*/  
 return Employee::pay(hoursWorked);  
}  
We *reused* the pay() method of Employee to define the pay() method of Manager.Note that when we call Employee's pay() method:Employee::pay(hoursWorked);  
we must explicitly specify the class from which it comes (i.e., from which it was inherited). Without doing so, we'd have an infinite recursive call:float Manager::pay(float hoursWorked) const  
{  
 ...  
 return pay(hoursWorked); // Calls Manager::pay()!  
}  
This new Manager class can be used just like our first attempt:#include "manager.h"  
  
...  
  
 Manager mgr("Jan Kovacs", 1200.0, true);  
  
 // Print out name and pay (based on 40 hours work).  
 cout << "Name: " << mgr.getName() << endl;  
 cout << "Pay: " << mgr.pay(40.0) << endl;  
Excitingly, it has methods from Employee, like getName(), that we did not declare or define in Manager...Remember, it *inherited* all the data and methods of an Employee! Thus, we have *reused* our definition of an employee to simplify defining a manager.

#### **Class Hierarchy:** Since we now have one class that inherits from another, we have the beginnings of a *class hierarchy*:Employee | Manager We say that Employee is the *base class* and Manager is a *derived class* of Employee. **Note:** Alternatively, we may call Employee the *superclass* and Manager the *subclass*. If needed, this hierarchy could be extended to include more classes. **Adding a Supervisor**To add another type of employee, such as a *supervisor*, a new class can be created. Two choices of where to place a Supervisor class in the hierarchy are:a) Employee b) Employee | / \ Manager Manager Supervisor | Supervisor

* 1. A supervisor is a kind of manager.  
     The Supervisor class directly inherits from Manager and *indirectly* inherits from Employee.
  2. A supervisor is just a special kind of employee.  
     Supervisor *directly* inherits from Employee.

**Aside:** We can say that Supervisor *inherits* from Employee when there is either a direct or indirect inheritance relationship.*Which hierarchy would we choose?*If a supervisor is viewed as part of management, then choice a) is probably your answer. Nonetheless, this is a decision not to be taken lightly. How one designs the inheritance hierarchy greatly affects what you can do with those classes later.

1. **Exercise:**  
   Take the code we've provided for the Employee class ([employee.h](https://www.cs.bu.edu/teaching/cpp/inheritance/intro/download/employee.h) and [employee.cpp](https://www.cs.bu.edu/teaching/cpp/inheritance/intro/download/employee.cpp)) and the Manager class ([manager.h](https://www.cs.bu.edu/teaching/cpp/inheritance/intro/download/manager.h) and [manager.cpp](https://www.cs.bu.edu/teaching/cpp/inheritance/intro/download/manager.cpp)).  
   Add methods to the classes named:  
   * setName()
   * setPayRate()
   * setSalaried()

that let users change the corresponding fields. Take advantage of the *inheritance* relationship between Employee and Manager--you only need add each of those methods to 1 class.Write a Supervisor class. A *supervisor* is responsible for employees in a specific department and must:

* + Have a field to store the *department name (as a string).*
  + *Have getDept() and setDept() methods to access the department field.*
  + *Always be salaried (i.e., pay for a single pay period is fixed, no matter how many hours are worked).*
  + *Have a constructor that takes initial values for all fields.*

*What class should Supervisor inherit from?Your code should compile and run correctly with the test program* [*empltest.cpp*](https://www.cs.bu.edu/teaching/cpp/inheritance/intro/download/empltest.cpp)*.*

*BU CAS CS - Introduction to Inheritance in C++*

*Copyright © 1993-2000 by Robert I. Pitts <rip at bu dot edu>. All Rights Reserved.*