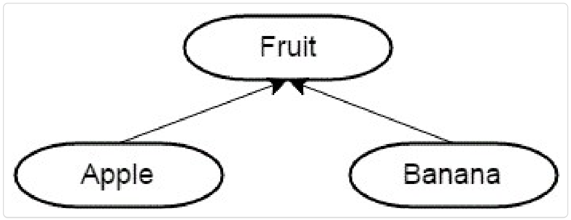
# 11.2 — Basic inheritance in C++

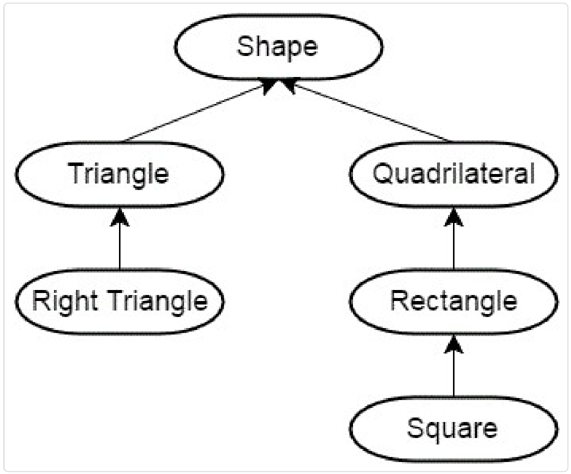
BY ALEX ON JANUARY 4TH, 2008 | LAST MODIFIED BY ALEX ON SEPTEMBER 24TH, 2018

Now that we've talked about what inheritance is in an abstract sense, let's talk about how it's used within C++.

Inheritance in C++ takes place between classes. In an inheritance (is-a) relationship, the class being inherited from is called the **parent class**, **base class**, or **superclass**, and the class doing the inheriting is called the **child class**, **derived class**, or **subclass**.



In the above diagram, Fruit is the parent, and both Apple and Banana are children.



In this diagram, Triangle is both a child (to Shape) and a parent (to Right Triangle).

A child class inherits both behaviors (member functions) and properties (member variables) from the parent (subject to some access restrictions that we'll cover in a future lesson). These variables and functions become members of the derived class.

Because child classes are full-fledged classes, they can (of course) have their own members that are specific to that class. We'll see an example of this in a moment.

**A Person class**

Here's a simple class to represent a generic person:

```
#include <string>

```

```cpp
class Person
{
public:
    std::string m_name;
    int m_age;

    Person(std::string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

};
```

Because this Person class is designed to represent a generic person, we've only defined members that that would be common to any type of person. Every person (regardless of gender, profession, etc…) has a name and age, so those are represented here.

Note that in this example, we've made all of our variables and functions public. This is purely for the sake of keeping these examples simple right now. Normally we would make the variables private. We will talk about access controls and how those interact with inheritance later in this chapter.

**A BaseballPlayer class**

Let's say we wanted to write a program that keeps track of information about some baseball players. Baseball players need to contain information that is specific to baseball players -- for example, we might want to store a player's batting average, and the number of home runs they've hit.

Here's our incomplete Baseball player class:

```cpp
class BaseballPlayer
{
public:
    double m_battingAverage;
    int m_homeRuns;

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
       : m_battingAverage(battingAverage), m_homeRuns(homeRuns)
    {
    }
};
```

Now, we also want to keep track of a baseball player's name and age, and we already have that information as part of our Person class.

We have three choices for how to add name and age to BaseballPlayer:
1) Add name and age to the BaseballPlayer class directly as members. This is probably the worst choice, as we're duplicating code that already exists in our Person class. Any updates to Person will have to be made in BaseballPlayer too.
2) Add Person as a member of BaseballPlayer using composition. But we have to ask ourselves, "does a BaseballPlayer have a Person"? No, it doesn't. So this isn't the right paradigm.
3) Have BaseballPlayer inherit those attributes from Person. Remember that inheritance represents an is-a relationship. Is a BaseballPlayer a Person? Yes, it is. So inheritance is a good choice here.

**Making BaseballPlayer a derived class**

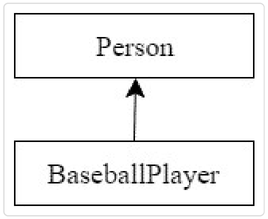
To have BaseballPlayer inherit from our Person class, the syntax is fairly simple. After the `class BaseballPlayer` declaration, we use a colon, the word "public", and the name of the class we wish to inherit. This is called *public inheritance*. We'll talk more about what public inheritance means in a future lesson.

```cpp
// BaseballPlayer publicly inheriting Person
```

```cpp
class BaseballPlayer : public Person
{
public:
    double m_battingAverage;
    int m_homeRuns;

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
       : m_battingAverage(battingAverage), m_homeRuns(homeRuns)
    {
    }
};
```

Using a derivation diagram, our inheritance looks like this:



When BaseballPlayer inherits from Person, BaseballPlayer acquires the member functions and variables from Person. Additionally, BaseballPlayer defines two members of its own: m_battingAverage and m_homeRuns. This makes sense, since these properties are specific to a BaseballPlayer, not to any Person.

Thus, BaseballPlayer objects will have 4 member variables: m_battingAverage and m_homeRuns from BaseballPlayer, and m_name and m_age from Person.

This is easy to prove:

```cpp
#include <iostream>
#include <string>

class Person
{
public:
    std::string m_name;
    int m_age;

    Person(std::string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

};

// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_battingAverage;
    int m_homeRuns;

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
       : m_battingAverage(battingAverage), m_homeRuns(homeRuns)
    {
    }
};

int main()
{
    // Create a new BaseballPlayer object
    BaseballPlayer joe;
    // Assign it a name (we can do this directly because m_name is public)
    joe.m_name = "Joe";
    // Print out the name
    std::cout << joe.getName() << '\n'; // use the getName() function we've acquired from the Person base class

    return 0;
}
```

Which prints the value:

```
Joe
```

This compiles and runs because joe is a BaseballPlayer, and all BaseballPlayer objects have a m_name member variable and a getName() member function inherited from the Person class.

**An Employee derived class**

Now let's write another class that also inherits from Person. This time, we'll write an Employee class. An employee "is a" person, so using inheritance is appropriate:

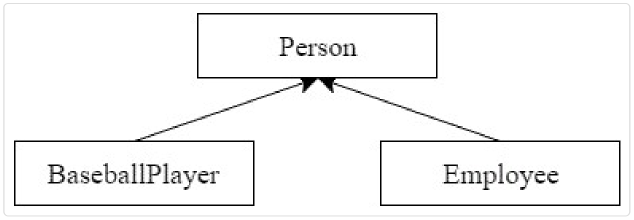
```cpp
// Employee publicly inherits from Person
class Employee: public Person
{
public:
    double m_hourlySalary;
    long m_employeeID;

    Employee(double hourlySalary = 0.0, long employeeID = 0)
        : m_hourlySalary(hourlySalary), m_employeeID(employeeID)
    {
    }

    void printNameAndSalary() const
    {
        std::cout << m_name << ": " << m_hourlySalary << '\n';
    }
};
```

Employee inherits m_name and m_age from Person (as well as the two access functions), and adds two more member variables and a member function of its own. Note that printNameAndSalary() uses variables both from the class it belongs to (Employee::m_hourlySalary) and the parent class (Person::m_name).

This gives us a derivation chart that looks like this:



Note that Employee and BaseballPlayer don't have any direct relationship, even though they both inherit from Person.

Here's a full example using Employee:

```cpp
#include <iostream>
#include <string>

class Person
{
public:
    std::string m_name;
    int m_age;

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

    Person(std::string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }
};

// Employee publicly inherits from Person
class Employee: public Person
{
public:
    double m_hourlySalary;
    long m_employeeID;

    Employee(double hourlySalary = 0.0, long employeeID = 0)
        : m_hourlySalary(hourlySalary), m_employeeID(employeeID)
    {
    }

    void printNameAndSalary() const
    {
        std::cout << m_name << ": " << m_hourlySalary << '\n';
    }
};

int main()
{
    Employee frank(20.25, 12345);
    frank.m_name = "Frank"; // we can do this because m_name is public

    frank.printNameAndSalary();

    return 0;
}
```

This prints:

```
Frank: 20.25
```

**Inheritance chains**

It’s possible to inherit from a class that is itself derived from another class. There is nothing noteworthy or special when doing so -- everything proceeds as in the examples above.

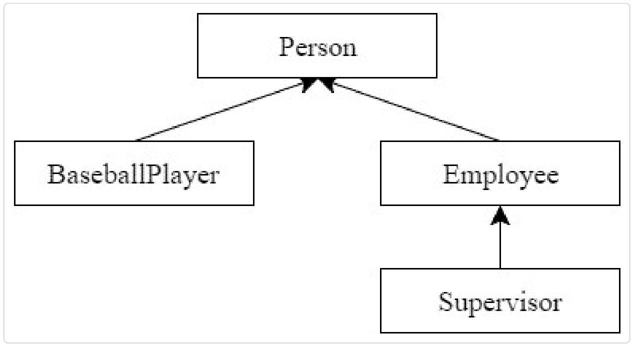
For example, let’s write a Supervisor class. A Supervisor is an Employee, which is a Person. We’ve already written an Employee class, so let’s use that as the base class from which to derive Supervisor:

```
class Supervisor: public Employee
{
public:
    // This Supervisor can oversee a max of 5 employees
    long m_overseesIDs[5];

    Supervisor()
    {
    }

};
```

Now our derivation chart looks like this:



All Supervisor objects inherit the functions and variables from both Employee and Person, and add their own m_overseesIDs member variable.

By constructing such inheritance chains, we can create a set of reusable classes that are very general (at the top) and become progressively more specific at each level of inheritance.

**Why is this kind of inheritance useful?**

Inheriting from a base class means we don’t have to redefine the information from the base class in our derived classes. We automatically receive the member functions and member variables of the base class through inheritance, and then simply add the additional functions or member variables we want. This not only saves work, but also means that if we ever update or modify the base class (e.g. add new functions, or fix a bug), all of our derived classes will automatically inherit the changes!

For example, if we ever added a new function to Person, both Employee and Supervisor would automatically gain access to it. If we added a new variable to Employee, Supervisor would also gain access to it. This allows us to construct new classes in an easy, intuitive, and low-maintenance way!

**Conclusion**

Inheritance allows us to reuse classes by having other classes inherit their members. In future lessons, we'll continue to explore how this works.

11.3 -- Order of construction of derived classes

Index

11.1 -- Introduction to inheritance

**Share this:**

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

**42 comments to 11.2 — Basic inheritance in C++**

**Nigel Booth**

October 13, 2018 at 4:15 am · Reply

Hi Alex,

In your base class (Person):

```
class Person
{
```

```
3   public:
4       std::string m_name;
5       int m_age;
6
7       std::string getName() const { return m_name; }
8       int getAge() const { return m_age; }
9
10      Person(std::string name = "", int age = 0)
11          : m_name(name), m_age(age)
12      {
13      }
14  };
```

you define int getAge() as a const.  Surely age would be variable and not const as it increases over time so you could have a function like:

```
int adage():public Person
{
bool isNewYear();
if isNewYear(true)
Age = Age+1;
}
```

**nascardriver**
October 13, 2018 at 7:31 am · Reply

Hi Nigel!

The "const" in line 8 doesn't affect @m_age. It means that @Person::getAge doesn't modify the object is can be called on a const object.

Aurora
September 22, 2018 at 3:52 am · Reply

Minor typo:

```
and add their own m_nOverseesIDs member variable.
```

instead of

```
and add their own m_OverseesIDs member variable.
```

Alex
September 24, 2018 at 7:42 am · Reply

Typo fixed! Thanks for the note.

Maxwell Ndirangu
August 14, 2018 at 11:50 pm · Reply

Hi please help me with this question;
Oxford University requires a program to store details of its employees/students.
Using C++ write a program with the full specifications;

i)A person's class that defines properties common to both students/employees.
ii)A student class that is derived from the person's class and is used to define properties that further define a student e.g. Aggregate grade & Admission number.

**nascardriver**
August 15, 2018 at 2:36 am · Reply

Hi Maxwell!

What exactly do you need help with? What do you have so far? What have you tried?

vibin
August 12, 2018 at 3:57 am · Reply

I am inheriting base1 and base2 as public. Base1 contains someFunction( ) in public and base2 contains someFunction( ) in private. So only base1 someFunction( ) is inherited. But when compiled,it shows error -

request for member 'someFunction' is ambiguous.
candidates are: void base2::someFunction()
void base1::someFunction()

```
// C++ program to implement
// Multiple Inheritance
#include <iostream>
using namespace std;
class base1
{
  public:
    void someFunction( )
    { cout<<"x"<<endl; }
};
class base2
{
   private:
   void someFunction( )
    { cout<<"y"<<endl; }
};
class derived : public base1, public base2
```

```
{

};

int main()
{
    derived obj;

    obj.someFunction(); // Error!

    return 0;
}
```

**nascardriver**
August 12, 2018 at 4:02 am · Reply

Hi Vibin!

Overload resolution is performed before access modifiers are taken into account.

vibin
August 12, 2018 at 4:18 am · Reply

But private data members are not inherited,right?.In base2 someFunction( ) is under private.

Quoxa
August 1, 2018 at 11:49 pm · Reply

Hi,
probably just a small typo:
in the sentence "We will talk about access controls and how those intersect with inheritance later in this chapter.", you meant interact instead of intersect right?

Alex
August 2, 2018 at 9:26 pm · Reply

I kinda think it works either way, but I like your suggestion better. Incorporated. Thanks!

Ned
April 16, 2018 at 4:30 am · Reply

Is

```
Employee(double hourlySalary = 0.0, long employeeID = 0)
        : m_hourlySalary(hourlySalary), m_employeeID(employeeID)
    {
    }
```

the same as

```
Employee(double hourlySalary = 0.0, long employeeID = 0)
```

nascardriver

April 16, 2018 at 4:35 am · Reply

Hi Ned!

Assuming you mean

```
Employee(double hourlySalary = 0.0, long employeeID = 0)
{
  m_hourlySalary = hourlySalary;
  m_employeeID = employeeID;
}
```

It's very similar, but not the same. See lesson 8.5a - Constructor member initializer lists.

Denis

March 1, 2018 at 6:07 am · Reply

Great lesson! Thank you for all the diagrams they were very helpful.

Lamont Peterson

September 19, 2017 at 4:19 pm · Reply

Alex,

I noticed that line 8 in the snippet initially showing the "Employee" class does not match line 26 in the "... full example using Employee:" which immediately follows; the first showing of Employee does not include default values in the Constructor.

Alex

September 21, 2017 at 3:48 pm · Reply

Fixed. Thanks!

Alexxx

July 20, 2017 at 7:48 am · Reply

awesome tutorials

"use the getName() function we've acquired from the Player base class". I think its inherited from Person class

Alex

July 20, 2017 at 11:17 pm · Reply

Indeed. Thanks for pointing this out.

Omri
June 26, 2017 at 12:55 pm · Reply

a. Note that the arrow tip is pointing opposite to what seems to me the "information flow direction". Somewhat counter intuitive... child_class aquires members from parent_class... Should the direction be interpreted as " arrow_base_class inherits from arrow_tip_class"?
b. As for now constructors can only be used with their corresponding classes. As for now employee did not inherit a person constructor in anyvway. Is this correct?

Alex
June 27, 2017 at 5:04 pm · Reply

a) Half the people think the arrows should go up, half of them think they should go down. It depends on whether you think about inheritance from the point of "derived inherits from base" or "base gives its stuff to derived" (the information flow perspective). In these tutorials, I use the arrow_base inherits from arrow_tip model.
b) Constructors aren't inherited. However, the derived class does have access to the base class constructor, and can use it to initialize the base portion of the derived object. We discuss this in lots more detail in the next lesson or two.

Tomas
November 21, 2016 at 1:06 am · Reply

There is a couple of typos in a employee class, line10 std::string getName() const { return m_bame; } should m_name

and I believe lane40 frank.m_name = 'Frank'; // it should be "Frank" shouldnt it?

Other from that really impressive work!

However, I have a question. In first example Person Class you use getName and getAge and use these functions to cout, wouldnt it be more useful to overload std::ostream& operator?

Alex
November 21, 2016 at 9:41 am · Reply

Thanks for pointing out the typos. They're fixed.

Yes, we could have just as easily provided an overloaded operator<<. However, using a normal function seemed more straightforward (no dealing with friends or overloaded operators) since that's not what the example was trying to show.

Matt
October 1, 2016 at 6:00 pm · Reply

In the code directly above the section titled "Inheritance chains", in the Employee class, 2nd member variable, you wrote "long m_lEmployeeID" instead of "long m_employeeID".

Alex
October 3, 2016 at 3:42 pm · Reply

Also fixed.

Matt

October 1, 2016 at 5:49 pm · Reply

For your Person class, in your getName() function, you returned "m_bame" instead of "m_name".

Alex
October 3, 2016 at 3:42 pm · Reply

Thanks! Fixed.

Vaibhav
January 25, 2016 at 9:51 pm · Reply

Hi Alex;
I don't get it why is it necessary to initialize a default constructor, why cant we define a parametrized one??

Vaibhav
January 26, 2016 at 11:26 am · Reply

I mean is it a rule for inheritance that the constructor type must be default?

Alex
January 26, 2016 at 5:44 pm · Reply

Nope. Inheritance doesn't care about constructors. However, if you construct a derived class, C++ will use the default constructor from the base class unless you tell it otherwise. We'll explore this (including how to tell it otherwise) in more detail in the next two lessons.

Alex
January 26, 2016 at 1:25 pm · Reply

I do not understand what you are asking. Can you clarify?

chandu
October 7, 2015 at 12:52 am · Reply

Hi,

please observe my code, it's follows:

```
[[

#include <iostream>

#include <string>

using namespace std;

class person
{
```

```
private:
     string c_mName;
     int c_mAge;
     bool c_mSex;

public:

    person(string c_fName, int c_fAge, bool c_fSex):
               c_mName (c_fName), c_mAge(c_fAge), c_mSex(c_fSex)
    {

    }

    string Get_mName(void)
    {
         return c_mName;
    }

    int Get_mAge(void)
    {
       return c_mAge;
    }

    bool Get_mSex(void)
    {
        return c_mSex;
    }

    void Set_mName(string c_fName)
    {
        c_mName = c_fName;
    }

    void Set_mAge(int c_fAge)
    {
        c_mAge = c_fAge;
    }

    void Set_mSex(bool c_fSex)
    {
        c_mSex = c_fSex;
    }

    friend istream& operator >> (istream &in, person &c_person)
    {
         in >> c_person.c_mName;
         in >> c_person.c_mAge;
         in >> c_person.c_mSex;
```

```
                return in;
            }

            friend ostream& operator << (ostream &out, person &c_person)
            {
                out << c_person.c_mName<<endl <<
                    c_person.c_mAge<<endl << c_person.c_mSex<<endl;
                return out;
            }


};
class cricket_player : public person
{
      private:
             string c_mState;
             int c_mJersyNum;

      public:
            cricket_player(string c_fName, int c_fAge, bool c_fSex, string c_fState, int c_fJersyNum) :
                        person(c_fName, c_fAge, c_fSex) , c_mState(c_fState), c_mJersyNum(c_fJersyNum)
            {

            }
             string Get_mPlayerState(void)
             {
                   return c_mState;
             }

             int Get_mPlayerJesryNum(void)
             {
                return c_mJersyNum;
             }

             friend ostream& operator << (ostream &out, cricket_player &c_player)
            {
                out  << c_player.c_mState<<endl <<
                        c_player.c_mJersyNum<<endl;
                return out;
            }
};
```

```
int main()
{
    cricket_player c_obj1("chandu", 24, 1, "telangana", 1);

    cout << "enter the name,age,sex of a player: ";

    cin >> c_obj1;

    cout << c_obj1;

    getchar();
    getchar();
}
]]
```

My question is how to overload the <<(out) operator of the base class with the derived class object? there is any possible way for it to print the base class members with the overloading output operator (or) we could access through member functions only??

please run the code & understand it???????

here my expected display is:
name
age
sex
state
jersynum with the help of overloading of out (>>) operator,
but i got the output like:
state
jersy num
how to get entire expected output with operator overloading concept? there is any possibility for it??????

Alex
October 7, 2015 at 12:21 pm · Reply

You can use a dynamic_cast to convert a reference to a derived class object into a reference to a base class object.

Add the following line to the top of your output operator for cricket_player:

```
out << dynamic_cast<person&>(c_player); // call the output operator for person
```

```

    // now output the cricket_player-specific fields
```

vish
September 18, 2015 at 3:23 am · Reply

Hey..make me clear here..in the inheritance chaining the supervisor can use the person class or not?
And if not how two classes can be inherited in one class?

Alex
September 18, 2015 at 8:53 am · Reply

Yes, a Supervisor is an Employee, and an Employee is a Person, so a Supervisor is a Person.

momalok
January 15, 2014 at 6:57 am · Reply

Hi Alex.
Im a bit confused with your employee supervisor example. Surely a supervisor both HAS an employee and IS an employee, so how would we know in this case which one of composition or inheritance to utilise?
I understand the rest of it fine, im just not sure on that little bit, or if the example exhibits chain inheritance well enough.
Thanks.

Alex
September 21, 2016 at 11:34 am · Reply

You'd use both. Because the Supervisor itself "is-an" Employee, it should use inheritance to become a subclass of Employee. Because the supervisor "has" employee IDs, it can use composition to store the Employee IDs.

edgeoftheworld
April 27, 2010 at 4:56 pm · Reply

Thank you very much for your tutorials. They are a great help :).
-cheers.

o.o!
December 9, 2008 at 7:11 am · Reply

class Supervisor: public Employee
{
public:
// This Supervisor can oversee a max of 5 employees
int m_nOverseesIDs[5]; // <== [4]?
};

**Alex**
December 9, 2008 at 9:02 am · Reply

It's correct as is. Declaring an array with [5] means there are 5 elements, numbered 0 through 4.

davidv
September 17, 2008 at 2:20 pm · Reply

Is the inheritance chart necessarily a tree? In other words, could Supervisor have inherited both BaseballPlayer and Employee at the same time?

davidv
September 17, 2008 at 3:44 pm · Reply

Nevermind, section 11.7 covers this. That's what happens with questions asked too early in a chapter. 🙂

**Alex**
September 21, 2008 at 10:05 am · Reply

It's better to leave it as a tree if you can. Although it technically doesn't have to be (using multiple inheritance), things stay much simpler if it is a tree.