

ISA, HASA, inheritance classes

Last changed 3/25/2022

Code examples can be found at

<https://engineering.purdue.edu/~smidkiff/ECE39595CPP/Code/L3Inheritance.zip>

What we will learn in this set of slides

- What are HASA and ISA relations?
- Polymorphism, Inheritance, base and derived classes, subclasses and superclasses, function overriding
- What are virtual functions
- Virtual function tables
- Virtual function calls

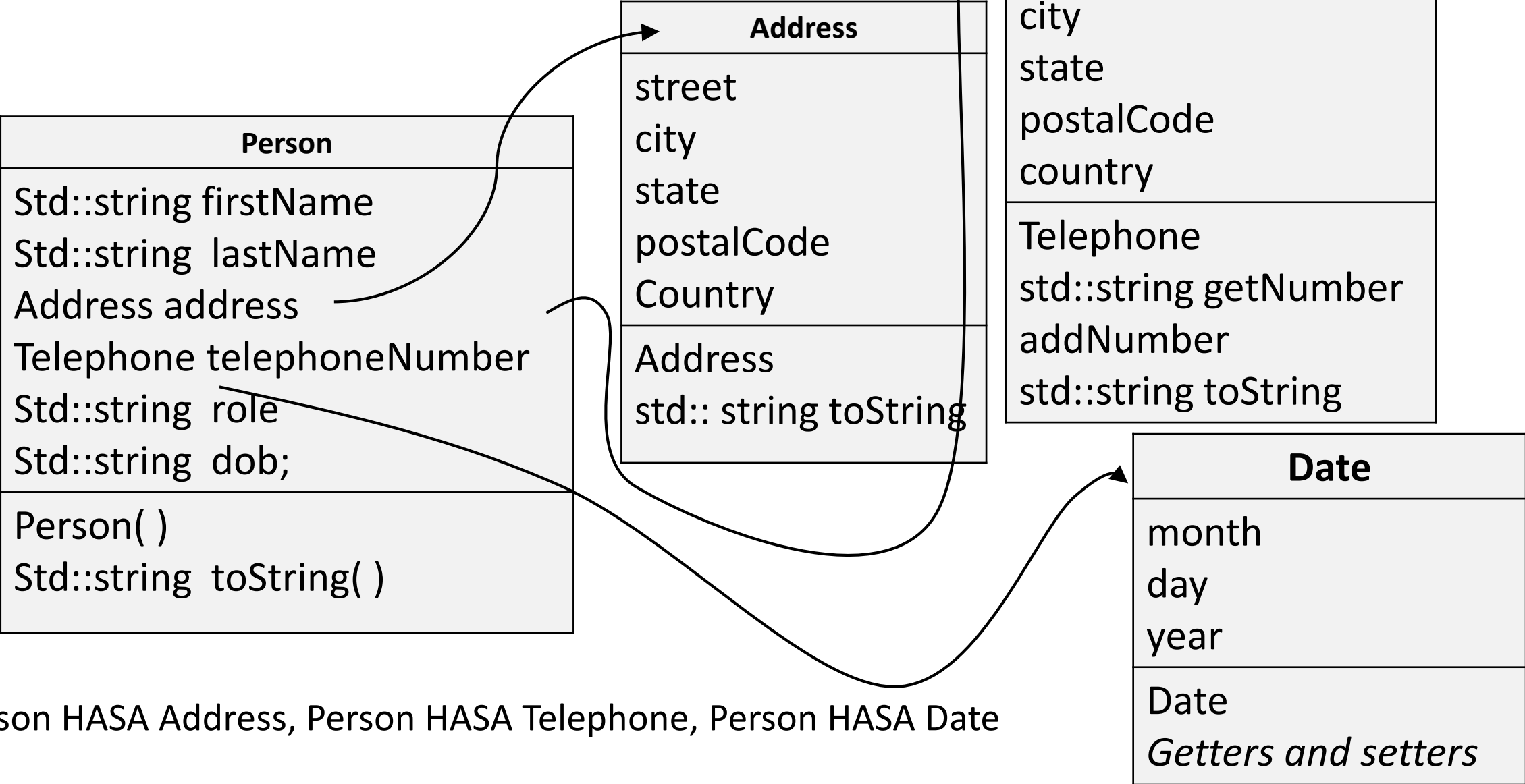
The HASA (Has A) Relationship

- HASA describes a class C using an object of another class H to hold info about class C
- If a class Person class has a date of birth, and uses a Date object to hold the date of birth, then Person HASA Date
- HASA allows a class such as Person to use the functionality of the other classes' objects (e.g., Date), through the object
- Let's look at an example of this.

Attributes of the Person class (See Example0 code)

```
class Person {  
  
private:  
    std::string firstName;  
    std::string lastName;  
    Address* address;  
    Telephone* telephoneNumber;  
    std::string role;  
    Date* dob;
```

UML for a person class



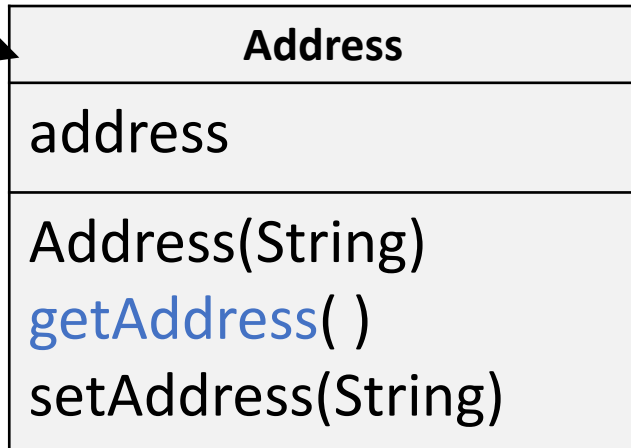
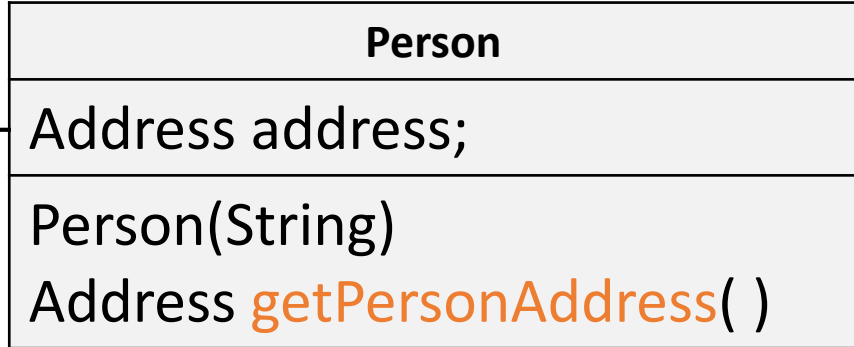
Person HASA Address, Person HASA Telephone, Person HASA Date

Let's look at a simplified Person and Address (See Example1 code)

```
class Person {  
private:  
    Address* address;  
  
public:  
    Person(Address* _address);  
    virtual Address* getPersonAddr( );  
};  
Person::Person(Address* _address) {  
    address = _address;  
}  
  
Address* Person::getPersonAddr( ) {  
    return address;  
}
```

```
class Address {  
private:  
    std::string address;  
public:  
    Address(std::string _address);  
    virtual std::string getAddress( );  
    virtual void setAddress(std::string _address);  
};  
  
Address::Address(std::string _address) {  
    address = _address;  
}  
  
std::string Address::getAddress( ) {  
    return address;  
}  
  
void Address::setAddress(std::string _address) {  
    address = _address;  
}
```

UML and some test code for these simplified classes



```
#include <iostream>
#include <string>
#include "Address.h"
#include "Person.h"
```

```
int main(int argc, char* argv[ ]) {
    Address* addr = new Address("the address");
    Person* person = new Person(addr);
    std::cout << person->getPersonAddr( )->getAddress( ) <<
    std::endl;
}
```

The bold code works, but it is a bad way to do it.

Why is this? It is because a class should be an *interface*, and Test shouldn't know the internals of how Person stores an Address. With this code Test has to know that there is an address object held by Person.

A better way to get the address (See Example2 code)

```
class Person {
private:
    Address* address;

public:
    Person(Address* _address);
    virtual std::string getAddress( );
};

Person::Person(Address* _address) {
    address = _address;
}

virtual Address* Person::getPersonAddr( )
{
    return address;
}

std::string Person::getAddress( ) {
    return address->getAddress( );
}
```

```
#include <iostream>
#include "Address.h"
#include "Person.h"
```

```
int main(int argc, char* argv[ ]) {
    Address* addr = new Address("the address");
    Person* person = new Person(addr);
    std::cout << person->getAddress( ) << std::endl;
}
```

- This is much better.
- Test now only knows that somehow a Person object can return an address. It doesn't know how it gets it, and whether or not Person has access to an Address object, unlike the previous slide.

The ISA relationship

- The ISA relationship is used when one class *IS A* class, and when objects of one class are also objects of another class.
- As an example, a real human student is also a real human person, and therefore it seems reasonable that if we have Person and Student objects, and Student shares Person properties, a Student ISA Person.
 - Person may have a date of birth, name and address
 - Student may also need to have these
 - A human student also has properties that are unique to a student, e.g., a GPA, classes being taken, etc., that are not part of a Person
- The ability for a class to be another class by *extending* or *deriving from* it is a fundamental OO concept

A simple Person and Student class (See Example3 code)

```
class Person {  
private:  
    std::string name;  
  
public:  
    Person(std::string _name);  
    virtual std::string getName( );  
};
```

```
Person::Person(std::string _name) {  
    name = _name;  
}
```

```
std::string Person::getName( ) {  
    return name;  
}
```

```
class Student : public Person {  
private:  
    float gpa;  
  
public:  
    Student(float _gpa, std::string name);  
    virtual float getGPA( );  
};
```

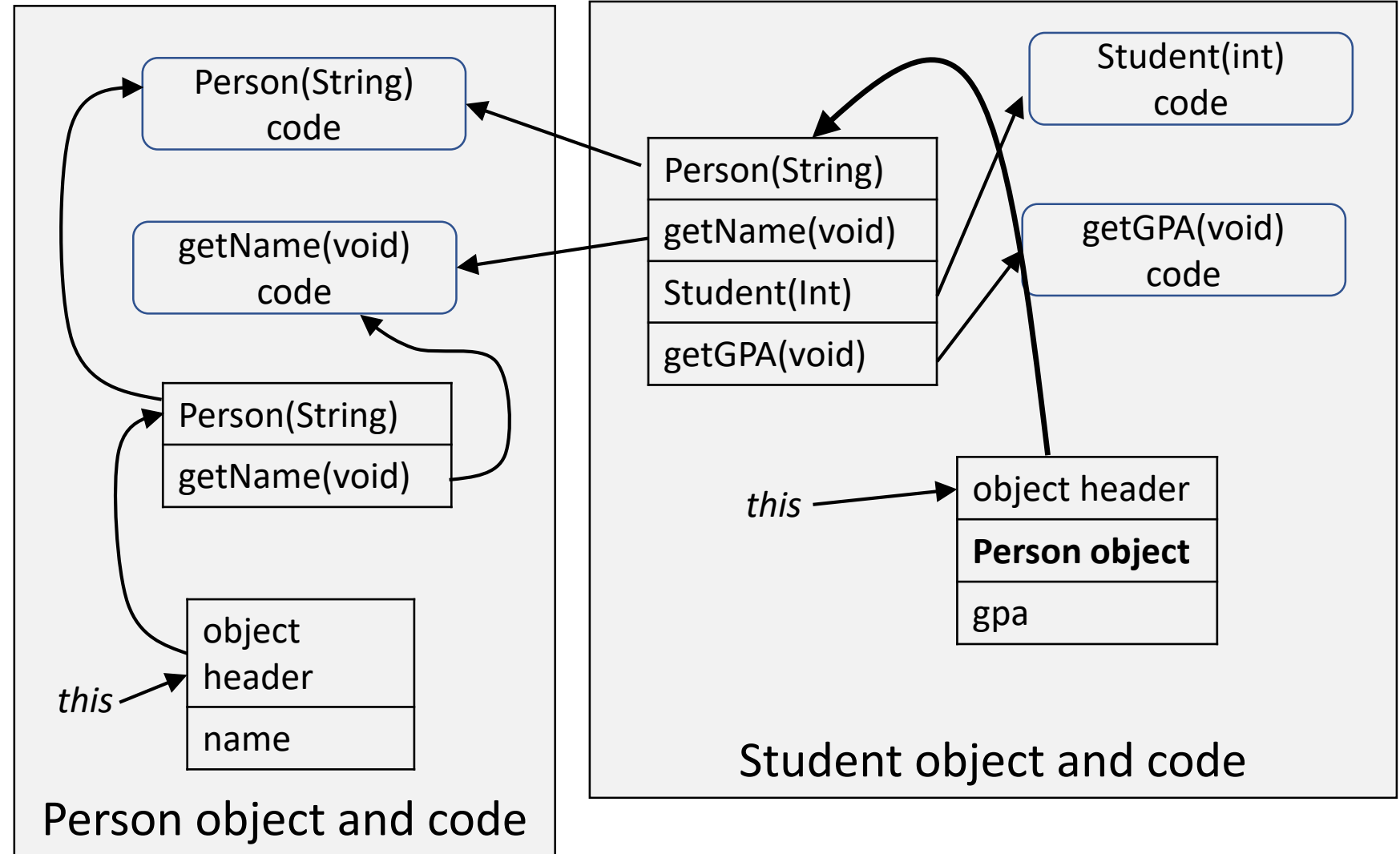
```
Student::Student(float _gpa, std::string name) : Person(name) {  
    gpa = _gpa;  
}
```

```
float Student::getGPA( ) {  
    return gpa;  
}
```

Student *ISA* Person

Person and Student object physical layout

- Because a Student ISA Person it has everything a Person has
 - It contains the attributes of a Person in addition to its own attributes
 - It can access the non-private attributes of the Person object it contains
 - It can access the non-private functions of a Person in addition to its own functions
- Because a Student object is a Person object, it can act like a Student object



Polymorphism

- Polymorphism is the property by which some type can act like more than one type
- That is, it has properties from more than one type

Polymorphism

- An object of type Student can act like a Person object
 - It acts like multiple things – Person and Student
 - Polymorphism describes this behavior
- When a Student object is acting like a Person, only attributes (variables) and functions present in a Person can be accessed.
- When a Student object acts like a Student, visible Student and Person attributes and functions can be accessed.
- The student object truly is both a Student and a Person.

Some terminology (1)

```
class Person {  
  
private:  
    std::string name;  
  
public:  
    Person(std::string _name);  
    virtual std::string getName( );  
};
```

Student ISA Person, i.e., student
is *derived from* Person or
Student ISA *subclass* of Person

Person is a *base* class of Student
or is the *superclass* of Student

```
class Student : public Person {  
  
private:  
    float gpa;  
  
public:  
    Student(float _gpa, std::string name);  
    virtual float getGPA( );  
};  
  
Student::Student(float _gpa, std::string name) : Person(name) {  
    gpa = _gpa;  
}  
  
float Student::getGPA( ) {  
    return gpa;  
}
```

Constructing derived objects (1)

```
class Person {  
  
private:  
    std::string name;  
  
public:  
    Person(std::string _name);  
    virtual std::string getName( );  
};
```

When constructing a Student object, its contained Person object must also be constructed. The first element in the Student *initializer list* should be a call to a Person constructor

```
class Student : public Person {  
  
private:  
    float gpa;  
  
public:  
    Student(float _gpa, std::string name);  
    virtual float getGPA( );  
};  
  
Student::Student(float _gpa, std::string name) : Person(name) {  
    gpa = _gpa;  
}  
  
float Student::getGPA( ) {  
    return gpa;  
}
```

Constructing derived objects (2) (See Example4 code)

```
class Person {  
  
private:  
    std::string name;  
  
public:  
    Person( );  
    virtual std::string getName( );  
};  
Person::Person( ) {  
    name = "Anne";  
}  
  
std::string Person::getName( ) {  
    return name;  
}
```

```
class Student : public Person {  
  
private:  
    float gpa;  
  
public:  
    Student(float _gpa);  
    virtual float getGPA( );  
};  
  
Student::Student(float _gpa) : Person() {  
    gpa = _gpa;  
}  
  
float Student::getGPA( ) {  
    return gpa;  
}
```

- If no base class constructor is called, C++ will insert a call to the zero arg base class constructor.
- If no zero arg base constructor exists, C++ will create one *if no other base constructor is defined*

Constructing derived objects (See Example5 code)

```
class Person {  
  
private:  
    std::string name;  
  
public:  
    Person( ); /* C++ inserts this */  
    virtual std::string getName( );  
};  
  
Person::Person( ) { } /* C++ inserts this */  
  
std::string Person::getName( ) {  
    return name;  
}
```

```
class Student : public Person {  
  
private:  
    float gpa;  
  
public:  
    Student(float _gpa);  
    virtual float getGPA( );  
};  
  
Student::Student(float _gpa) : Person( ) {  
    gpa = _gpa;  
}  
  
float Student::getGPA( ) {  
    return gpa;  
}
```

- C++ creates a base class zero arg constructor for a class that sets all uninitialized variables to their default values if no zero arg constructor is specified *and no other constructors are declared in the class*.
- C++ default initializations are to the machine 0.

C++ default values

- C++ default values are the machine zeros for each type
- 0 for pointers, 0 for integers, 0.0 for floats and doubles, etc.
- Because they are the machine zeros, it is possible to write values to a file on one machine and have them not be in the correct format on another machine
 - As the world has converged on X86 and little-endian is less of a problem that it used to be

Constructor call order (1) (See

Example6 code)

```
class Person {  
  
private:  
    std::string name;  
  
public:  
    Person(std::string _name);  
    virtual std::string getName( );  
};  
  
Person::Person(std::string _name) {  
    name = _name;  
    std::cout << "Person" << std::endl;  
}  
  
std::string Person::getName( ) {  
    return name;  
}
```

```
class Student : public Person {  
  
private:  
    float gpa;  
  
public:  
    Student(float _gpa, std::string name);  
    virtual float getGPA( );  
};  
  
Student::Student(float _gpa, std::string name) : Person(name) {  
    gpa = _gpa;  
    std::cout << "Student" << std::endl;  
}  
  
float Student::getGPA( ) {  
    return gpa;  
}
```

Output:

Person
Student


The base class constructor is called at the very start of the derived class constructor, it executes, and upon its return the rest of the derived class constructor executes

Constructor call order (2)

```
class Person {  
  
private:  
    std::string name;  
  
public:  
    Person( );  
    virtual std::string getName( );  
};  
  
Person::Person( ) {  
    name = "Anne";  
    std::cout << "Person" << std::endl;  
}  
  
std::string Person::getName( ) {  
    return name;  
}
```

```
class Student : public Person {  
  
private:  
    float gpa;  
  
public:  
    Student(float _gpa);  
    virtual float getGPA( );  
};  
  
Student::Student(float _gpa) {  
    gpa = _gpa;  
    std::cout << "Student" << std::endl;  
}  
  
float Student::getGPA( ) {  
    return gpa;  
}
```

Output:
Person
Student



Note there is no explicit call to Person() -- C++ inserts a call to the zero arg constructor

Polymorphism and virtual function calls

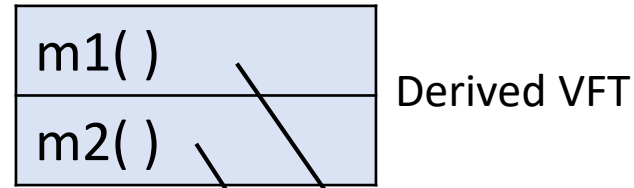
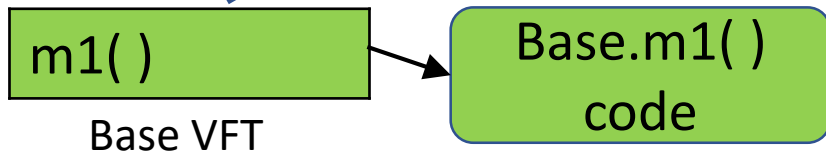
- With polymorphism, Objects can act like what their derived class says, or they can act like base objects
- Let's look at the case where a function is defined in the base class and the derived class.
 - Which function is called when an object is acting like a base object?
 - Which function is called when an object is acting like a derived object?
 - What mechanisms are used to decide what function is called?
- We'll first look at the simple case where everything is public, and then go to the slightly more complicated case when base functions are private.

Two simple, contrived classes – physical layout (See Example8 code)

```
class Base {  
public:  
    virtual void m1( );  
};
```

```
void Base::m1( ) {  
    std::cout << "Base.m1" << std::endl;  
}
```

These tables of pointers
to the code for functions
are *Virtual Function
Tables*



Derived.m1()
code

Derived.m2()
code

main(. . .)
code

```
class Derived : public Base {  
public:  
    void m1( ); // why no virtual?  
    virtual void m2( );  
};
```

```
void Derived::m1( ) {  
    std::cout << "Derived.m1" << std::endl;  
}
```

```
void Derived::m2( ) {  
    std::cout << "Derived.m2" << std::endl;  
}
```

Derived m1() is said to *override* Base's m1().

Behavior of the classes when they execute

```
class Base {  
public:  
    virtual void m1( );  
};
```

```
void Base::m1( ) {  
    std::cout << "Base.m1" << std::endl;  
}
```

```
class Derived : public Base {  
public:  
    void m1( );  
    virtual void m2( );  
};
```

```
void Derived::m1( ) {  
    std::cout << "Derived.m1" << std::endl;  
}
```

```
void Derived::m2( ) {  
    std::cout << "Derived.m2" << std::endl;  
}
```

Output from running the program (with the
ERROR line commented out)

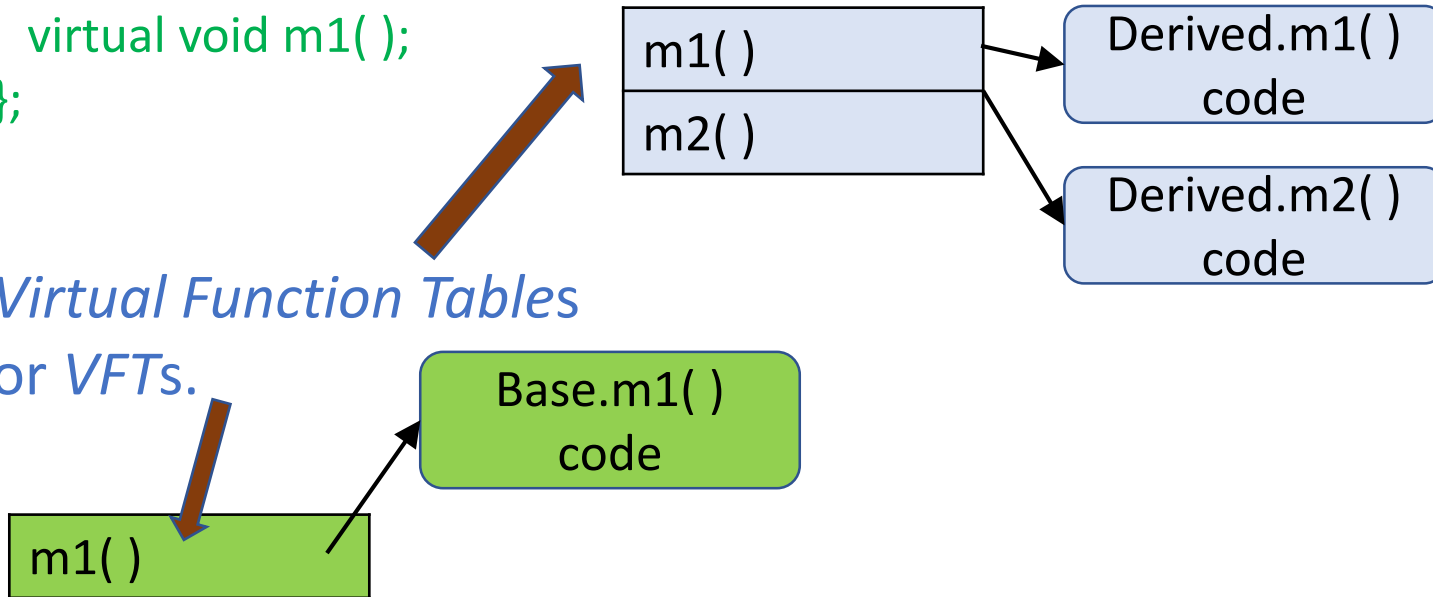
Derived.m1
Derived.m2(changed)
Derived.m1

```
int main(int argc, char* argv[ ]) {  
    Derived* d = new Derived( );  
    d->m1( ); // Derived->m1  
    d->m2( ); // Derived->m2  
    Base* b = d;  
    b->m1( ); // Derived->m1  
// b->m2( ); error, remove the commenting and compile to see the error.  
}
```

The call to d.m1()

```
class Base {  
public:  
    virtual void m1( );  
};
```

*Virtual Function Tables
or VFTs.*



```
class Derived : public Base {  
public:  
    void m1( );  
    virtual void m2( );  
};
```

```
void Derived::m1( ) {  
    std::cout << "Derived.m1" << std::endl;  
}
```

```
void Derived::m2( ) {  
    std::cout << "Derived.m2" << std::endl;  
}
```

```
int main(int argc, char* argv[ ]) {  
    Derived* d = new Derived( );  
    d->m1( ); // Derived->m1  
    ...  
}
```

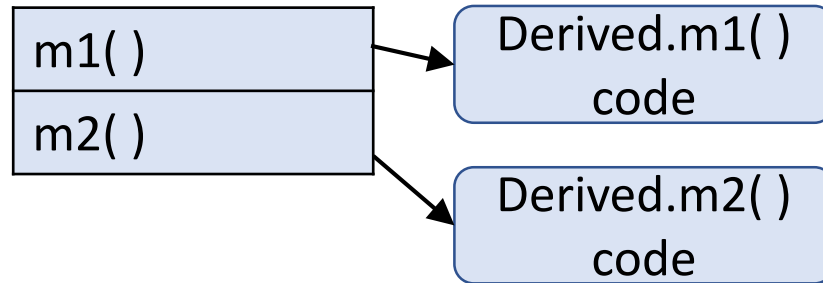
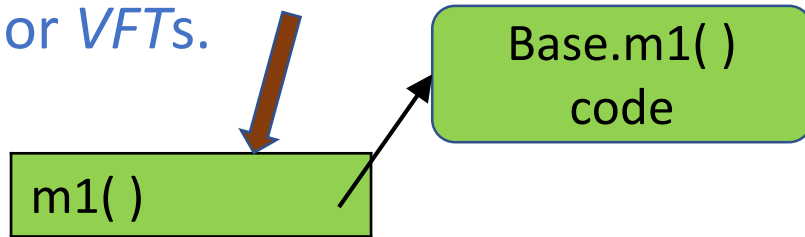
At compile time, when g++ sees the call to d.m1(), it:

1. Looks in the class Derived to see if there is a function m1 that is visible there
2. There is, so g++ determines m1's position *P* in the *VFT for the class that is the type of the pointer* (a Derived pointer, therefore Derived's VFT)
3. g++ then generates code to call the function at position *P* in Derived's VFT. At runtime, Derived.m1() is called

The call to d.m2()

```
class Base {  
public:  
    virtual void m1( );  
};
```

*Virtual Function Tables
or VFTs.*



```
class Derived : public Base {  
public:  
    void m1( );  
    void m2( );  
};  
void Derived::m1( ) {  
    std::cout << "Derived.m1" << std::endl;  
}  
void Derived::m2( ) {  
    std::cout << "Derived.m2" << std::endl;  
}  
int main(int argc, char* argv[ ]) {  
    Derived* d = new Derived( );  
    ...  
    d->m2( );  
}
```

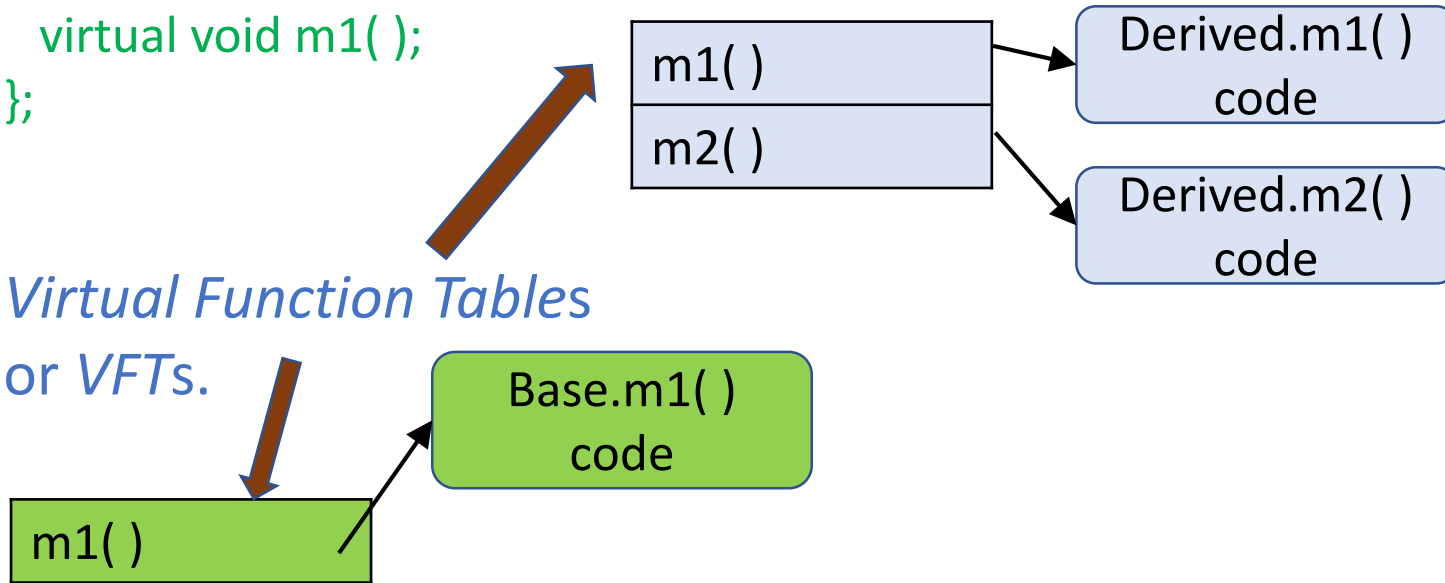
At compile time, when g++ sees the call to d.m2(), it:

1. Looks in the class Derived to see if there is a function **m2** that is visible there
2. There is, so g++ determines m2's position *P* in the VFT *for the class that is the type of the pointer, therefore Derived's VFT*
3. It then generates code to call the function at position *P* in Derived's VFT. At runtime, Derived.m2() is called

The call to b.m1()

```
class Base {  
public:  
    virtual void m1( );  
};
```

*Virtual Function Tables
or VFTs.*



```
class Derived : public Base {  
public:  
    void m1( );  
    void m2( );  
};  
void Derived::m1( ) {  
    std::cout << "Derived.m1" << std::endl;  
}  
void Derived::m2( ) {  
    std::cout << "Derived.m2" << std::endl;  
}  
int main(int argc, char* argv[ ]) {  
    Derived* d = new Derived( );  
    ...  
    Base* b = d;  
    b->m1( ); // Derived->m1
```

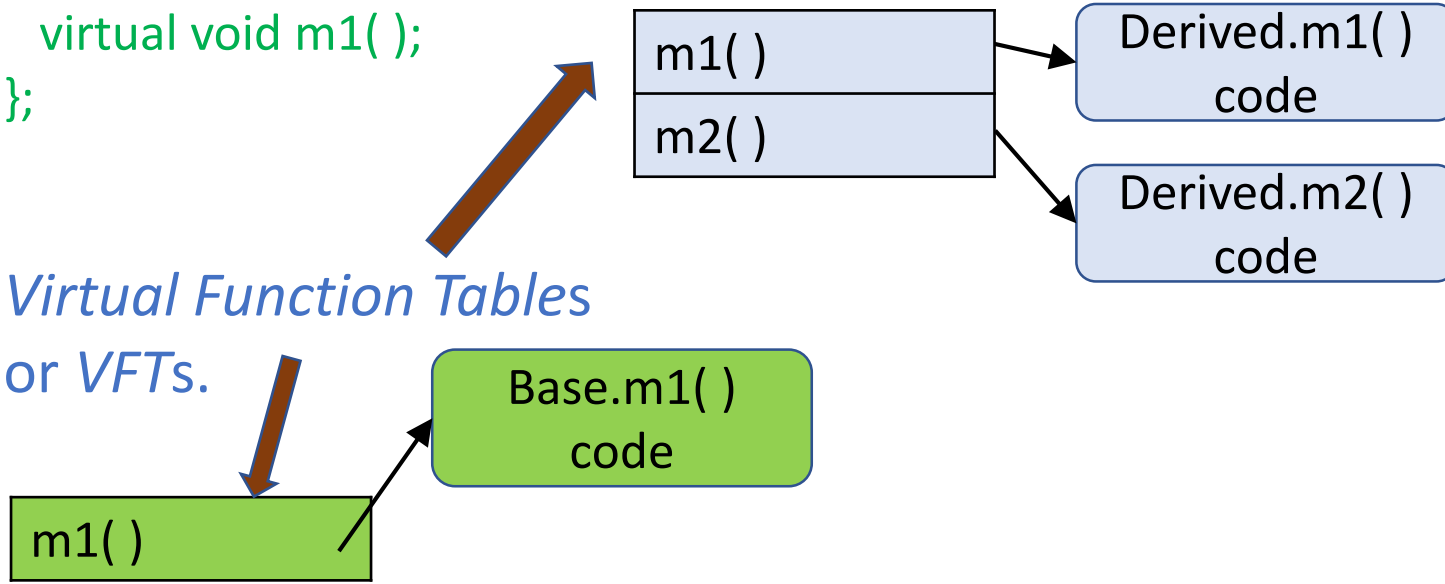
At compile time, when g++ sees the call to b->m1(), it:

1. Looks in the class Base to see if there is a function m1 that is visible there
2. There is, so g++ determines m2's position *P* in the VFT *for the class that is the type of the pointer* (a Base pointer, therefore Base's VFT)
3. It then generates code to call the function at position *P* in VFT of the object actually pointed to – a **Derived** VFT. At runtime, **Derived.m1()** is called

The attempt to call to b.m2()

```
class Base {  
public:  
    virtual void m1( );  
};
```

*Virtual Function Tables
or VFTs.*



At compile time, when g++ sees the call to b->m2(), it:

1. Looks in the class Base (the class that is the type of the pointer) to see if there is a function m2 that is visible
2. There is not, so an error is issued

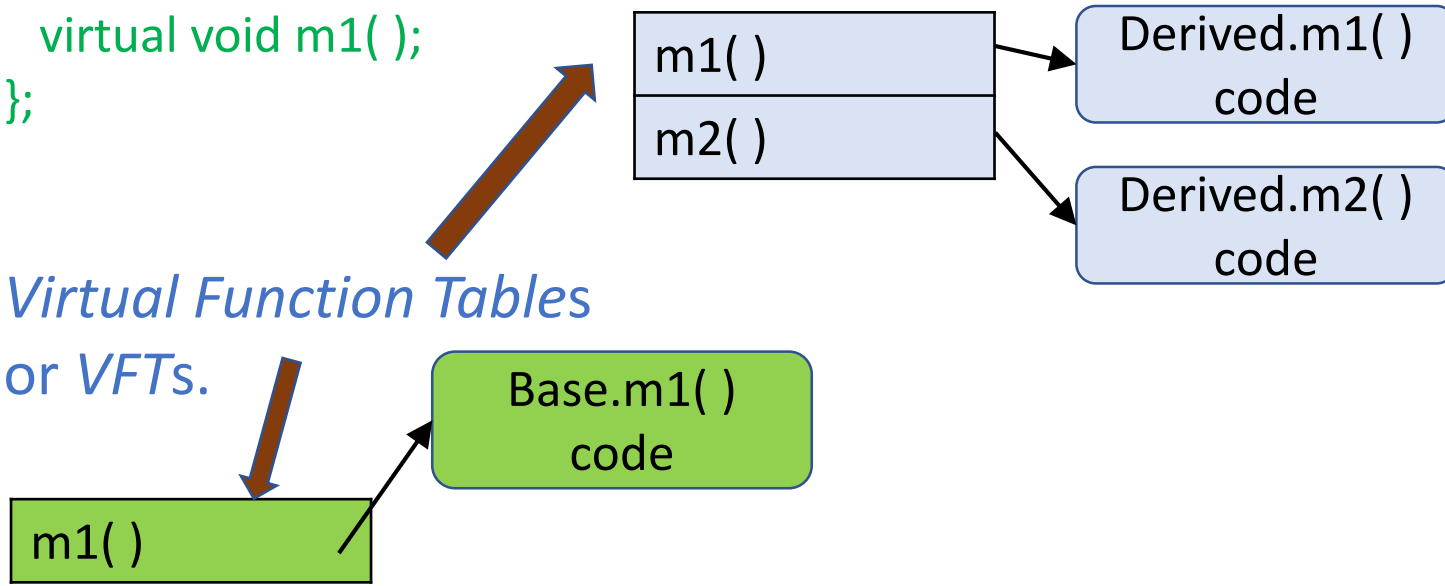
Note that at compile time a compiler is not able, in general, to know what object, and what kind of object will be referenced at runtime. Therefore, it cannot tell if the referenced object has an m2 or not. Therefore, if an object is accessed through a base pointer only base functions can be called

```
class Derived : public Base {  
public:  
    void m1( );  
    void m2( );  
};  
void Derived::m1( ) {  
    std::cout << "Derived.m1" << std::endl;  
}  
void Derived::m2( ) {  
    std::cout << "Derived.m2" << std::endl;  
}  
int main(int argc, char* argv[ ]) {  
    Derived* d = new Derived( );  
    ...  
    Base* b = d;  
    b->m2( );  
}
```

The attempt to call to b.m2()

```
class Base {  
public:  
    virtual void m1( );  
};
```

Virtual Function Tables
or VFTs.



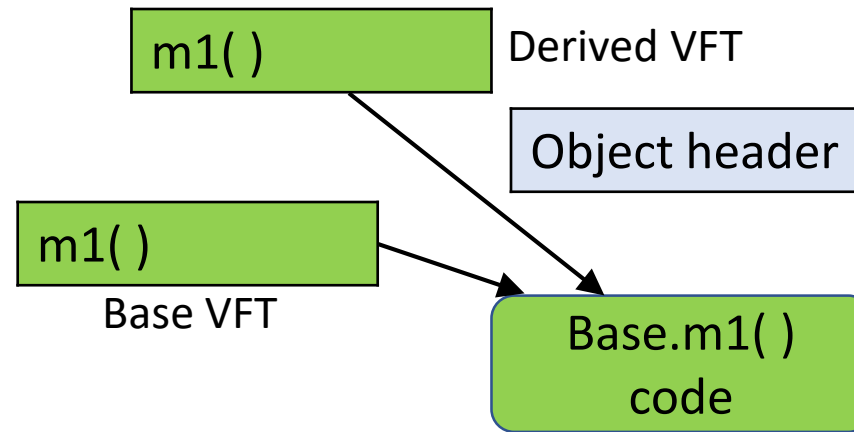
Note that at *compile time* a compiler is not able, in general, to know what object, and what kind of object will be referenced at runtime. Therefore, it cannot tell if the referenced object has an m2 or not. Therefore, if an object is accessed through a base pointer only base functions can be called. This ensures there is a valid VFT entry to call the function.

```
class Derived : public Base {  
public:  
    void m1( );  
    void m2( );  
};  
void Derived::m1( ) {  
    std::cout << "Derived.m1" << std::endl;  
}  
void Derived::m2( ) {  
    std::cout << "Derived.m2" << std::endl;  
}  
int main(int argc, char* argv[ ]) {  
    Derived* d = new Derived( );  
    ...  
    Base* b = new Base( );  
    if (rand( ) > 0.5) {  
        b = d;  
    }  
    b->m2( );  
}
```

When only the base class defines a function (Example 9 code)

```
class Base {  
  
public:  
    virtual void m1( );  
};
```

```
void Base::m1( ) {  
    std::cout << "Base.m1" << std::endl;  
}
```



```
class Derived : public Base { };
```

```
int main(int argc, char* argv[ ]) {  
    Derived* d = new Derived( );  
    Base* b = d;  
    b->m1( );  
    d->m1( );  
}
```

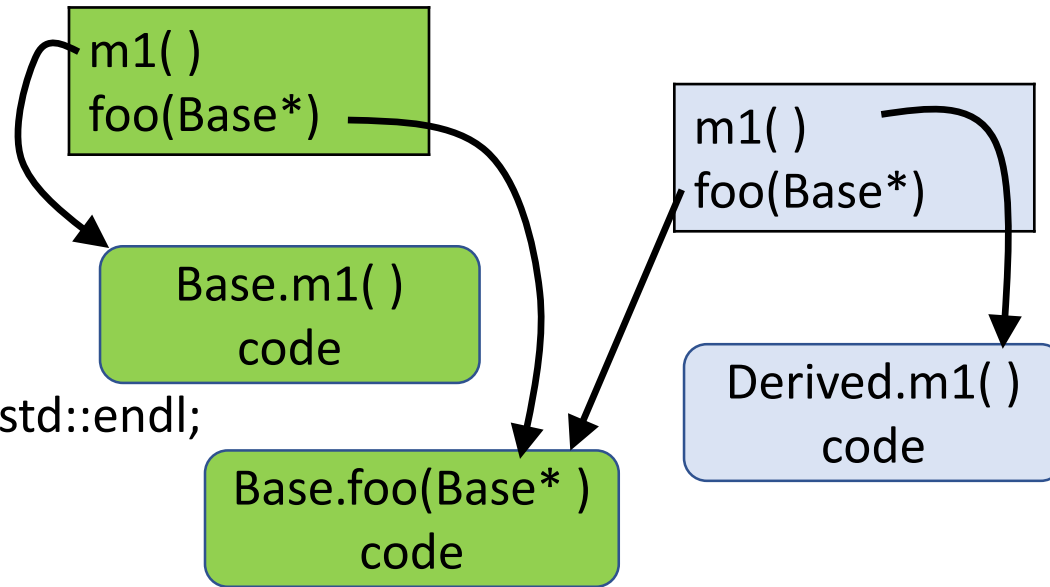
The C++ compiler sees the call to `d->m1()`. The Base `m1` is visible in `Derived`, and the function to be called at runtime is the one in the `m1()` position of `Derived's` VFT, i.e., Base's `m1`.

The C++ compiler sees the call to `b->m1()`. `m1` is defined in `Base`, and the function to be called at runtime is the one in the `m1()` position of **Derived's** VFT, i.e., Base's `m1`.

Private virtual functions (See Example9 code)

```
class Base {  
private:  
    virtual void m1( );  
public:  
    virtual void foo(Base* b);  
};  
void Base::m1( ) {  
    std::cout << "Base.m1" << std::endl;  
}
```

```
void Base::foo(Base* b) {  
    b->m1( );  
}
```



```
class Derived : public Base {  
private:  
    void m1( );  
};  
void Derived::m1( ) {  
    std::cout << "Derived.m1" <<  
        std::endl;  
}
```

```
int main(int argc, char* argv[ ]) {  
    Derived* d = new Derived( );  
    Base* b = d;  
    // b->m1( ); ERROR  
    // d->m1( ); ERROR  
    b->foo(b); // Derived->m1  
    b->foo(d); // Derived->m1  
}
```

When private virtual functions are called, they work just like non-private functions – we find the type of the pointer, check to see if the function is visible in the corresponding class, if it is, find the functions position in the VFT and then call through the VFT of the pointed to object.

What if a function is not virtual? (1) (See

Example10 code)

```
class Base {
public:
    void m1( );
};

void Base::m1( ) {
    std::cout << "Base.m1" << std::endl;
}
```

Output:

Derived.m1
Derived.m2

- For S1, the C++ compiler sees the pointer is of type Derived
 - It goes to the Derived class, and finds Derived::m1() which is not virtual
 - It generates code to call Derived::m1 directly()
- For S2, the C++ compiler sees the pointer is of type Derived
 - It goes to the Derived class, and finds Derived::m2() which is not virtual
 - It generates code to call Derived::m1 directly(), i.e., NOT through the VFT

```
class Derived : public Base {
public:
    void m1( );
    void m2( );
};

void Derived::m1( ) {
    std::cout << "Derived.m1" << std::endl;
}

void Derived::m2( ) {
    std::cout << "Derived.m2" << std::endl;
}

int main(int argc, char* argv[ ]) {
    Derived* d = new Derived( );
    d->m1( ); // S1 Derived->m1
    d->m2( ); // S2 Derived->m2
    Base* b = d;
    b->m1( ); // S3 Base->m1
    // b->m2( ); S4 error
}
```

What if a function is not virtual?

```
class Base {
public:
    void m1( );
};

void Base::m1( ) {
    std::cout << "Base.m1" << std::endl;
}
```

Output:

Derived.m1
Derived.m2
Base.m1

- For S3, a Base pointer b points to the Derived object. The compiler goes to Base, finds an m1, it is not virtual and the base m1 is called directly
- For S4, a Base pointer b points to the Derived object. The compiler goes to Base, does not find an m2, and an error is given

```
class Derived : public Base {

public:
    void m1( );
    void m2( );
};

void Derived::m1( ) {
    std::cout << "Derived.m1" << std::endl;
}

void Derived::m2( ) {
    std::cout << "Derived.m2" << std::endl;
}

int main(int argc, char* argv[ ]) {
    Derived* d = new Derived( );
    d->m1( ); // S1 Derived->m1
    d->m2( ); // S2 Derived->m2
    Base* b = d;
    b->m1( ); // S3 Base->m1
    // b->m2( ); S4 error
}
```


What if a function is not virtual? (2) (see Example11 code)

```
class Base {  
private:  
    void m1( );
```

```
public:  
    void foo(Base* b);  
};
```

```
void Base::m1( ) {  
    std::cout << "Base.m1" << std::endl;  
}
```

```
void Base::foo(Base* b) {  
    b->m1( );  
}
```

```
class Derived : public Base {  
public:  
    void m1( );  
};  
  
void Derived::m1( ) {  
    std::cout << "Derived.m1" << std::endl;  
}
```

```
int main(int argc, char* argv[ ]) {  
    Derived* d = new Derived( );  
    Base* b = d;  
    // b->m1( ); S1 ERROR  
    d->m1( ); S2 Derived m1( )  
    b->foo(b); // S3 Base foo and m1  
    b->foo(d); // S4 Base foo and m1  
}
```

Output:

Derived.m1

Base.m1

- For S1, a Base pointer b is used to call the function, so the compiler looks into class Base, sees that m1 is private, and issues an error
- For S2, a Derived pointer is used, so the compiler looks into class Derived, finds a non-virtual *public* m1, and calls it directly – it does not go through the VFT.

What if a function is not virtual? (3)

```
class Base {
private:
    void m1( );

public:
    void foo(Base* b);
};

void Base::m1( ) {
    std::cout << "Base.m1" << std::endl;
}

void Base::foo(Base* b) {
    b->m1( );
}
```

Output:

Derived.m1
Base.m1

```
class Derived : public Base {
public:
    void m1( );
};

void Derived::m1( ) {
    std::cout << "Derived.m1" << std::endl;
}

int main(int argc, char* argv[ ]) {
    Derived* d = new Derived( );
    Base* b = d;
    // b->m1( ); S1 ERROR
    d->m1( ); S2 Derived m1( )
    b->foo(b); // S3 Base foo and m1
    b->foo(d); // S4 Base foo and m1
}
```

- For S3, a Base pointer b is used to call the foo, and is passed as the argument
 - The compiler looks into Base, sees a matching foo, and generates code to call foo with b
 - In foo, the pointer b used to call m1 is of type Base, so the compiler looks into the Base class, sees a matching m1 that is non-virtual, and calls it directly – *even though the pointer b points to a Derived object, and Derived defines an m1!*

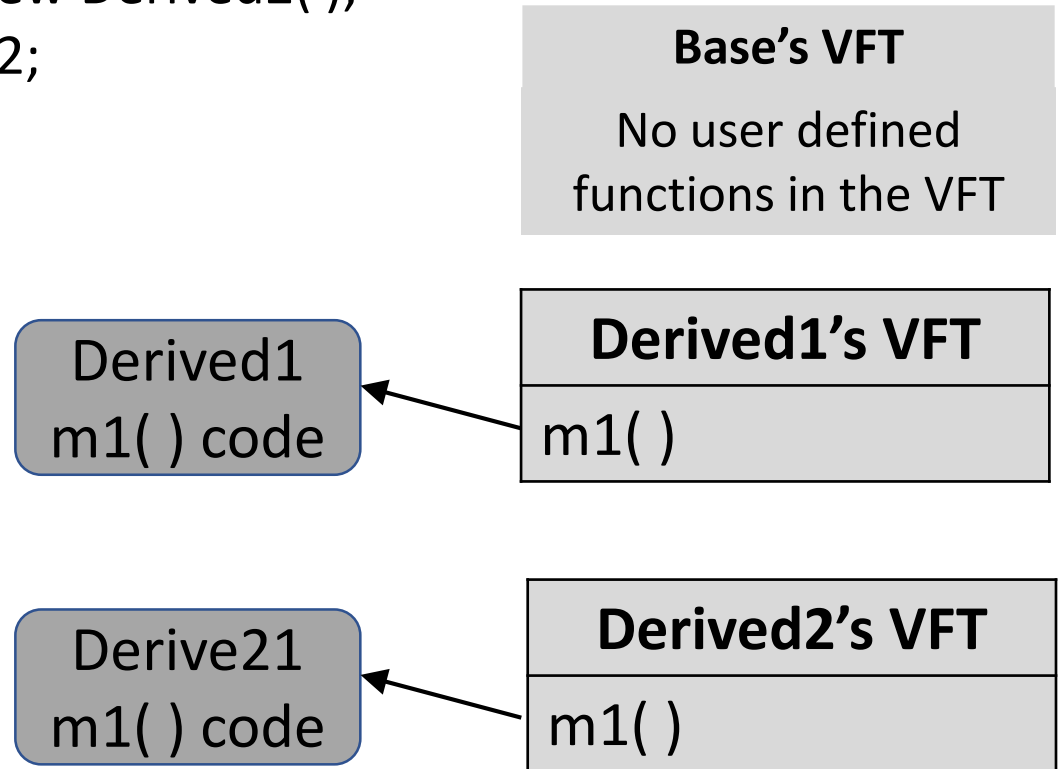
What if the function is virtual only in Derived classes (this is a terrible thing to do) (See Example13 code)

```
class Base {  
public:  
    void m1( );  
};
```

```
class Derived1 : public Base {  
public:  
    virtual void m1( );  
};
```

```
class Derived2 : public Derived1 {  
public:  
    virtual void m1( );  
};
```

```
int main(int argc, char* argv[ ]) {  
    Derived2* d2 = new Derived2( );  
    Derived1* d1 = d2;  
    Base* b = d2;  
    b->m1( ); // S1  
    d1->m1( ); // S2  
}
```



It's a terrible thing to do because it makes it hard to determine what is called!

```
class Base {  
public:  
    void m1( );  
};
```

```
class Derived1 : public Base {  
public:  
    virtual void m1( );  
};
```

```
class Derived2 : public Derived1 {  
public:  
    virtual void m1( );  
};
```

```
int main(int argc, char* argv[ ]) {  
    Derived2* d2 = new Derived2( );  
    Derived1* d1 = d2;  
    Base* b = d2;  
    b->m1( ); // S1  
    d1->m1( ); // S2  
}
```

Base.m1
Derived2.m1

- When calling through a Base pointer, the matched m1 is not virtual
 - Base's m1 is called directly
- When calling through a Derived1 pointer, the matched m1 is virtual
 - Call through the pointed to object's virtual function table
 - Derived2's m1 is called

Upcasts and downcasts with pointers to objects (See Example14 code)

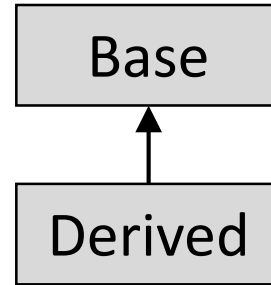
```
class Base {  
public:  
    int iv;  
    Base(int);  
};
```

```
Base::Base(int i) : iv(i) { }
```

```
Class Derived : public Base {  
public:  
    int jv;  
    Derived(int, int);  
};
```

```
Derived::Derived(int i, int j) : Base(i), jv(j) { }
```

```
int main(int argc, char* argv[ ]) {  
    Base* b = new Base(5);  
    Derived* d = new Derived(4, 5);  
    b = d; // OK  
    d = b; // Compile time ERROR  
    d = (Derived*) b; // Allowed, but  
                      // be careful  
}
```



- `b = d` is an upcast because it goes up the inheritance chain.
 - Always legal
 - No explicit cast needed
- Derived ISA Base is what makes this legal

- `D = b` is a downcast because it goes down the inheritance chain
 - Allowed with an explicit cast
 - Legal only if `b` points to a Derived object or something that ISA Derived object, i.e., inherits from Derived

Object attributes and inheritance (See Example15 code)

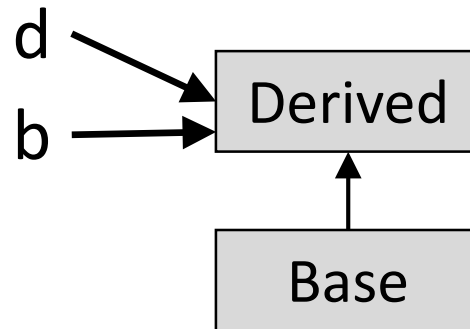
```
class Base {  
public:  
    int iv;  
    Base( );  
};
```

```
Base::Base( ) {iv = 0;}
```

```
Class Derived : public Base {  
public:  
    int iv;  
    Derived( );  
};
```

```
Derived::Derived( ) {iv = 1;}
```

```
int main(int argc, char* argv[ ]) {  
    Derived* d = new Derived( );  
    Base* b = d;  
    std::cout << "b->iv: "+b->iv << std::endl;  
    std::cout << "d->iv: "+d->iv << std::endl;  
}
```



- When accessing object attributes, the type of the pointer says what attribute will be used.
- For `d->iv`, the pointer type is `Derived`, so `Derived's iv` is used.
- For `b->iv`, the pointer type is `Base`, and so `Base's iv` is printed, even though `b` points to a `Derived` object
 - This is done for efficiency – you don't want every variable access to have one or more indirections.

What we learned in this set of slides

- What are HASA and ISA relations?
 - HASA is the *has a* relationship. Class A HASA class B if A has an attribute that is a reference to a B object
 - ISA is the *is a* relationship. Class A ISA class B if A extends (inherits from, or derives from) class B.
- Polymorphism, Inheritance, base and derived classes, subclasses and superclasses, function overriding
 - Polymorphism is the property that an object can act like an object of its class, or classes its class derives from.
 - Inheritance is the act of a class extending and specializing the behavior of a base class.
 - A base or superclass is the class that is extended or inherited from
 - A derived or subclass is the class that extends or derives from a base or superclass
 - A base class function is overridden when a derived class function is defined that has the same name and arguments as the base class function

What we learned in this set of slides

- Virtual function table

- A virtual function table (VFT) for a class is a table of functions that can be called by objects of that class. In C++, only functions declared virtual, or that override a function declared as virtual, are called using the VFT.
- static functions, which we will discuss soon, are an exception to this.

- Virtual function calls

The virtual function call `d.m(args)`, where `d` is a `D` pointer, and points to an `E` object, is made through the VFT as follows (this will get added to in later lectures):

1. Examine class `D`. If a function that matches `m`
2. `(args)` is visible in `D` then
 - I. Find the position of `m(args)` in `D`'s virtual function table.
 - II. Generate code to call `m(args)` through the virtual function table of `E` object `d` references
 - III. At runtime, call `m(args)` through the `E` VFT.

A simple way to figure out what virtual function is being called

- Virtual function calls

The virtual function call $d \rightarrow m(\text{args})$, where d is a D pointer, and points to an E object
Examine class D .

If a function that matches $m(\text{args})$ is found in D

1. Find the $m(\text{args})$ that is visible in E and call it through the VFT of the object pointed to.

else give a compile time error that the function $m(\text{args})$ is not found