SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220328<A|D>

Replace <A|D> with this section's letter

Inheritance II

CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

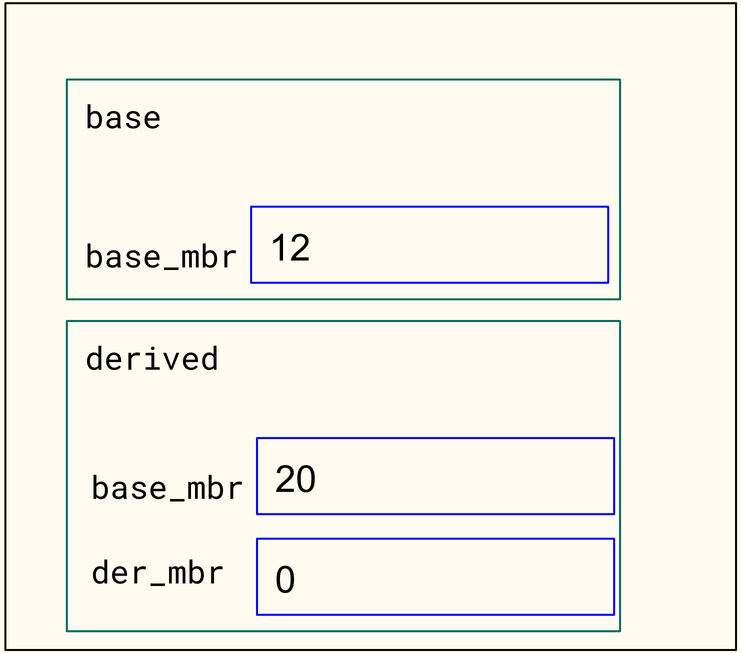
Agenda

- The slicing problem
- Polymorphism
- Constructors
- Protected members
- Interfaces

The slicing problem

Slicing

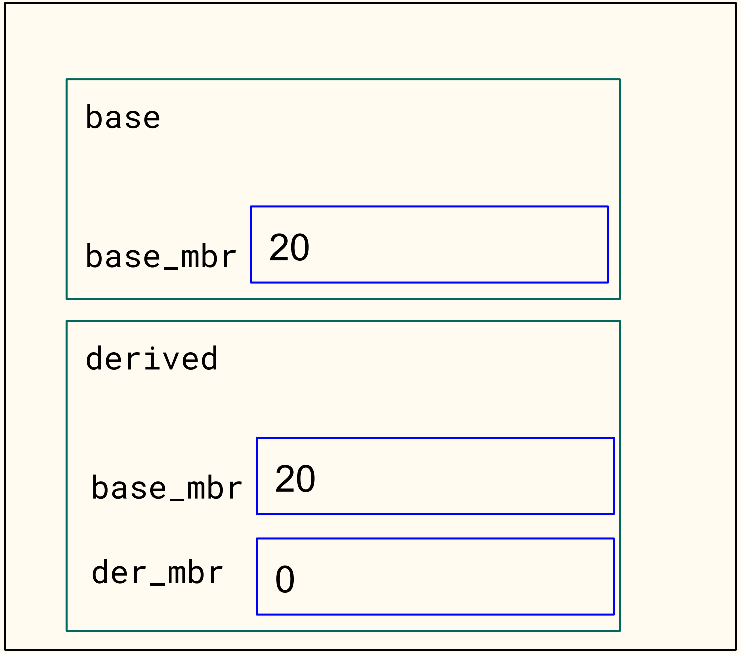
```
class Base {
private:
    int base_mbr;
};

class Derived: public Base {
private:
    int der_mbr;
};

int main() {
    Base base;
    Derived derived;
    ...
    base = derived;
    ...
}
```



Slicing

```
class Base {
private:
    int base_mbr;
};

class Derived: public Base {
private:
    int der_mbr;
};

int main() {
    Base base;
    Derived derived;

    base = derived;
    ...
}
```

Base class instance
has no der_mbr



base

base_mbr 20

derived

base_mbr 20

der_mbr 0

Inheritance assignment rules

- derived class instance to base class instance ✔

```
class Base {
private:
    int base_mbr;
};

class Derived: public Base {
private:
    int der_mbr;
};
```

```
int main() {
    Derived der;
    Base base;

    base = der;
}
```

slicing

Think Principle of Substitutability

- `Derived` is a `Base`

Inheritance assignment rules

- derived class instance to base class instance ✔
- base class instance to derived class instance ✘

```
class Base {
private:
    int base_mbr;
};

class Derived: public Base {
private:
    int der_mbr;
};
```

```
int main() {
    Derived der;
    Base base;

    der = base;
}
```

compilation error!

Think Principle of Substitutability

- `Base` is not a `Derived`

Inheritance assignment rules

- derived class instance to base class instance ✔
- base class instance to derived class instance ✘
- address of derived class instance to base class pointer ✔

```
class Base {
private:
    int base_mbr;
};

class Derived: public Base {
private:
    int der_mbr;
};
```

```
int main() {
    Derived der;
    Base base;

    Base* bp;
    Derived* dp;

    bp = &der;
}
```

Think Principle of Substitutability

- `Derived` is a `Base`

Inheritance assignment rules

- derived class instance to base class instance ✔
- base class instance to derived class instance ✘
- address of derived class instance to base class pointer ✔
- address of base class instance to derived class pointer ✘

```
class Base {
private:
    int base_mbr;
};

class Derived: public Base {
private:
    int der_mbr;
};
```

```
int main() {
    Derived der;
    Base base;

    Base* bp;
    Derived* dp;

    dp = &base;
}
```

compilation error!

Think Principle of Substitutability

- `Base` is not a `Derived`

Inheritance assignment rules (elaboration)

```
class Animal {
public:
    void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
    void purr() { cout << "purrr\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

```
int main() {
    Bear yogi;
    Lion leo;

    Animal* an_ptr = &leo; ✔

    an_ptr->purr(); ✘
}
```

cannot invoke derived class method via base pointer

Inheritance assignment rules (elaboration)

```
class Animal {
public:
    void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
    void purr() { cout << "purrr\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

```
int main() {
    Bear yogi;
    Lion leo;

    Animal* an_ptr = &leo;  ✔

    an_ptr = &yogi;  ✔

    an_ptr->purr();  ✘  would allow bear to purr
}
```

Inheritance assignment rules (elaboration)

```
class Animal {
public:
    void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
    void purr() { cout << "purrr\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

```
int main() {
    Bear yogi;
    Lion leo;

    Animal* an_ptr = &leo;  ✔

    an_ptr = &yogi;  ✔

    Lion* lion_ptr = nullptr;

    lion_ptr = an_ptr;  ✘

    lion_ptr->purr();
}
```

assigning base pointer to derived class pointer would allow bear to purr

Inheritance assignment rules

- derived class instance to base class instance ✔
- base class instance to derived class instance ✘
- address of derived class instance to base class pointer ✔
- address of base class instance to derived class pointer ✘

All are compile time considerations

(i.e. errors occur during compilation)

Polymorphism

Polymorphism

Principle of Substitutability

- Derived class instance can be used in place of a base class instance

```
class Animal {};

class Lion : public Animal {};

class Tiger : public Animal {};

class Bear : public Animal {};
```

any where an Animal is expected, a Lion, Tiger, or Bear can be provided at runtime

Declared type vs actual type

- Compiler evaluates code based on declared type
- Actual instance provided at **runtime** can be of a derived type

Slicing (again)

```
class Animal {
public:
    void eat() { cout << "Animal eating\n"; }
};
class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};
class Tiger : public Animal {
public:
    void eat() { cout << "Tiger eating\n"; }
};
class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

```
int main() {
    Lion fred;
    Tiger tigger;
    Bear pooh;

    vector<Animal> animals;

    animals.push_back(fred);
    animals.push_back(tigger);
    animals.push_back(pooh);

    for (size_t i = 0; i < animals.size(); ++i) {
        animals[i].eat();
    }
}
```

observed

```
Animal eating
Animal eating
Animal eating
```

wanted

```
Lion eating
Tiger  eating
Bear  eating
```

Slicing (again)

```
class Animal {
public:
    void eat() { cout << "Animal eating\n"; }
};
class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};
class Tiger : public Animal {
public:
    void eat() { cout << "Tiger eating\n"; }
};
class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

```
int main() {
    Lion fred;
    Tiger tigger;
    Bear pooh;

    vector<Animal> animals;

    animals.push_back(fred);
    animals.push_back(tigger);
    animals.push_back(pooh);

    for (size_t i = 0; i < animals.size(); ++i) {
        animals[i].eat();
    }
}
```

let's try pointers!

observed

```
Animal eating
Animal eating
Animal eating
```

wanted

```
Lion eating
Tiger  eating
Bear  eating
```

Slicing (again)

```
class Animal {
public:
    void eat() { cout << "Animal eating\n"; }
};
class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};
class Tiger : public Animal {
public:
    void eat() { cout << "Tiger eating\n"; }
};
class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

```
int main() {
    Lion fred;
    Tiger tigger;
    Bear pooh;

    vector<Animal*> animals;

    animals.push_back(&fred);
    animals.push_back(&tigger);
    animals.push_back(&pooh);

    for (size_t i = 0; i < animals.size(); ++i) {
        animals[i]->eat();
    }
}
```

observed

```
Animal eating
Animal eating
Animal eating
```

wanted

```
Lion eating
Tiger  eating
Bear  eating
```

Dynamic binding using the `virtual` keyword

```
class Animal {
public:
    void eat() {
        cout << "Animal eating\n";
    }
};
class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};
class Tiger : public Animal {
public:
    void eat() { cout << "Tiger eating\n"; }
};
class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

```
int main() {
    Lion fred;
    Tiger tigger;
    Bear pooh;

    vector<Animal*> animals;

    animals.push_back(&fred);
    animals.push_back(&tigger);
    animals.push_back(&pooh);

    for (size_t i = 0; i < animals.size(); ++i) {
        animals[i]->eat();
    }
}
```

Dynamic binding using the `virtual` keyword

method can be redefined in subclass
and bound to object at **runtime**

```cpp
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};
class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};
class Tiger : public Animal {
public:
    void eat() { cout << "Tiger eating\n"; }
};
class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

```cpp
int main() {
    Lion fred;
    Tiger tigger;
    Bear pooh;

    vector<Animal*> animals;

    animals.push_back(&fred);
    animals.push_back(&tigger);
    animals.push_back(&pooh);

    for (size_t i = 0; i < animals.size(); ++i) {
        animals[i]->eat();
    }
}
```

wanted
&
observed

```
Lion eating
Tiger  eating
Bear  eating
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_I() { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

must be method of Base class in order to compile

```
% g++ -std=c++11 override.cpp -o override.o
% override.o
Base
Base
```

instance of Derived class can be used at runtime

TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220328<A|D>

Replace <A|D> with this section's letter

Why is the base class method being invoked each time the `foo()` function is called?

```
class Base {
public:
    virtual void where_am_i() { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_I() { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

```
% g++ -std=c++11 override.cpp -o override.o
% override.o
Base
Base
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_I() { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

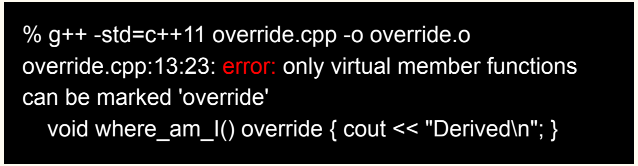
different function names

```
% g++ -std=c++11 override.cpp -o override.o
% override.o
Base
Base
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void ~~where_am_I()~~ override { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

different function names



```
% g++ -std=c++11 override.cpp -o override.o
override.cpp:13:23: error: only virtual member functions
can be marked 'override'
    void where_am_I() override { cout << "Derived\n"; }
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_i() override { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

same function name

```
% g++ -std=c++11 override.cpp -o override.o
% ./override.o
Base
Derived
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_i() override { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_i() override { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_i() override { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

different function signatures

compilation error!

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_I() { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

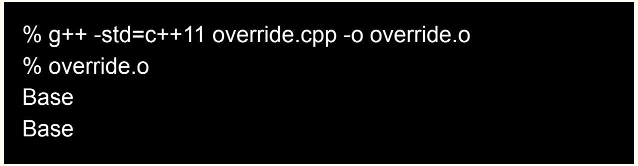
Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_I() const { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_I() const { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

different function names



```
% g++ -std=c++11 override.cpp -o override.o
% override.o
Base
Base
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_I() const ___ { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_I() const _1_ { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

Which keyword replaces blank #1 to indicate that the `where_am_I()` method is meant to override a method with the same signature?

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_I() const _1_ { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

```
% g++ -std=c++11 override.cpp -o override.o
% override.o
Base
Base
```

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_I() const override { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

compilation error!

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_i() const override { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

~~compilation error!~~

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_i() const override { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

```
% g++ -std=c++11 override.cpp -o override.o
% ./override.o
Base
Derived
```

Polymorphism and function parameters

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};

void feed_animal(Animal an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

```
int main() {
   Lion leo;

   feed_animal(leo);
}
```

```
Feeding the animal
_1_
```

What is output (replacing blank #1) for this program?

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};

void feed_animal(Animal an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

```
int main() {
    Lion leo;

    feed_animal(leo);
}
```

```
Feeding the animal
_1_
```

Polymorphism and function parameters

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

pass-by-value results in slicing

```
void feed_animal(Animal an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

```
int main() {
    Lion leo;

    feed_animal(leo);
}
```

```
Feeding the animal
Animal eating
```

Polymorphism and function parameters

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

pass-by-reference

```
void feed_animal(Animal& an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

```
int main() {
    Lion leo;

    feed_animal(leo);
}
```

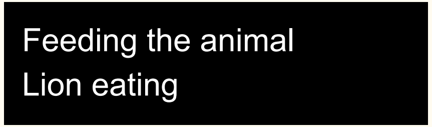
```
Feeding the animal
Lion eating
```

Polymorphism and function parameters

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};

void feed_animal(Animal& an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

```
int main() {
    Lion leo;

    feed_animal(leo);
}
```

can pass Animal, Lion, or Bear instance at runtime



```
Feeding the animal
Lion eating
```

eat() must be defined for Animal class for code to compile

Polymorphism and function parameters

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};

void feed_animal(Animal& an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

Polymorphism and pointers

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
    void climb() { cout << "Lion climbing\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
void feed_animal(Animal& an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

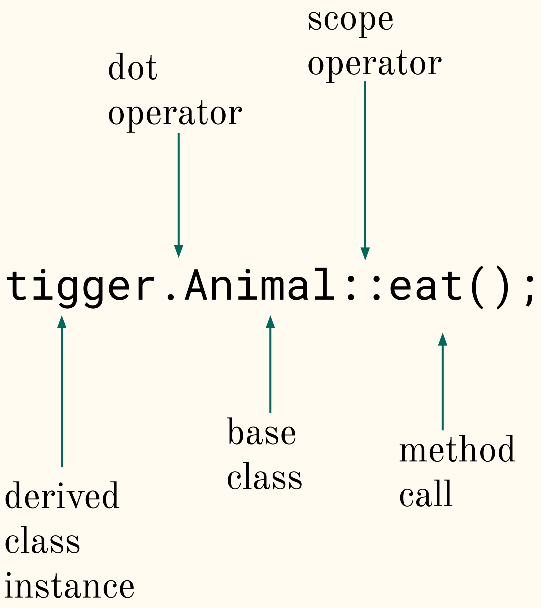
```
int main() {
    Animal* an_ptr = new Lion();

    an_ptr->climb();   compilation error!
}
```

Calling a base class method (outside of class)

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Tiger : public Animal {
public:
    void eat() { cout << "Tiger eating\n"; }
};
```

```
int main() {
    Tiger tigger;
    tigger.eat();
    tigger.Animal::eat();
}
```

```
Tiger  eating
Animal eating
```



tigger.Animal::eat();

Calling a base class method (inside of class)

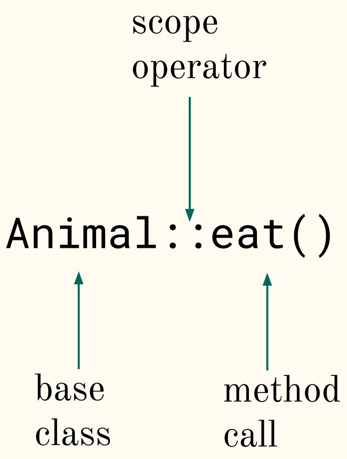
```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Tiger : public Animal {
public:
    void eat() {
        cout << "Tiger eating\n";
    }
};
```

```
int main() {
    Tiger tigger;

    tigger.eat();
}
```

Calling a base class method (inside of class)

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Tiger : public Animal {
public:
    void eat() {
        cout << "Tiger!!!\n";
        Animal::eat();
    }
};
```

```
int main() {
    Tiger tigger;

    tigger.eat();
}
```


scope operator

`Animal::eat()`

base class

method call

```
Tiger!!!
Animal eating
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base { };

int main() {
    Derived der;
    der.foo(17);
}
```

```
% g++ -std=c++11 hiding1.cpp -o hiding1.o
% ./hiding1.o
Base::foo(num)
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo(int num) const { cout << "Derived::foo()\n"; }
};

int main() {
    Derived der;
    der.foo(17);
}
```

```
% g++ -std=c++11 hiding2.cpp -o hiding2.o
% ./hiding2.o
Derived::foo()
```

Why is the derived class definition being invoked each time the `foo()` method is called?

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo(int num) const { cout << "Derived::foo()\n"; }
};

int main() {
    Derived der;
    der.foo(17);
}
```

```
% g++ -std=c++11 hiding2.cpp -o hiding2.o
% ./hiding2.o
Derived::foo()
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo(int num) const { cout << "Derived::foo()\n"; }
};

int main() {
    Derived der;
    der.foo(17);
}
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
};

int main() {
    Derived der;
    der.foo(17);  compilation error!
}
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
};

int main() {
    Derived der;
    der.foo(17);  compilation error!

    der.Base::foo(17); // call Base foo() directly
}
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
};

int main() {
    Derived der;
    der.foo(17);  compilation error!
}
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
    void foo(int num) const { Base::foo(num); }
};

int main() {
    Derived der;
    der.foo(17);
}
```

~~compilation error!~~

```
% g++ -std=c++11 hiding4.cpp -o hiding4.o
% ./hiding4.o
Base::foo(num)
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
   ~~ void foo(int num) const { Base::foo(num~~); }
     using Base::foo;
};

int main() {
    Derived der;
    der.foo(17);
}
```

```
% g++ -std=c++11 hiding4.cpp -o hiding4.o
% ./hiding4.o
Base::foo(num)
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
    using Base::foo;
};

int main() {
    Derived der;
    der.foo(17);
}
```

```
% g++ -std=c++11 hiding4.cpp -o hiding4.o
% ./hiding4.o
Base::foo(num)
```