Introduction to Programming (C/C++)

06: Inheritance & Polymorphism

Huanchen Zhang

清華大學
Tsinghua University

交叉信息研究院
Institute for Interdisciplinary Information Sciences

Recap

Encapsulation & Abstraction

Class vs. Object

Data member

Function member

Access Control

Scope Operator

Static member

- Constructor
- Copy Constructor
- Destructor
- Getter & Setter

Friendship

- Allow functions and classes outside the class to access private members

Friendship

➜ Allow functions and classes outside the class to access private members

```
class A {
 public:
  A(int val) : val_(val) {}
  int GetVal() const { return val_; }

 private:
  int val_;
};
```

```
bool IsEqual(const A &a1,
             const A &a2) {
  return (a1.GetVal() ==
          a2.GetVal());
}
```

Friendship

➜ Allow functions and classes outside the class to access private members

```
class A {
 public:
  A(int val) : val_(val) {}


 private:
  int val_;
};
```

```
bool IsEqual(const A &a1,
             const A &a2) {
  return (a1.GetVal() ==
          a2.GetVal());
}
```

Friendship

➜ Allow functions and classes outside the class to access private members

```
class A {
 public:
  A(int val) : val_(val) {}


 private:
  int val_;
};
```

```
bool IsEqual(const A &a1,
             const A &a2) {
  return (a1.GetVal() ==
          a2.GetVal());
}
```

?

Friendship

➜ Allow functions and classes outside the class to access private members

```
class A {
 public:
  A(int val) : val_(val) {}
  friend bool IsEqual(const A &a1,
                      const A &a2);
 private:
  int val_;
};
```

```
bool IsEqual(const A &a1,
             const A &a2) {
  return (a1.val_ == a2.val_);
}
```

Friendship

➜ Allow functions and classes outside the class to access private members

```
class A {
 public:
  A(int val) : val_(val) {}
  friend bool IsEqual(const A &a1,
                      const A &a2);

 private:
  int val_;
};
```

NOT a member

```
bool IsEqual(const A &a1,
             const A &a2) {
  return (a1.val_ == a2.val_);
}
```

Friendship

➜ Allow functions and classes outside the class to access private members

```
class A {
 public:
  A(int val) : val_(val) {}
  friend bool IsEqual(const A &a1,
                      const A &a2);

 private:
  int val_;
};
```

NOT a member

```
bool IsEqual(const A &a1,
             const A &a2) {
  return (a1.val_ == a2.val_);
}
```

Friendship

➜ Allow functions and classes outside the class to access private members

```
class A {
 public:
  A(int val) : val_(val) {}
  friend class B;

 private:
  int val_;
};
```

```
class B {
 public:
  B(int c) : count_(c) {}
  void Accumulate(const A &a) {
    count_ += a.val_;
  }

 private:
  int count_;
};
```

Friendship

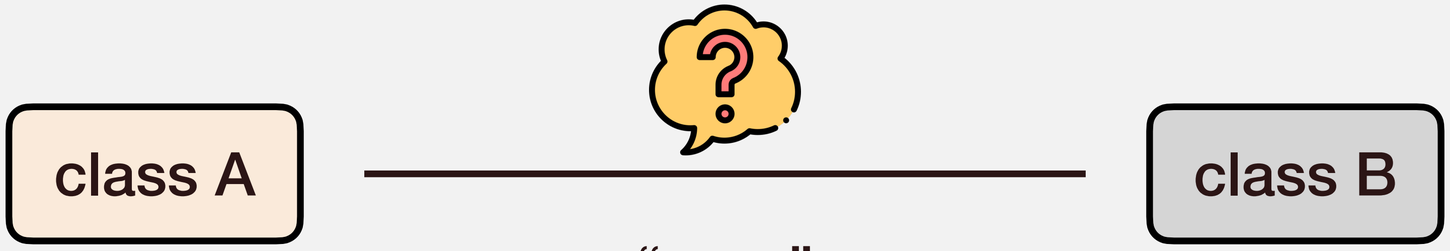
➜ Allow functions and classes outside the class to access private members

```
class A {
 public:
  A(int val) : val_(val) {}
  friend void B::Accumulate(const A &a);

 private:
  int val_;
};
```

```
class B {
 public:
  B(int c) : count_(c) {}
  void Accumulate(const A &a) {
    count_ += a.val_;
  }

 private:
  int count_;
};
```

Class Relationships

Class Relationships



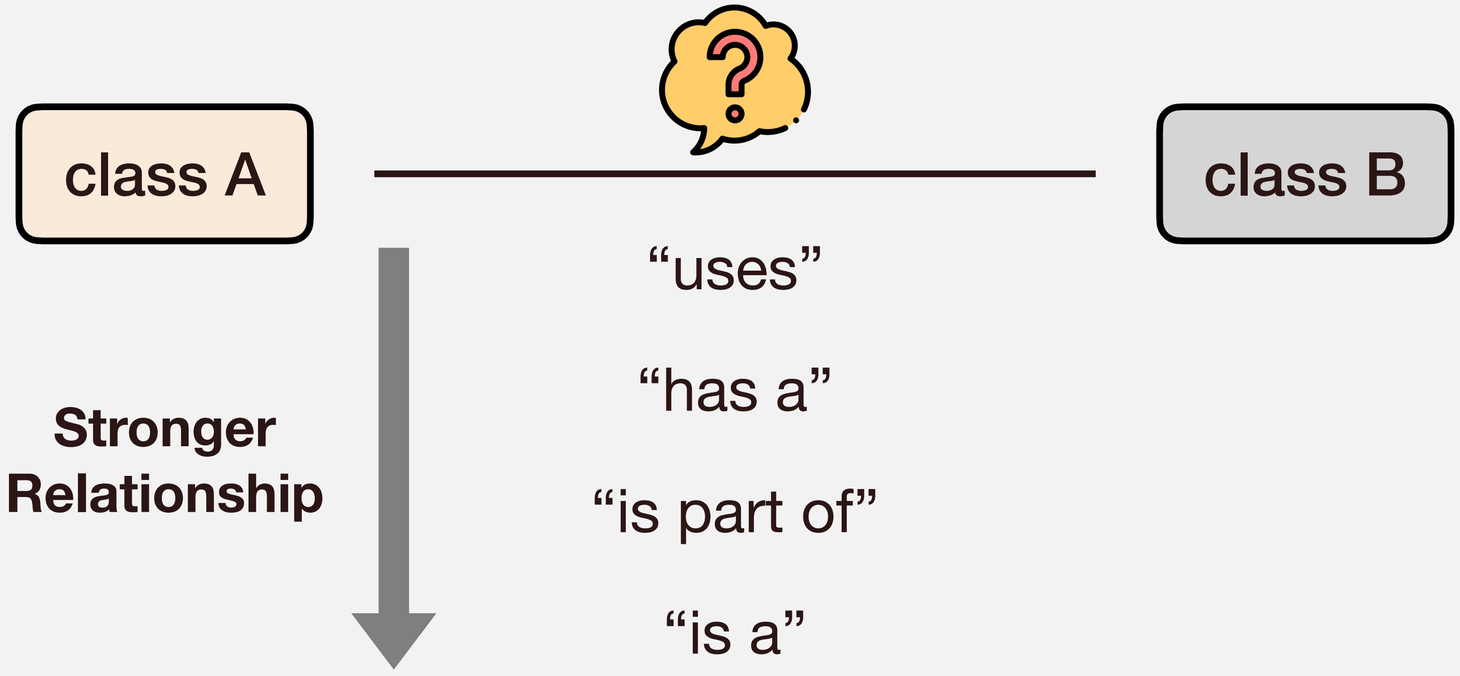
“uses”

“has a”

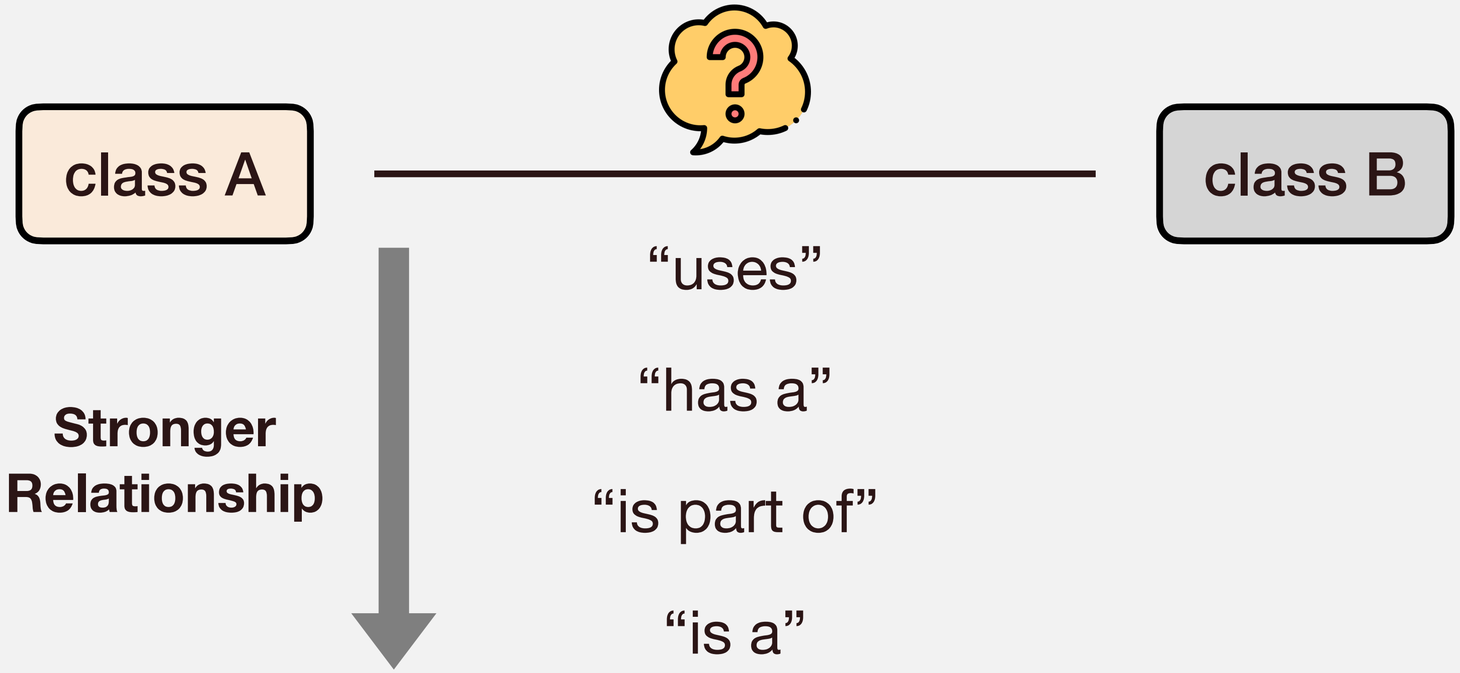
“is part of”

“is a”

Class Relationships



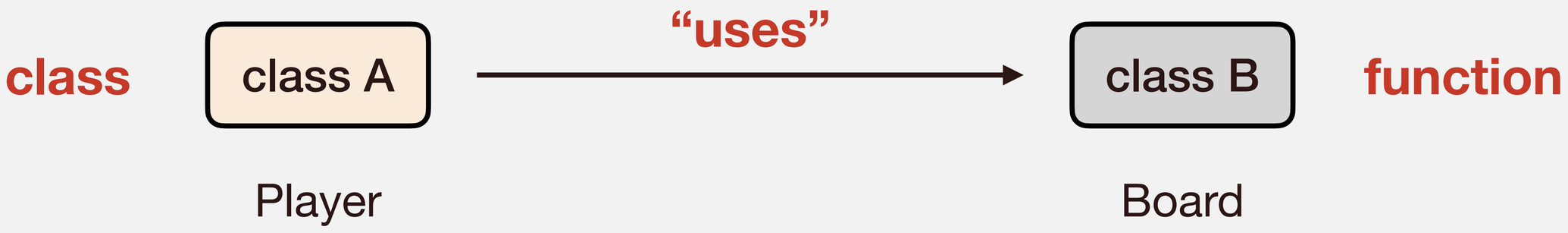
Class Relationships



Goal: to facilitate code **reuse**

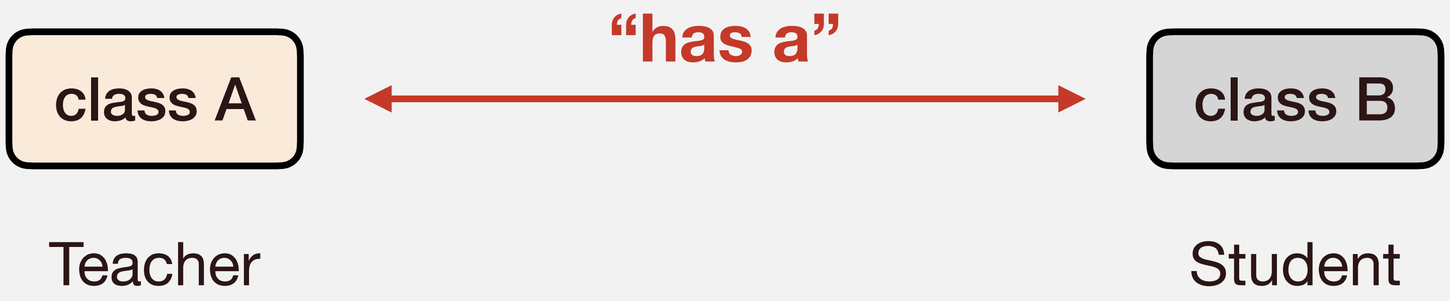
- build new “concepts” based on existing ones

Dependency

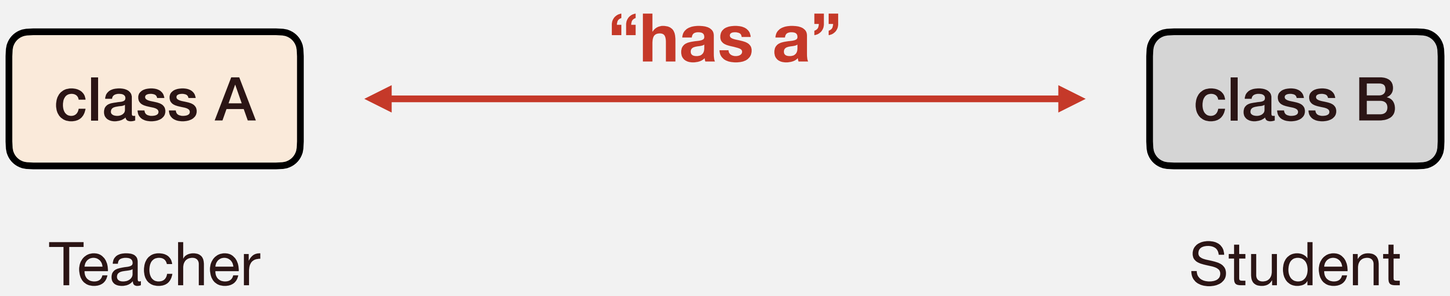



```
class Player {
 public:
   ...
   void Play(Board *board);
   ...
};
```

Association



Association




```
class Teacher {
 public:
  ...
 private:
  Student *s_[100];
};
```

```
class Student {
 public:
  ...
 private:
  Teacher *t_[10];
};
```

Association

class A ← "has a" → class B

Teacher

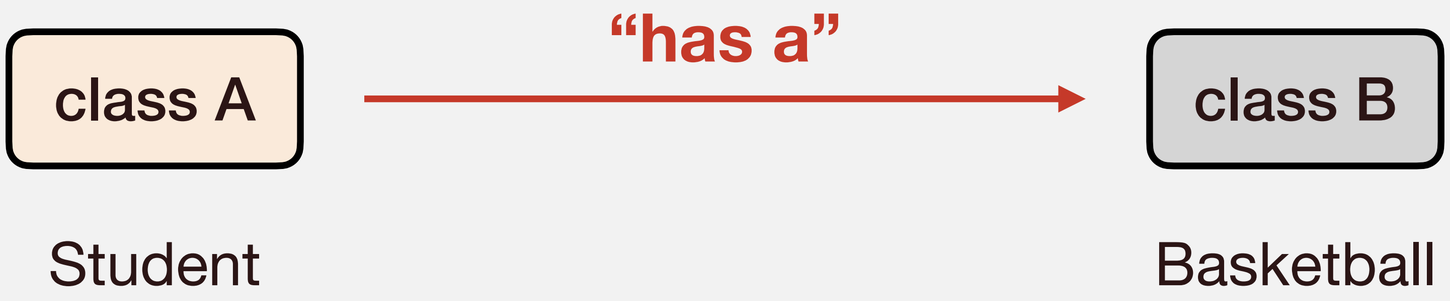
Student

- Typically pointer or reference members
- NOT responsible for creating/destroying the associated objects

```
class Teacher
 public:
  ...
 private:
  Student *s_[
};
```

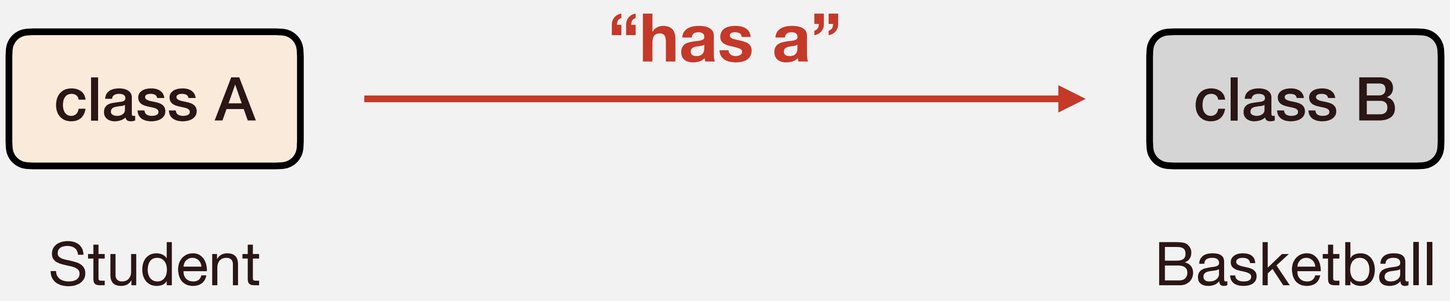
```
ss Student {
blic:
 .
ivate:
  Teacher *t_[10];
};
```

Aggregation




```
class Student {
 public:
   ...
 private:
  Basketball *ball_;
};
```

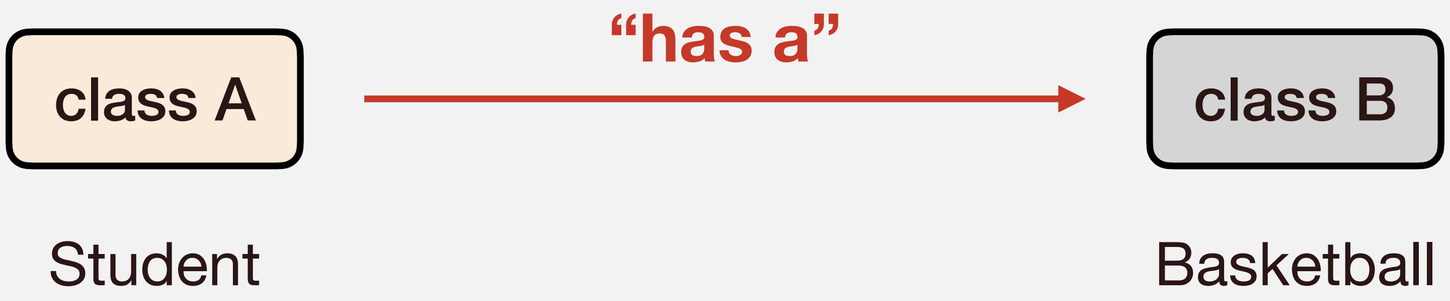
Aggregation




```
class Student {
 public:
   ...
 private:
  Basketball *ball_;
};
```

- Basketball is unaware of the Student object

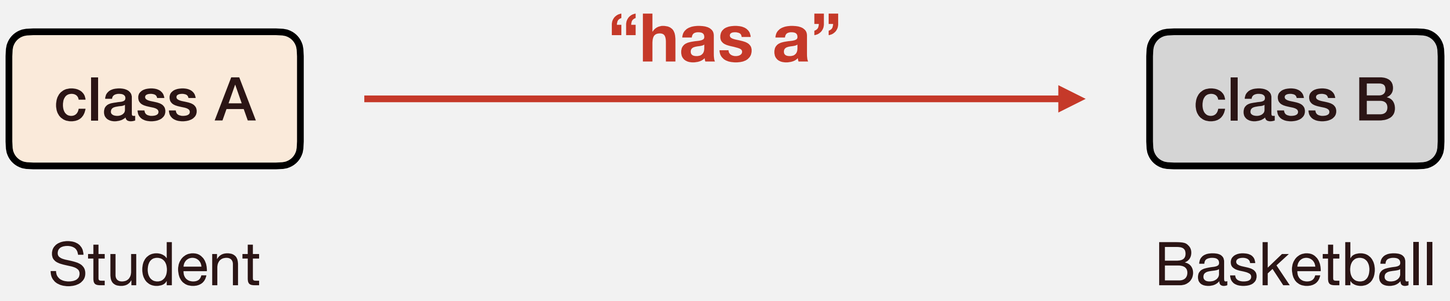
Aggregation




```
class Student {
 public:
  ...
 private:
  Basketball *ball_;
};
```

- Basketball is unaware of the Student object
- Basketball may be in ≥ 1 Student(s)

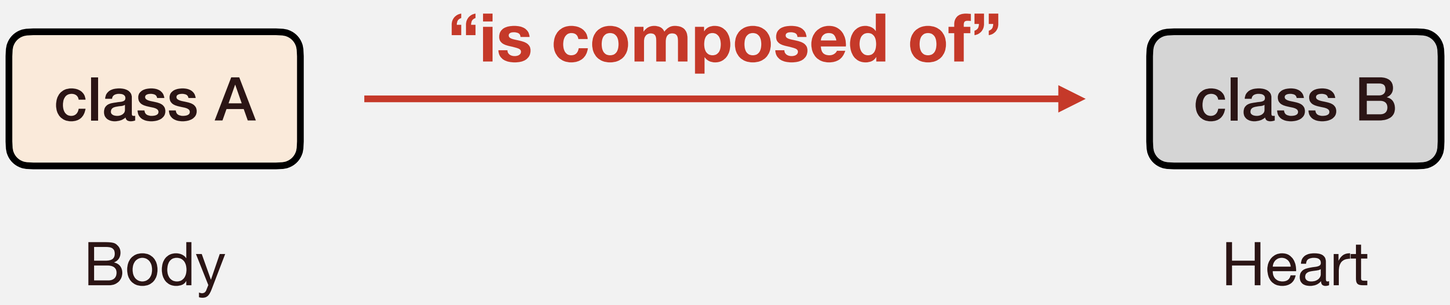
Aggregation




```
class Student {
 public:
  ...
 private:
  Basketball *ball_;
};
```

- Basketball is unaware of the Student object
- Basketball may be in ≥ 1 Student(s)
- Student is not responsible for creating/ destroying the Basketball

Composition




```
class Body {
  ...
 private:
  Heart heart_;
  Brain brain_;
  ...
};
```

Composition

Body

Heart

Brain

```
class Body {
  ...
 private:
  Heart heart_;
  Brain brain_;
  ...
};
```

- Heart is unaware of the body object

Composition

Body

Heart

Brain

```
class Body {
  ...
 private:
  Heart heart_;
  Brain brain_;
  ...
};
```

- Heart is unaware of the body object
- Heart can only be in one body

Composition

Body

Heart

Brain

```
class Body {
  ...
 private:
  Heart heart_;
  Brain brain_;
  ...
};
```

- Heart is unaware of the body object
- Heart can only be in one body
- The existence of Heart is managed by Body

Put Together

	Association	Aggregation	Composition
Relationship Type	`Equal`	`Whole→Part`	`Whole→Part`
Direction			
Manage member existence?			
Member can belong to multiple classes?			
Example	`Doctor` `Patient`	`Class` `Student`	`Car` `Engine`

Put Together

	Association	Aggregation	Composition
Relationship Type	`Equal`	`Whole→Part`	`Whole→Part`
Direction	$\longleftrightarrow$		
Manage member existence?			
Member can belong to multiple classes?			
Example	`Doctor` `Patient`	`Class` `Student`	`Car` `Engine`

Put Together

	Association	Aggregation	Composition
Relationship Type	`Equal`	`Whole→Part`	`Whole→Part`
Direction	$\longleftrightarrow$	$\longrightarrow$	$\longrightarrow$
Manage member existence?			
Member can belong to multiple classes?			
Example	`Doctor` `Patient`	`Class` `Student`	`Car` `Engine`

Put Together

	Association	Aggregation	Composition
Relationship Type	`Equal`	`Whole→Part`	`Whole→Part`
Direction	$\longleftrightarrow$	$\longrightarrow$	$\longrightarrow$
Manage member existence?	`NO`		
Member can belong to multiple classes?			
Example	`Doctor` `Patient`	`Class` `Student`	`Car` `Engine`

Put Together

	Association	Aggregation	Composition
Relationship Type	`Equal`	`Whole→Part`	`Whole→Part`
Direction	$\longleftrightarrow$	$\longrightarrow$	$\longrightarrow$
Manage member existence?	`NO`	`NO`	
Member can belong to multiple classes?			
Example	`Doctor` `Patient`	`Class` `Student`	`Car` `Engine`

Put Together

	Association	**Aggregation**	**Composition**
Relationship Type	`Equal`	`Whole→Part`	`Whole→Part`
Direction	$\longleftrightarrow$	$\longrightarrow$	$\longrightarrow$
Manage member existence?	`NO`	`NO`	`YES`
Member can belong to multiple classes?			
Example	`Doctor` `Patient`	`Class` `Student`	`Car` `Engine`

Put Together

	Association	Aggregation	Composition
Relationship Type	`Equal`	`Whole→Part`	`Whole→Part`
Direction	$\longleftrightarrow$	$\longrightarrow$	$\longrightarrow$
Manage member existence?	`NO`	`NO`	`YES`
Member can belong to multiple classes?	`YES`		
Example	`Doctor` `Patient`	`Class` `Student`	`Car` `Engine`

Put Together

	Association	Aggregation	Composition
Relationship Type	`Equal`	`Whole→Part`	`Whole→Part`
Direction	$\longleftrightarrow$	$\longrightarrow$	$\longrightarrow$
Manage member existence?	`NO`	`NO`	`YES`
Member can belong to multiple classes?	`YES`	`YES`	
Example	`Doctor` `Patient`	`Class` `Student`	`Car` `Engine`

Put Together

	Association	**Aggregation**	**Composition**
Relationship Type	`Equal`	`Whole→Part`	`Whole→Part`
Direction	$\longleftrightarrow$	$\longrightarrow$	$\longrightarrow$
Manage member existence?	`NO`	`NO`	`YES`
Member can belong to multiple classes?	`YES`	`YES`	`NO`
Example	`Doctor` `Patient`	`Class` `Student`	`Car` `Engine`

Inheritance

```
class Dog {
 public:
  Dog(...);
  void Eat(Food *food);
  void WagTail();

 private:
  std::string name_;
  int age_;
  Person *owner_;
  bool walked_today_;
};
```

Inheritance

```
class Dog {
 public:
  Dog(...);
  void Eat(Food *food);
  void WagTail();

 private:
  std::string name_;
  int age_;
  Person *owner_;
  bool walked_today_;
};
```

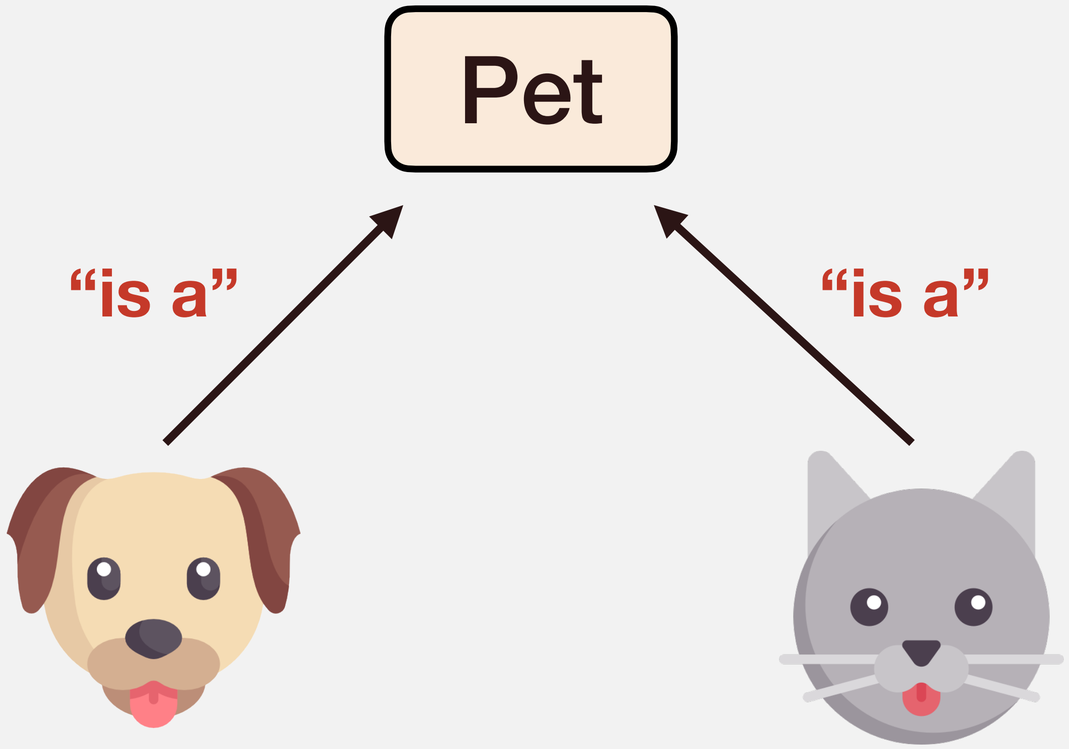
```
class Cat {
 public:
  Cat(...);
  void Eat(Food *food);
  void Purr();

 private:
  std::string name_;
  int age_;
  Person *owner_;
  int num_mouse_killed_;
};
```

Inheritance

```
class Dog {
 public:
  Dog(...);
  void Eat(Food *food);
  void WagTail();

 private:
  std::string name_;
  int age_;
  Person *owner_;
  bool walked_today_;
};
```

```
class Cat {
 public:
  Cat(...);
  void Eat(Food *food);
  void Purr();

 private:
  std::string name_;
  int age_;
  Person *owner_;
  int num_mouse_killed_;
};
```

Inheritance



Inheritance

Pet

```
class Pet {
 public:
  Pet(...);
  void Eat(Food *food);

 private:
  std::string name_;
  int age_;
  Person *owner_;
};
```

Inheritance

Pet

```
class Pet {
 public:
  Pet(...);
  void Eat(Food *food);

 private:
  std::string name_;
  int age_;
  Person *owner_;
};
```

```
class Dog : public Pet {
 public:
  Dog(...);
  void WagTail();

 private:
  bool walked_today_;
};
```

Inheritance

Pet

```
class Pet {
 public:
  Pet(...);
  void Eat(Food *food);

 private:
  std::string name_;
  int age_;
  Person *owner_;
};
```

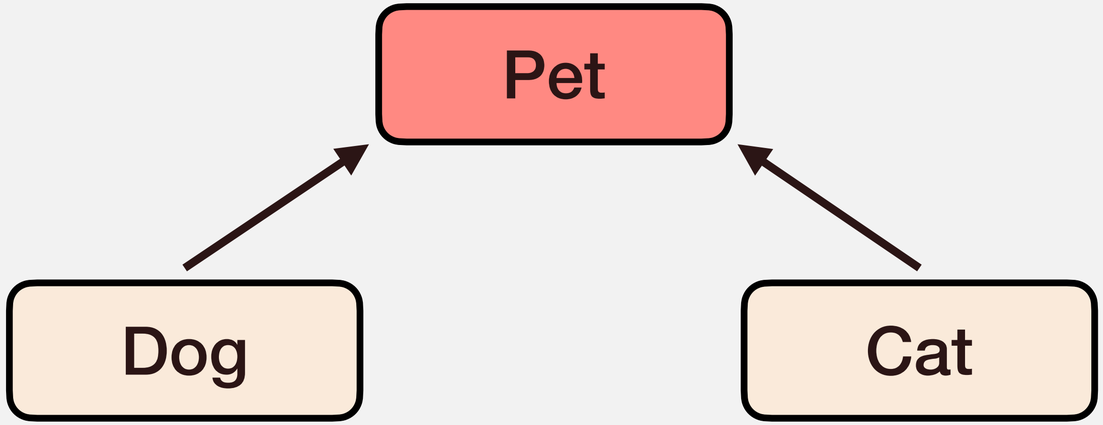
```
class Dog : public Pet {
 public:
  Dog(...);
  void WagTail();

 private:
  bool walked_today_;
};
```

```
class Cat : public Pet {
 public:
  Cat(...);
  void Purr();

 private:
  int num_mouse_killed_;
};
```

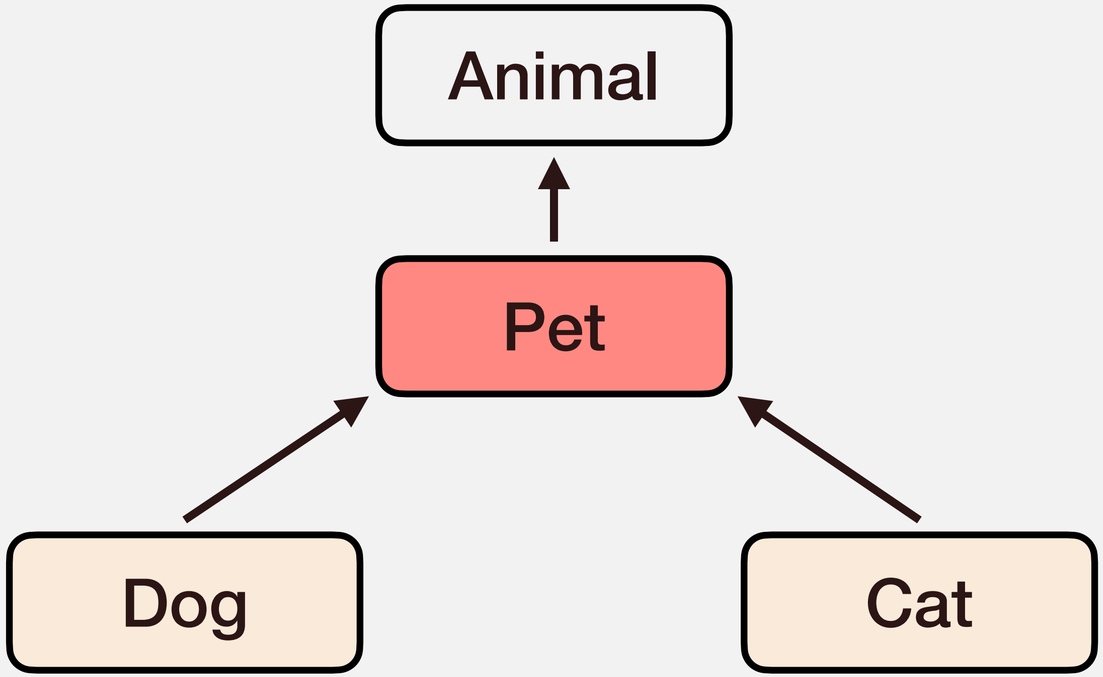
Why is inheritance useful?

To avoid code duplicate

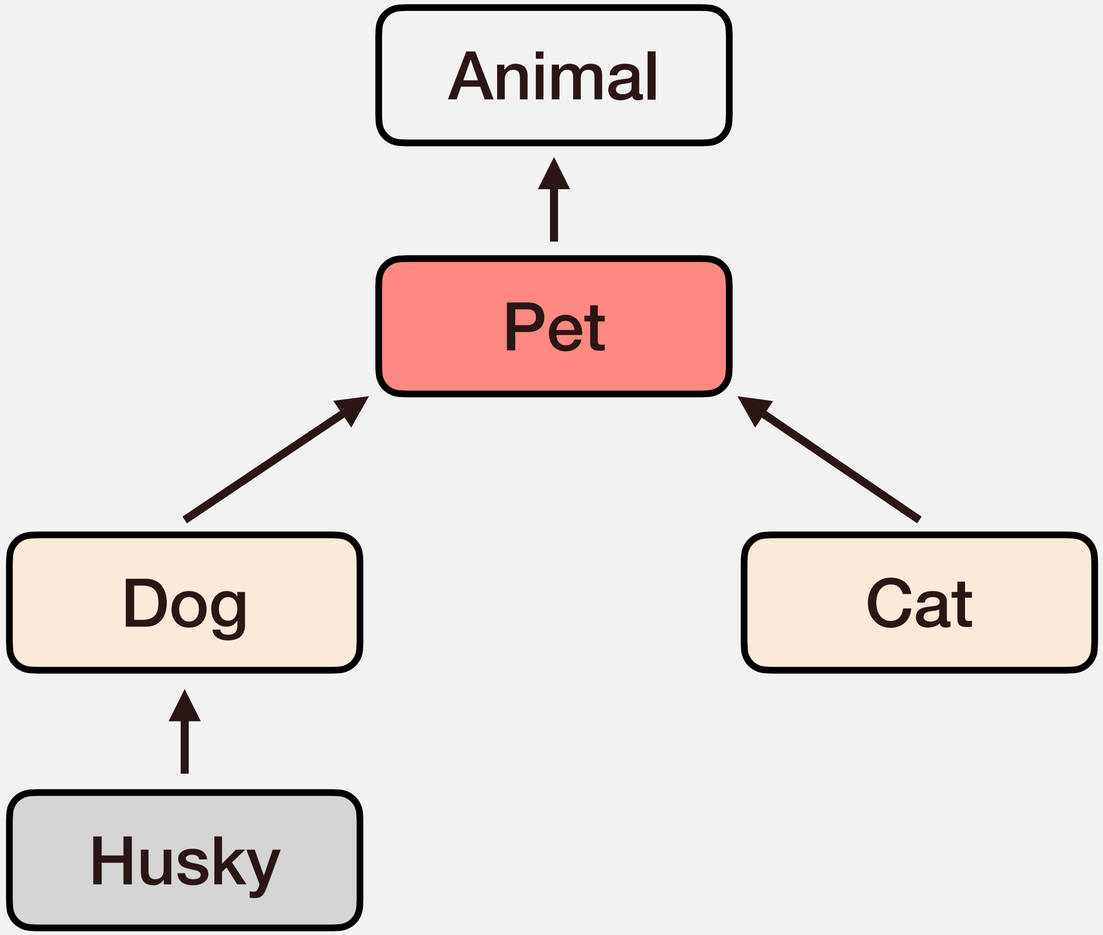
Inheritance Hierarchy



Inheritance Hierarchy



Inheritance Hierarchy



Base and Derived Classes

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 private:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(long c);
  long GetCount() const;

 private:
  long count_;
};
```

Base and Derived Classes

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 private:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(long c);
  long GetCount() const;

 private:
  long count_;
};
```

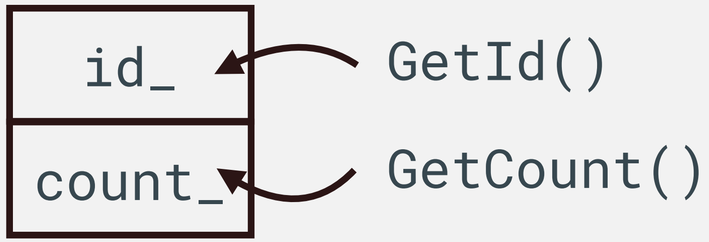
Inherits both **properties** and **behaviors**

Base and Derived Classes

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 private:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(long c);
  long GetCount() const;

 private:
  long count_;
};
```

Inherits both **properties** and **behaviors**



Access Control — Revisit

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 private:
  int id_;
};
```

Access Control — Revisit

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 private:
  int id_;
};
```

← Can be accessed by **anyone**

Access Control — Revisit

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 private:
  int id_;
};
```

← Can be accessed by **anyone**

← Can only be accessed by **members** or **friends**

Access Control with Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 private:
  int id_;
};
```

← Can be accessed by **anyone**

← Can only be accessed by **members** or **friends**

Derived class CANNOT access

Access Control with Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

← Can be accessed by **anyone**

← Can only be accessed by **members**, **friends** or **derived classes**

Access Control with Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

← Can be accessed by **anyone**

← Can only be accessed by **members**, **friends** or **derived classes**

Favor `private` over protected

Constructing a Derived Object

```
class Base {
 public:
  Base();
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(long c);
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(long c)
    : count_(c) {}



int main() {
  Derived d_obj(5);
}
```

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(long c)
    : count_(c) {}



int main() {
  Derived d_obj(5);
}
```

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(long c) ❶
    : count_(c) {}



int main() {
  Derived d_obj(5);
}
```

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}  ❷

Base::Base(int id) : id_(id) {}

Derived::Derived(long c)  ❶
    : count_(c) {}


int main() {
  Derived d_obj(5);
}
```

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}  ❷

Base::Base(int id) : id_(id) {}

Derived::Derived(long c)  ❶
    : count_(c) {}


int main() {
  Derived d_obj(5);
}
```

Derived Object

Base int id_ = 0
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}  ❷

Base::Base(int id) : id_(id) {}

Derived::Derived(long c)  ❶
    : count_(c) {}  ❸


int main() {
  Derived d_obj(5);
}
```

Derived Object

Base int id_ = 0
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}  ❷

Base::Base(int id) : id_(id) {}

Derived::Derived(long c)  ❶
    : count_(c) {}  ❸
```

```
int main() {
  Derived d_obj(5);
}
```

Derived Object

`Base` `int id_ = 0`
`Derived` `long count_ = 5`

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(long c)
    : count_(c) {}



int main() {
  Derived d_obj(5);
}
```

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(int id, long c)
    : id_(id), count_(c) {}


int main() {
  Derived d_obj(1, 5);
}
```

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(int id, long c)
    : id_(id), count_(c) {}
```

Can appear in initializer list **only once**

```
int main() {
  Derived d_obj(1, 5);
}
```

Derived Object

`Base` `int id_`
`Derived` `long count_`

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(int id, long c)
    : count_(c) {
  id_ = id;
}

int main() {
  Derived d_obj(1, 5);
}
```

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(int id, long c)
    : count_(c) {
  id_ = id;
}

int main() {
  Derived d_obj(1, 5);
}
```

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(int id, long c)
    : count_(c) {
  id_ = id;
}

int main() {
  Derived d_obj(1, 5);
}
```

but **Bad Practice**

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(int id, long c)
    : count_(c) {
  id_ = id;
}

int main() {
  Derived d_obj(1, 5);
}
```

but **Bad Practice**

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(int id, long c)
    : count_(c) {
  id_ = id;
}

int main() {
  Derived d_obj(1, 5);
}
```

but **Bad Practice**

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(int id, long c)
    : Base(id),
      count_(c) {}


int main() {
  Derived d_obj(1, 5);
}
```

Derived Object

Base int id_
Derived long count_

Construction Order

```
Base::Base() : id_(0) {}

Base::Base(int id) : id_(id) {}

Derived::Derived(int id, long c)
    : Base(id),
      count_(c) {}


int main() {
  Derived d_obj(1, 5);
}
```

1. Memory allocation for derived object
2. Call derived constructor
3. Call (specified) base constructor
4. Base constructor initializer list
5. Base constructor body
6. Derived constructor initializer list
7. Derived constructor body

Inheritance Chain

```
class A {
 public:
  A(int a)
      : a_(a) {}
 private:
  int a_;
};
```

Inheritance Chain

```
class A {
 public:
  A(int a)
      : a_(a) {}
 private:
  int a_;
};
```

```
class B : public A {
 public:
  B(int a, int b)
      : A(a), b_(b) {}
 private:
  int b_;
};
```

Inheritance Chain

```
class A {
 public:
  A(int a)
      : a_(a) {}
 private:
  int a_;
};
```

```
class B : public A {
 public:
  B(int a, int b)
      : A(a), b_(b) {}
 private:
  int b_;
};
```

```
class C : public B {
 public:
  C(int a, int b, int c)
      : B(a, b), c_(c) {}
 private:
  int c_;
};
```

Inheritance Chain

```
class A {
 public:
  A(int a)
      : a_(a) {}
 private:
  int a_;
};
```

```
class B : public A {
 public:
  B(int a, int b)
      : A(a), b_(b) {}
 private:
  int b_;
};
```

```
class C : public B {
 public:
  C(int a, int b, int c)
      : B(a, b), c_(c) {}
 private:
  int c_;
};
```

C cannot call A constructor

Public Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

Derived Object

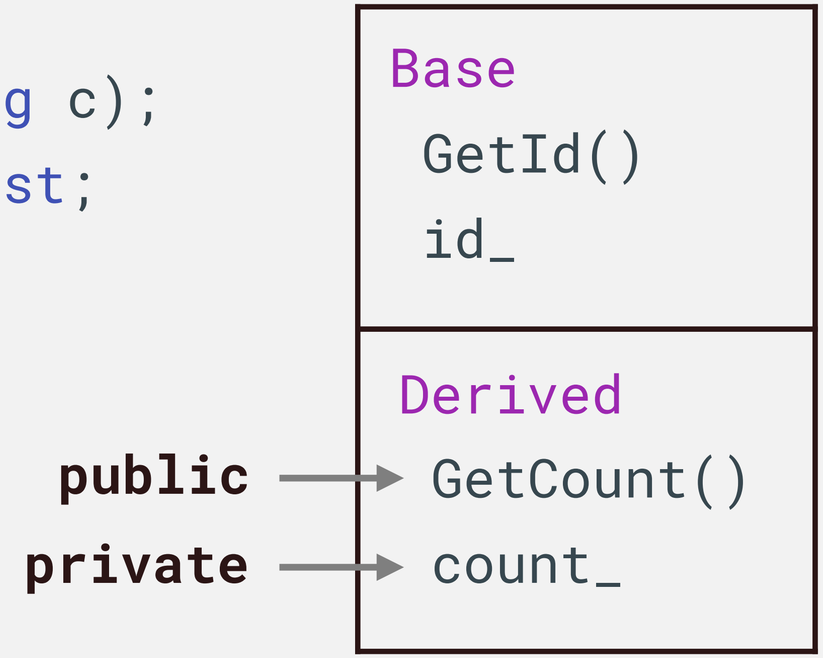
Base GetId() id_
Derived GetCount() count_

Public Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

Derived Object

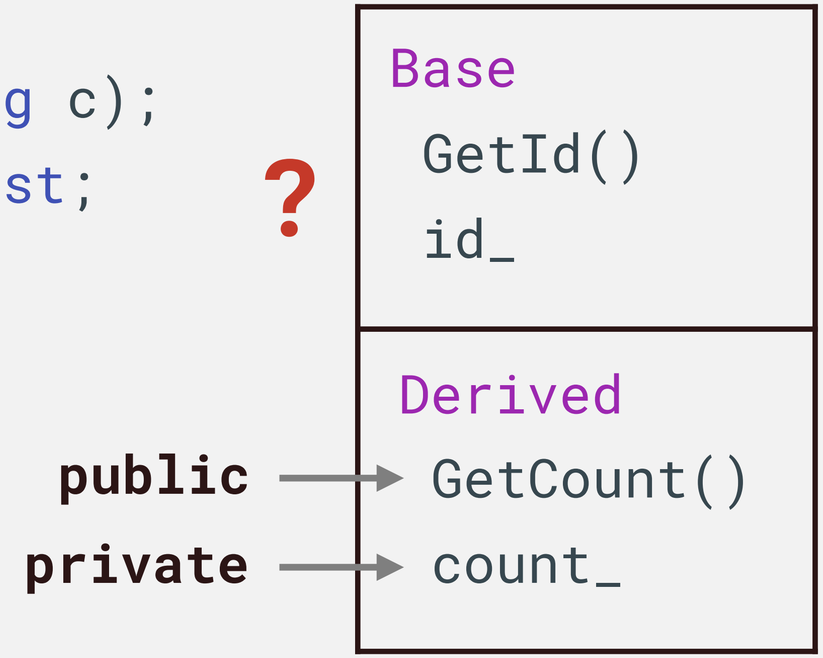


Public Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

Derived Object



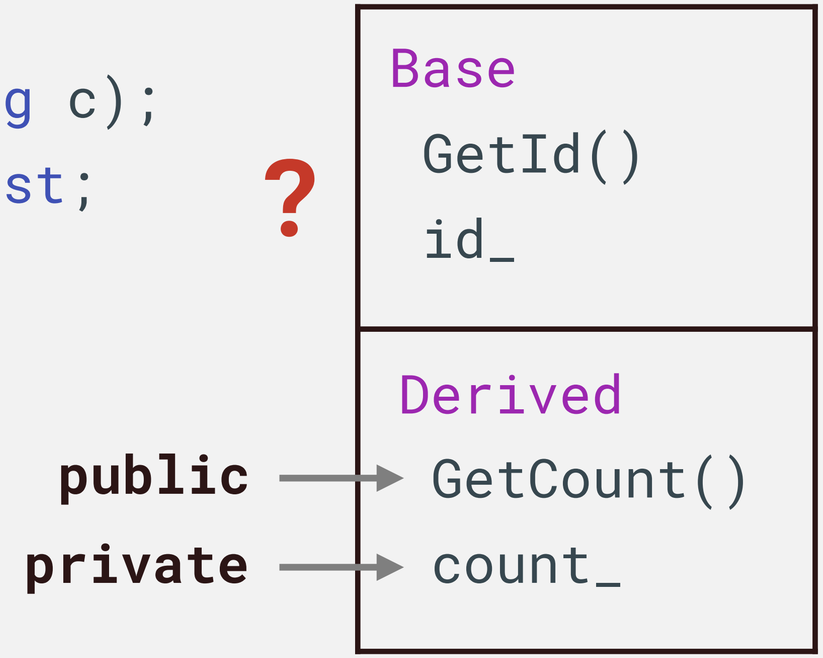
Public Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

public → public

Derived Object



Public Inheritance

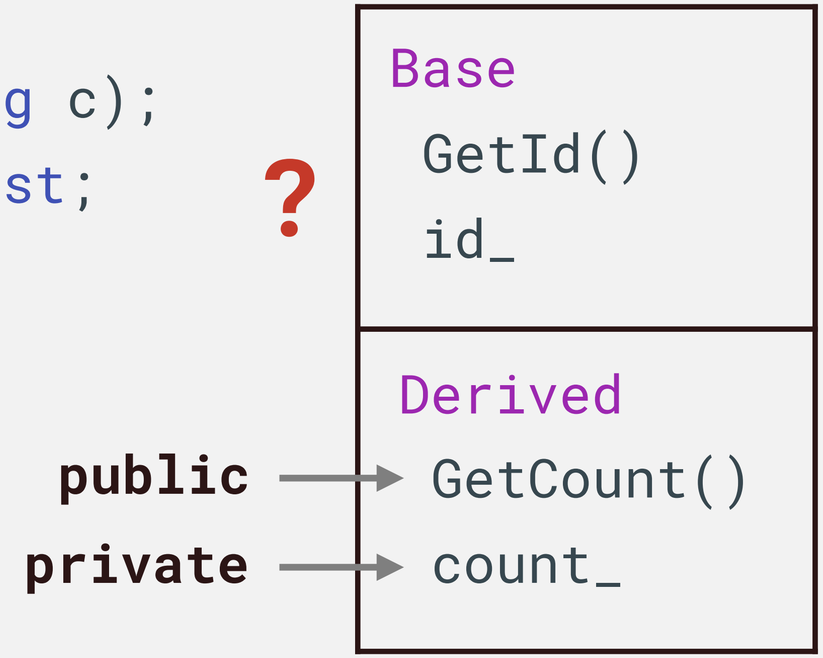
```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

public → public

protected → protected

Derived Object



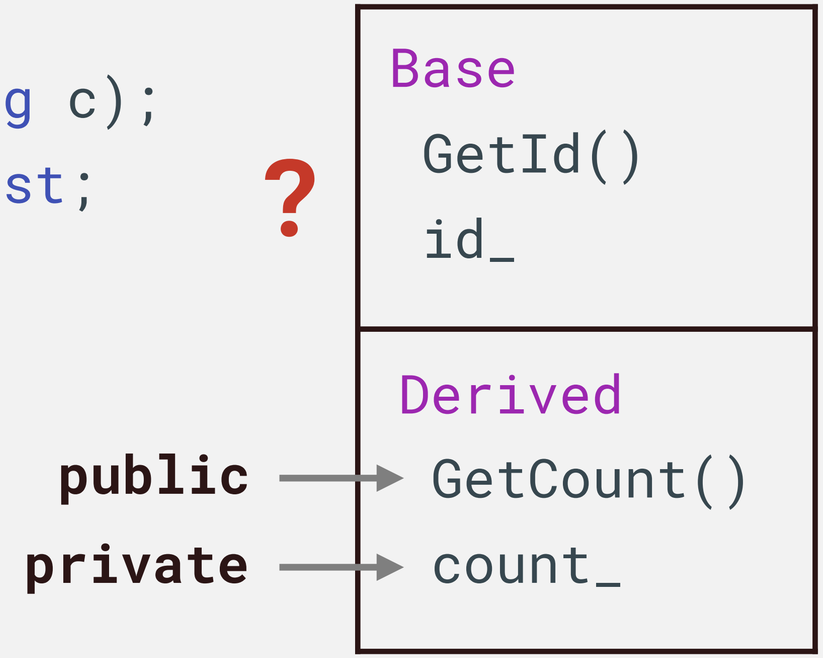
Public Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

public	→	public
protected	→	protected
private	→	inaccessible

Derived Object



Public Inheritance

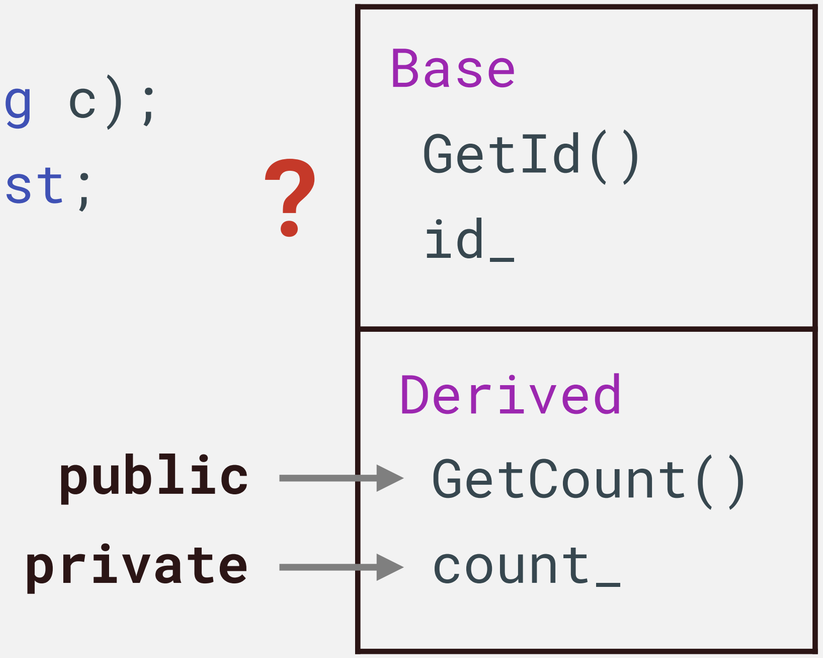
Most Common

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

public	→	public
protected	→	protected
private	→	inaccessible

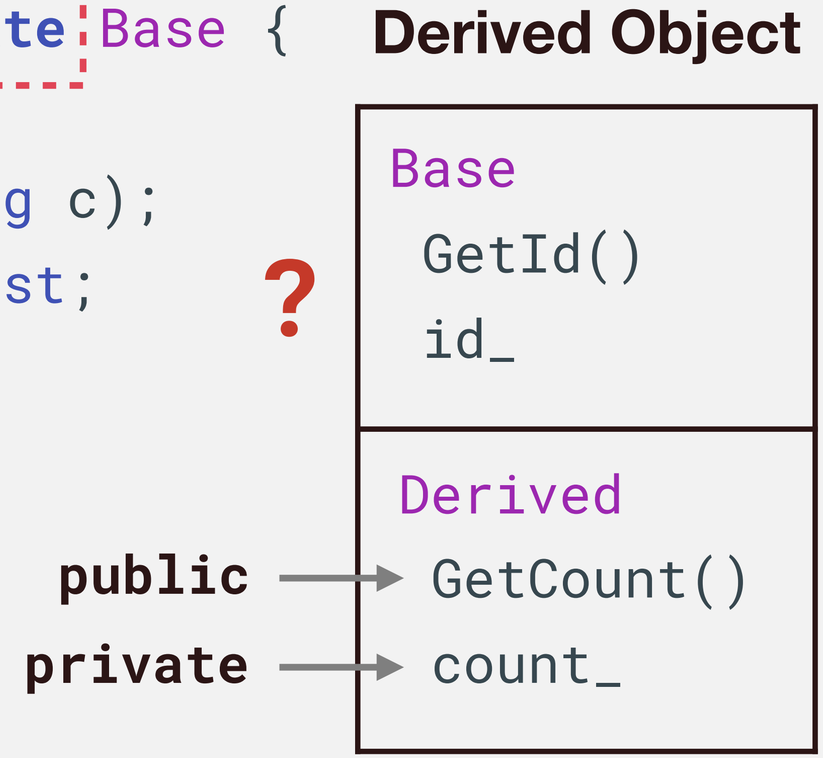
Derived Object



Private Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : private Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```



Private Inheritance

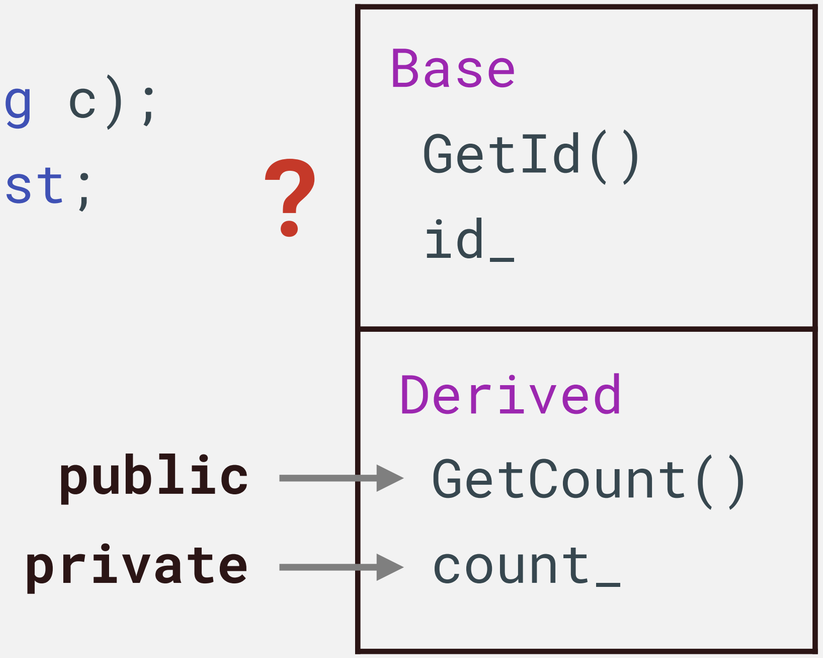
```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : private Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

public → private

?

Derived Object



Private Inheritance

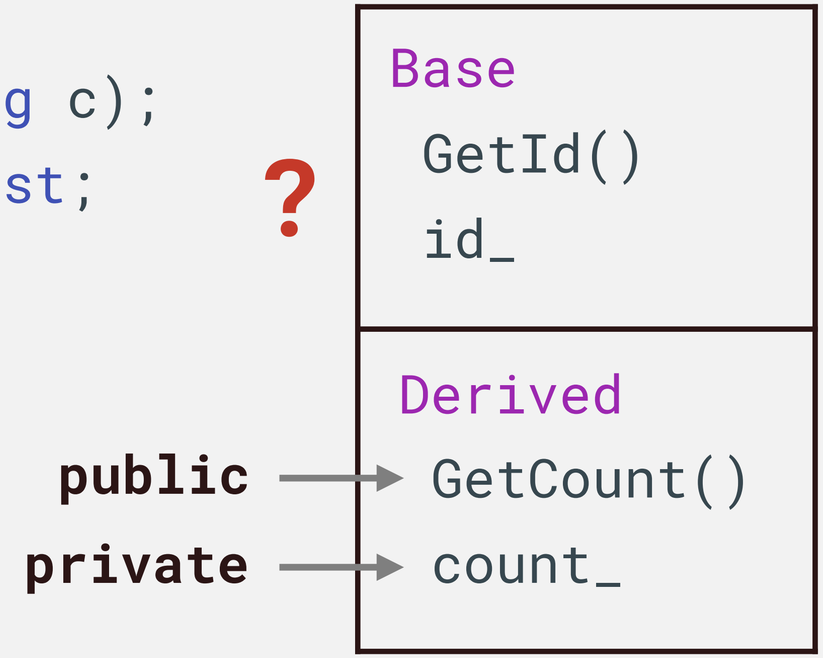
```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : private Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

public → private

protected → private

Derived Object

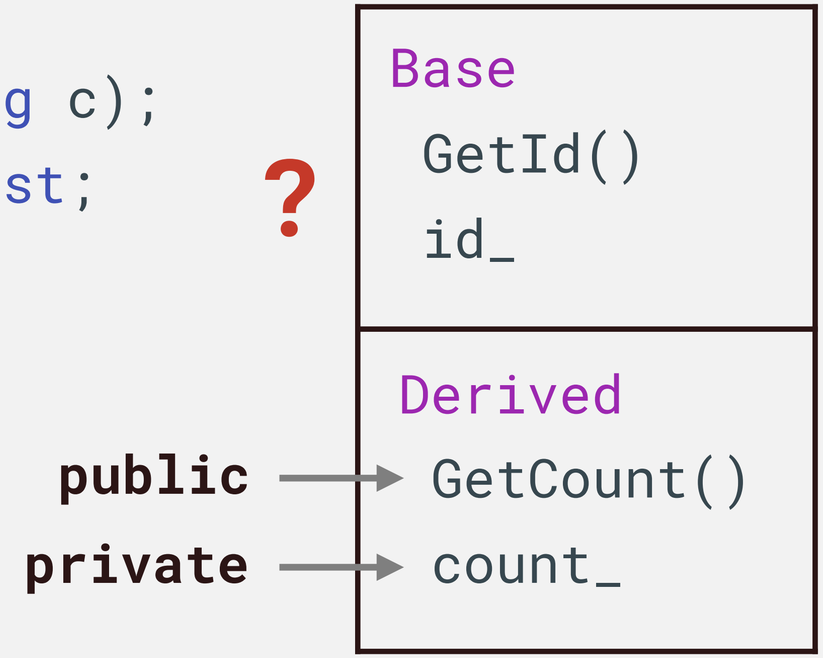


Private Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : private Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

Derived Object



public	→	private
protected	→	private
private	→	inaccessible

Private Inheritance

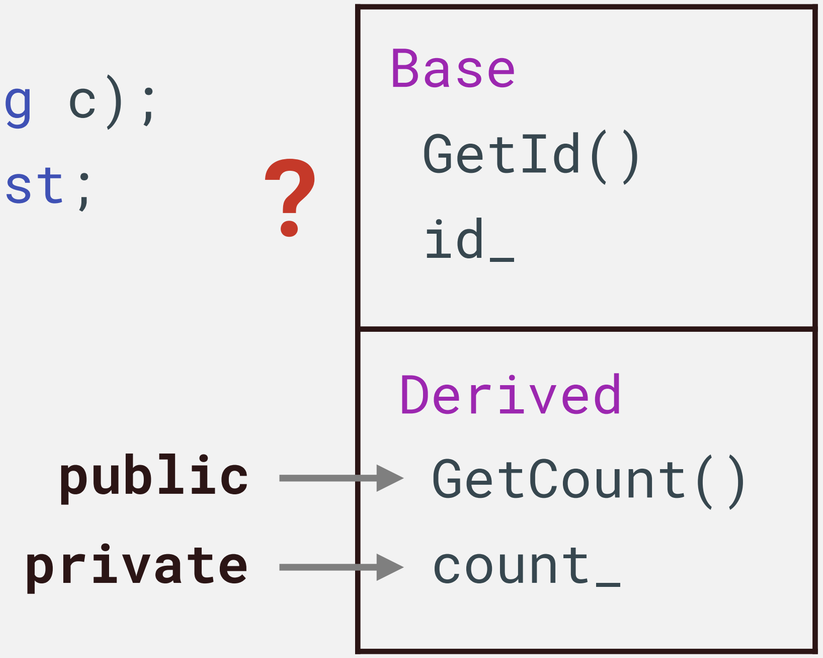
Rare

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : private Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

?

Derived Object



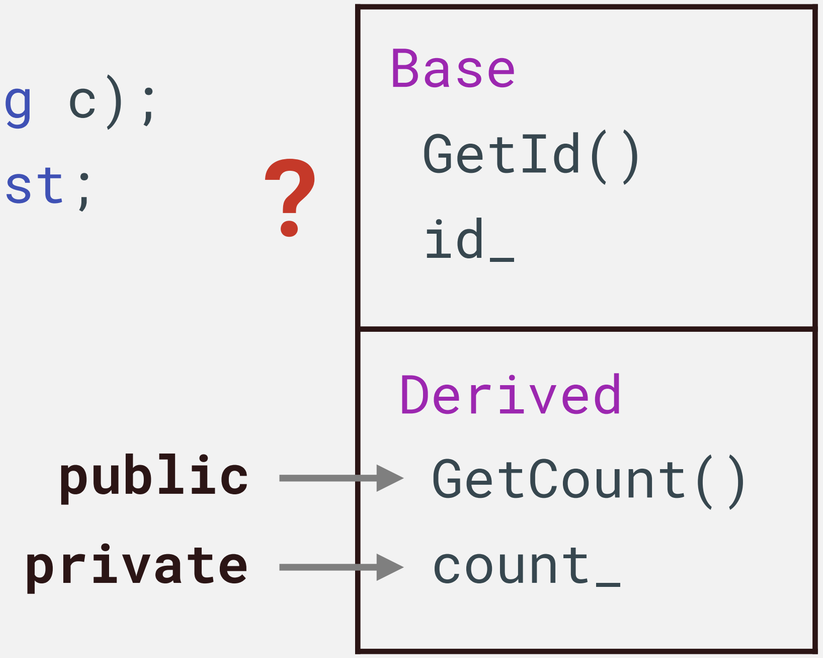
public	→	private
protected	→	private
private	→	inaccessible

Protected Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : protected Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

Derived Object

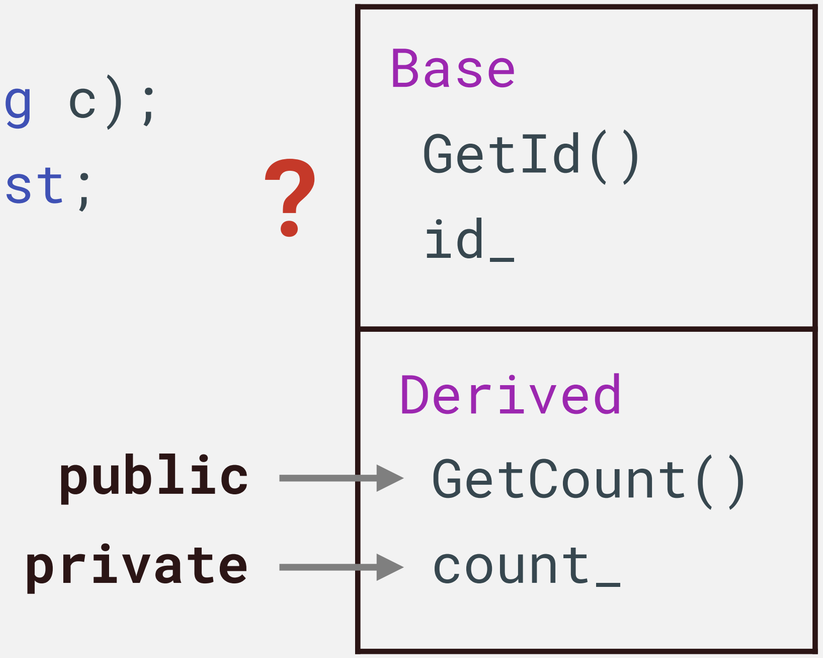


Protected Inheritance

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : protected Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

Derived Object



public	→	protected
protected	→	protected
private	→	inaccessible

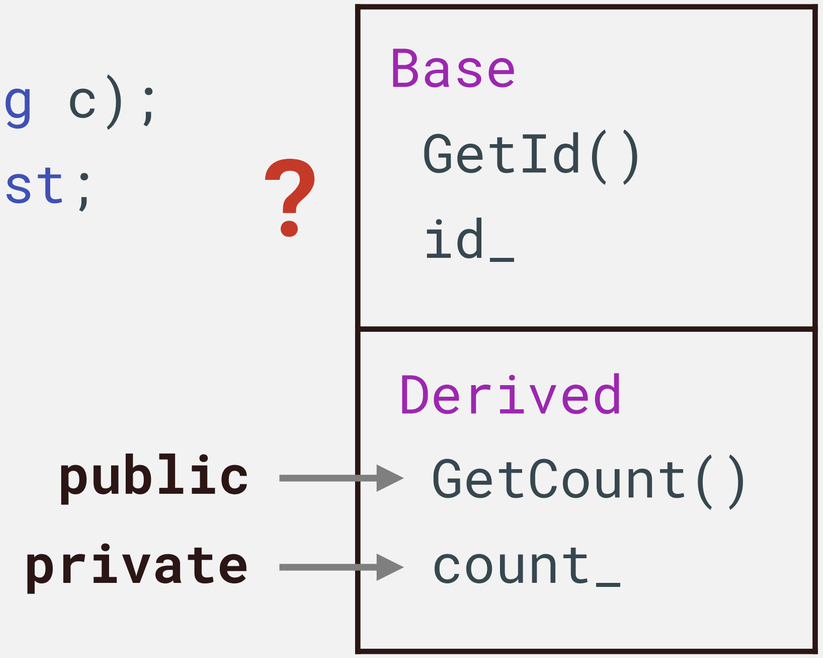
Protected Inheritance

Almost Never

```
class Base {
 public:
  Base(int id);
  int GetId() const;

 protected:
  int id_;
};
```

```
class Derived : protected Base {
 public:
  Derived(int id, long c);
  long GetCount() const;

 private:
  long count_;
};
```

Derived Object



public	→	protected
protected	→	protected
private	→	inaccessible

Redefine Behaviors

```
class Base {
 public:
  Base();
  void Print() const {
    std::cout << “Base\n”;
  }
};
```

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

```
int main() {
  Derived d_obj;
  d_obj.Print();
}
```

Redefine Behaviors

```
class Base {
 public:
  Base();
  void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  Derived();
  void Print() const {
    std::cout << “Derived\n”;
  }
};

int main() {
  Derived d_obj;
  d_obj.Print();
}
```

Redefine Behaviors

```
class Base {
 public:
  Base();
 protected:
  void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  Derived();
  void Print() const {
    std::cout << “Derived\n”;
  }
};

int main() {
  Derived d_obj;
  d_obj.Print();
}
```

Redefine Behaviors

```
class Base {
 public:
  Base();
 protected:
  void Print() const {
    std::cout << “Base\n”;
  }
};
```

public

```
class Derived : public Base {
 public:
  Derived();
  void Print() const {
    std::cout << “Derived\n”;
  }
};

int main() {
  Derived d_obj;
  d_obj.Print();
}
```

Redefine Behaviors

```
class Base {
 public:
  Base();
 protected:
  void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  Derived();
  void Print() const {
    Base::Print();
    std::cout << “Derived\n”;
  }
};
int main() {
  Derived d_obj;
  d_obj.Print();
}
```

Pets Speak!

```
class Pet {
 public:
  Pet(std::string name)
      : name_(name) {}
  void SayHi() const {
    std::cout << name_
      << “says ???\n”;
  }

 protected:
  std::string name_;
};
```

```
class Dog : public Pet {
 public:
  Dog(std::string name) : Pet(name) {}
  void SayHi() const {
    std::cout << name_ << “says WOOF\n”;
  }
};
```

```
class Cat : public Pet {
 public:
  Cat(std::string name) : Pet(name) {}
  void SayHi() const {
    std::cout << name_ << “says MEOW\n”;
  }
};
```

Base Pointer

```
int main() {
  Dog erha(“Erha”);
  Pet *pet = (Pet *)&erha;
  pet->SayHi();
}
```

Base Pointer

```
int main() {
  Dog erha("Erha");
  Pet *pet = (Pet *)&erha;
  pet->SayHi();
}
```

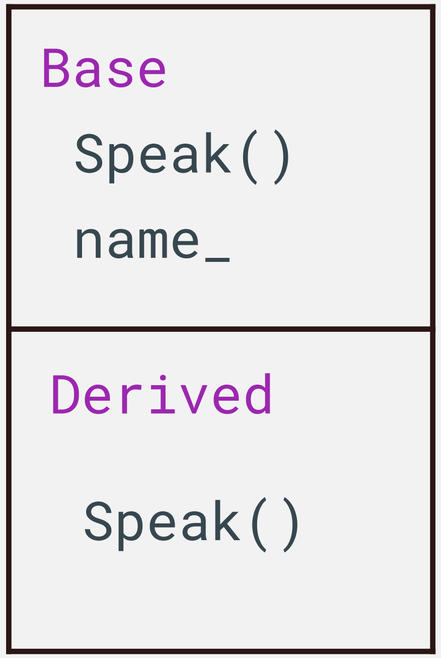
allowed

Base Pointer

```
int main() {
  Dog erha("Erha");
  Pet *pet = (Pet *)&erha;
  pet->SayHi();
}
```

allowed

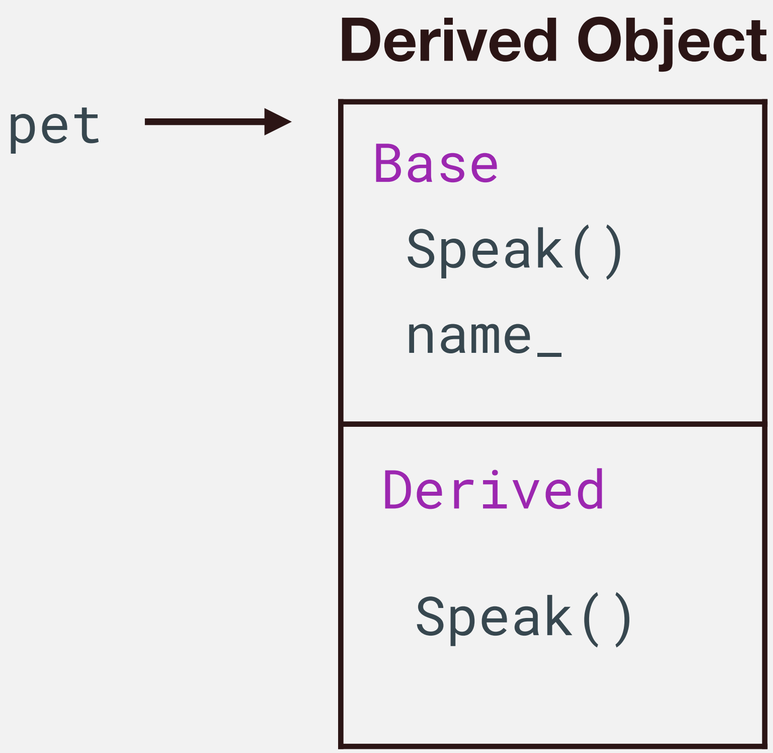
Derived Object



Base Pointer

```
int main() {
  Dog erha("Erha");
  Pet *pet = (Pet *)&erha;
  pet->SayHi();
}
```

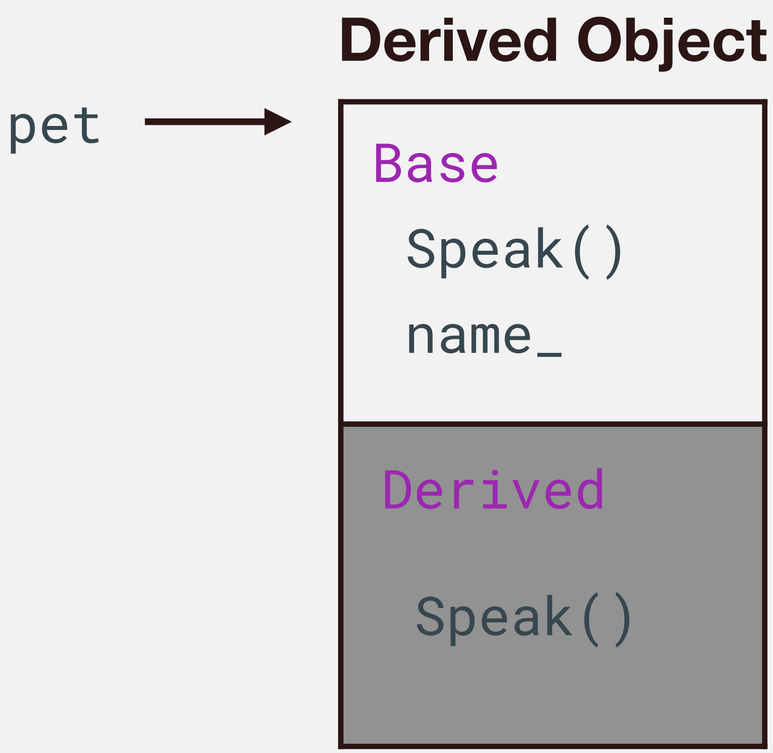
allowed



Base Pointer

```
int main() {
  Dog erha("Erha");
  Pet *pet = (Pet *)&erha;
  pet->SayHi();
}
```

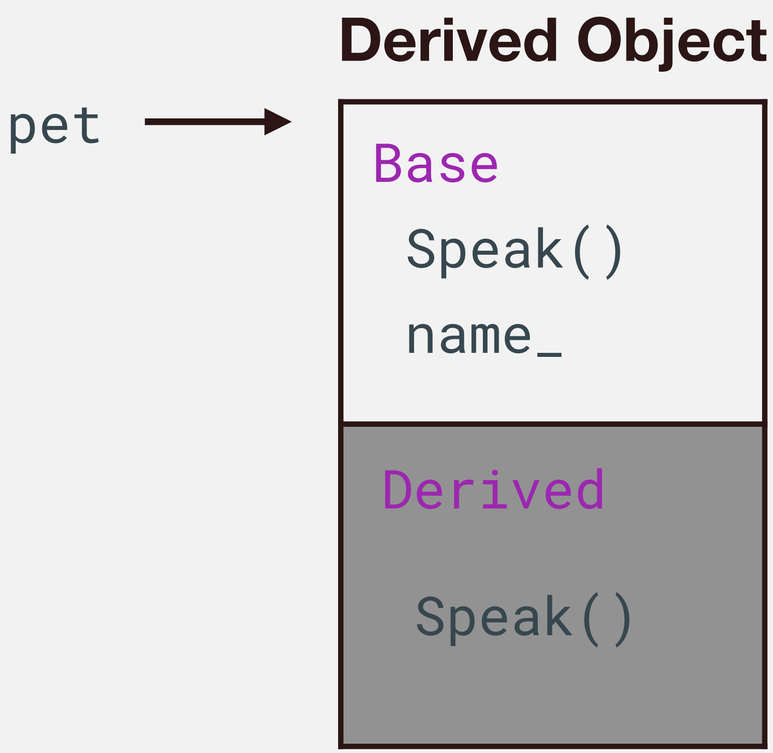
allowed



Base Pointer

```
int main() {
  Dog erha("Erha");
  Pet *pet = (Pet *)&erha;
  pet->SayHi();
}

>> Erha says ???
```

allowed



Pets Speak!

```
int main() {
  Dog erha(“Erha”);
  Dog wangchai(“Wangchai”);
  ...
  Cat mimi(“Mimi”);
  Cat zhaocai(“Zhaocai”);
  ...





}
```

Pets Speak!

```
int main() {
  Dog erha("Erha");
  Dog wangchai("Wangchai");
  ...
  Cat mimi("Mimi");
  Cat zhaocai("Zhaocai");
  ...
  Dog dogs[10] = {erha, wangchai, ...};
  for (int i = 0; i < 10; i++) {
    dogs[i].SayHi();
  }



}
```

Pets Speak!

```
int main() {
  Dog erha("Erha");
  Dog wangchai("Wangchai");
  ...
  Cat mimi("Mimi");
  Cat zhaocai("Zhaocai");
  ...
  Dog dogs[10] = {erha, wangchai, ...};
  for (int i = 0; i < 10; i++) {
    dogs[i].SayHi();
  }
  Cat cats[10] = {mimi, zhaocai, ...};
  for (...)
}
```

Pets Speak!

```
int main() {
  Dog erha("Erha");
  Dog wangchai("Wangchai");
  ...
  Cat mimi("Mimi");
  Cat zhaocai("Zhaocai");
  ...
  Dog dogs[10] = {erha, wangchai, ...};
  for (int i = 0; i < 10; i++) {
    dogs[i].SayHi();
  }
  Cat cats[10] = {mimi, zhaocai, ...};
  for (...)
}
```

```
>> Erha says WOOF
   Wangchai says WOOF
   ...
   Mimi says MEOW
   Zhaocai says MEOW
   ...
```

Pets Speak!

```
int main() {
  Dog erha("Erha");
  Dog wangchai("Wangchai");
  ...
  Cat mimi("Mimi");
  Cat zhaocai("Zhaocai");
  ...
  Dog dogs[10] = {erha, wangchai, ...};
  for (int i = 0; i < 10; i++) {
    dogs[i].SayHi();
  }
  Cat cats[10] = {mimi, zhaocai, ...};
  for (...)
}
```

⋮

What we hope to work …

```
int main() {
  Dog erha("Erha");
  Dog wangchai("Wangchai");
  ...
  Cat mimi("Mimi");
  Cat zhaocai("Zhaocai");
  ...
  Pet *pets[20] = {&erha, &mimi, &wangchai, &zhaocai, ...};
  for (int i = 0; i < 20; i++) {
    pets[i]->SayHi();
  }
}
```

🐶 🐱 🐶 🐱 ...

What we hope to work …

```
int main() {
  Dog erha("Erha");
  Dog wangchai("Wangchai");
  ...
  Cat mimi("Mimi");
  Cat zhaocai("Zhaocai");
  ...
  Pet *pets[20] = {&erha, &mimi, &wangchai, &zhaocai, ...};
  for (int i = 0; i < 20; i++) {
    pets[i]->SayHi();
  }
}
```

🐶 🐱 🐶 🐱 ...

```
>> Erha says ???
   Mimi says ???
   Wangchai says ???
   Zhaocai says ???
   ...
```

Virtual Functions

```
class Base {
 public:
  void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  void Print() const {
    std::cout << “Derived\n”;
  }
};

int main() {
  Derived d_obj;
  Base *b_ptr = (Base *)&d_obj;
  b_ptr->Print();
}
```

Virtual Functions

```
class Base {
 public:
  void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  void Print() const {
    std::cout << “Derived\n”;
  }
};

int main() {
  Derived d_obj;
  Base *b_ptr = (Base *)&d_obj;
  b_ptr->Print();
}
```

```
>> Base
```

Virtual Functions

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  void Print() const override {
    std::cout << “Derived\n”;
  }
};

int main() {
  Derived d_obj;
  Base *b_ptr = (Base *)&d_obj;
  b_ptr->Print();
}
```

Virtual Functions

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

Find the most-derived version of the function to execute

```
class Derived : public Base {
 public:
  void Print() const override {
    std::cout << “Derived\n”;
  }
};

int main() {
  Derived d_obj;
  Base *b_ptr = (Base *)&d_obj;
  b_ptr->Print();
}
```

Virtual Functions

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

Find the most-derived version of the function to execute

```
class Derived : public Base {
 public:
  void Print() const override {
    std::cout << “Derived\n”;
  }
};
```

Since C++11

```
int main() {
  Derived d_obj;
  Base *b_ptr = (Base *)&d_obj;
  b_ptr->Print();
}
```

Virtual Functions

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

Find the most-derived version of the function to execute

```
class Derived : public Base {
 public:
  void Print() const override {
    std::cout << “Derived\n”;
  }
};
```

Since C++11

```
int main() {
  Derived d_obj;
  Base *b_ptr = (Base *)&d_obj;
  b_ptr->Print();
}
```

```
>> Derived
```

Virtual Functions

```
class A {
 public:
  virtual void Print() const { std::cout << “A\n”; } };

class B : public A {
 public:
  void Print() const override { std::cout << “B\n”; } };

class C : public B {
 public:
  void Print() const override { std::cout << “C\n”; } };

class D : public C {
 public:
  void Print() const override { std::cout << “D\n”; } };
```

```
int main() {
  C c_obj;
  A *a_ptr = (A *)&c_obj;
  a_ptr->Print();
}
```

Virtual Functions

```
class A {
 public:
  virtual void Print() const { std::cout << “A\n”; } };

class B : public A {
 public:
  void Print() const override { std::cout << “B\n”; } };

class C : public B {
 public:
  void Print() const override { std::cout << “C\n”; } };

class D : public C {
 public:
  void Print() const override { std::cout << “D\n”; } };
```

```
int main() {
  C c_obj;
  A *a_ptr = (A *)&c_obj;
  a_ptr->Print();
}

>> C
```

Destructors under Inheritance

```
class Base {
 public:
  Base(int id) : id_(id) {}
  ~Base() {}
 private:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int len) : a_() {
    a_ = new int[len];
  }
  ~Derived() { delete[] a_; }
 private:
  int *a_;
};
```

Destructors under Inheritance

```
class Base {
 public:
  Base(int id) : id_(id) {}
  ~Base() {}
 private:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int len) : a_() {
    a_ = new int[len];
  }
  ~Derived() { delete[] a_; }
 private:
  int *a_;
};
```

```
int main() {
  Base *b_ptr = (Base *)(new Derived(10));
  delete b_ptr;
}
```

Destructors under Inheritance

```
class Base {
 public:
  Base(int id) : id_(id) {}
➜ ~Base() {}
 private:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int len) : a_() {
    a_ = new int[len];
  }
  ~Derived() { delete[] a_; }
 private:
  int *a_;
};
```

```
int main() {
  Base *b_ptr = (Base *)(new Derived(10));
  delete b_ptr;
}
```

Destructors under Inheritance

```
class Base {
 public:
  Base(int id) : id_(id) {}
➜ ~Base() {}
 private:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int len) : a_() {
    a_ = new int[len];
  }
  ~Derived() { delete[] a_; }
 private:
  int *a_;
};
```

```
int main() {
  Base *b_ptr = (Base *)(new Derived(10));
  delete b_ptr;
}
```

Memory Leak!

Destructors under Inheritance

```
class Base {
 public:
  Base(int id) : id_(id) {}
  virtual ~Base() {}
 private:
  int id_;
};
```

```
class Derived : public Base {
 public:
  Derived(int len) : a_() {
    a_ = new int[len];
  }
➜ ~Derived() override { delete[] a_; }
 private:
  int *a_;
};
```

```
int main() {
  Base *b_ptr = (Base *)(new Derived(10));
  delete b_ptr;
}
```

Destructors under Inheritance

```
class Base {
 public:
  Base(int id) : id_(id) {}
  virtual ~Base() {}
 private:
  int id_;
};


int main() {
  Base *b_ptr = (Base *)(new Derived(10));
  delete b_ptr;
}
```

```
class Derived : public Base {
 public:
  Derived(int len) : a_() {
    a_ = new int[len];
  }
➜ ~Derived() override { delete[] a_; }
 private:
  int *a_;
};
```

Always make destructors virtual when dealing with inheritance

Overrides

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  void Print() const override {
    std::cout << “Derived\n”;
  }
};
```

- Declare Intention
- Keep Function Virtual

Overrides

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  void PrintOut() const override {
    std::cout << “Derived\n”;
  }
};
```

Overrides

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  void PrintOut() const override {
    std::cout << “Derived\n”;
  }
};
```

Compile Error

Overrides

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  void Print() override {
    std::cout << “Derived\n”;
  }
};
```

Compile Error

Final

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived final : public Base {
 public:
  void Print() const override {
    std::cout << “Derived\n”;
  }
};
```

Final

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived final : public Base {
 public:
  void Print() const override {
    std::cout << “Derived\n”;
  }
};
```

Prevent others from inheriting this class

Final

```
class Base {
 public:
  virtual void Print() const {
    std::cout << “Base\n”;
  }
};
```

```
class Derived : public Base {
 public:
  void Print() const override final {
    std::cout << “Derived\n”;
  }
};
```

Prevent derived classes to override this function

Pets Speak! — Revisit

```
class Pet {
 public:
  Pet(std::string name)
      : name_(name) {}
  void SayHi() const {
    std::cout << name_
      << "says ???\n";
  }

 protected:
  std::string name_;
};
```

```
class Dog : public Pet {
 public:
  Dog(std::string name) : Pet(name) {}
  void SayHi() const {
    std::cout << name_ << "says WOOF\n";
  }
};
```

```
class Cat : public Pet {
 public:
  Cat(std::string name) : Pet(name) {}
  void SayHi() const {
    std::cout << name_ << "says MEOW\n";
  }
};
```

Pets Speak! — Revisit

```
class Pet {
 public:
  Pet(std::string name)
      : name_(name) {}
  virtual void SayHi() const {
    std::cout << name_
      << "says ???\n";
  }

 protected:
  std::string name_;
};
```

```
class Dog : public Pet {
 public:
  Dog(std::string name) : Pet(name) {}
  void SayHi() const override final {
    std::cout << name_ << "says WOOF\n";
  }
};
```

```
class Cat : public Pet {
 public:
  Cat(std::string name) : Pet(name) {}
  void SayHi() const override final {
    std::cout << name_ << "says MEOW\n";
  }
};
```

Pets Speak! — Revisit

```
int main() {
  Dog erha("Erha");
  Dog wangchai("Wangchai");
  ...
  Cat mimi("Mimi");
  Cat zhaocai("Zhaocai");
  ...
  Pet *pets[20] = {&erha, &mimi, &wangchai, &zhaocai, ...};
  for (int i = 0; i < 20; i++) {
    pets[i]->SayHi();
  }
}
```

🐶 🐱 🐶 🐱 ...

```
>> Erha says WOOF
   Mimi says MEOW
   Wangchai says WOOF
   Zhaocai says MEOW
   ...
```

Pets Speak! — Revisit

```
int main() {
  Dog erha("Erha");
  Dog wangchai("Wangchai");
  ...
  Cat mimi("Mimi");
  Cat zhaocai("Zhaocai");
  ...
  Pet *pets[20] = {&erha, &mimi, &wangchai, &zhaocai, ...};
  for (int i = 0; i < 20; i++) {
    pets[i]->SayHi();
  }
}
```

🐶 🐱 🐶 🐱 ...

```
>> Erha says WOOF
   Mimi says MEOW
   Wangchai says WOOF
   Zhaocai says MEOW
   ...
```

Polymorphism

Polymorphism

`poly • morphism`

many forms

Polymorphism

poly • morphism

many forms

```
int main() {
  ...
  Pet *pets[20] = {&erha, &mimi, &wangchai, &zhaocai, ...};
  for (int i = 0; i < 20; i++) {
    pets[i]->SayHi();
  }
}
```

Polymorphism

poly • morphism

many forms

```
int main() {
  ...
  Pet *pets[20] = {&erha, &mimi, &wangchai, &zhaocai, ...};
  for (int i = 0; i < 20; i++) {
    pets[i]->SayHi();
  }
}
```

Which function definition to execute is determined at **runtime**

Binding

- Map **identifiers** (e.g., variable, function names) to memory **addresses**

Binding

➔ Map **identifiers** (e.g., variable, function names) to memory **addresses**

Address Space

```
0x305C   void func() {
         ...
              ⋮
➔ 0x7A80 int main() {
           func();
         }
```

Binding

➔ Map **identifiers** (e.g., variable, function names) to memory **addresses**

Address Space

0x305C	`void func() {`
	`...`
	⋮
➔ 0x7A80	`int main() {`
	`  func();` 🔗 0x305C
	`}`

Binding

➔ Map **identifiers** (e.g., variable, function names) to memory **addresses**

Early Binding

- Binding is completed at **compile time**

Address Space

```
0x305C   void func() {
         ...
                ⋮
➔ 0x7A80 int main() {
           func(); 🔗 0x305C
         }
```

Binding

➜ Map **identifiers** (e.g., variable, function names) to memory **addresses**

Early Binding

- Binding is completed at **compile time**

Late Binding

- Binding happens at **runtime**
- Impossible to know which specific function will be called at compile time

Address Space

```
0x305C   void func() {
         ...
                ⋮
➜ 0x7A80 int main() {
           func(); 🔗 0x305C
         }
```

Function Pointer

- A type of pointer containing the address of a function

Function Pointer

➜ A type of pointer containing the address of a function

```
void funcA(int c) { std::cout << “A” << c; }
void funcB(int c) { std::cout << “B” << c; }

int main() {
  int op;
  std::cin >> op;
  void (*f_ptr)(int) = nullptr;
  switch (op) {
    case 0: f_ptr = funcA; break;
    case 1: f_ptr = funcB; break;
  }
  f_ptr(10);
}
```

Function Pointer

➜ A type of pointer containing the address of a function

```
void funcA(int c) { std::cout << “A” << c; }
void funcB(int c) { std::cout << “B” << c; }

int main() {
  int op;
  std::cin >> op;
  void (*f_ptr)(int) = nullptr;
  switch (op) {
    case 0: f_ptr = funcA; break;
    case 1: f_ptr = funcB; break;
  }
  f_ptr(10);
}
```

← Create a function pointer

Function Pointer

➜ A type of pointer containing the address of a function

```
void funcA(int c) { std::cout << "A" << c; }
void funcB(int c) { std::cout << "B" << c; }

int main() {
  int op;
  std::cin >> op;
  void (*f_ptr)(int) = nullptr;
  switch (op) {
    case 0: f_ptr = funcA; break;
    case 1: f_ptr = funcB; break;
  }
  f_ptr(10);
}
```

← Create a function pointer

Late Binding

Function Pointer

➜ A type of pointer containing the address of a function

```
void funcA(int c) { std::cout << "A" << c; }
void funcB(int c) { std::cout << "B" << c; }

int main() {
  int op;
  std::cin >> op;
  void (*f_ptr)(int) = nullptr;
  switch (op) {
    case 0: f_ptr = funcA; break;
    case 1: f_ptr = funcB; break;
  }
  f_ptr(10);
}
```

← Create a function pointer

Late Binding

← Call the bound function

How does polymorphism work?

```
class Base {
 public:
  virtual void funcA() {
    Base_impl_A
  }
  virtual void funcB() {
    Base_impl_B
  }
};
```

```
class D1 : public Base {
 public:
  void funcA() override {
    D1_impl_A
  }
};

class D2 : public Base {
 public:
  void funcB() override {
    D2_impl_B
  }
};
```

How does polymorphism work?

```
class Base {
 public:
  virtual void funcA() {
    Base_impl_A
  }
  virtual void funcB() {
    Base_impl_B
  }
  VirtualTable *__vtable;
};
```

Hidden Pointer

```
class D1 : public Base {
 public:
  void funcA() override {
    D1_impl_A
  }
};

class D2 : public Base {
 public:
  void funcB() override {
    D2_impl_B
  }
};
```

How does polymorphism work?

```
class Base {
 public:
  virtual void funcA() {
    Base_impl_A
  }
  virtual void funcB() {
    Base_impl_B
  }
  VirtualTable *__vtable;
};
```

Virtual Tables

Base
*funcA
*funcB

D1
*funcA
*funcB

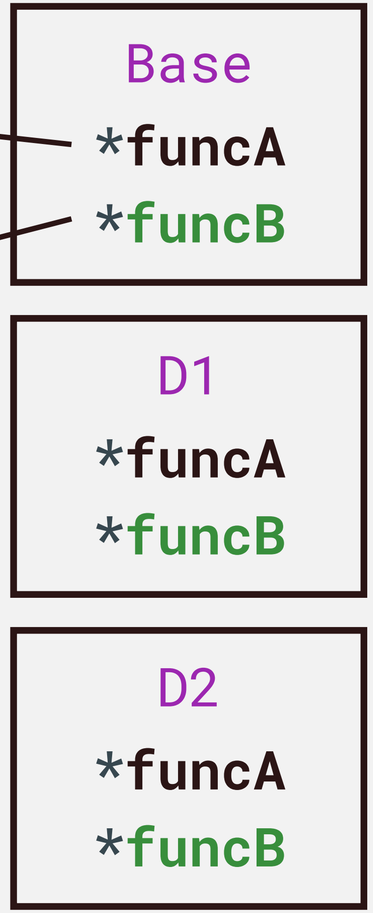
D2
*funcA
*funcB

```
class D1 : public Base {
 public:
  void funcA() override {
    D1_impl_A
  }
};

class D2 : public Base {
 public:
  void funcB() override {
    D2_impl_B
  }
};
```

How does polymorphism work?

```cpp
class Base {
 public:
  virtual void funcA() {
    Base_impl_A
  }
  virtual void funcB() {
    Base_impl_B
  }
  VirtualTable *__vtable;
};
```

Virtual Tables

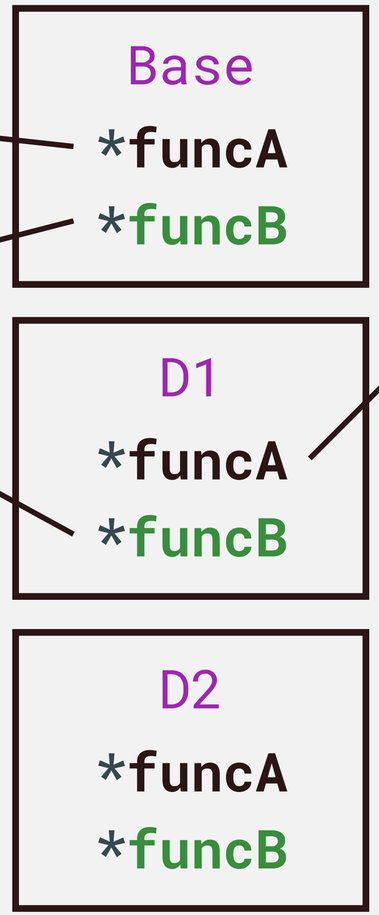



```cpp
class D1 : public Base {
 public:
  void funcA() override {
    D1_impl_A
  }
};

class D2 : public Base {
 public:
  void funcB() override {
    D2_impl_B
  }
};
```

How does polymorphism work?

```
class Base {
 public:
  virtual void funcA() {
    Base_impl_A
  }
  virtual void funcB() {
    Base_impl_B
  }
  VirtualTable *__vtable;
};
```

Virtual Tables

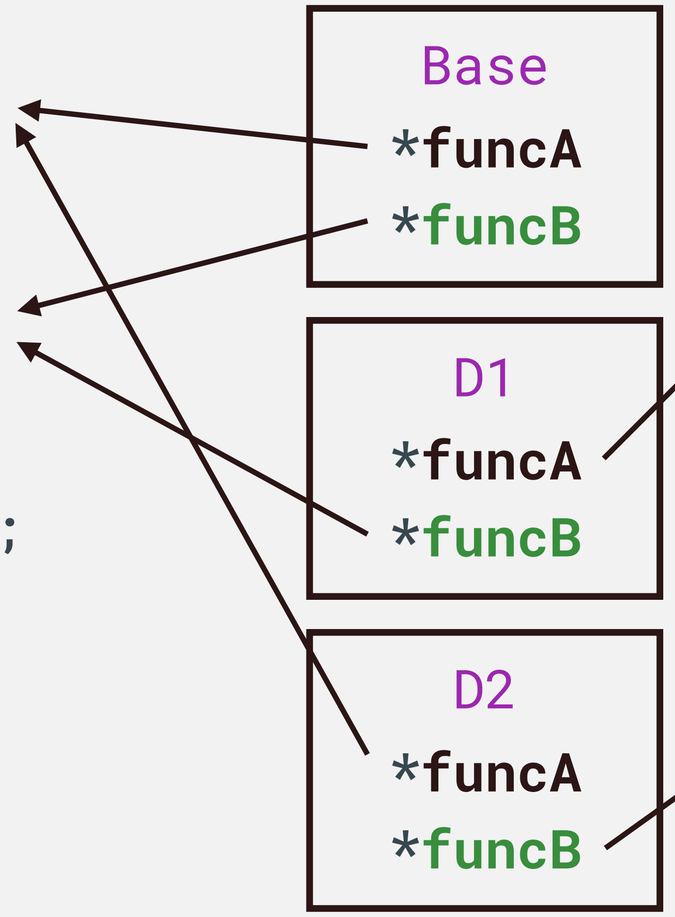



```
class D1 : public Base {
 public:
  void funcA() override {
    D1_impl_A
  }
};

class D2 : public Base {
 public:
  void funcB() override {
    D2_impl_B
  }
};
```

How does polymorphism work?

```
class Base {
 public:
  virtual void funcA() {
    Base_impl_A
  }
  virtual void funcB() {
    Base_impl_B
  }
  VirtualTable *__vtable;
};
```

Virtual Tables




```
class D1 : public Base {
 public:
  void funcA() override {
    D1_impl_A
  }
};

class D2 : public Base {
 public:
  void funcB() override {
    D2_impl_B
  }
};
```

How does polymorphism work?

```
int main() {
  D1 d1_obj;



}
```

```
void funcA() {
  Base_impl_A
}

void funcB() {
  Base_impl_B
}

void funcA() {
  D1_impl_A
}
```

How does polymorphism work?

```
int main() {
  D1 d1_obj;


}
```

D1 Object

```
Base
 ...
 *__vtable
```

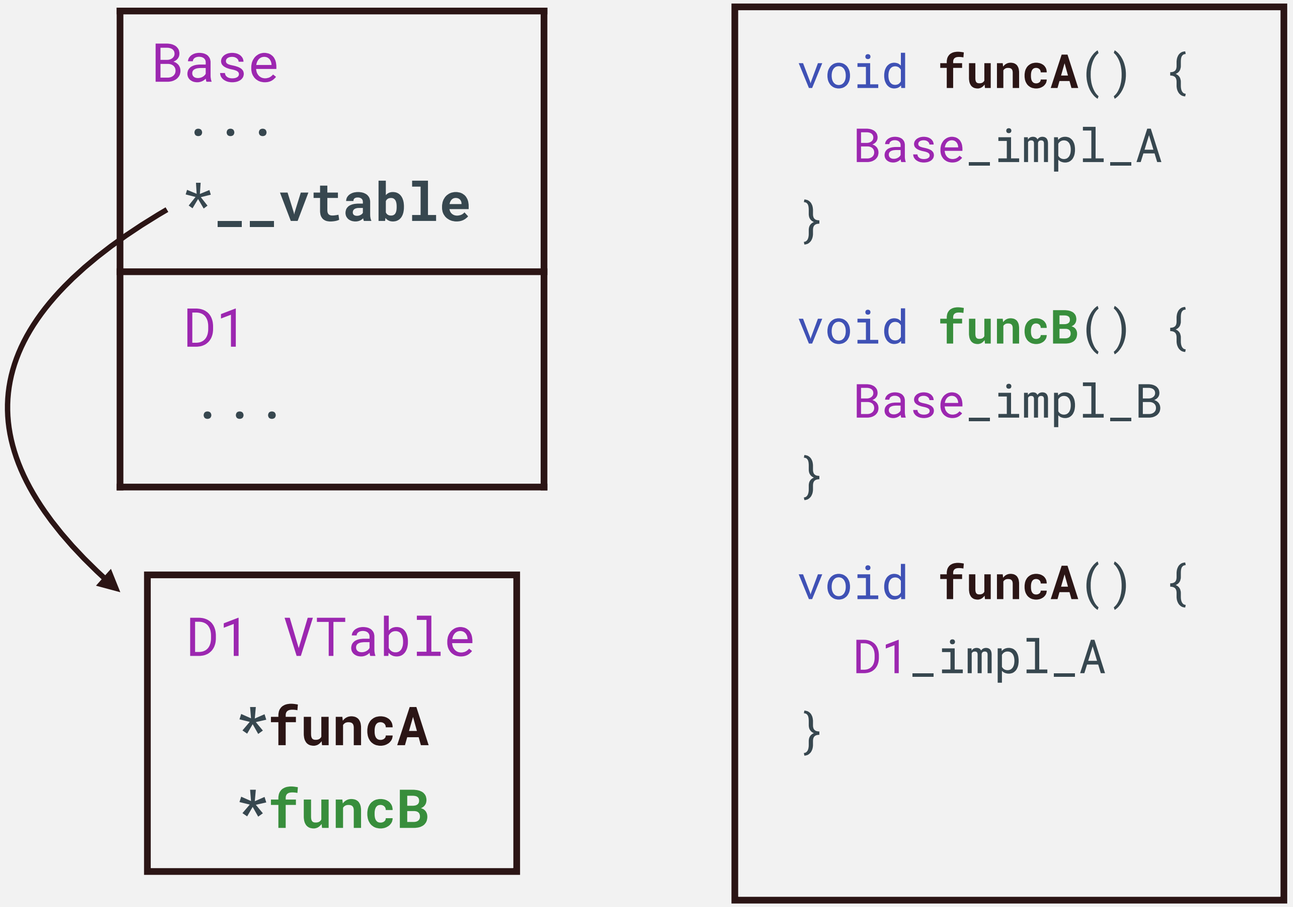
```
D1
 ...
```

```
void funcA() {
  Base_impl_A
}

void funcB() {
  Base_impl_B
}

void funcA() {
  D1_impl_A
}
```

How does polymorphism work?

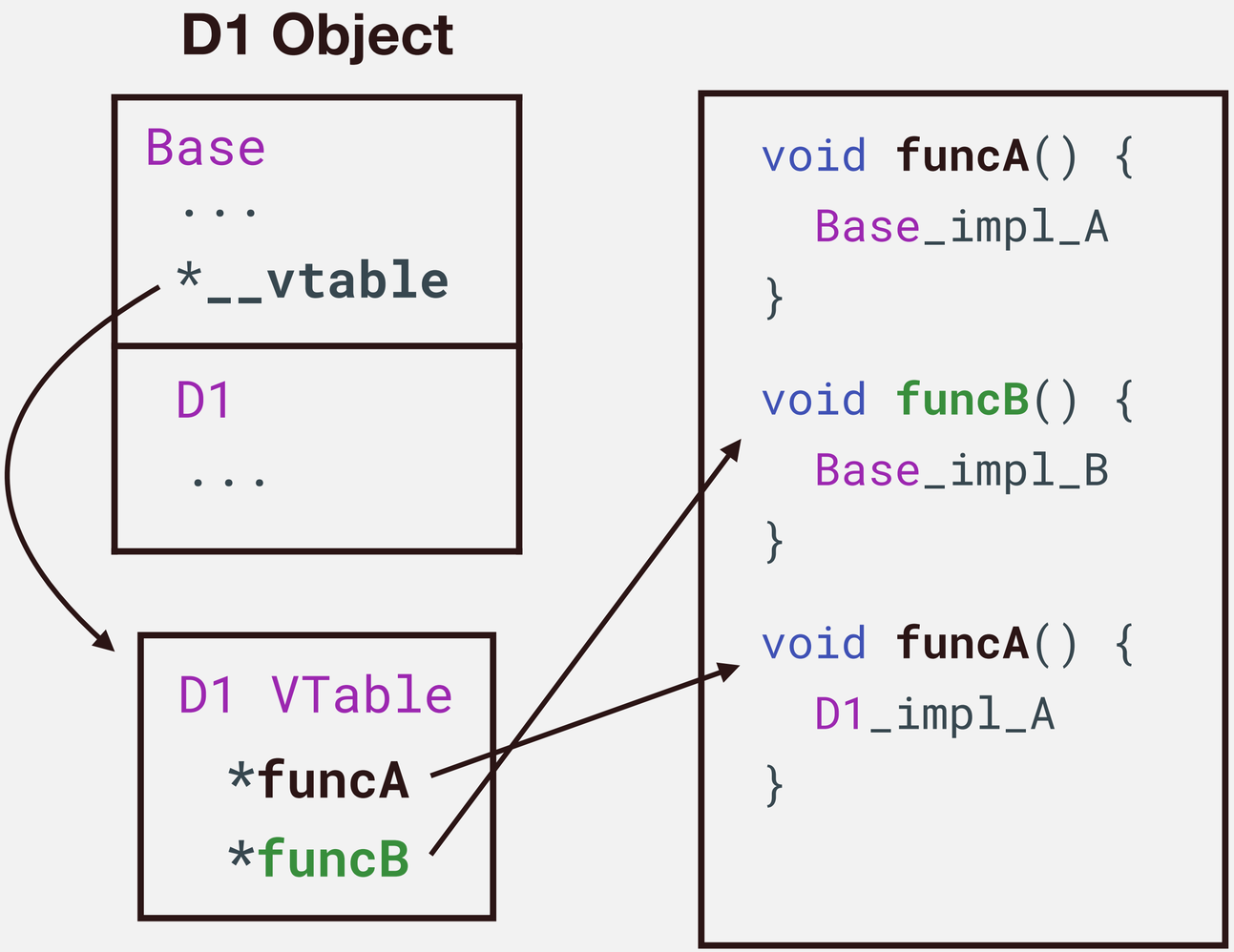
```
int main() {
  D1 d1_obj;


}
```

D1 Object



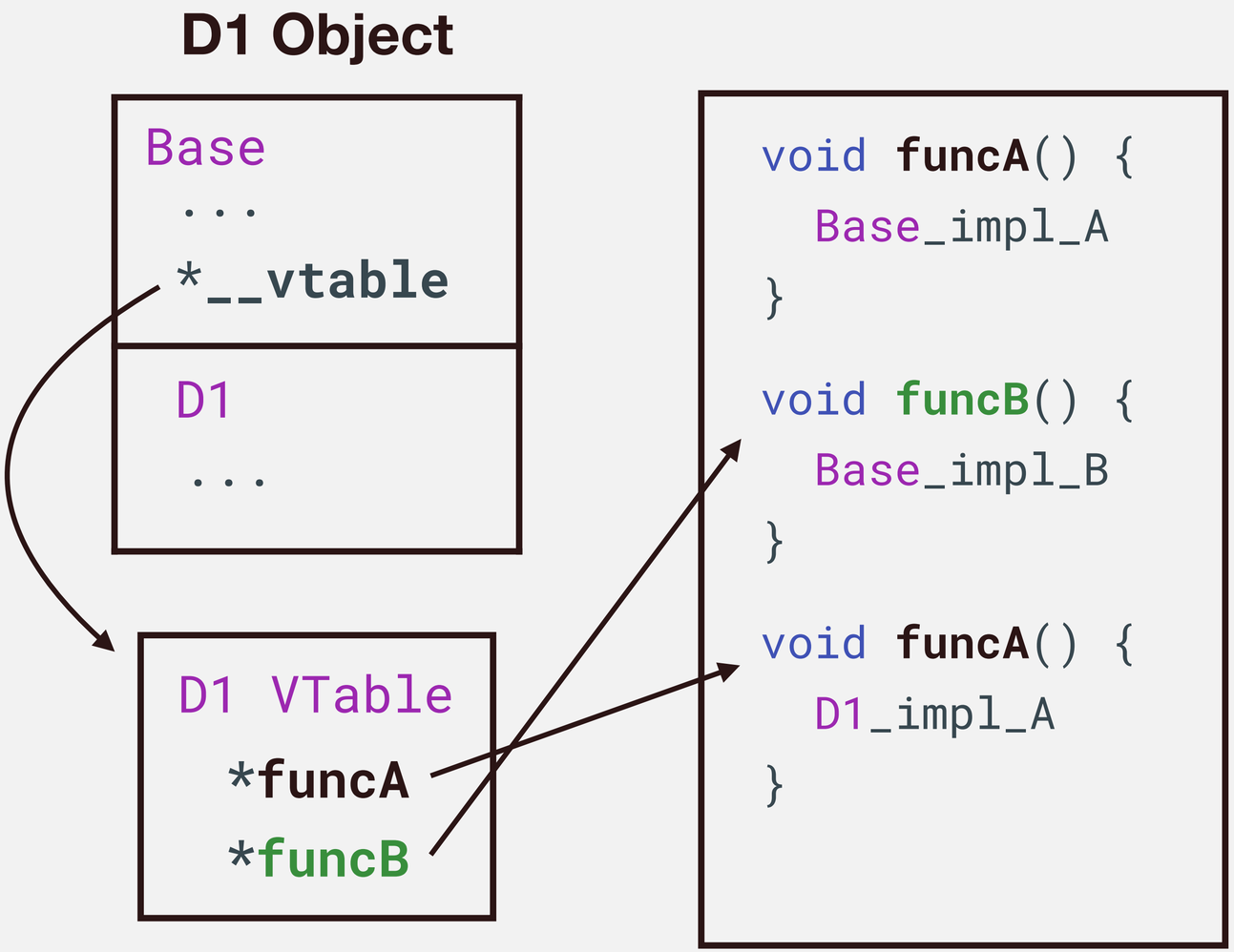
How does polymorphism work?

```
int main() {
  D1 d1_obj;


}
```



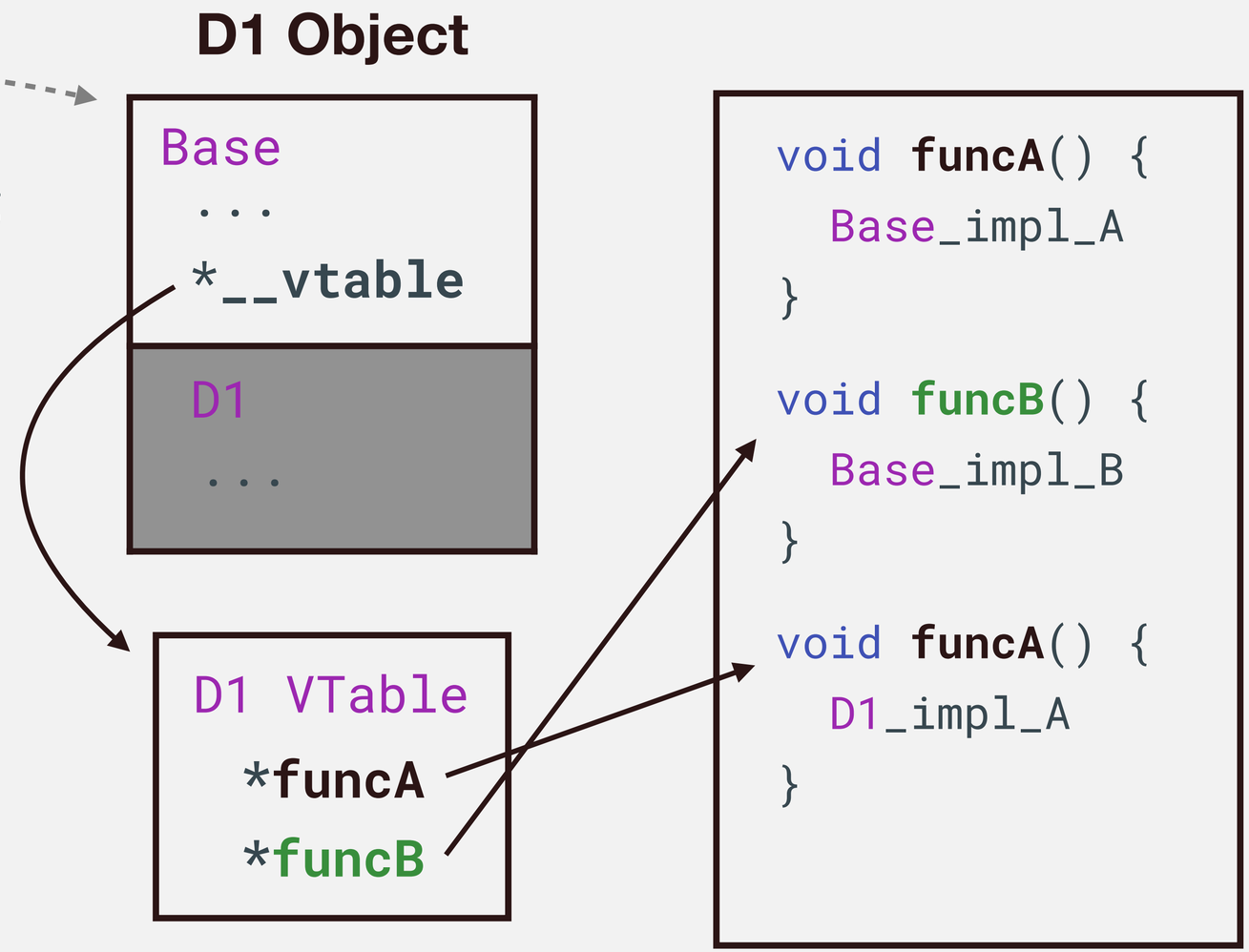
How does polymorphism work?

```
int main() {
  D1 d1_obj;
  Base *b_ptr = (Base *)&d1_obj;

}
```



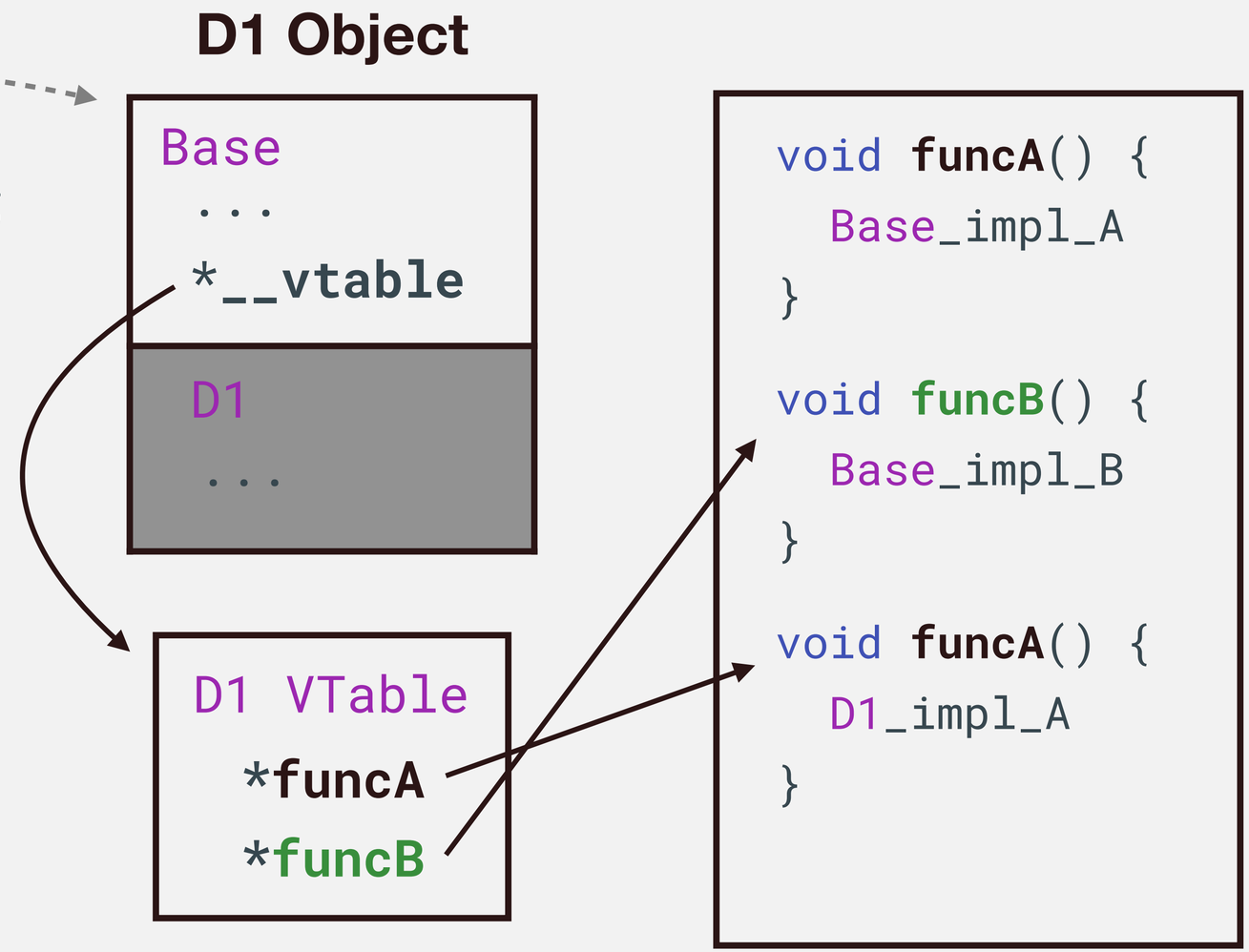
How does polymorphism work?

```
int main() {
  D1 d1_obj;
  Base *b_ptr = (Base *)&d1_obj;

}
```



How does polymorphism work?

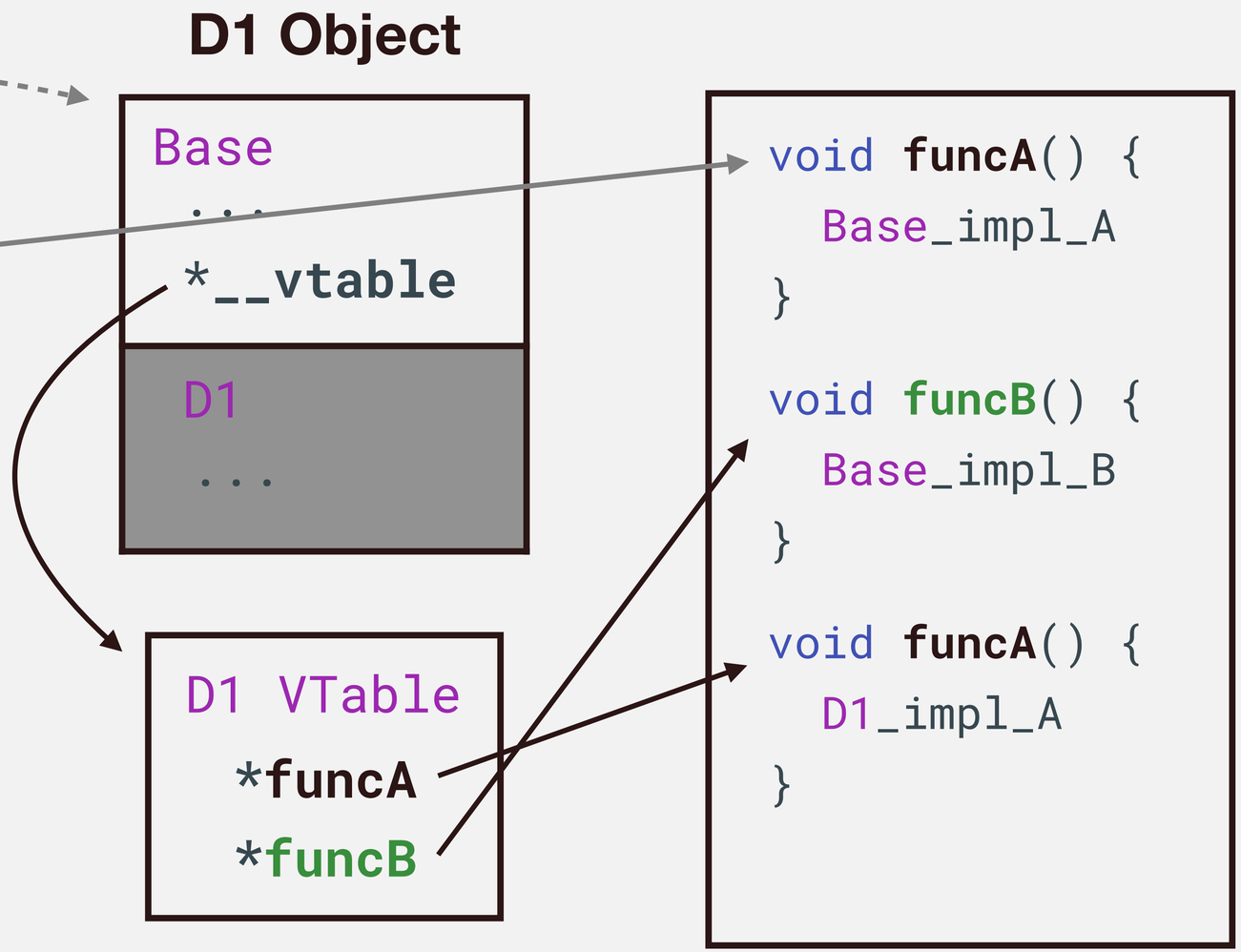
```
int main() {
  D1 d1_obj;
  Base *b_ptr = (Base *)&d1_obj;
  b_ptr->funcA();
}
```



How does polymorphism work?

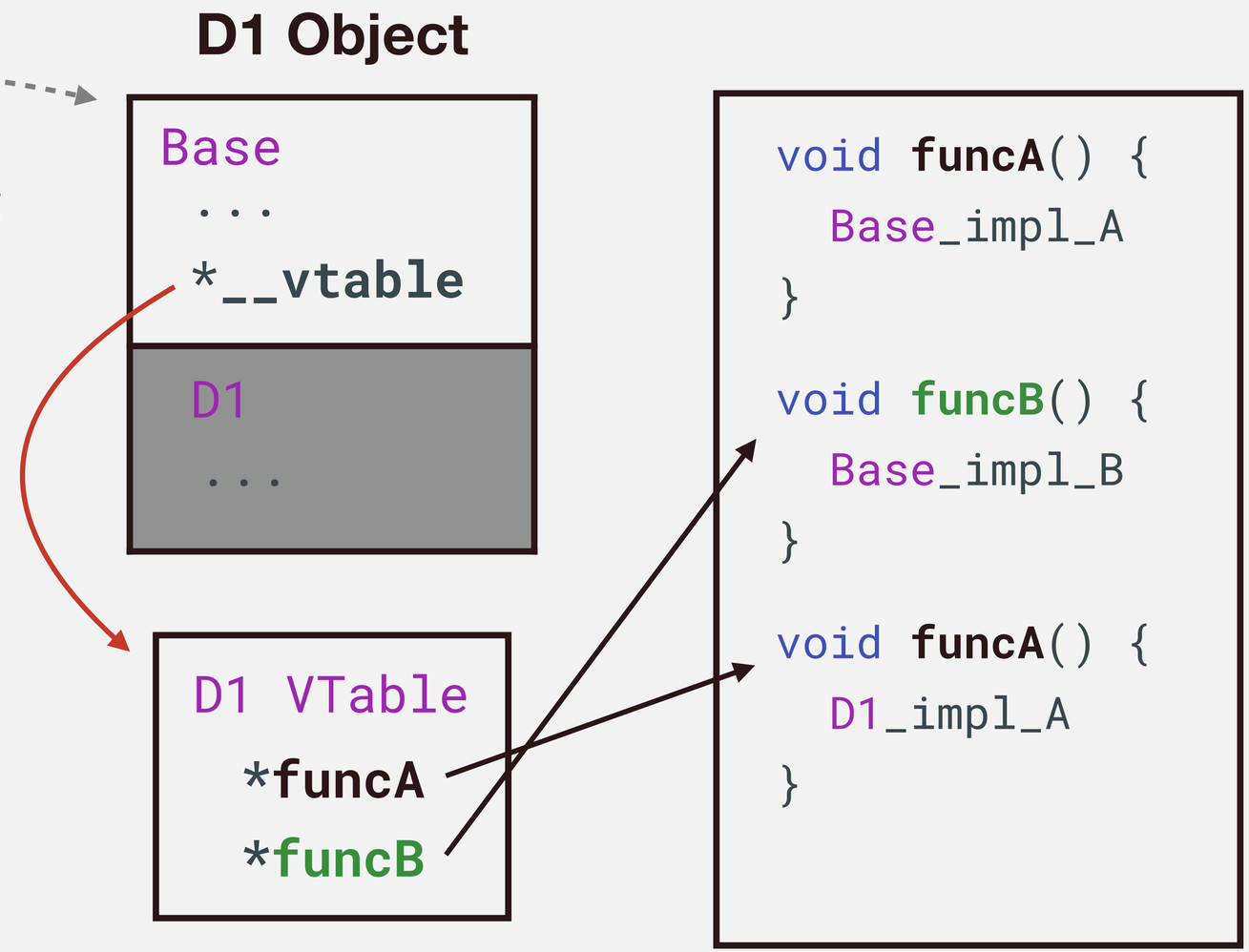
```
int main() {
  D1 d1_obj;
  Base *b_ptr = (Base *)&d1_obj;
  b_ptr->funcA();
}
```

If Early Binding



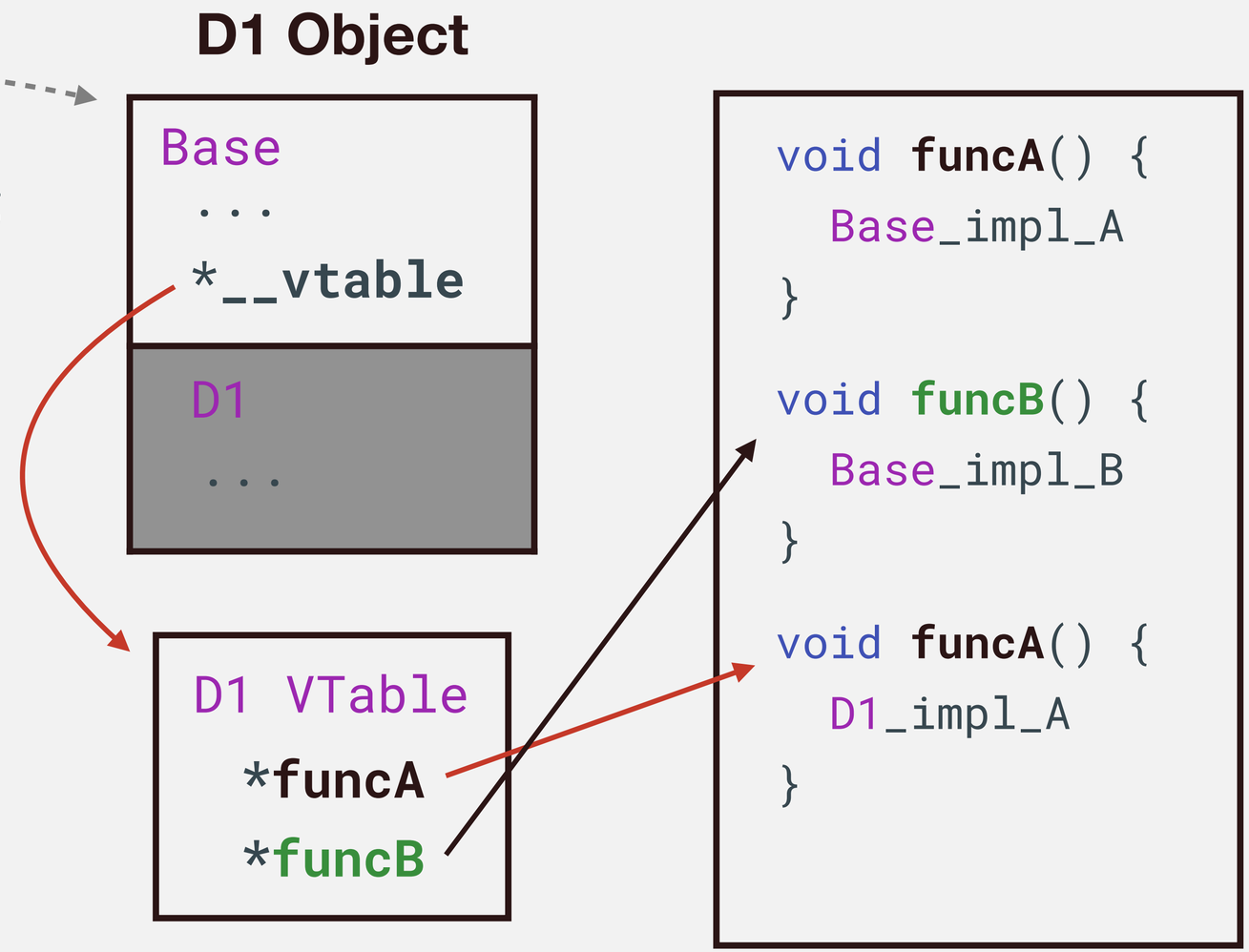
How does polymorphism work?

```
int main() {
  D1 d1_obj;
  Base *b_ptr = (Base *)&d1_obj;
  b_ptr->funcA();
}
```



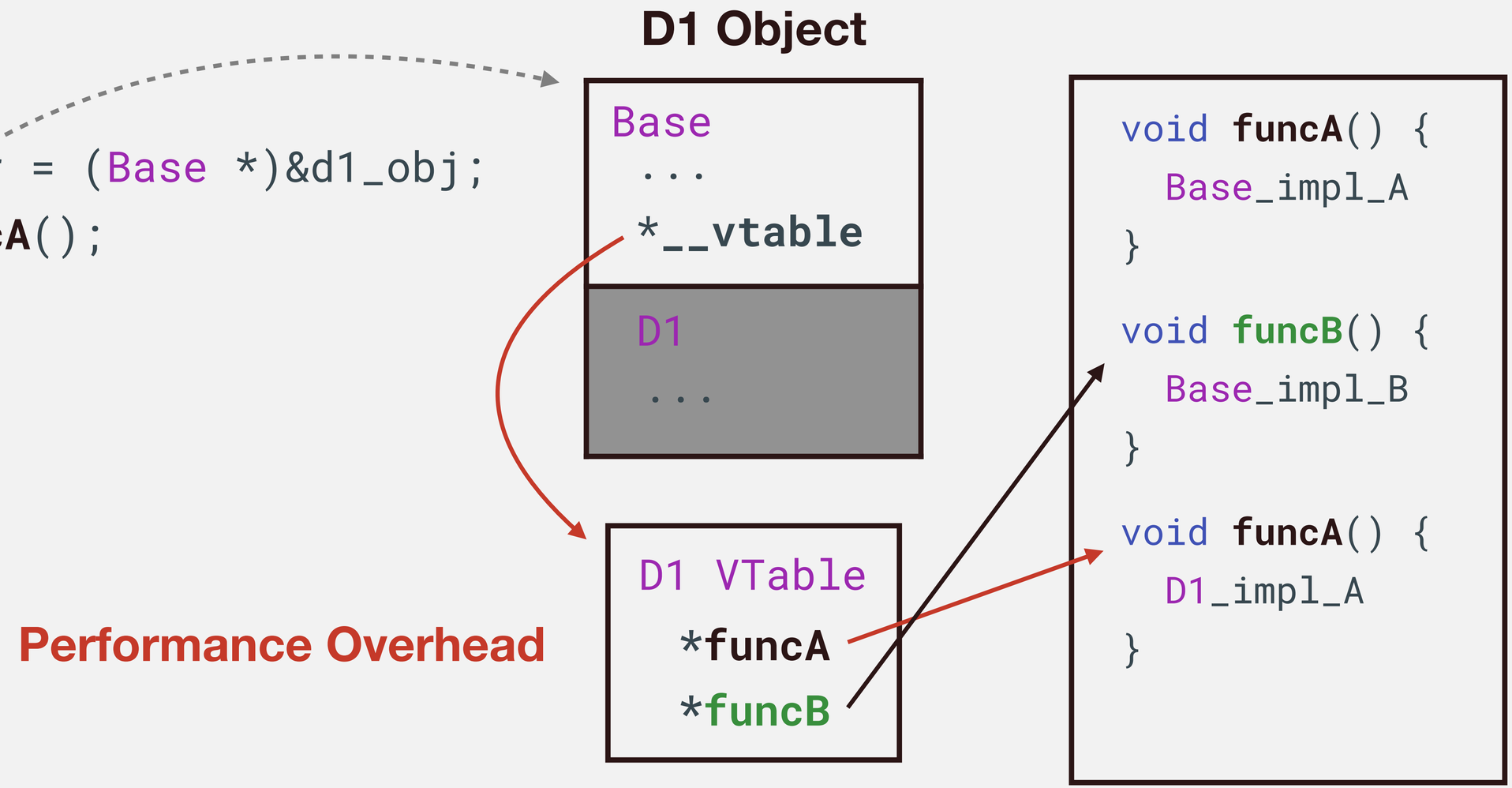
How does polymorphism work?

```
int main() {
  D1 d1_obj;
  Base *b_ptr = (Base *)&d1_obj;
  b_ptr->funcA();
}
```



How does polymorphism work?

```
int main() {
  D1 d1_obj;
  Base *b_ptr = (Base *)&d1_obj;
  b_ptr->funcA();
}
```



Performance Overhead

Pure Virtual Functions

```
class Pet {
 public:
  Pet(std::string name) : name_(name) {}
  virtual void SayHi() const {
    std::cout << name_ << “says ???\n”;
  }

 protected:
  std::string name_;
};
```

Pure Virtual Functions

```
class Pet {
 public:
  Pet(std::string name) : name_(name) {}
  virtual void SayHi() const = 0;

 protected:
  std::string name_;
};
```

Pure Virtual Functions

```
class Pet {
 public:
  Pet(std::string name) : name_(name) {}
  virtual void SayHi() const = 0;

 protected:
  std::string name_;
};
```

1 Becomes an **abstract** class, cannot be instantiated

Pure Virtual Functions

```
class Pet {
 public:
  Pet(std::string name) : name_(name) {}
  virtual void SayHi() const = 0;

 protected:
  std::string name_;
};
```

1. Becomes an **abstract** class, cannot be instantiated
2. Derived classes **must override** this function

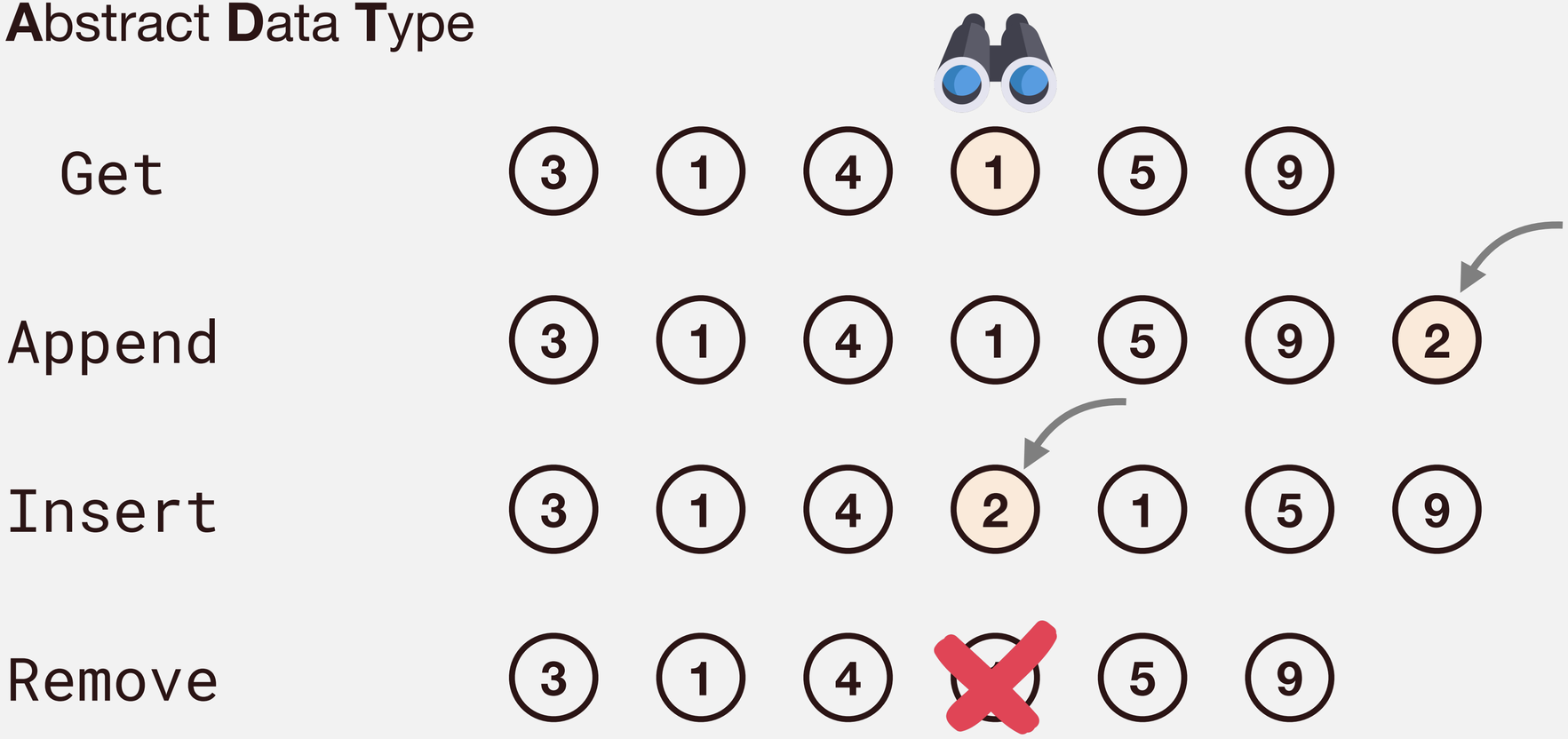
Interface Classes

➜ No data member, all virtual functions

```
class Champion {
 public:
  virtual bool Attack(Champion *enemy) = 0;
  virtual bool Defend() = 0;
  virtual bool Move(int x, int y) = 0;
  virtual int Heal() = 0;
  virtual ~Champion() {}
};
```

ListADT

➜ **ADT** = **A**bstract **D**ata **T**ype



ListADT

```
class List {
 public:
  virtual void Append(int val) = 0;
  virtual bool Insert(int pos, int val) = 0;
  virtual bool Remove(int pos) = 0;
  virtual bool Get(int pos, int &val) const = 0;
  virtual bool GetSize() const = 0;
  virtual ~List() {}
};
```

Summary

- **Inheritance**
  - To reduce code duplication
  - “is-a”, protected, construction order, overwrite base functions
- **Polymorphism**
  - Avoid distinguish object types for common interfaces
  - Late/Dynamic binding, virtual functions, virtual table

Templates

Motivation For Templates

```
int max(int a, int b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max(2, 3);
}
```

Motivation For Templates

```
int max(int a, int b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
}
```

Motivation For Templates

```
int max(int a, int b) {
  return (a > b) ? a : b;
}

double max(double a, double b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
}
```

Motivation For Templates

```
int max(int a, int b) {
  return (a > b) ? a : b;
}

double max(double a, double b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
  Vec a_vec(2, 3);
  Vec b_vec(3, 4);
  Vec c_vec = max(a_vec, b_vec);
}
```

Motivation For Templates

```
int max(int a, int b) {
  return (a > b) ? a : b;
}

double max(double a, double b) {
  return (a > b) ? a : b;
}

Vec max(Vec a, Vec b) {
  return (a > b) ? a : b;
}
```

Overloaded

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
  Vec a_vec(2, 3);
  Vec b_vec(3, 4);
  Vec c_vec = max(a_vec, b_vec);
}
```

Motivation For Templates

```
int max(int a, int b) {
  return (a > b) ? a : b;
}

double max(double a, double b) {
  return (a > b) ? a : b;
}

Vec max(Vec a, Vec b) {
  return (a > b) ? a : b;
}
```

⋮

Overloaded

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
  Vec a_vec(2, 3);
  Vec b_vec(3, 4);
  Vec c_vec = max(a_vec, b_vec);
}
```

Motivation For Templates

```
int max(int a, int b) {
  return (a > b) ? a : b;
}

double max(double a, double b) {
  return (a > b) ? a : b;
}

Vec max(Vec a, Vec b) {
  return (a > b) ? a : b;
}
```

Overloaded

⋮

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
  Vec a_vec(2, 3);
  Vec b_vec(3, 4);
  Vec c_vec = max(a_vec, b_vec);
}
```

Avoid Repetition?

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Function Templates

Template/Generic Type

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Function Templates

Template/Generic Type

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Template Parameter Declaration

Function Templates

Template/Generic Type

```
          <class T>
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Template Parameter Declaration

Function Templates

Template/Generic Type

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Template Parameter Declaration

➜ The code is NOT compiled or executed directly

Function Templates

Template/Generic Type

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Template Parameter Declaration

➜ The code is NOT compiled or executed directly

➜ It is used by the compiler to **generate/instantiate function instances**

Function Templates

Template/Generic Type

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Template Parameter Declaration

➜ The code is NOT compiled or executed directly

➜ It is used by the compiler to **generate/instantiate function instances**

- on-demand, let compiler do the repetitive work

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max<int>(2, 3);
}
```

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max<int>(2, 3);
}
```

Template Argument

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max<int>(2, 3);
}
```

Template Argument

Generated by Compiler:

```
int max(int a, int b) {
  return (a > b) ? a : b;
}
```

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max<int>(2, 3);
  double d = max<double>(2.5, 3.5);
}
```

Generated by Compiler:

```
int max(int a, int b) {
  return (a > b) ? a : b;
}
```

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max<int>(2, 3);
  double d = max<double>(2.5, 3.5);
}
```

Generated by Compiler:

```
int max(int a, int b) {
  return (a > b) ? a : b;
}
double max(double a, double b) {
  return (a > b) ? a : b;
}
```

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Generated by Compiler:

```
int max(int a, int b) {
  return (a > b) ? a : b;
}
double max(double a, double b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max<int>(2, 3);
  double d = max<double>(2.5, 3.5);
  Vec a_vec(2, 3);
  Vec b_vec(3, 4);
  Vec c_vec
      = max<Vec>(a_vec, b_vec);
}
```

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Generated by Compiler:

```
int max(int a, int b) {
  return (a > b) ? a : b;
}
double max(double a, double b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
  Vec a_vec(2, 3);
  Vec b_vec(3, 4);
  Vec c_vec = max(a_vec, b_vec);
}
```

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Generated by Compiler:

```
int max(int a, int b) {
  return (a > b) ? a : b;
}
double max(double a, double b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
  Vec a_vec(2, 3);
  Vec b_vec(3, 4);
  Vec c_vec = max(a_vec, b_vec);
}
```

Template Argument Deduction

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  double c = max(2.5, 3);
}
```

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  double c = max(2.5, 3);
}
```

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

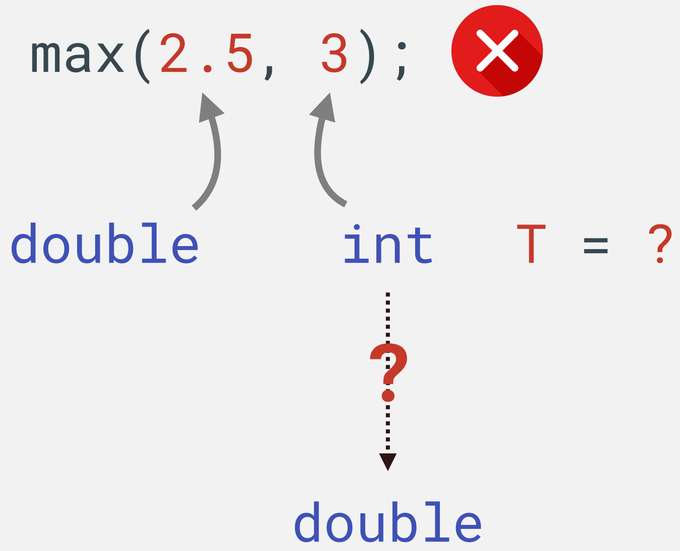
```
int main() {
  double c = max(2.5, 3);
}
```

double int T = ?

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

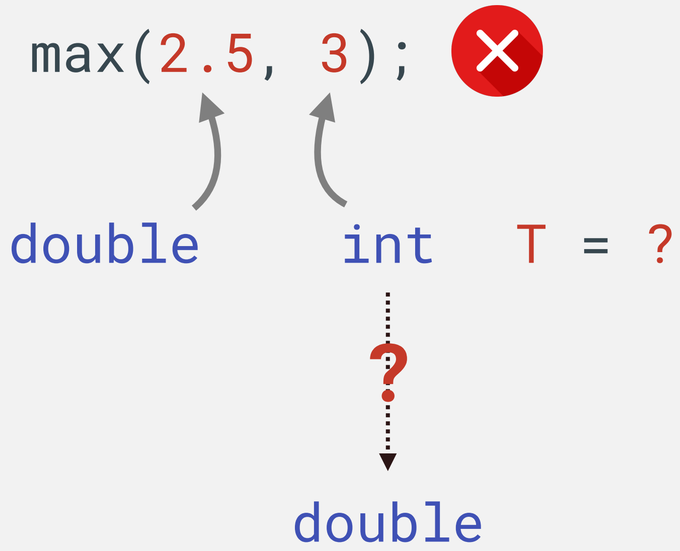
```
int main() {
  double c = max(2.5, 3);
}
```



Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  double c = max(2.5, 3);
}
```



Type conversion is not performed automatically during template argument deduction

Function Templates

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  double c = max(2.5, (double)3);
}
```

Function Templates

```
template <typename T, typename U>
T max(T a, U b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  double c = max(2.5, 3);
}
```

Function Template Specialization

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
}
```

Function Template Specialization

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
  char s1[10] = “tea”;
  char s2[10] = “garden”;
  char *s = max(s1, s2);
}
```

Function Template Specialization

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}


template <>
char *max(char *a,  char *b) {
  return (strcmp(a, b) > 0) ? a : b;
}
```

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
  char s1[10] = “tea”;
  char s2[10] = “garden”;
  char *s = max(s1, s2);
}
```

Function Template Specialization

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}


template <>
char *max(char *a,  char *b) {
  return (strcmp(a, b) > 0) ? a : b;
}
```

```
int main() {
  int c = max(2, 3);
  double d = max(2.5, 3.5);
  char s1[10] = “tea”;
  char s2[10] = “garden”;
  char *s = max(s1, s2);
}
```

Template Classes

```
class ArrayList {
 public:
  ArrayList();
  bool Insert(int pos, int val);
  bool Remove(int pos);
  int Get(int pos) const;

 private:
  int size_, last_idx_;
  int *list_;
};
```

Template Classes

```
class ArrayList {
 public:
  ArrayList();
  bool Insert(int pos, int val);
  bool Remove(int pos);
  int Get(int pos) const;

 private:
  int size_, last_idx_;
  int *list_;
};
```

Only works for integer items

Template Classes

```
template <typename T>
class ArrayList {
 public:
  ArrayList();
  bool Insert(int pos, int val);
  bool Remove(int pos);
  int Get(int pos) const;

 private:
  int size_, last_idx_;
  int *list_;
};
```

Template Classes

```
template <typename T>
class ArrayList {
 public:
  ArrayList();
  bool Insert(int pos, T val);
  bool Remove(int pos);
  T Get(int pos) const;

 private:
  int size_, last_idx_;
  T *list_;
};
```

Template Classes

```
template <typename T>
class ArrayList {
 public:
  ArrayList();
  bool Insert(int pos, T val);
  bool Remove(int pos);
  T Get(int pos) const;

 private:
  int size_, last_idx_;
  T *list_;
};
```

```
int main() {
  ArrayList<int> int_list;
  ArrayList<double> double_list;
}
```

Template Classes

```
template <typename T>
class ArrayList {
 public:
  ArrayList();
  bool Insert(int pos, T val);
  bool Remove(int pos);
  T Get(int pos) const;

 private:
  int size_, last_idx_;
  T *list_;
};
```

```
int main() {
  ArrayList<int> int_list;
  ArrayList<double> double_list;
}
```

```
template <typename T>
T ArrayList<T>::Get(int pos) const {
    assert(pos >= 0 && pos < size_);
    return list_[pos];
}
```

Template Classes

```
template <typename T>
class ArrayList {
 public:
  ArrayList();
  bool Insert(int pos, T val);
  bool Remove(int pos);
  T Get(int pos) const;

 private:
  int size_, last_idx_;
  T *list_;
};
```

```
int main() {
  ArrayList<int> int_list;
  ArrayList<double> double_list;
}
```

```
template <typename T>
T ArrayList<T>::Get(int pos) const {
    assert(pos >= 0 && pos < size_);
    return list_[pos];
}
```