Inheritance & Polymorphism

Cristian Cibils
(ccibils@stanford.edu)

Why Inheritance

Say I am given a Person class, and want to extend it and implement a Student class

Why Inheritance

Option 1: Create Student independently from Person

Why Inheritance

Option 1: Create Student independently from Person

BAD idea: Lot's of repeated code and extremely messy

Why Inheritance

Option 2: Create a student class that has a Person as a member variable

```
class Student {
public:
   string getName() {
      return person.getName();
   }
private:
   Person person;
};
```

Why Inheritance

This works, but is annoying for a few reasons:

Why Inheritance

This works, but is annoying for a few reasons:

- Must copy every single method from Person class
- Every time Person changes, we must also change student
- Someone extending Student further must implement all of Student's methods and all of Person's methods
  - This gets gross really fast

Why Inheritance

Option 3: Use **Inheritance**

```
class Student:public Person {
   //person already defined so no need
   //to implement getName
};
```

Inheritance

Inheritance is the process of
creating a derived class that inherits all the members of a base class on top
of which it can add its own members

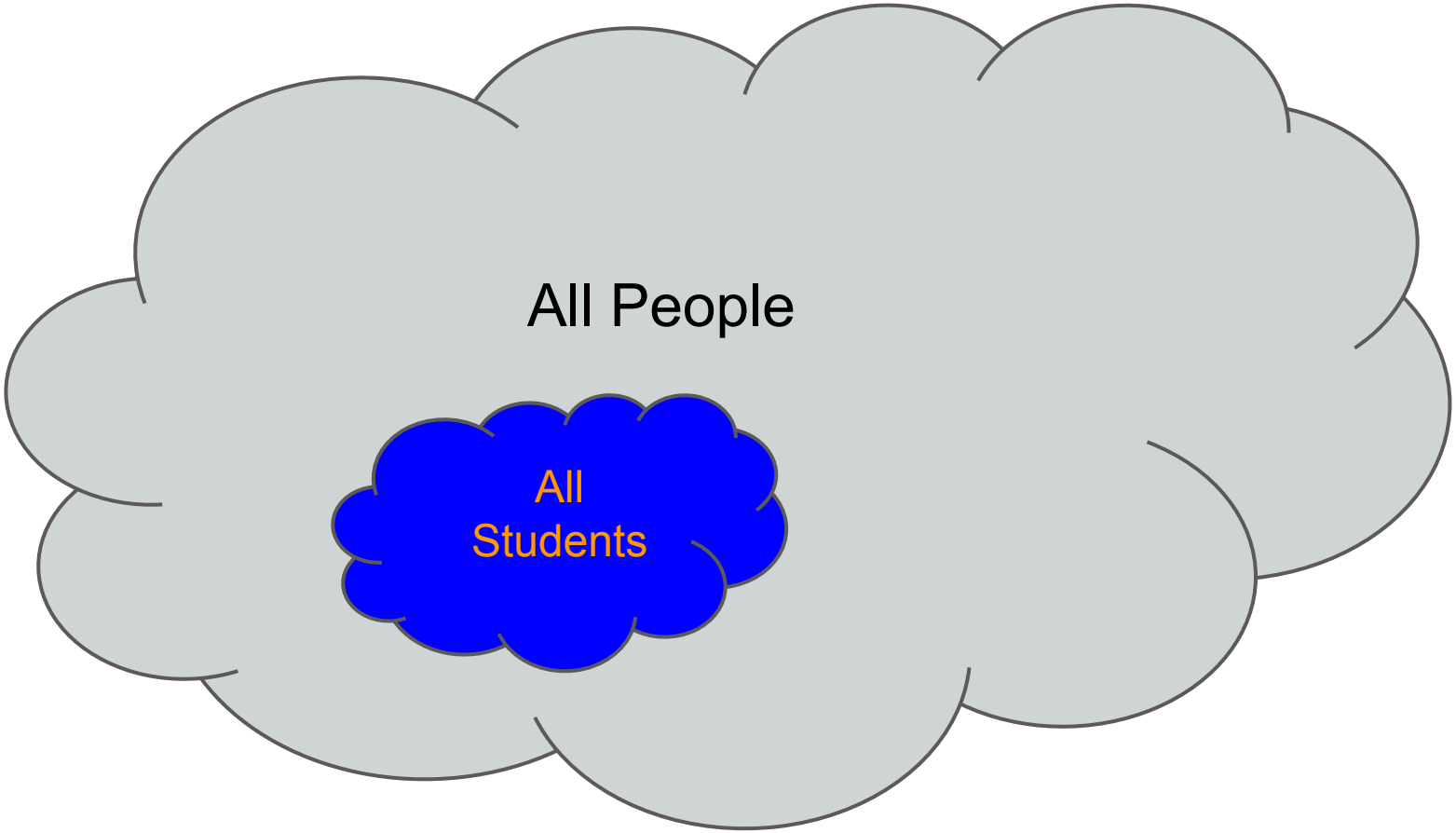
Important!

Most Important point:

Public Inheritance means “is a”

Student is a person

Important



Inheritance

```
class Student:public Person {
public:
   string getID() {
      return id;
   }
private:
   string id;
};
```

Inheritance

```
//The syntax is class Derived:public Base
class Student:public Person {
public:
   string getID() {
      return id;
   }
private:
   string id;
};
```

Inheritance

To call a method on the base class directly, you can use Base::Method

```
string getInfo() {
   return Person::getName() + “ ” + id;
}
```

Inheritance

Constructors should almost always call their base constructor

```
class Derived:public Base {
  Derived():Base::Base() {}
  //Other member functions and things
};
```

Inheritance

Let’s put it together and make our student class

Polymorphism

One really cool part of inheritance is the idea of **Polymorphism** which means that you can refer to a derived class as a base class (only works with pointers and references)

```
Person* p = new Student;

Student s;
Person& otherP = s;
```

Polymorphism

One really cool part of inheritance is the idea of **Polymorphism** which means that you can refer to a derived class as a base class (only works with pointers and references)

At runtime, the program will still know that p is a student

```
Person* p = new Student;
```

Polymorphism

If you don’t use pointers or references, however, **slicing** happens which means the program forgets that p2 is a student as well as a person

```
Person* p = new Student;//p is Student
Person p2 =  *p;//Slicing: p2 is not a
                    //student
```

Polymorphism

Why is this useful?

- Since a student “is a” person, any method that takes in a person as a parameter can take in a student
- Can have many different classes of People that can all be stored together
  - A Vector<Person*> can store a person, student, or any other class derived from person

Overriding Methods

So far so good, but what if we want to override a method in a base class?

Say we have a Bird class and want to create a Penguin class

Bird has a method fly()

Overriding Methods

For an instance of a bird, having a general fly method makes sense

Overriding Methods

For our Penguin class, however, flying works a little differently

Virtual Methods

Marking a method as virtual will mean that the program will decide at run time which method to use instead of deciding at compile time

This means that the actual type is used instead of the declared type

Virtual Methods

```
struct Bird {
   void fly();
}
struct Penguin:public Bird {
   void fly();
}
//Bird's fly method will be called
Bird *b = new Penguin;
b->fly();
```

Virtual Methods

```
struct Bird {
   virtual void fly();
}
struct Penguin:public Bird {
   virtual void fly();
}
//Penguin's fly method will be called
Bird *b = new Penguin;
b->fly();
```

Virtual Methods

Let’s see this in action!

Abstract Classes

Unlike normal methods, virtual methods do not have to be implemented in the base class

A class that does not implement all of its methods is called an **abstract class**

Cannot be created directly and must instead create derived classes

Abstract Classes

To signal that a method will not be implemented, use = 0; after the declaration

```
class SuperHero {
public:
   virtual void usePower() = 0;
};
SuperHero s; //will NOT compile
```

Abstract Classes

```
class Thing:public SuperHero {
public:
   virtual void usePower() {
      cout << “IT’S CLOBBERING TIME!”
           << endl;
};

SuperHero* s = new Thing; //compiles
s->usePower();
```

Multiple Inheritance

In C++, unlike Java and many other languages, a class can inherit from multiple classes.

iostream, for example, inherits from both istream and ostream.

Works the same as regular inheritance but a little tricky since methods can have the same name

Advice when Using Inheritance

If you are using inheritance, here are some key pieces of advice:

- If you have a destructor in your base type, make it virtual. Non-virtual destructors are asking for memory leaks

Advice when Using Inheritance

If you are using inheritance, here are some key pieces of advice:

- If you have a destructor in your base type, make it virtual. Non-virtual destructors are asking for memory leaks
- Never create a method in your derived class with the same signature as one in your base class unless it is virtual

Advice when Using Inheritance

If you are using inheritance, here are some key pieces of advice:

- If you have a destructor in your base type, make it virtual. Non-virtual destructors are asking for memory leaks
- Never create a method in your derived class with the same signature as one in your base class unless it is virtual
- Always call the base constructor in the derived's constructor

Casting

In Java, C, and many other languages, we convert from one type to another (**cast**) with:

```
int x = 5;
double d = (double)x;

oldType oldV
newType newV = (newType)oldV
```

Casting

In C++, this kind of casting, called **C-style casting** is considered terrible style

Instead, use one of the following:

- const_cast
- static_cast
- dynamic_cast
- reinterpret_cast

Casting

In C++, this kind of casting, called **C-style casting** is considered terrible style

Instead, use one of the following:

- ~~const_cast (covered in const lecture)~~
- static_cast
- dynamic_cast
- reinterpret_cast

Casting

static_cast is used to do 90% of the casting you are used to. Use it to cast one type to another if both types are declared at compile time:

```
int x;
double y = static_cast<double>(x);

oldT V;
newT newV = static_cast<oldType>(v);
```

Casting

In C++, this kind of casting, called **C-style casting** is considered terrible style

Instead, use one of the following:

- ~~const_cast (covered in const lecture)~~
- ~~static_cast~~
- dynamic_cast
- reinterpret_cast

Casting

dynamic_cast is used to cast a base class to a derived class. Only works if base class is actually derived class at run time

```
Base* b = new Derive;
Derived* d = dynamic_cast<Derive*>(b);
```

Casting

In C++, this kind of casting, called **C-style casting** is considered terrible style

Instead, use one of the following:

- ~~const_cast (covered in const lecture)~~
- ~~static_cast~~
- ~~dynamic_cast~~
- reinterpret_cast

Casting

In C++, this kind of casting, called **C-style casting** is considered terrible style

Instead, use one of the following:

- ~~const_cast (covered in const lecture)~~
- ~~static_cast~~
- ~~dynamic_cast~~
- ~~reinterpret_cast (don't use)~~