Inheritance

Section 3

1

What is Inheritance?

- Inheritance enables classes to inherit attributes and methods from other classes. It allows the sharing of common elements between classes without having to repeat definitions for each class.

*Inheritance occurs between classes **NOT** Objects!*

2

Derived & Base Classes (1)

- A class does not contain state values – it only defines the methods and attributes which may be used in a class. State values are contained in individual objects.
- A **derived** class inherits the attributes and methods of a **base** class.
- We may build on the derived class by adding attributes and methods.

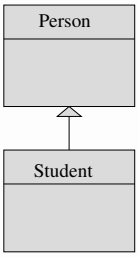
3

Derived & Base Classes (2)

- **Base class:** a class form which another class inherits.
- Base classes don't have to represent anything concrete which could be instantiated as an object of its own.
- **Derived class objects** do not inherit any state values from base class objects.
- The derived class may be extended without effecting the original class..

4

public inheritance



Every student is a person, but not every person is a student.

C++ compiler enforces this!

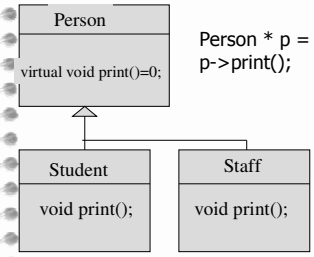
5

is-a

```
class Person {
   void play(const Person &);
};
class Student : public Person {
   void study(const Student & s);
};

Person p;
Student s;
p.play();          // okay
s.play();          // no problem
s.study();         // yup!
p.study();         // error!
```

6

Typical design




```
Person * p = new Student("Hank");
p->print();
```

7

Passing parameters to base class constructor

```
class Person {
public:
   Person(const string & n) : name(n) {}
private:
   string name;
};
class Student {
public:
   Student(const string & n, float g) : Person(n), gpa(g)
{}
private:
   float gpa;
};
```

8

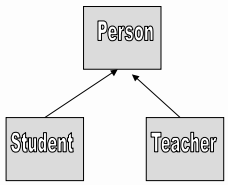
Typical uses of inheritance

- Single inheritance – one base class
- attributes that describe all classes go in the base class
- attributes that are "special" go in derived class

9

Software Reuse

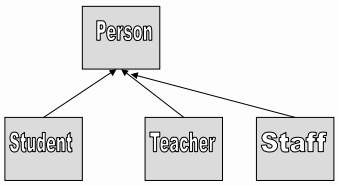
Derived classes reuse attributes in the base class.



10

Easy to extend

Can add a new derived class, extending the framework, without changing any existing code: extensibility

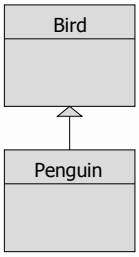


11

is-a doesn't always work right!

- fact: birds can fly
- fact: a penguin is a bird

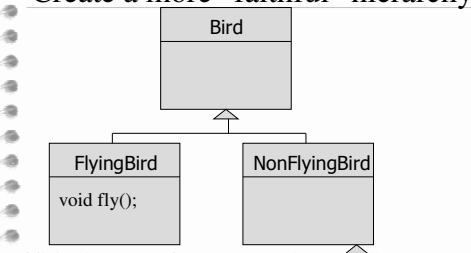
```
class Bird {
   virtual void fly();
};

class Penguin : public Bird {
};
```



What's wrong with this?

12

One solution:
Create a more "faithful" hierarchy



If flying isn't part of our system and never will, then the previous hierarchy is okay.

Problem: no ONE ideal design for all software!

13

Second Approach

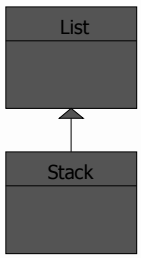
Redefine "fly" so that it generates a runtime error:

```
void error(const string & msg);

class Penguin : public Bird {
public:
  virtual void fly() { error("Penguins can't fly!");

};
```

Better to detect errors at compile time, rather than at runtime.

14

Another anti-intuitive example:
a stack is-a list?



What's wrong with this picture?

compliments of Rumbaugh, et al.

15

16

C++ Syntax for Inheritance

- : denotes inheritance(also called specialisation or derivation or extension)
- public derivations and private derivations are possible.
- The protected keyword allows derived classes to access inherited attributes.

17

Public Inheritance in C++

```
class intObj{
private:
int val;
public:
intObj(){val =0;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<val<<endl;}
};
class SubintObj: public intObj{
//...
}
int main(){
intObj obj; obj.print();
SubintObj Sobj; Sobj.print();
}
```

The common syntax is:

derive SubintObj as a subclass of intObj

it is public because it means that the public members of the base class become public members of the derived class.

Question: what can we do with the derived class?

18

Public or Private Derivations?

Public derivations:

- Allow objects of a derived class access to the public section of the base class.

Private derivations:

- Allow a derived object to use the methods defined in the derived class, not those inherited from the base class.

19

Public Inheritance in C++

```
class intObj{
private:int val;
public:
intObj(){val =0;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<val<<endl;}
};

class SubintObj: public intObj{
//...
};

int main(){
SubintObj Sobj; Sobj.print();
}
```

Question: How can we make the derived class different from the base class?

Put something in its class definition

But WHAT, and HOW?

20

Public Inheritance in C++:add a method

```
class intObj{
private:int val;
public:
intObj(){val =0;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<val<<endl;}
};

class SubintObj: public intObj{
public:
void inc(){val++;}
};

int main(){
SubintObj Sobj; Sobj.print();
Sobj.inc();Sobj.print();
}
```

Put another method in:

increment the internal value of the integer

But, the compiler will return an error:

```
intObj.cc: In method
`void SubintObj::inc()':
intObj.cc:20: member
`val' is private
```

21

Introducing protected data members

The SubintObj class has a problem --

- it cannot access the private val data member of its base class (intObj)
- it *needs* to access it to implement the inc method … (Question: does it?)

By saying that the val is a *protected* member (in the base class) this will allow all subclasses (derived classes) access to it.

When we do this (see over) the code compiles and works as required.

22

Public Inheritance in C++: add a method

```
class intObj{
protected: int val;
public:
intObj(){val =0;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<val<<endl;}
};

class SubintObj: public intObj{
public:
void inc(){val++;}
};

int main(){
SubintObj Sobj; Sobj.print();
Sobj.inc();Sobj.print();
}
```

protected : to allow access

This will print out

```
0

1
```

23

Public Inheritance in C++: add a data member

```
class intObj{
private:int val;
public:
intObj() {val =0;}
int getint() const  {return val;}
void setint(int x) {val = x;}
void print() {cout
<<val<<endl;}
};

class SubintObj: public intObj{
public:
int count;
};

int main(){
SubintObj Sobj; Sobj.print();
cout << Sobj.count;
}
```

Put another data member in:

the number of times it has been printed, for example

This will output (typically):

```
0
2285548
```

where the second number shows that the count has not been initialised

24

Public Inheritance in C++: add a data member

```
class intObj{
private:int val;
public:
intObj(){val =0;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<val<<endl;}
};

class SubintObj: public intObj{
public: int count;
SubintObj(){count =0;}
};

int main(){
SubintObj Sobj; Sobj.print();
cout << Sobj.count;
}
```

Question: how can we make sure that the added data type gets initialised correctly when an object of the derived class is constructed?

Answer: we write a constructor for the derived class (in this case a default constructor with no parameters).

The output is now:

0

0

Question: how did val get initialised on slide 11?

Parent constructors and destructors

When an object of a derived class (a subclass) is declared, then the parent's **default** constructor is called before the required constructor of the derived class

If the parent has a parent then the grandparent's **default** is called followed by the parent's **default** followed by the derived class constructor

In effect, for an arbitrary number of parents, the parent **default** constructors are called in a top-down fashion.

The same thing happens with the destructors except it is down bottom-up (with the derived child class calling its own destructor first)

NOTE: This can get very complicated (esp. with *multiple inheritance*)

Parameters?

- A derived class will always inherit the constructor of a base class, as well as having its own. The base class constructor is always called first, followed by the constructor of the derived class, and so on down the tree.
- If the base constructor takes no parameters inheritance is implicit.
- If it takes parameters these must be stated explicitly in each derived class.

An Example:

```
class customer
{
private:
     char name[30];
  public:
      {   customer(char * name_in)
          strncpy(name,name_in,29);
          name[29] = '/0';
      }
}
```

Inheriting the constructor.

```
class accountcustomer:public customer
{ private:
      int account_number;
  public:
      accountcustomer(char* name_in);
};
accountcustomer::accountcustomer(char*
  name_in):customer(name_in)
{ // constructor body
}
```

29

30

31

32


33


34


35


36

37

DOMINATING (overridden) member functions

```
class intObj{
protected:int val;
public:
intObj(){val =0;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<val<<endl;}
};
class SubintObj: public intObj{
public: int count;
SubintObj(){count =0;}
void print(){cout <<val<<endl;count++;
};
int main(){
SubintObj Sobj; Sobj.intObj::print();
cout << Sobj.count;}
```

We wanted to use count to count the number of times print was called so we have to override the base class print method in the derived class.

But C++ lest us override the override … ie dominate!

This produces the output:

0

0

38

OVERRIDING and DOMINATING data members

As well as being able to override methods/functions, we can also do it to data members.

You will not need to do this!

It is (generally) a bad thing to do

39

`virtual` functions & polymorphism

```
// SubintObj.cc
#include<iomanip>
#include<iostream>
class intObj{
private:int val;
public:
intObj(){val =0;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<"intObj:"<<val<<endl;}
};
class SubintObj: public intObj{
public:
void print(){cout <<"SubintObj:"<<getint()<<endl;}
};
int main(){
intObj obj; obj.print();
SubintObj Sobj; Sobj.print();}
```

This will produce the output:

```
intObj:0
SubintObj:0
```

The print functions are *polymorphic* because **they work on different types (classes)**

We have already seen *polymorphic* functions in the same class, where **they have different parameter types**

40

`virtual` functions & polymorphism

```
class intObj{
private:int val;
public:
intObj(){val =0;}
void print(){cout <<"intObj:"<<val<<endl;}
bool equals(intObj obj){obj.print(); print();
                        return (obj.getint()==val);}
};
class SubintObj: public intObj{
public:
void print(){cout <<"SubintObj:"<<getint()<<endl;}
};
int main(){
intObj obj; SubintObj Sobj;
obj.print(); Sobj.print();
if (obj.equals(Sobj)) cout <<"They are
equal"<<endl;
  else cout <<"They are not equal"<<endl;
}
```

The output is:

```
intObj:0
SubintObj:0
intObj:0
intObj:0
They are equal
```

What happened here?

`virtual` functions and polymorphism

What happened here?

```
class intObj{
private:int val;
public:
intObj(){val =0;}
void print(){cout <<"intObj:"<<val<<endl;}
bool equals(intObj obj){obj.print(); print();
                        return (obj.getint()==val);}
};
class SubintObj: public intObj{
public:
void print(){cout
<<"SubintObj:"<<getint()<<endl;}
};
int main(){
intObj obj; SubintObj Sobj;
obj.print(); Sobj.print();
if (obj.equals(Sobj)) cout << "They are
equal"<<endl;
  else cout <<"They are not equal"<<endl;
}
```

The Sobj object is passed as a parameter to the equals method of the intObj class. However, the type of the parameter is specified to be an intObj. Normally, this would give a type mismatch compiler error. However, as a SubintObj *is-a subclass* of an intObj, anywhere we would like an intObj we can use a SubIntObj. But, it is then used/copied as an intObj and *gets* its methods.

`virtual` functions and polymorphism…

This becomes even more complicated when we consider pointers to objects. We can declare a variable to be a pointer, p say, to an object of class C

Now, if S is a subclass of C then we can point p at any object of class S

However, C++ *confuses* matters when it comes to deciding whether a p of class C which points to an object of class S should execute the methods of C or the methods of S.

Question: what do you think should be done?

- Always use the base class methods
- Always use the methods of the class of the object being pointed at
- A mixture of the two approaches

I am sure you can guess what happens in C++ !!!! … We get a mixture!!!

Why use virtual functions

- place the burden of knowing/choosing the object type on the compiler
- compiler constructs a vtbl for each class w/ virtual function
- one table per class; each class has ptr to vtbl
- vtbl holds pointers to all virtual functions
- vptrs init in constructor
- each virtual function is invoked thru its vptr

Do virtual functions have performance penalty?

- vptr must be init in constructor
- virtual function is invoked via pointer indirection
- cannot inline virtual functions
- more later…

45

advantage of virtual functions

```
class ZooAnimal {
public:
   virtual void draw() const = 0;
   int resolveType()   const { return myType; }
private:
enum animalTypes  myType { BEAR, MONKEY };
};
class Bear : public ZooAnimal {
public:
   Bear(const string & name)
         : myName(name), myType(BEAR) {}
   void draw() {  cout << "I'm a bear" << endl;
};
```

46

maintenance headache

```
void drawAllAnimals(ZooAnimal *pz) {
   for (ZooAnimal *p=pz; p; p = p->next) {
      switch (p->resolveType() ) {
         case BEAR : { ((Bear *)p)->draw(); break; }
         case MONKEY : { ((Monkey *)p)->draw();
break; }

         ... handle all other animals currently in the
zoo     have to change the switch statement
      }    every time an animal arrives/leavs
   }                    the zoo!
}
```

47

Use virtual functions

```
void drawAllAnimals(ZooAnimal *pz) {
   for (ZooAnimal *p=pz; p; p = p->next) {
      p->draw();
   }
}
```

48

RULE: Never redefine an inherited non-virtual function

```
class Person {
public:
   string getName() const { return name; }
private:
   string name;
};
class Student : public Person {};

Student stu;
Person * p = &stu;
Student * s = &stu;
p->getName();
s->getName();
```

These two should behave the same!

49

Never redefine an inherited non-virtual function

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

Student stu;
Person * p = &stu;
Student * s = &stu;
p->getName();
s->getName();
```

Now they behave differently!

Non-virtual functions are **statically** bound;
virtual functions are **dynamically** bound

50

`virtual` functions and polymorphism

```
class intObj{
private:int val;
public:
intObj(){val =10;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<"intObj:"<<val<<endl;}
};
class SubintObj: public intObj{
public:
SubintObj(){setint(20);}
void print(){cout <<"SubintObj:"<<getint()<<endl;}
};

int main(){
intObj objPt; SubintObj Sobj; SubintObj* SobjPt;
objPt.print(); Sobj.print(); SobjPt->print();
}
```

```
The output is:
intObj:10
SubintObj:20
SubintObj:16
```

Note: no constructive initialisation of the pointer value!!!

51

`virtual` functions and polymorphism

```
class intObj{
private:int val;
public:
intObj(){val =10;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<"intObj:"<<val<<endl;}
};

class SubintObj: public intObj{
public:
SubintObj(){setint(20);}
void print(){cout <<"SubintObj:"<<getint()<<endl;}
};

int main(){
intObj objPt; SubintObj Sobj; SubintObj* SobjPt;
SobjPt = &objPt; SobjPt->print();
}
```

Question: why does the compiler give the following error message -

```
SubintObj.cc: In
function `int main()':
SubintObj.cc:23: type
`SubintObj' is not a
base type for type
`intObj'
```

Answer: we cannot redirect the pointer at a superclass object we can only repoint at a subclass object

52

`virtual` functions and polymorphism

```
class intObj{
private:int val;
public:
intObj(){val =10;}
int getint() const {return val;}
void setint(int x){val = x;}
void print(){cout <<"intObj:"<<val<<endl;}
};
class SubintObj: public intObj{
public:
SubintObj(){setint(20);}
void print(){cout <<"SubintObj:"<<
getint()<<endl;}
};
int main(){
SubintObj Sobj; intObj* objPt;
objPt = &Sobj; objPt->print();
}
```

The output is:

```
intObj:20
```

So, in this case we have a bizzare mixture:

the print() method of the intObj class has been called for an objPt which is pointing at a SubintObj

Question: why can't the method of the object being pointed at be used?

This would be called *dynamic binding*

Dynamic binding: `virtual` functions and polymorphism

- when the class of the object being pointed at is used (at run time) to decide which method gets executed (its own method)
- it is dynamic because pointers can be used to point at any subclass object
- So, for example, if we define a Pets class with 2 subclasses (dog and cat). Each offers a method makenoise(). If we declare an object, Pugsy say, to be a Pet (or a pointer to a pet), then Pugsy may bark or purr depending on whether it is a dog or a cat.
- The key is that we cannot tell at compile time whether Pugsy is a dog or a cat so the correct method can only be decided at run time
- Unless we want the Pet method makenoise every time (which we usually do not) then we need some mechanism to tell the compiler to bind the method at run time.
- This mechanism is a *virtual* function in C++.

`virtual` functions and polymorphism

```
class intObj{
private:int val;
public:
intObj(){val =10;}
int getint() const {return val;}
void setint(int x){val = x;}
void virtual print(){cout <<"intObj:"<<val<<endl;}
};
class SubintObj: public intObj{
public:
SubintObj(){setint(20);}
void print(){cout <<"SubintObj:"<<getint()<<endl;}
};
int main(){
intObj obj;
SubintObj Sobj; intObj* objPt;
objPt = &Sobj; objPt->print();
objPt =&obj; objPt->print();
}
```

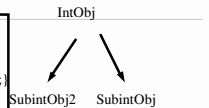
The output is:

```
SubintObj:20
intObj:10
```

So, in this case we have the 'correct' method used at run time

`virtual` functions and polymorphism

```
class SubintObj: public intObj{
public:
SubintObj(){setint(20);}
void print(){cout <<"SubintObj:"<<getint()<<endl;}
};
class SubintObj2: public intObj{
public:
SubintObj2(){setint(30);}
void print(){cout <<"SubintObj2:"<<
getint()<<endl;}
};
int main(){
SubintObj2 Sobj2;
SubintObj Sobj; intObj* objPt;
objPt = &Sobj; objPt->print();
objPt =&Sobj2; objPt->print();
}
```



objPt can point at an object of:

- its own class
- any of its 2 subclasses

If print is *virtual* in IntObj the output is:

```
SubintObj:20
SubintObj2:30
```

- Pets and virtual functions example.

57