Inheritance and Polymorphism

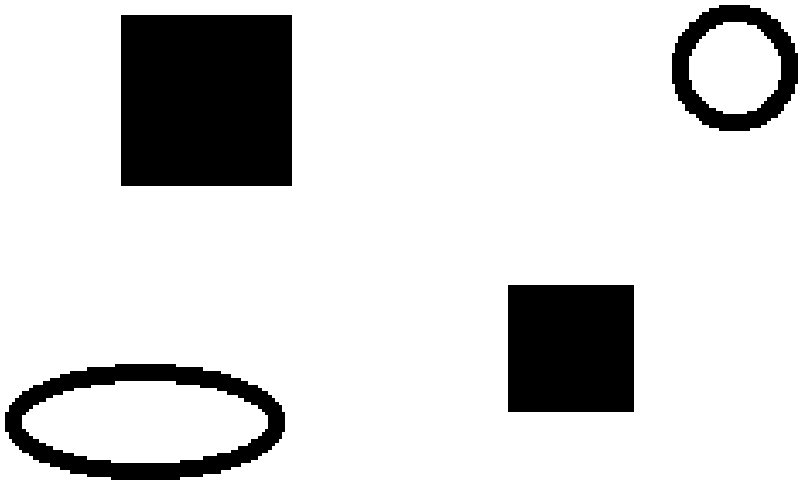
PIC 10B, UCLA

Inheritance and Polymorphism

Very often when a class is written, there are other relevant classes that satisfy the “is a” relationship to the original class. Managing graphics is one of the nicest applications of polymorphism. All entities on a canvas could be considered **Shape**s; there are different types of shapes: **Rectangle**s, **Ellipse**s, etc.; in turn, a **Square** is a **Rectangle**, a **Cirle** is an **Ellipse**, etc. We may wish to do different things to collections of them and have those functions apply to all **Shapes** or all **Squares**, etc.

Inheritance and Polymorphism

Very often when a class is written, there are other relevant classes that satisfy the “is a” relationship to the original class. Managing graphics is one of the nicest applications of polymorphism. All entities on a canvas could be considered **Shape**s; there are different types of shapes: **Rectangle**s, **Ellipse**s, etc.; in turn, a **Square** is a **Rectangle**, a **Cirle** is an **Ellipse**, etc. We may wish to do different things to collections of them and have those functions apply to all **Shapes** or all **Squares**, etc.



Inheritance and Polymorphism

Very often when a class is written, there are other relevant classes that satisfy the “is a” relationship to the original class. Managing graphics is one of the nicest applications of polymorphism. All entities on a canvas could be considered **Shape**s; there are different types of shapes: **Rectangle**s, **Ellipse**s, etc.; in turn, a **Square** is a **Rectangle**, a **Cirle** is an **Ellipse**, etc. We may wish to do different things to collections of them and have those functions apply to all **Shapes** or all **Squares**, etc.

Increase all Squares by 50%

Inheritance and Polymorphism

Very often when a class is written, there are other relevant classes that satisfy the “is a” relationship to the original class. Managing graphics is one of the nicest applications of polymorphism. All entities on a canvas could be considered **Shape**s; there are different types of shapes: **Rectangle**s, **Ellipse**s, etc.; in turn, a **Square** is a **Rectangle**, a **Cirle** is an **Ellipse**, etc. We may wish to do different things to collections of them and have those functions apply to all **Shapes** or all **Squares**, etc.

Shrink all Ellipses by 20%

Inheritance and Polymorphism

Very often when a class is written, there are other relevant classes that satisfy the “is a” relationship to the original class. Managing graphics is one of the nicest applications of polymorphism. All entities on a canvas could be considered **Shape**s; there are different types of shapes: **Rectangle**s, **Ellipse**s, etc.; in turn, a **Square** is a **Rectangle**, a **Cirle** is an **Ellipse**, etc. We may wish to do different things to collections of them and have those functions apply to all **Shapes** or all **Squares**, etc.

Move all Circles down

Inheritance and Polymorphism

Very often when a class is written, there are other relevant classes that satisfy the “is a” relationship to the original class. Managing graphics is one of the nicest applications of polymorphism. All entities on a canvas could be considered **Shape**s; there are different types of shapes: **Rectangle**s, **Ellipse**s, etc.; in turn, a **Square** is a **Rectangle**, a **Cirle** is an **Ellipse**, etc. We may wish to do different things to collections of them and have those functions apply to all **Shapes** or all **Squares**, etc.

Move all Shapes right

Inheritance and Polymorphism

The word **polymorphism** literally means "the condition of occurring in several different forms."

With **polymorphism**, we manage collections of objects according to their level of specification.

By storing a collection of **Shape**s while being aware of whether a given shape is itself a **Circle**, **Ellipse**, etc., various actions can be carried out on specific class types.

Point and Point-Mass Classes

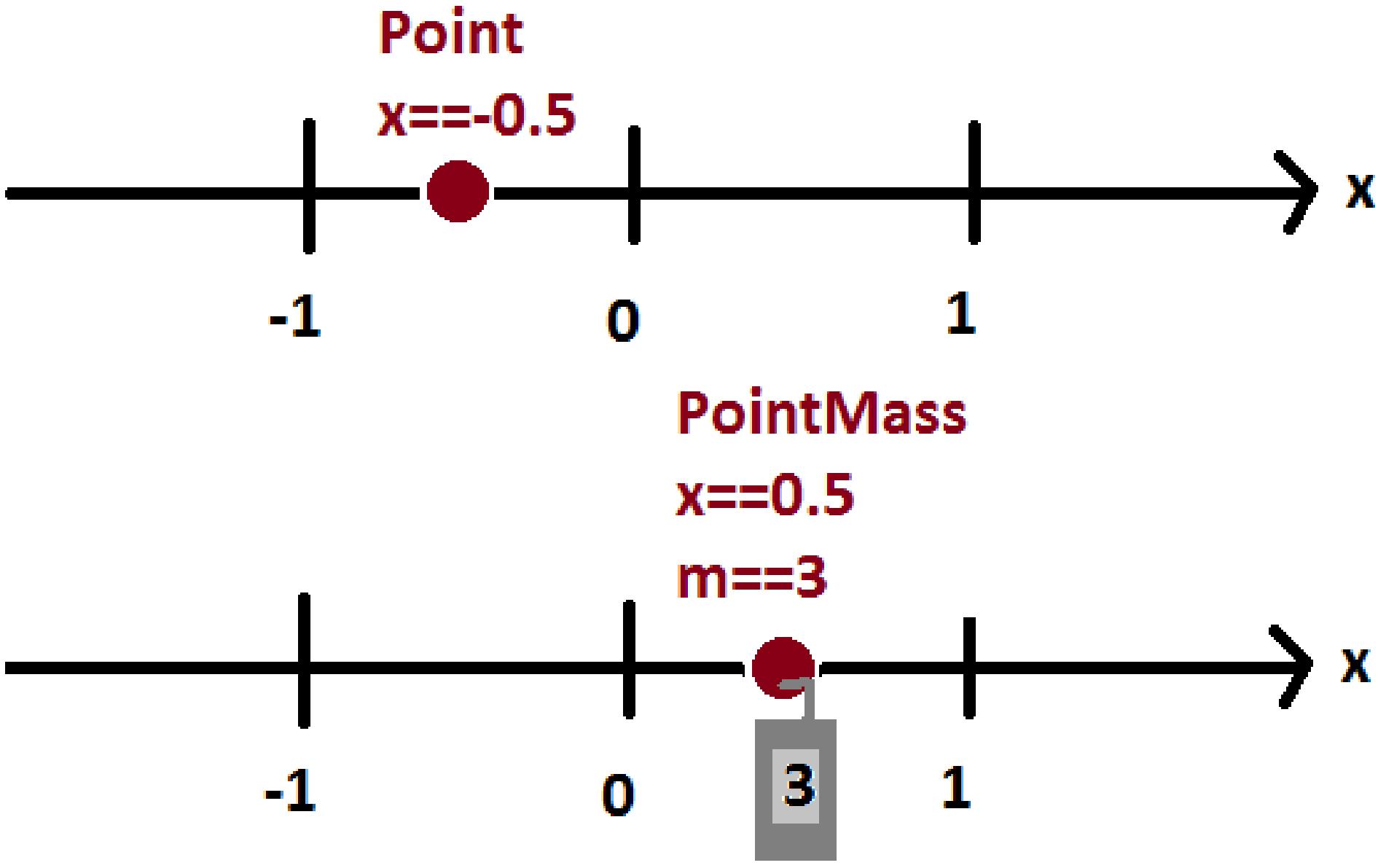
We begin this exploration with a very simple example to illustrate the basic concept of inheritance and a few pieces of syntax. We consider **Point** and **PointMass** classes:
A **Point** will

- store a value **x** representing a position along the x-axis;
- have a single constructor to set **x**;
- have a function **shift_x** to update **x**; and
- have a function **display**, printing the position to the console.

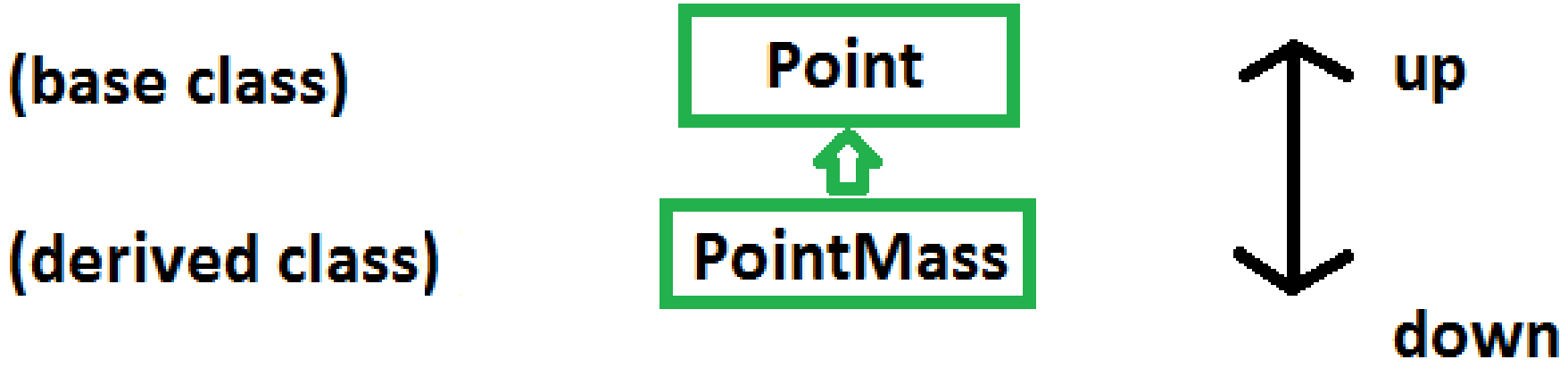
A **PointMass** *is a* **Point** but the point has a mass associated to it. A **PointMass** will

- store a value **m** representing its mass (**x** will be managed by **Point**);
- have a single constructor to set **x** and **m**;
- have a function **change_m** to update **m**; and
- *override* the function **display** to print both the mass and position.

Point and Point-Mass Classes



Point and Point-Mass Classes



In terminology, we say that **Point** is a **base class** to **PointMass** and **PointMass inherits** from **Point**. There is an "is a" relationship here: a point mass is a point (just more specialized).

We say that **PointMass** is a **derived class** from **Point**.

Later on, we'll need a sense of "direction": up means towards more base classes; down means towards more derived classes.

Point and Point-Mass Classes

Here is our **Point** class:

```
class Point {
private:
    double x;

public:
    Point(double _x) : x(_x) { }
    void shift_x(double dx) { x += dx; }

    virtual void display() const {
        std::cout <<"position = " <<x <<'\n';
    }

    virtual ~Point() = default;
};
```

Point and Point-Mass Classes

The constructor and **shift_x** are obvious in what they do. But we need to introduce the **virtual** keyword.

```
virtual void display() const
```

When a member function is marked as **virtual** as above, it tells the compiler that a derived class may give a more specialized version of the function. Thus, **PointMass**, to be defined, can give its own rendition of **display**.

Point and Point-Mass Classes

virtual ~Point() = default

Whenever an object is polymorphic, i.e., it is the base class of some inheritance structure, **its destructor must be marked as virtual to avoid undefined behaviour**. More on this later.

The **= default** instructs the compiler to do what it would normally do when destructing the object (almost as though we had given an empty body or did not even mention the destructor). The important thing, though, is the destructor is virtual.

Remark: we could have also just given a virtual destructor with empty body as in:
virtual ~Point() { }
It would do the same thing.

Point and Point-Mass Classes

If the declaration and definition of a virtual function are separated, the virtual keyword is not repeated at the definition:

```
struct X {
    virtual ~X() = default;
    virtual void good() const;
    virtual void bad() const;
};

// ...

void X::good() const { } // okay...

virtual void X::bad() const { } // ERROR
```

Point and Point-Mass Classes

Here is our **PointMass** class:

```
class PointMass : public Point {
private:
    double m;

public:
    PointMass(double _x, double _m) : Point(_x), m(_m) { }
    void change_m(double new_m) { m = new_m; }

    void display() const override {
        std::cout <<"mass of " <<m <<" at ";
        Point::display();
    }
};
```

Point and Point-Mass Classes

class PointMass : public Point

This tells the compiler that **PointMass** *inherits* from **Point**. It does so through **public inheritance**. This means

- any member function/variable that was **public** in **Point** is also **public** in **PointMass** and accessible to **PointMass**;
- any member function/variable that was **private** in **Point** is inaccessible to the **PointMass** class; and
- any member function/variable that was **protected** in **Point** is **protected** in **PointMass** and accessible to **PointMass**.

The **protected** keyword is an **access specifier** like **public** and **private**. To all external functions/classes, it behaves like **private**. But to derived classes, it behaves like **public**.

In most cases, **protected** should be avoided because a derived class could mismanage the data of its base class.

Point and Point-Mass Classes

```
PointMass(double _x, double _m) : Point(_x), m(_m) { }
```

A derived class *first and foremost ensures that its direct base class is constructed*.

The **PointMass** class must first construct its **Point** component. To do so, a derived class calls the constructor of its base class with appropriate parameters, hence **Point(_x)**.

Only *after the base class has been constructed* can a derived class initialize its own members such as with the **m(_m)**! Just like with normal classes, these members are initialized in the order declared within the class.

Point and Point-Mass Classes

void display() const override

The **display** function is **virtual**, so **PointMass** can give its own rendition of the function, which we do here. The **override** keyword tells the reader of the code and the compiler that we *think* that we are overriding a virtual function from a base class.

From a debugging standpoint, if the function we wrote was not overriding a virtual function, the compiler could generate an error and save us from subtle bugs.

Point and Point-Mass Classes

void display() const override

From a readability standpoint, it is important for someone looking at the function to know it is a **virtual** function being given a more specialized implementation.

The **override** keyword is not required for the code to compile and run. It is, however, good practice as it adds clarity and robustness to code.

Remark: once a function has been marked as **virtual**, it is **virtual** in all derived classes, even ten levels deep, say.

Point and Point-Mass Classes

```
void display() const override {
    std::cout <<"mass of " <<m <<" at ";
    Point::display();
}
```

If we want to use a virtual function from a base class directly, we can. Here, in the **PointMass::display** function, we directly execute the instructions of the **Point::display** function.

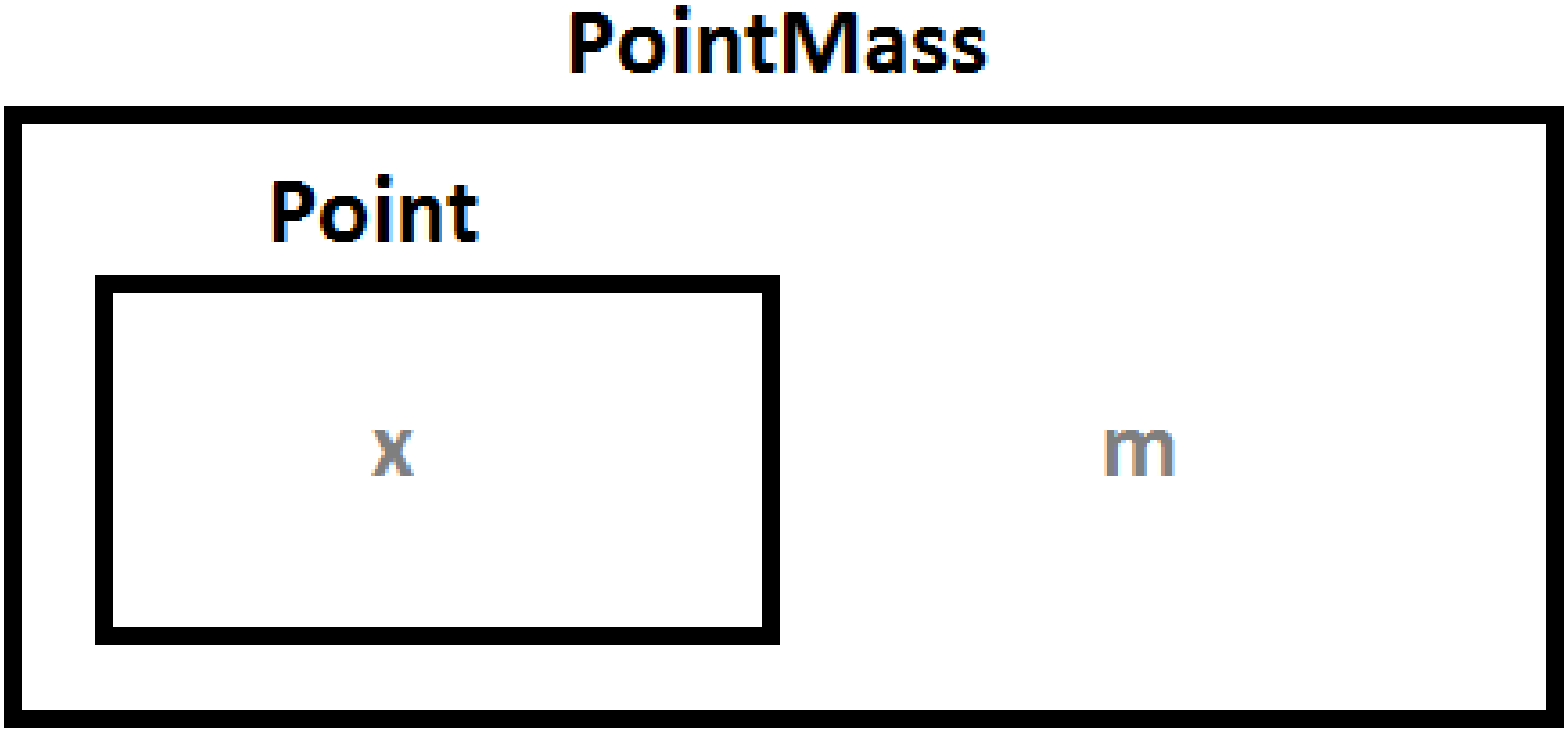
The **Point::** is important here as otherwise calling **display()** would resolve to **PointMass::display** calling itself, which calls itself, which calls itself, ... and it would never end!

Point and Point-Mass Classes

Like the **virtual** keyword, **override** (and **final**, to come), are not repeated outside of the class interface if the declarations/definitions are separated.

Point and Point-Mass Classes

Note that all members of the base class are part of the derived class, even if the base cannot access them directly.



Point and Point-Mass Classes

Here we consider an example of using the classes.

```
Point p{3};
p.shift_x(5);
p.display();

PointMass pm{2,4};
pm.shift_x(20);
pm.change_m(10);
pm.display();
```

Output:

```
position = 8
mass of 10 at position = 22
```

Point and Point-Mass Classes

But the real power of inheritance comes from functions like:

```
void print(const Point& pref) {
    pref.display();
}

void print(const Point* pptr) {
    pptr->display();
}
```

Note how "in writing", this overloaded function can either accept a reference to const **Point** or a pointer to const **Point**.

We say the **static type**s of **pref** and **pptr** are **const Point&** and **const Point***. There is no mention of **PointMass**...

Point and Point-Mass Classes

But...

```
Point x{1};
PointMass y{2,3};

print(x);
print(y);

print(&x);
print(&y);
```

produces the output:

```
position = 1
mass of 3 at position = 2
position = 1
mass of 3 at position = 2
```

Point and Point-Mass Classes

PointMass inherits from **Point.** Thus, a function accepting a pointer or reference to **Point** can equally well accept a pointer or reference to **PointMass** *and preserve* the **dynamic type** of the object.

When calling **print** with either **y** or **&y**, the true underlying type is **const PointMass&** or **const PointMass***. This underlying type is the **dynamic type** of the object.

When a virtual function is called on a pointer or reference type, the most derived version of that function, appropriate to the dynamic type of the object, will be invoked.

A Pet Class

Now consider a **pet** class to represent a family pet (either a cat or dog).

- Every **pet** has a **name**.
- Every **pet** can also **talk**, but how they talk depends on being a cat or a dog...

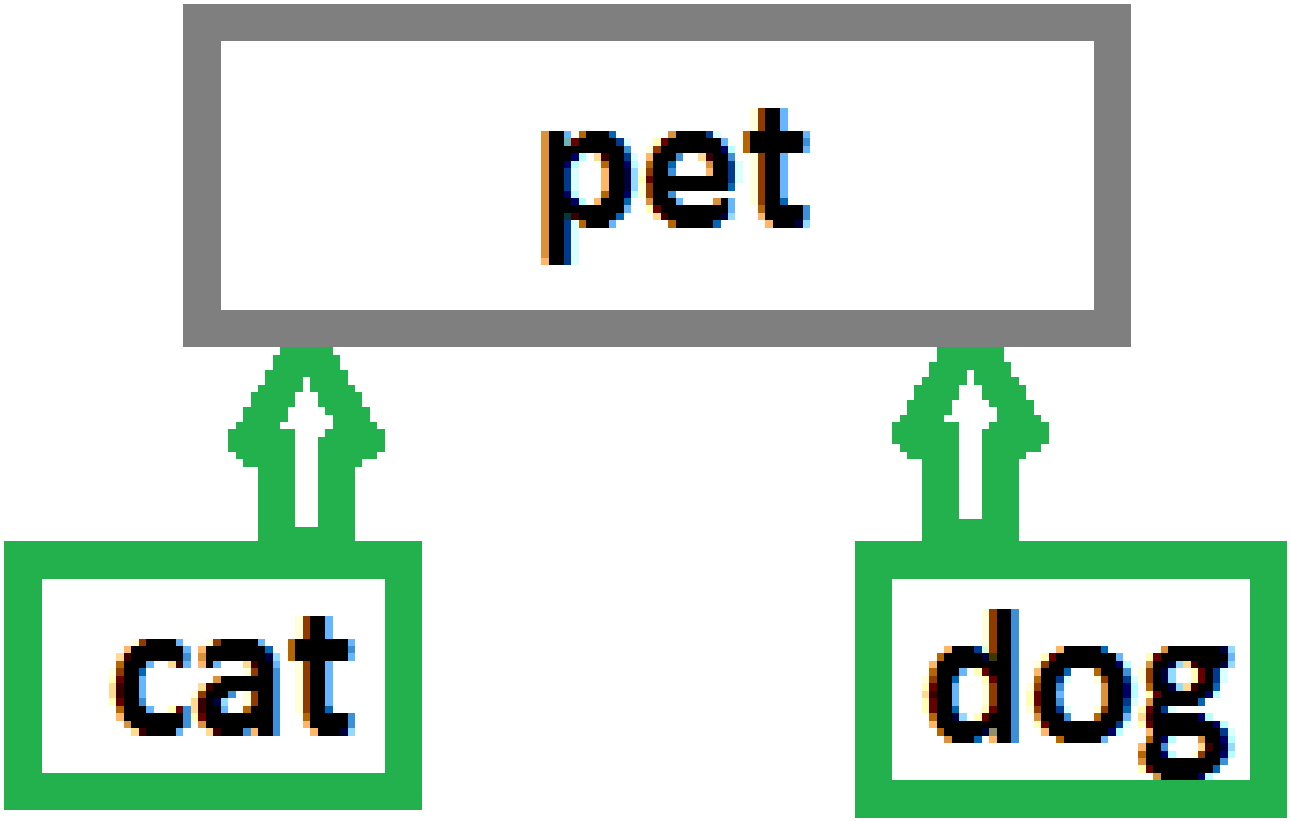
A **cat** *is a* **pet**.

- When the **cat** talks, it should say "meow".
- A **cat** can also scratch furniture.

A **dog** *is a* **pet**.

- When the **dog** talks, it should say "woof".
- A **dog** can also growl at strangers.

A Pet Class



We choose to colour **pet** gray as it is an abstract class, to be discussed.

A Pet Class

```
class pet {
private:
   std::string name; // its name

public:
   pet(std::string _name) : name( std::move(_name) ) { }

   const std::string& get_name() const { return name; }

   virtual void talk() const = 0;

   virtual ~pet() = default;
   pet(pet&&) noexcept = default; // re-enable move constructor
   pet &operator=(pet&&) noexcept = default; // and move assignment
   pet(const pet&) = default; // re-enable copy constructor
   pet &operator=(const pet&) = default; // and copy assignment
};
```

A Pet Class

There are some things to explain.

```
pet(std::string _name) : name( std::move(_name) ) { }
```

The constructor parameter **_name** is copy initialized! Then **_name** is cast to an rvalue reference to invoke the **std::string move** constructor.

This is actually more efficient!

A Pet Class

If we feed an **std::string lvalue** to

```
pet(std::string _name) : name( std::move(_name) ) { }
```

- **_name** is **copied** from that **lvalue** and
- **name** is **move constructed** from **_name**

resulting in **one copy** and **one move**;

and by feeding a **std::string lvalue** to

```
pet(const std::string& _name) : name( _name ) { }
```

- **_name** is a reference to const for that **lvalue** and
- **name** is **copy constructed** from **_name**

resulting in **one reference** and **one copy**.

In this case, the two are roughly equivalent.

A Pet Class

If we feed an **std::string rvalue** to

```
pet(std::string _name) : name( std::move(_name) ) { }
```

- **_name** is **move constructed** from that **rvalue** (efficient) and
- **name** is move constructed from **_name**

resulting in **two moves**;

and by feeding a **std::string rvalue** to

```
pet(const std::string& _name) : name( _name ) { }
```

- **_name** is **move constructed** from that **rvalue** and
- **name** is **copy constructed** from **_name**

resulting in **one copy** and **one move**.

The top version is more efficient as it saves a copy.

A Pet Class

If we feed a string literal to

```
pet(std::string _name) : name( std::move(_name) ) { }
```

- **_name** is constructed from the string literal and
- **name** is move constructed from **_name**;

resulting in **one new construction** and **one move**;

and by feeding a string literal to

```
pet(const std::string& _name) : name( _name ) { }
```

- **_name** is constructed from the string literal and
- **name** is **copy constructed** from **_name**

resulting in **one new construction** and **one copy**.

The top version is more efficient as it saves a copy.

A Pet Class

```
const std::string& get_name() const { return name; }
```

This member function is *not virtual*: we intend for all derived classes to have the same implementation.

A Pet Class

```
virtual void talk() const = 0;
```

The **void talk() const** function is marked as **purely virtual** with the **=0;**. A class with one or more pure virtual function is called an **abstract class** and instances of the class cannot be constructed.

```
pet p("fluffy"); // ERROR: cannot make a pet, it is abstract
```

A Pet Class

```
virtual void talk() const = 0;
```

The pure virtual **void talk() const** function forces all non-abstract derived classes to give an implementation of this function otherwise they will also be abstract.

Making a function to be pure virtual is a way to enforce compliance to a given interface for the derived classes.

A Pet Class

```
pet(pet&&) noexcept = default; // re-enable move constructor
pet &operator=(pet&&) noexcept = default; // and move assignment
pet(const pet&) = default; // re-enable copy constructor
pet &operator=(const pet&) = default; // and copy assignment
```

By declaring a destructor, the compiler does not provide us with move semantics (move constructor and assignment).
We re-enable them by specifying them as **=default**, but in doing so the compiler deletes copy semantics (copy constructor and assignment) so we re-enable those as well.

A Pet Class

```
pet(pet&&) noexcept = default; // re-enable move constructor
pet &operator=(pet&&) noexcept = default; // and move assignment
pet(const pet&) = default; // re-enable copy constructor
pet &operator=(const pet&) = default; // and copy assignment
```

The **noexcept** keyword tells the compiler that an exception will not arise when the function is executed. This allows for some important optimizations.

Any function that will not throw an exception can be marked as **noexcept**, but we'll focus mostly on those pertaining to move semantics.

A Pet Class

```
pet(pet&&) noexcept = default; // re-enable move constructor
pet &operator=(pet&&) noexcept = default; // and move assignment
pet(const pet&) = default; // re-enable copy constructor
pet &operator=(const pet&) = default; // and copy assignment
```

Any base class that has member variables like **std::string**s, **std::vector**s, i.e., the classes that manage resources, should re-enable the move/copy semantics.

For a class that has only simple fundamental type member variables like **int**, **double[5]**, etc., this isn't necessary: copy semantics will be given by default (unless we declare move semantics) and move semantics don't offer benefit.

A Pet Class

Now we write the **cat** class.

```
class cat : public pet {
public:
   cat(std::string _name ) : pet( std::move(_name) ) { }

   void talk() const override { std::cout <<get_name() << "says: meow"
      << '\n'; }

   void scratch_furniture() const { std::cout << "scratch scratch"<< '\n';    }
};
```

As we did not consider other member variables for **cat**, its constructor only needs to initialize the base class.

A Pet Class

```
void talk() const override {
    std::cout << get_name() + " says: meow"<< '\n';
}
```

Since **const std::string& get_name() const** is a **public** member of **pet**, **cat** has direct access to this function and can use it to access the name. It would be an error to try to access the **name**, which is **private**, directly:

```
// not gonna work!
void talk() const override {
    std::cout << name + " says: meow"<< '\n';
}
```

A Pet Class

```
void scratch_furniture() const {
    std::cout << "scratch scratch\n";
}
```

Cats like scratching furniture, but it isn't a general property of a **pet**. This is a member function that belongs to **cat** only. It is not a virtual function, either, just a regular member function.

A Pet Class

```
cat kitty("Fluffy");
std::cout << kitty.get_name() << '\n';

kitty.talk();

kitty.scratch_furniture();
```

```
Fluffy
Fluffy says:  meow
scratch scratch
```

A Pet Class

We similarly define **dog**.

```
class dog : public pet {
public:
    dog(std::string _name) : pet( std::move(_name) ) { }

    void talk() const override {
        std::cout << get_name() + " says: woof"<< '\n';
    }

    void growl_at_stranger() const {
        std::cout <<"grr!\n";
    }
};
```

A Pet Class

Yet again, we have polymorphic behaviour:

```
void make_talk(const pet& a_pet) { a_pet.talk(); }

void make_talk(const pet* a_pet) { a_pet->talk(); }
```

If we call the function as below:

```
cat c{"Fluffy"};
dog d{"Molly"};

make_talk(c);
make_talk(&d);
```

We generate:

```
Fluffy says:  meow
Molly says:  woof
```

A Pet Class

Although we cannot directly create an instance of a **pet** class, we can bind a reference to a **pet** to both **cat** and **dog** objects because both are **pet**s. The same holds with pointers.

Since **talk** is a virtual function, during runtime the proper **talk** function is invoked.

The **static type** (what is directly written) of **a_pet** is **const pet&** (in the first overload) and **const pet*** (in the second overload); however, the **dynamic type** (what the object actually is) could be **cat** or **dog** (in the first overload) or **cat*** and **dog*** (in the second overload). A **virtual** function ensures the appropriate implementation is given, depending on the dynamic type of the object.

Slicing

We cannot convert a base class into a derived class, but we can “slice” and convert a derived class into a base class at the loss of some information.

```
PointMass pm(0, 10);

Point p = pm; // okay, but information is lost...
p.display();
```

```
position = 0
```

We lost all the information about the mass, but the object is still valid.

Slicing

The conversion from a point mass to a point makes sense; a point mass *is a* type of point and we may lose info, but we can live with that.

On the other hand, a point is not necessarily a point mass and there would be information to fill/make up that we wouldn't have inputs for by making a conversion from point to point massr.

```
PointMass pm2 = p; // ERROR: cannot do this conversion
```

Polymorphism is Lost on Copies

Polymorphism only applies to references/pointers. Otherwise only the version of a function appropriate to the static type of the object is called, even if the function is virtual.

```
void print(Point p) {
    // whether passed a Point or PointMass, always displays via
    // Point::display
    p.display();
}

// ...

PointMass pm(3,4);
print(pm); // position = 3
```

We can think of this as the derived implementations being sliced away.

Upcasting and Downcasting

A pointer (or reference) can be **upcast** (cast from a derived to base class) but cannot be **downcast** (cast from a base to a derived class) without a **dynamic cast**.

Suppose that:
pm1 is a **PointMass***
pm2 is a **PointMass** or **PointMass&**

```
/* UPCASTING */

Point *p1 = pm1; // okay
Point& p2 = pm2; // okay
```

Upcasting and Downcasting

Suppose that:
p3 is a **Point ***
p4 is a **Point&**

```
/* DOWNCASTING: all errors below! */

PointMass* pm3 = p3;
PointMass& pm4 = p4;
```

Upcasting and Downcasting

With **downcasting**, we *don't know* if the dynamic type is "derived enough" to be stored statically as the new reference/pointer type. Consider what would happen if this were allowed without careful checking:

```
Point p(7);

PointMass* pm = &p; // let's just imagine this was somehow okay...

pm->display(); // so what we we use for mass?
```

Dynamic Casts

A class is a **polymorphic type** if it has at least one virtual member.

A **dynamic cast** allows for the conversion between pointers and references to **polymorphic types**. It allows us to differentiate between the true dynamic types of polymorphic objects.

A *virtual destructor counts as a virtual member* function!

Dynamic Casts

Dynamic casts to pointers take the form

dynamic_cast<type*>(expression)

- If the conversion is permissible, i.e., the dynamic type of **expression** is, or points to a class derived from **type**, then the result is a pointer of static type **type*** with the *same dynamic type as* **expression**.
- If the conversion is not permissible, i.e., the dynamic type of **expression** is neither **type*** nor a pointer to a class derived from **type**, the result is **nullptr**.

If the dynamic type of the resulting pointer differs from its static type, the pointer still exhibits polymorphic behaviour. Thus the virtual methods of the most derived class to which it points will be used.

Dynamic Casts

Her.- do you have a dog or a cat?
me.- I don't know.

Dynamic Casts

```
void cat_or_dog(const pet* a_pet) {
   const cat* catPtr = dynamic_cast<const cat*>(a_pet); // try for cat

   if(catPtr) { // not null then a cat
      catPtr->scratch_furniture(); // as a cat, can scrach furniture
   }
   else { // then maybe a dog?
      const dog* dogPtr = dynamic_cast<const dog*>(a_pet); // try for dog

      if(dogPtr) { // not null then a dog
         dogPtr->growl_at_stranger(); // as a dog, growls at strangers
      }
   }
}
```

Dynamic Casts

We can safely call **scratch_furniture** in the first branch since we know it's a cat. And we can safely call **growl_at_stranger** in the second branch since we know it's a dog.

Smart Pointers and Inheritance

Smart pointers also obey polymorphism. Consider the code below:

```
// store Points polymorphically
std::vector< std::shared_ptr< Point > > points;

points.push_back( std::make_shared<Point>(3);
points.push_back( std::make_shared<PointMass>(5,6);
for ( const auto& point : points ) { // for each point
   point->display(); // display it
}
```

```
position = 3
mass of 6 at position = 5
```

Dynamic Casts

Smart pointers can also manage dynamic casting.

In managing **std::shared_ptr**s, we use the function **std::dynamic_pointer_cast**.

A **std::unique_ptr** to a base class can point to a derived class and proper virtual functions will be invoked. But **std::dynamic_pointer_cast** cannot be done for **std::unique_ptr**s since we would be allowing another **std::unique_ptr** to point to the same object.

Dynamic Casts

```
void make_massless(const std::shared_ptr< Point>& p) {

    // try to make into PointMass
    auto point_mass = std::dynamic_pointer_cast<PointMass>(p);

    if(point_mass) { // if PointMass (or more derived)
        point_mass->change_m(0);
        point_mass->display();
    }
    else { // cannot change mass, not a PointMass
        std::cout << "not a point mass: no mass to set!\n";
    }
}
```

Dynamic Casts

With the preceding function, we can write:

```
std::shared_ptr<Point> p1 = std::make_shared<Point>(5);
std::shared_ptr<Point> p2 = std::make_shared<PointMass>(10,20);

make_massless(p1);
make_massless(p2);
```

And obtain:

```
not a point mass:  no mass to set!
mass of 0 at position = 10
```

Dynamic Casts

While the **static type** of **p** within the function call is **std::shared_ptr<Point>**, the **dynamic type** in the second call is really **std::shared_ptr<PointMass>** hence the **display** function providing the detail at the level of **PointMass** and not just **Point.**

Account Classes

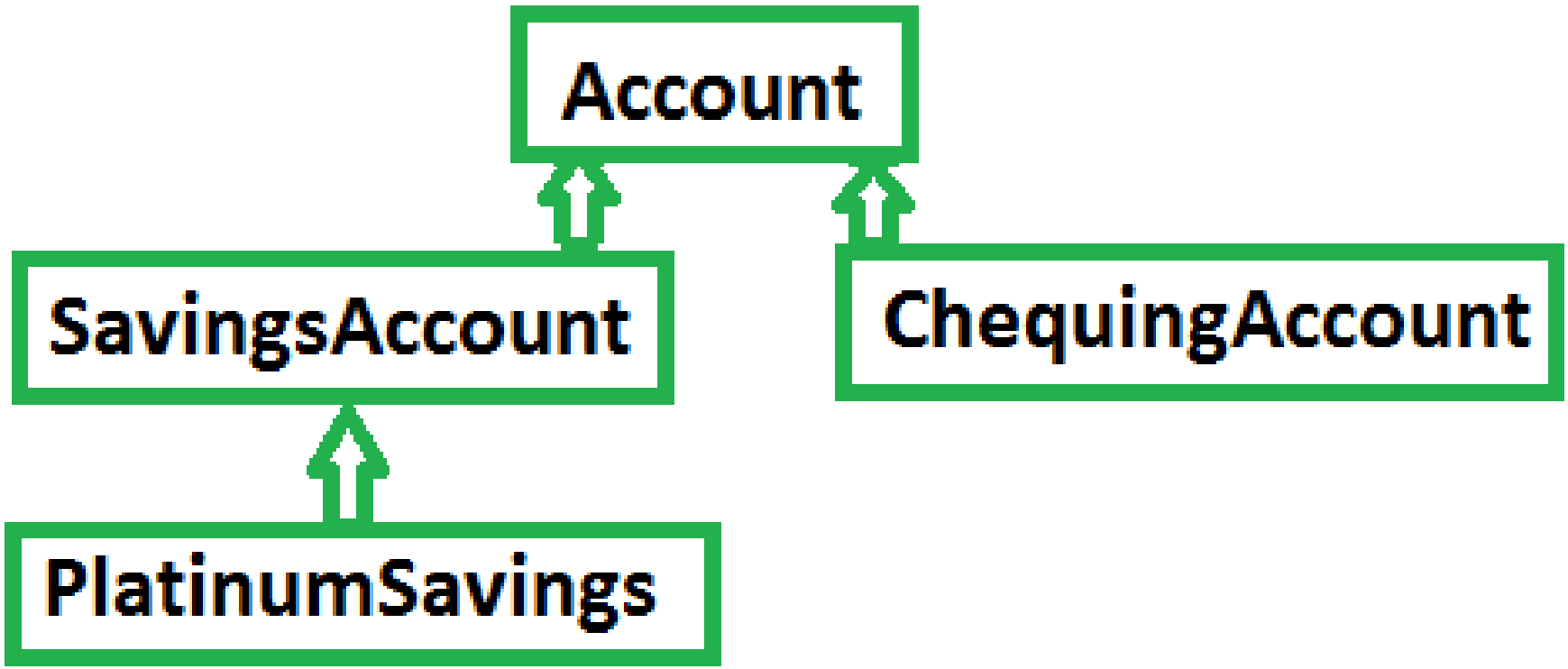
We now consider a more complex inheritance structure here to describe bank accounts. This example will also illustrate the use of **static** member variables/functions.

Account will be a base class, representing a bank account of generic type storing the name, balance, and providing some basic functionalities. *It will also allow us to query the total amount of money in the bank!*

SavingsAccount will inherit from **Account**, offering some earned interest. More specialized than this will be **PlatinumSavings** that offers loan opportunities.

ChequingAccount will inherit from **Account**, offering the ability to write cheques.

Account Classes



Account Classes I

We begin with the **Account** class:

```
class Account{
protected:
    using money_type = double; // money_type means double!

private:
    static money_type bank_money; // total money in bank

    std::string name; // their name
    money_type balance; // money in account
```

Account Classes II

```
protected:
    static void update_bank_money(money_type change) {
        bank_money += change;
    }

    money_type get_balance() const { return balance; }

    const std::string& get_name() const { return name; }
```

Account Classes III

```
public:
    virtual ~Account() = default;

    Account(Account&&) noexcept = default;
    Account(const Account&) = default;
    Account& operator=(Account&&) noexcept = default;
    Account& operator=(const Account&) = default;
```

Account Classes IV

```
  Account(std::string _name, double _balance = 0) :
     name( std::move(_name) ), balance(_balance) {
     update_bank_money(balance);
  }

  void update_balance(money_type change) {
     update_total_money(change);
     balance += change;
  }

  virtual void display(std::ostream& out) const {
     out <<name <<"\tbalance $" <<balance;
  }

  static money_type get_bank_money() { return bank_money; }
};
```

Account Classes

We only discuss the new features of this interface.

```
protected:
    using money_type = double;
```

Within **Account** and its derived classes, **money_type** is a typedef for **double**. It has a nice semantic meaning here. Anywhere we write **money_type**, that is really of type **double**.

Account Classes

```
static money_type bank_money;
```

bank_money is a **static member variable** of **Account**. It can be used to track the sum of all the account balances.

Instead of each **Account** class storing its own version of **bank_money**, there will be a single value stored as a **static variable** and *all instances of the class share this variable.* If somehow **bank_money** changed, all **Account**s would see this effect immediately.

Account Classes

```
static money_type bank_money;
```

Static member variables are usually defined outside the class interface (in a **.cpp** file, say). We must specify their data type and class scope at their initialization. For example:

Account.cpp

```
#include "Account.h"

Account::money_type Account::bank_money = 0;
```

Account Classes

```
protected:
   void update_bank_money(money_type change) const {
      bank_money += change;
   }
```

This function updates the value of the static member **bank_money**.

Note how it is marked as const! A static member variable isn't a "real" member variable of the class; as such, the **const** that we write only describes the **name** and **balance** member variables.

Account Classes

```
static money_type get_bank_money() { return bank_money; }
```

This is a **static member function**. Static member functions only have access to static member variables. They cannot be marked as **const**.

With static member functions, we *do not need to construct a class in order to invoke them!*

```
std::cout <<Account::get_bank_money(); // okay!
```

Above, there aren't even necessarily **Account** member variables in existence and we can call the function with the **Account::** scope.

Account Classes

```
Account(std::string _name, double _balance = 0)
```

Constructors, like other functions, can have default arguments. If a second argument is not given, **_balance** is set to 0.

Account Classes

```
virtual void display(std::ostream& out) const
```

This function accepts a **reference to std::ostream**. Since references can work polymorphically, we can pass types that derive from **std::ostream** such as **std::ofstream** and **std::ostringstream**.

```
Account a("Joe Bruin", 10000);

std::ofstream fout("joe.txt");
a.display(fout); // writes to the file
fout.close();
```

Account Classes I

Now we write the **SavingsAccount** class:

```
class SavingsAccount : public Account {
private:
   static constexpr double rate = 0.005;
public:
   SavingsAccount(std::string _name, double _balance = 0) :
      Account( std::move(_name), _balance) {}

   void gain_interest() {
      // calculate increment
      money_type diff = rate * get_balance();
      update_balance(diff); // now increment
   }
};
```

Account Classes

```
static constexpr double rate = 0.0005;
```

Rather than just having a static member variable, shared among all classes, to store the minuscule interest rate, we optimize with **constexpr** so that the value is set at compile time, too.

A **static constexpr** member variable must be initialized in the interface.

Remark: a member variable that is **constexpr** must also be **static**!

Account Classes I

Now we write the **PlatinumSavings** class:

```
class PlatinumSavings final : public SavingsAccount {
private:
    money_type loaned; // amount on loan

public:
    // initializes Account and makes loan zero
    PlatinumSavings(std::string _name, double _balance = 0) :
        SavingsAccount( std::move(_name), _balance), loaned(0) {}
```

Account Classes II

```
    // updates how much as been loaned
    void update_loan(money_type change) {
        loaned += change;
        update_bank_money(-change); // bank money goes down
    }

    void display(std::ostream& out) const override {
        Account::display(out);
        out <<"\tloaned $" <<loaned;
    }
};
```

Account Classes I

class PlatinumSavings final : public SavingsAccount

When a class is being defined, the **final** keyword can be appended after its name to tell the compiler that no further classes can inherit from it. The **final** keyword can also be used on a virtual function to tell the compiler that the function cannot be overridden.

Account Classes II

```
struct B {
    virtual ~Base() = default;
    virtual int foo() const { return 0; }
};

struct D1 : public B {
    int foo() const final { return 1; }
};

struct D2 final : public B { };

struct X final { };
```

Account Classes III

```
struct D3 : public D1 {
    // ERROR: override final
    int foo() const override { return 2; }
};

struct D4 : public D2 { }; // ERROR: inherit from final
struct Y : public X { };

struct D5 : public B {
    // ERROR: not overriding function of same signature
    void foo() const override;
};
```

Account Classes

Finally we write the **ChequingAccount** class:

```
class ChequingAccount final : public Account {
public:
    ChequingAccount(std::string _name, double _balance = 0) :
        Account( std::move(_name), _balance) {}

    void write_cheque(money_type amount, const std::string& to) const {
        std::cout <<" pay $" <<amount <<" to " <<to <<'\n';
    }
};
```

Protected

In many cases, it is bad practice to use the **protected** keyword. If member variables are protected, it means derived classes can modify them and possibly put the object into an invalid state.

That doesn't mean member functions and **using**s/**typedef**s should not be protected. We just need to be judicious.

Just remember that **encapsulation** is one of the big paradigms of object oriented programming: give access on a need-to-know basis to prevent data corruption.

Polymorphism Subtleties: Preserving Dynamic Type I

All **dynamic_cast**s preserve the dynamic type of an object.

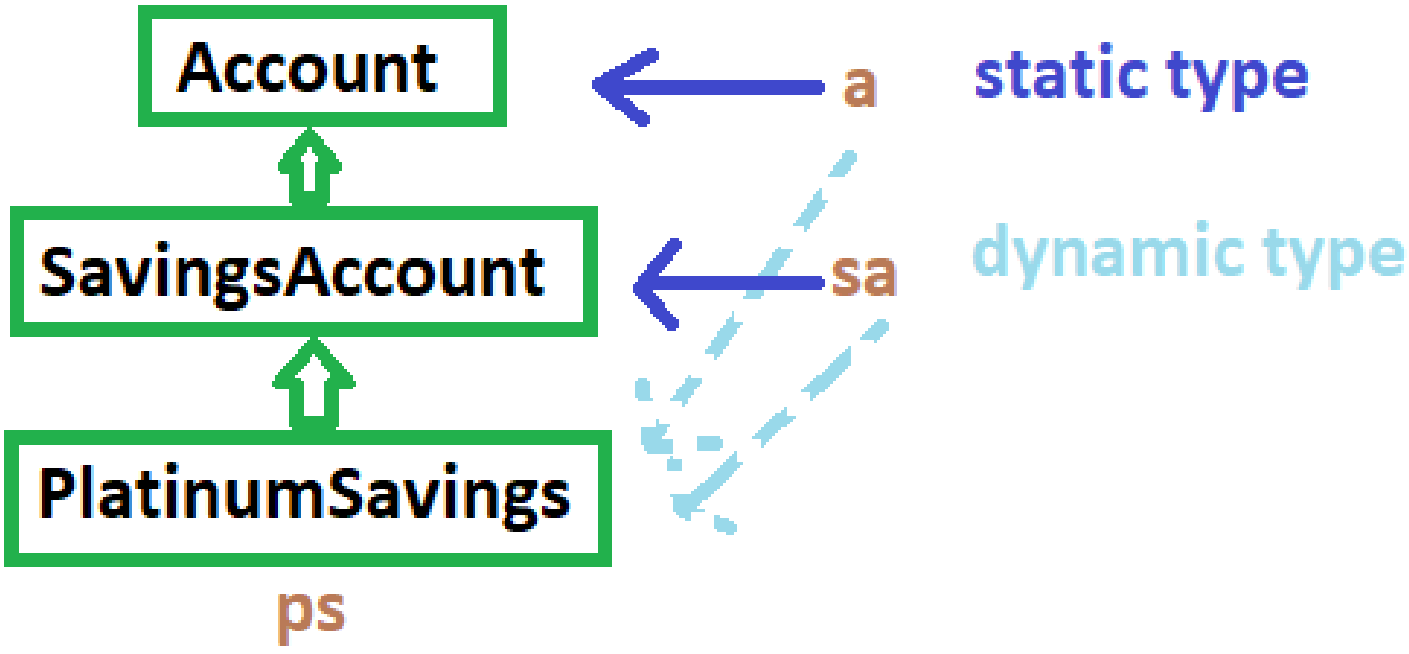
```
PlatinumSavings ps("Jane Doe", 32000);

// static type: Account*, dynamic type: PlatinumSavings*
Account *a = &ps;

// static type: SavingsAccount*, dynamic type: PlatinumSavings*
SavingsAccount *sa = dynamic_cast<SavingsAccount*>(a);
```

Only the static type changes!

Polymorphism Subtleties: Preserving Dynamic Type II



sa->display(std::cout); calls **PlatinumSavings::display**!

Polymorphism Subtleties I

If a function is not virtual in a base class, derived classes can implement their own functions but they mask the function in the base class; such function calls will then apply only to the static object type.

Polymorphism Subtleties II

```
struct B {
    virtual ~B() = default;
    void foo() const { std::cout << 'B'; } // NOT VIRTUAL
};

struct D : public B {
    void foo() const { std::cout << 'D'; }
};

// ...
B *bp = new D;
bp->foo(); // static type used: calls B::foo()
D *dp = new D;
dp->foo(); // static type used, calls D::foo()
```

BD

Polymorphism Subtleties III

```
struct B {
    virtual ~B() = default;
    virtual void foo() const { std::cout << 'B'; } // VIRTUAL
};

struct D : public B {
    void foo() const override { std::cout << 'D'; }
};

// ...
B *bp = new D;
bp->foo(); // dynamic type used: calls D::foo()
D *dp = new D;
dp->foo(); // dynamic type used, calls D::foo()
```

DD

Polymorphism Subtleties IV

For the class **B**, we did not bother with setting the copy and move functionalities to default. The class stores no data so what the compiler generates in the way of copy semantics is sufficient for efficiency. Were the class to store non-trivial member variables like **std::string**s, **std::vector**s, etc., then we would need to. Even with fundamental types like **int**s, **long double**s, etc., there wouldn't be much point in declaring the copy/move semantics since copying would be just as efficient.

Polymorphism Subtleties I

Calling a virtual function from a constructor (or destructor) can be delicate: only the most derived version of that function available can be used, where this availability can only extend as far as the class currently being constructed.

Polymorphism Subtleties II

```
struct A {
   virtual ~A() { }
   virtual void print() const { std::cout << "A"; }
   A () { print(); /* ALWAYS calls A::print, NEVER C::print */ }
};

struct B : public A {
   B () { print(); /* ALWAYS calls A::print, NEVER C::print */ }
};

struct C : public B {
   void print() const override { std::cout << "C"; }
   C () { print(); /* ALWAYS calls C::print, NEVER A::print */ }
};
```

Polymorphism Subtleties III

From the previous classes, by writing

C c;

we would have an output of

AAC

When **C::C()** is called and we do not specifiy how to make **B** in the initlizer list, **B::B()** is called; in turn **B::B()** calls **A::A()**.

From **A::A()**, we invoked **A::print()** printing A;
from **B::B()** after the **A** part is constructed, **A::print()** is invoked printing **A**; and
from **C::C()** after the **B** part is constructed, **C::print()** is invoked printing **C**.

Polymorphism Subtleties IV

C++ Standard: *Member functions, including virtual functions, can be called during construction or destruction. When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, and the object to which the call applies is the object (call it x) under construction or destruction, the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access and the object expression refers to the complete object of x or one of that object's base class subobjects but not x or one of its base class subobjects, the behavior is undefined.*

Polymorphism Subtleties I

But after construction, a member function can invoke a virtual function and that virtual call will be to the most derived function.

Polymorphism Subtleties II

```
struct A {
    virtual ~A() { }
    virtual void print() const { std::cout << "A"; }
    void bar() const { print(); /* COULD BE A::print or B::print */ }
};

struct B : public A {
    void print() const override { std::cout << "B"; }
};

A a;
a.bar(); // A::print invoked, prints A

B b;
b.bar(); // B::print invoked, prints B
```

Polymorphism Subtleties III

The reason is that for, as an example, **b.bar()**, when that function is invoked for **b**, there is a call to **this->bar()**; and **this** is a **B***, which can be properly invoked as **B::bar**.

Virtual Destructor Details

The importance of virtual destructors is best illustrated by an example with the classes below:

```
class Base {
public:
    virtual ~Base() { std::cout <<"~Base"; }
};
class Derived : public Base {
public:
    ~Derived() { std::cout <<"~Derived"; }
};
```

With code such as:

```
Base *bp = new Derived;
delete bp;
```

The output is

```
~Derived~Base
```

Thus, the destruction happens from the most derived class up.

Virtual Destructor Details

On the other hand, executing delete on a pointer to a base class with a dynamic type of a derived class when the destructor has not been declared virtual has **undefined behaviour**.

In some instances, the output might be something like

```
~Base
```

where only the base part of the object would be deleted and the rest of the object would remain on the heap, leaking memory.

Strictly speaking, though, the output is undefined.

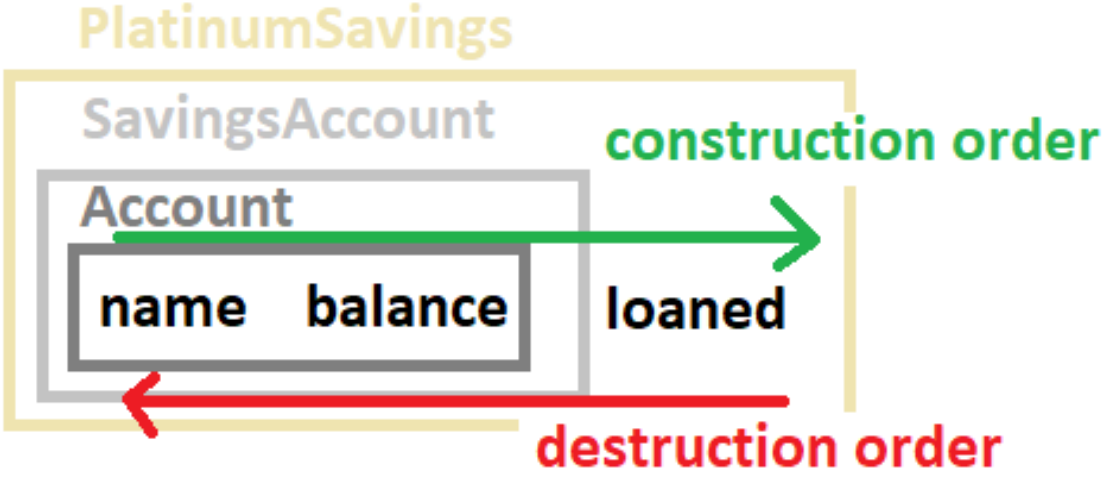
Virtual Destructor Details

During construction: each derived class invokes the constructor of its base class.

A base class initializes its member variables in the order declared.

Each derived class initializes its own member variables in the order they are declared, once the base class has been constructed.

During destruction, if done properly, the class destructors are invoked from most derived upwards to the base, with the local variables to each class being destructed in the reverse order of their declaration.



Try and Catch

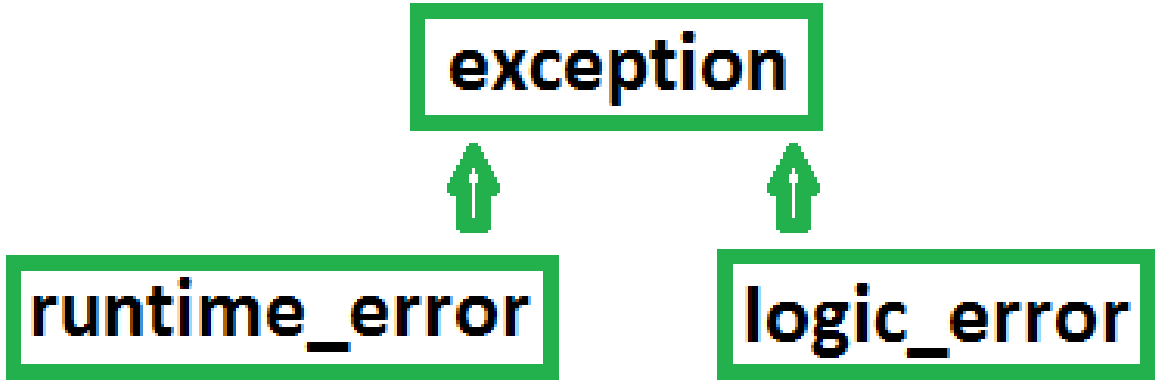
Errors are managed through **try** and **catch** statements. If an exception is thrown, an object, possibly storing error information, is generated and this can be captured by the program and used to manage the error. The pattern is often something like

```
try {
   // stuff to try
}
catch ( const std::exception& e) {
   // stuff to do if failed: usually managing memory issues, etc.
}
```

Try and Catch

The **std::exception** class is defined in the **<exception>** header.

This serves as the base class for many derived classes found within **<stdexcept>** such as **std::logic_error**, **std::runtime_error**, or **std::bad_cast**.



The line is often blurred but a **std::logic_error** typically references logical errors in the code and errors that could be preventable; a **std::runtime_error** references errors that can only be known at runtime.

Try and Catch

Exceptions can be thrown through built-in features of the objects used in the program, or we can throw our own exceptions with the **throw** keyword.

When exception objects derived from **std::exception** are created, it is possible to specify the specific nature of the error by passing a **std::string** to the constructor.

The **std::exception** class itself does not have this type of constructor (except on some compilers).

Try and Catch

An exception object has a **what** member function to return the error description.

When an exception is thrown, a **catch** block will be looked for to handle the exception. If one is not found, the program will terminate.

Try and Catch

Sometimes we handle the exception within the catch block alone and the program can proceed to function correctly; in other cases, though, the program will have to terminate so we can throw the exception again without a catch block and it will end. An error is rethrown by **throw;**.

Multiple catch statements can be used, each catching a specific type of error with **...** indicating the most general catch.

Once an exception has been caught, if the error is thrown again, it is up to an *enclosing catch block*, if there is one, to manage the error.

A program terminates if an exception is uncaught.

Try and Catch I

```
try { // outermost try block
    try {
        // something bad happens
        throw std::runtime_error("something bad");
    }
    catch ( const std::logic_error &LE ) { // logic error?
        std::cerr <<"logic error: "+ LE.what() <<std::endl; // display message
        throw; // and throw again
    }
    catch ( const std::runtime_error& RE) { // or maybe runtime error?
        std::cerr <<"runtime error: "+ RE.what() <<std::endl;
        throw; // and throw again
    }
```

Try and Catch II

```
    catch (const std::exception &E) { // or just any exception?
        std::cerr <<"exception: "+ E.what() <<std::endl;
        throw; // and throw again
    }
    catch (...) { // or anything else?
        std::cerr <<"..."<<std::endl;
        throw; // and throw again
    }
}

catch(...) { // catch anything rethrown from inner try/catches
    std::cerr << "outer catch"<<std::endl;
    throw;
}

std::cout <<"hi";
```

Try and Catch

We throw an exception of type **std::runtime_error** with the statement

```
throw std::runtime_error("something bad");
```

and the description of the error, i.e. what can appear when **what** is called, is set to "something bad".

Only a catch block designed to capture **std::runtime_error**s can capture and manage the exception.

Try and Catch

The runtime error is thrown; the logic error is not suitable (they are separate classes) but the next catch block applies perfectly as it looks for runtime errors:

```
catch( const std::runtime_error& RE) {
    // ...
}
```

Had this block been absent, the next catch for a **const std::exception&** would apply as it serves a base class for **std::runtime_error**. Had this block been absent, the **...** would apply. If nothing suitable were found, the program would just terminate.

Try and Catch

Within the catch block, we have

```
std::cerr <<"runtime error: " <<RE.what() <<std::endl;
throw;
```

std::cerr does effectively the same thing as **std::cout**, printing to the screen, but it is a stream used for reporting errors. It also has a semantic meaning of being an error. Using **std::endl** ensures that anything in the buffer is displayed to the console. In reporting errors this may be quite relevant.

Try and Catch

After printing that message, the extra **throw;** throws the error again; this time, it must be caught by an enclosing **catch** block. Thus, the **catch(...)** in the outermost **try**/**catch** applies and "..." is printed. Then, the exception is thrown yet again but it is not caught so the program terminates.

If we had not added the **throw;** statement within the runtime error catch block, the program would have continued running after printing the error to the screen and printed "hi", i.e. it would have moved past the **try**/**catch** and proceeded to run normally.

The output is:

```
runtime error:  something bad
outer catch
```

and then the program terminates.

Try and Catch I

Consider our **basic::string** class with our **at** member function overloaded on const. We will now throw exceptions if the index is invalid.

Try and Catch II

```
class string {
   // stuff

   char& at(size_t index) {
      /* sz is the size of the string object including the null
      ptr is the pointer to the first element of the dynamic array */

      if( (index < sz-1) ) { // if index within string
         return ptr[index];
      } else { // then subscripting at/beyond null char
         std::cerr <<"index out of range"<< std::endl;
         throw std::logic_error("index out of range");
      }
   }
```

Try and Catch III

```
   char at(size_t index) const {
      /* sz is the size of the string object including the null
      ptr is the pointer to the first element of the dynamic array */

      if( (index < sz-1) ) { // if index within string
         return ptr[index];
      } else { // then subscripting at/beyond null char
         std::cerr <<"index out of range"<< std::endl;
         throw std::logic_error("index out of range");
      }
   }
};
```

Dynamic Casts with References

When the conversion is valid, a dynamic cast to a reference type returns a reference of the desired static type of object (the dynamic type could still differ); otherwise, an exception **std::bad_cast** is thrown. This exception type defined in the **<typeinfo>** header. Dynamic casts to references take the form

dynamic_cast<type&>(expression)
(provided “expression” is an lvalue) or

dynamic_cast<type&&> (expression)
(provided “expression” is an rvalue).

Dynamic Casts with References

Suppose that **acc** is of type **Account&**. We could then have:

```
try { // if we succed, dynamic type is ChequingAccount
    const ChequingAccount &cheque =
        dynamic_cast<const ChequingAccount &>(acc);

    // can do this since ChequingAccount
    cheque.write_cheque(2000, "Joe Bruin");
}

// failure means conversion not possible
catch ( const std::bad_cast &bad) {
    std::cout << "cannot write cheque";
}
```

Dynamic Casts with References

Remark: in the case of casting, an exception is not a "bad thing"; it is the only natural mechanism to indicate a cast cannot be done. Pointers can be null, but not references.

Dynamic Casts

Constness matters. One cannot cast a reference to const to a reference (or a pointer to const to an ordinary pointer) by means of static or dynamic casts.

Suppose **acc** is a **const Account&**.

```
PlatinumSavings& plat =
   dynamic_cast<PlatinumSavings&>(acc); // ERROR: acc is const!
```

A **dynamic_cast** is only to be used to convert between polymorphic types and the constness must be protected in the process.

Exceptions and Stack Unwinding

When an exception is thrown, the function in which it occurs (or the **try** block is exited); before the exiting process, the destructors of all objects that had been created before the exception and which are local to that scope are called.

As the local variables are stack-based, they are destroyed in the reverse order of their creation.

Exceptions and Stack Unwinding

When *RAII classes* such as the C++ container classes have their *destructors called*, they also *destruct each element* that they store.

Fundamental types don't have destructors!!! But they still cease to exist when they are destroyed, either because their container destroys them or because they go out of scope.

When **raw pointers are destroyed**, **delete** or **delete []** are not called!!!

Exceptions and Stack Unwinding

Then, the exception is passed to an appropriate **catch** block or to the **function that called it** if no catch is available.

This process repeats, destroying local variables and exiting functions and **try**s, until an appropriate catch block is found or the exception is taken to **main** and no **catch** is found in which case the program would terminate.

This process is called **stack unwinding**.

Exceptions and Stack Unwinding I

```
struct X{
    char c;

    X(char _c) : c(_c) {}

    ~X() { std::cout << c; }
};
```

Exceptions and Stack Unwinding II

```
void Foo(){
    X x('C');
    throw std::exception();
    X x2('R');
}

void Bar() {
    X x('A');
    Foo();
    X x2('B');
}
```

Exceptions and Stack Unwinding III

```
int main() {
    // ...

    try {
        X x('S');
        X x2('T');
        Bar();
        X x3('M');
    }
    catch (...){ std::cout << "!"; }

    std::cout << "\nstillrunning";

    // ...

}
```

Exceptions and Stack Unwinding

The preceding output is:

```
CATS!
still running
```

In **main**, two **X**s are created storing **'S'** and **'T'**, respectively, then **Bar()** is called.

Inside **Bar()**, an **X** is created storing **'A'** and then **Foo()** is called.

Inside **Foo()**, an **X** is created storing **'C'** and then an exception is thrown. No more of **Foo()** is executed because there is no **catch** block so the second **X** is never created!

Exceptions and Stack Unwinding

From the above, the scope of **Foo()** must be exited, so the **X** with **'C'** is destoyed, resulting in the first **char** printed.

The exception is passed onto the caller of **Foo()**, namely **Bar()**.

Bar() does not have a mechanism to manage exceptions so its scope must also be exited now. In the process, the **X** with **'A'** is destroyed, printing the second **char**. **x2** is never created and the exception is passed to **Bar()**'s caller, **main()**.

Exceptions and Stack Unwinding

There is an exception-handling mechanism in **main**.

First, the local scope of its **try** must be exited, destructing **x2** and **x** (the reverse order of their creation because they are stack based), printing "TS". The **x3** is never created.

The exception is passed to the **catch** and **"!"** is printed.

As the exception was not rethrown, the program continues to run, printing the next message.

Exceptions and Stack Unwinding

Further note: if a constructor throws an exception that is not managed within the constructor call, the object's destructor is *not necessarily called* and this can lead to a memory leak.

C++ Standard: *If the initialization ... of an object ... is terminated by an exception, the destructor is invoked for each of the object's direct subobjects and, for a complete object, virtual base class subobjects, whose initialization has completed... Such destruction is sequenced before entering a handler of the function-try-block of the constructor ... if any.*

The destructor is invoked for each automatic object of class type constructed ... since the try block was entered. ... The objects are destroyed in the reverse order of the completion of their construction.

If the object was allocated by a new-expression, the matching deallocation function, if any, is called to free the storage occupied by the object.

Exceptions and Stack Unwinding

For simplicity, we consider a rather contrived example of where/why this is relevant. Basically, *constructors should throw exceptions* if they would create an object of invalid state. Consider the class **Int** that wraps around an **int** and is constructed by reading from a stream:

```
struct Int {
    int i;
    Int(std::istream& in) { // recall: streams are always passed by reference!
        in >> i;
        if(in.fail()) { // did not read correctly
            throw std::runtime_error("not valid"); // invalid int
        }
    }
};
```

Exceptions and Stack Unwinding

Now, for whatever reason, we have a class **TwoInts** that stores two **Int** pointers. We might use the class like:

```
std::vector<TwoInts> v;
size_t items = 0; // how many in vector

while(items < 10) { // until we have 10 of these
   try { // attempt to make a TwoInt and append it

      v.push_back( TwoInts(std::cin) ); // read from console might throw
      ++items; // successful so increase count
   }
   catch(...) { // problem reading values
      // call a function that fixes the std::cin stream
      some_function ( std::cin );
   }
}
```

Exceptions and Stack Unwinding

The real question, though, is: how do we write **TwoInts** to be exception safe and not leak memory?

Just because an exception is thrown, it does not mean a program must terminate. We are, after all, managing the exceptions so it is quite reasonable to expect the program to run, not crash, *and not leak memory!*

Exceptions and Stack Unwinding

With smart pointers, the solution is easy:

```
class TwoInts {
private:
   std::unique_ptr<Int> i1, i2;
public:
   TwoInts( std::istream& in ) : i1( new Int( in ) ), i2 ( new Int( in ) ) { }
};
```

If the first **new** fails due to an exception, **i1** is never constructed and the potential memory for **i1** is freed.

If the second **new** fails, **i1** is destructed and since it is a smart pointer, the memory is freed up.

Exceptions and Stack Unwinding

On the surface, with raw pointers, this looks correct (but it is not!):

```
class TwoInts {
private:
   Int *i1, *i2;
public:
   TwoInts( std::istream& in ) : i1( new Int( in ) ), i2 ( new Int( in ) ) { }
   ~TwoInts() { delete i1; delete i2; }
};
```

If the first **new** fails then the memory is freed and **i1** and **i2** were never constructed...

But if the second **new** fails, since *the class destructor will not be invoked*, **i1** is destroyed as a raw pointer. But this does not free memory!

Exceptions and Stack Unwinding

Here is the correct way with raw pointers.

```
class TwoInts {
private:
   Int *i1, *i2;
public:
   TwoInts( std::istream& in ) : i1(nullptr), i2(nullptr) {
      try { // try to make two valid objects
         i1 = new Int( in );
         i2 = new Int( in );
      }
      catch(...) { // free memory if there is a problem
         delete i1; delete i2; i1 = i2 = nullptr;
         throw; // throw exception onwards!
      }
   }
   ~TwoInts() { delete i1; delete i2; }
};
```

Exceptions and Stack Unwinding

The pointers are initially **nullptr**. If any exception is thrown, although we do not proceed to the destructor, we still call **delete**, owing to the **catch(...)**. And we can safely delete **nullptr**, just in case a pointer had not been set before the exception occurs.

It is necessary to **throw** again within the constructor or else we would have created a "zombie object" by allowing the program to continue running normally, despite having **TwoInts** storing two **nullptr** values.

VTables

Whenever a class has a virtual function, or is derived from a class with virtual functions, it is given a **vtable** (a.k.a. virtual function table, or virtual method table, or dispatch table).

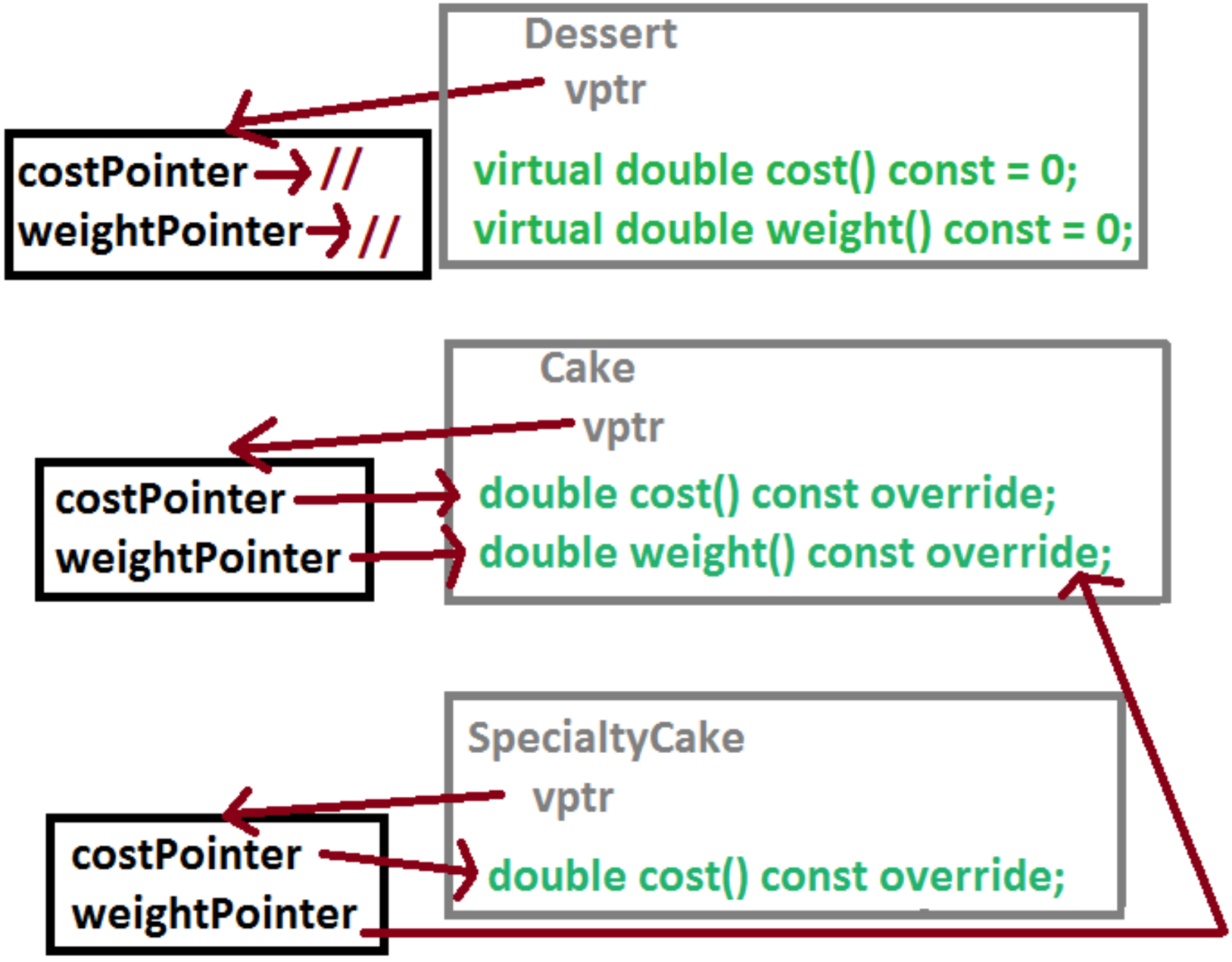
A vtable stores a static array of function pointers (pointers that point to the location in memory for a function).

The base class is given a pointer to this vtable and all derived classes have a pointer to their own respective vtables.

The function pointers of the vtables point to the most derived version of the virtual function possible for that class.

Due to the extra pointer-tracing, virtual functions can be slower than non-virtual functions; on modern computers, this is seldom an issue, however.

VTables


Dessert
vptr
virtual double cost() const = 0;
virtual double weight() const = 0;
costPointer → //
weightPointer → //
Cake
vptr
double cost() const override;
double weight() const override;
costPointer
weightPointer
SpecialtyCake
vptr
double cost() const override;
costPointer
weightPointer

VTables

```
// assume SpecialtyCake has a default constructor
std::shared_ptr<Dessert> dessert( new SpecialtyCake );

auto cake = std::dynamic_pointer_cast<Cake>( dessert );

if(cake) { // either Cake or SpecialtyCake
    cake->cost(); // invokes SpecialtyCake::cost
    cake->weight(); // invokes Cake::cost
}
```

Function Pointers

We'll discuss function pointers more broadly.

When a function is written, it is given a place in code memory during the execution of a program. This function can be accessed by an address, just like a variable. Consider the function

```
int add3(int a, int b, int c) {
    return a+b+c;
}
```

Function Pointers

The function type of **add3** is **int(int, int, int)**, i.e. its signature.

A pointer to a such a function has type **int (*)(int, int, int)** - note the parenthesis around the asterisk!

int* (int, int, int) denotes a function taking in 3 **int** inputs and returning a pointer to an **int**!

Function Pointers

```
int (*g)(int, int, int) = &add3; /* g points to the totalSeconds
    function */
time_t (*h)(time_t *) = std::time // h points to the std::time function
```

When initializing a function pointer, the preceding address-of operator is not necessary: a function name is treated as a pointer to that function (*more precisely, the function "decays" into a pointer*).

To call the pointed-to function, the dereferencing is optional:

```
g(1,2,3); // okay
(*g)(2,4,6); // okay
```

In essence, *function pointers allow us to treat functions as objects*. We can pass a function as a parameter.

A function is an lvalue!

Aside: PRvalue vs Xvalue

Polymorphism is one time we care about the difference between a **prvalue** and **xvalue**, as opposed to grouping them both as **rvalues**.

```
Account as_pr(const PlatinumSavings& ps){
    return ps; // copy initializes the prvalue returned, does a slice
}

Account&& as_x(PlatinumSavings& ps) {
    return std::move(t); // function returns an xvalue
}
```

If **joe_account** is a **PlatinumSavings** then

```
as_pr(joe_account).display(std::cout); // does Account::display
as_x(joe_account).display(std::cout); // does PlatinumSavings::display
```

print differently: the first only prints the **Account** part of **joe_account** as prvalues are not polymorphic; the second prints **joe_account** as a **PlatinumSavings** as xvalues can be polymorphic.

Aside: Pure Virtual Functions

The purpose of a pure virtual function is to enforce compliance in derived classes. It *does not mean a pure virtual function cannot be defined!*

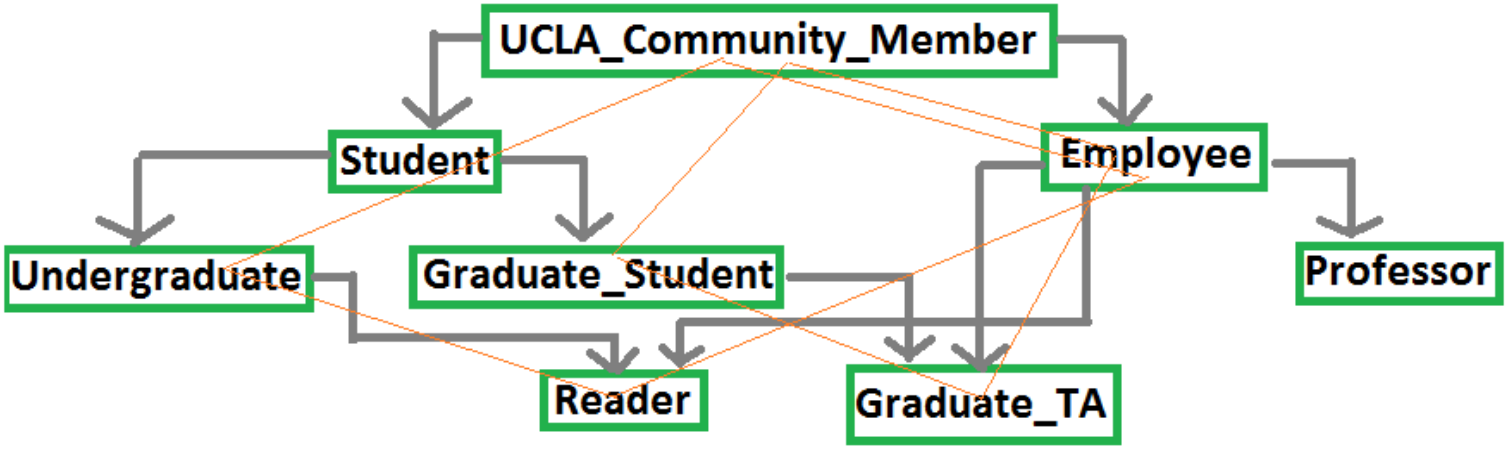
```
struct Base{
    virtual ~Base() = default;
    virtual void foo() const = 0;
};

struct Derived : Base {
    void foo() const override { Base::foo(); }
};

// definition of pure virtual function...
void Base::foo() const { std::cout <<"foo"; }
```

Aside: Multiple Inheritance

It doesn't take much for an inheritance heirarchy to become vastly more complex. A class can inherit from multiple classes, too. But this brings up issues such as the **"deadly diamond of death"** and the need for **virtual inheritance** to avoid multiple copies of a base class appearing in a derived class.



We won't discuss this, but when a class inherits from multiple classes, care is necessary.

Aside: Different Types of Inheritance

Besides **public** inheritance, there also exist **protected** and **private** inheritance (and **virtual** inheritance in the case of multiple inheritances).

```
class X : public Base { /* stuff */ };
```

What was **public** within **Base** is **public in X**.

What was **protected** within **Base** is **protected in X**.

What was **private** within **Base** is **not directly accessible from X** (but could be through public or protected members of the base).

With no keyword specified, the default is **public** for **struct** inheritance.

Aside: Different Types of Inheritance

```
class Y : protected Base { /* stuff */ } ;
```

What was **public** within **Base** is **protected in Y**.

What was **protected** within **Base** is **protected in Y**.

What was **private** within **Base** is **not directly accessible from Y** (but could be through public or protected members of the base).

Aside: Different Types of Inheritance

```
class Z : private Base { /* stuff */ };
```

What was **public** within **Base** is **private in Z**.

What was **protected** within **Base** is **private in Z**.

What was **private** within **Base** is **not directly accessible from Z** (but could be through public or protected members of the base).

With no keyword specified, the default is **private** for **class** inheritance.

Summary

- When a class satisfies the *is a* relationship with another class, it could inherit from that class.
- Polymorphism is the ability to deal with objects of many types (with the same underlying base class).
- Functions that are virtual can be given their own implementation in derived classes, different to their base class.
- The destructor should be declared as virtual when a class will serve as a base class otherwise the derived classes not properly be destroyed when dealing with polymorphism.
- When a virtual function is invoked upon a pointer or reference to a base class, the most derived form of the function will be invoked; behind the scenes this is managed by vtables.

Summary

- Dynamic casts allow for conversion between polymorphic types; it is always possible to upcast pointers or references (towards the base class) but not possible to downcasat (to derived class) without dynamic casts.
- **try** and **catch** statements allow for error handling.
- When exceptions occur, the stack unwinds, destroying objects and exiting scopes to find a suitable **catch**.

Exercises

1. Summarize the meaning/usage of **virtual**, **=0**, **override**, **final**, **static**.
2. Explain why virtual destructors are important; also explain why re-enabling copy/move semantics may be necessary.
3. Write ...
   - an abstract class **Shape** with **get_area** and **get_perimeter** functions;
   - **Square** and **Circle** classes, which have their side lengths/ radii of **double** type assigned at construction, and which define the **get_area** and **get_perimeter** functions; and
   - **ColouredSquare** and **ColouredCircle** classes, which store an additional **std::string** for their colour, which is assigned during construction along with their lengths/radii, and which can be returned from a **get_colour** function.
4. Add to the classes above in such a way so that **Shape::get_total_area** and **Shape::get_total_perimeter** will return the total area/perimeter of all shapes in the program.
5. Write a function that accepts either (smart) pointers or references to a **Shape** and which prints the type of object "Square", "ColouredCircle", etc., to the console.

Exercises

6. While preserving (the admittedly contrived) steps below, make the code below safe with respect to memory leaks and exceptions (do it with smart pointers and with raw pointers, separately):

```
std::vector<int*> v;
for(size_t i=0; i < 10; ++i) {
  v.push_back( new int( std::rand() ) );
}
for (int* i :  v ) { std::cout <<*i; }
for (int* ip :  v) { delete ip; }
```

7. What happens when a virtual function is invoked in a constructor?
8. Write functions **double_value** and **triple_value** that accept **int**s and double/triple their arguments, respectively. Now, write a function **change_vector** that accepts a **std::vector**<**int**> and a *function pointer* and performs the operation on each element of the vector. If **v** is a **std:vector**<**int**> then **change_vector(v, double_value)** should double all of its values.