C++: Initialisation, cleanup & inheritance

David Marshall

School of Computer Science & Informatics
Cardiff University

CM2204

All Lecture notes, code listings on CM2204 Web page

More on Classes: Initialisation

Consider the `BankAccount` class we defined in Week 8 (BankAccountCppStruct.h, BankAccountCppStruct.cpp & BACStructTest.cpp):

- We provided a function to initialise objects:

```
BankAccount a, b;
a.initialise("Stuart");
b.initialise("Bob");
```

- This approach is **not ideal**:
  - Inconvenient syntax;
  - Allows the object to be accessed before initialisation;
  - Lack of control – *e.g.* we would like to ensure all bank accounts have a `name` **defined**.
- These can be addressed by defining **constructors**.

Constructors

- Constructors have the same **name** as the **class**
- Constructors are called **automatically** when a **variable** is **declared**
- Constructors can take **arguments**
- Constructors can be **overloaded**
- Constructors have **no return value**
- If **no** constructor is provided, a **default** constructor with no arguments is **automatically generated**
- If any constructor is provided, the **default** constructor is **not available**

Simple Constructor Example

ConstructorExample.cpp

```
#include <iostream>
using namespace std;

class A {
private:
int a;
public:
A(); // Constructor
};

int main() {
A a;

}
```

Destructors

- Destructors perform **cleanup**, e.g.:
  - **Free** any memory allocated in a constructor (or elsewhere)
  - **Close** files, database connections, etc.
- Destructors have the same name as the class **with** a **prefixed ~**
- **Called** when the object goes out of **scope**

Simple Destructor Example

DestructorExample.cpp

```
#include <iostream>
using namespace std;

class A {
private:
int* a;
public:
A(int size);   // Declare Constructor
~A();          // Declare Destructor
};

A::A(int size) { // Constructor
cout << "Creating" << endl;
a = new int[size];
}

A::~A() { //  Destructor
cout << "Deleting" << endl;
delete [] a;
}

int main() {
A a(50);
}
```

A Complete Constructor/Destructor Example

Constructor1.cpp

```
#include <iostream>
using namespace std;

class Tree {
  int height;
public:
  Tree(int initialHeight);  // Constructor
  ~Tree();  // Destructor
  void grow(int years);
  void printsize();
};

Tree::Tree(int initialHeight) {
  height = initialHeight;
}

Tree::~Tree() {
  cout << "inside Tree destructor" << endl;
  printsize();
}
```

A Complete Constructor/Destructor Example Cont.

Constructor1.cpp Cont.

```
void Tree::grow(int years) {
  height += years;
}

void Tree::printsize() {
  cout << "Tree height is " << height << endl;
}

int main() {
  cout << "before opening brace" << endl;
  {
    Tree t(12);
    cout << "after Tree creation" << endl;
    t.printsize();
    t.grow(4);
    cout << "before closing brace" << endl;
  }
  cout << "after closing brace" << endl;
}
```

Inheritance vs composition

- So far the **classes** we have defined have been **composed** of other types

```
class X {
  string s; }
```

  - I.e. each object of type `X` **has a** string `s`
- When using **inheritance** we model the **is a** relationship between classes
- The syntax used is:

```
class Y : public X {}
```

- We call `Y` the *subclass* and `X` the *base class*
- The class `Y` automatically gets all members of `X` (data and functions)

Back to our bank account example

Building on previous bank account example:

- A new subclass `SavingsAccount`

BankAccount1.h

```
#include <string>
#include <sstream>

.......... BankAccount Class Definition ..........

class SavingsAccount : public BankAccount {

private:
        float interestRate;

public:
        SavingsAccount(float rate);
        void addInterest();
};
```

For full code listing and example call see: BankAccount1.h, BankAccount1.cpp & BA1Test.cpp

Construction is a little untidy here though!

BankAccount1.cpp

```
SavingsAccount::SavingsAccount(float rate) {
        interestRate = rate;
        balance = 0;
}

void SavingsAccount::addInterest() {
        balance = balance * (1.0 + interestRate);
}
```

Inheritance and constructors

- Constructors guarantee correct Initialisation
- When an object `x` is composed of **sub-objects**, the constructor for `x` will call the constructors for the sub-objects
- When using **inheritance**, we (usually) **need** to **ensure** the constructors for any **base classes** are **called**
- The syntax for doing this is :

```
class Y : public X {}
Y::Y() : X() {}
```

- The same syntax can also be used to **initialize** members
- There is no need to explicitly call destructors for base classes – this is done automatically

A Second Version of SavingsAccount Inheritance

BankAccount1.h

```
class BankAccount {

protected:
        std::string name;   //  WE ADD a name
        float balance;   // Account balance

.......... rest BankAccount Class Definition ..........

};

class SavingsAccount : public BankAccount {

private:
        float interestRate;

public:
        SavingsAccount(std::string inName, float rate);
        void addInterest();
};
class SavingsAccount : public BankAccount {

private:
        float interestRate;

public:
        SavingsAccount(float rate);
        void addInterest();
};
```

Cleaner Inheritance for SavingsAccount

```
BankAccount::BankAccount(std::string inName) {
        name = inName;
        balance = 0; // ... or implicitly
}

.............

SavingsAccount::SavingsAccount(std::string inName, float
    rate) : BankAccount(inName) {
        interestRate = rate;
}

void SavingsAccount::addInterest() {
        balance = balance * (1.0 + interestRate);
}
```

For full code listing and example call see: BankAccount2.h, BankAccount2.cpp & BA2Test.cpp

Function overriding

- Sometimes we want to redefine the behaviour of a function when we inherit a class – this is called **overriding**
- Suppose we have classes `X` and `Y` where

```
class Y : public X {
```

- We use the `virtual` keyword in the definition of `Y` to denote that it is available to be overridden

```
  virtual void doSomething() {
       // Base class implementation }
```

- This can then be overridden in the definition of `Y`:

```
  void doSomething() { // Sub class implementation }
```

A `SavingsAccount` `virtual` function example

Reporting `rate` in `SavingsAccount` `report()`

- The `report()` function (inherited from `BankAccount`) **should** report `rate`.

A SavingsAccount virtual function example: Class Definitions

BankAccount3.h

```
class BankAccount {

protected:
        std::string name;
        float balance;  // Account balance

public:
        BankAccount(std::string inName);
        ............
        virtual std::string report();
};

class SavingsAccount : public BankAccount {

private:
        float interestRate;

public:
        SavingsAccount(std::string inName, float rate);
        void addInterest();
        std::string report();
};
```

Overriding the `report()` function

BankAccount3.cpp

```
BankAccount::BankAccount(std::string inName) {
        name = inName;
        balance = 0; // ... or implicitly
}

.........

std::string BankAccount::report() {
        std::stringstream ss (std::stringstream::in | std::stringstream::out);
        ss << name << " " << balance;
        return ss.str();
}

SavingsAccount::SavingsAccount(std::string inName, float rate) : BankAccount(
    inName) {
        interestRate = rate;
}

........

std::string SavingsAccount::report() {
        std::stringstream ss (std::stringstream::in | std::stringstream::out);
        ss << name << " " << interestRate << " " << balance;
        return ss.str();
}
```

Full code listing: BankAccount3.h, BankAccount3.cpp & BA3Test.cpp

Upcasting and `virtual` functions

Upcasting:

Taking the address of an object (either a pointer or a reference) and treating it as the address of the base type.

- Arises because of the way inheritance trees are drawn with the base class at the top.

Upcasting Problem (and solution)

Consider the following code:

Instrument2.cpp

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
```

Upcasting Problem Cont.

Instrument2.cpp Cont.

```
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
}
```

What is the Problem Here?

- The problem with Instrument2.cpp can be seen by running the program.
  The output is

```
Instrument::play
```

  This is clearly **not** the **desired** output.
- We know object is actually a `Wind` and **not** just an `Instrument`.
- The ideal output **should** produce

```
Wind::play
```

Any object of a class derived from Instrument should have its version of `play( )` .

Solution: Function call binding

A `virtual` rescue:

Late binding with the `virtual` keyword.

- binding occurs at runtime, based on the type of the object.

Late binding is also called **dynamic binding** or **runtime binding**

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
```

Note: Defensive Programming Principle

```
class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
```

- `const` specifies that the function **cannot change** any members
  - **Defensive Programming**!

Late Binding Example Cont.

```
// Wind objects are Instruments because they have the same
class Wind : public Instrument {
public:
  // Override interface function:
  void play(note) const { cout << "Wind::play" << endl;}
};

void tune(Instrument& i) { // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

Note: `void tune()` function does not care what **Instrument** plays — all **Instrument**s have play

Abstract classes

- Inheritance allows us to specify an *interface* that subclasses will adhere to
- Often, it doesn't make sense to have objects of the base class
  - E.g. what should the implementation for the `Instrument` class contain?
- A class that can't be instantiated is called *abstract*
- This is done by making one of it's functions a *pure virtual function*:

```
virtual void doSomething() = 0;
```

- Any subclass must then provide a definition of `doSomething`, or it will also be abstract

Abstract class example

Shapes.cpp

```
#include <iostream>
using namespace std;

class Shape {

public:
        virtual float getArea() = 0; // Pure virtual
};

class Square : public Shape {
private:
        int length;

public:
        Square(int n) {
                length = n;
        }
        float getArea();
};
```

Abstract class example cont.

Shapes.cpp cont.

```
float Square::getArea() {
        return length * length;
}

class Rectangle : public Shape {
private:
        int length;
        int width;

public:
        Rectangle(int n, int m) {
                length = n;
                width = m;
        }
        float getArea();
};
```

Abstract class example cont.

Shapes.cpp cont.

```
float Rectangle::getArea() {
        return length * width;
}

void printArea(Shape& s) {
        cout << s.getArea() << endl;
}

int main() {
        // Shape bob;
        Square s(5);
        printArea(s);
        Rectangle r(5, 4);
        printArea(r);

}
```

Multiple Inheritance

- C++ supports Multiple Inheritance
  - although this is not often the best way to program (see below)
- Inheritance from two classes could be via:

Multilevel Inheritance

```
class A
{ .... ... .... };
class B : public/private/.. A
{ .... ... .... };
class C : public/private/.. B
{ .... ... .... };
```

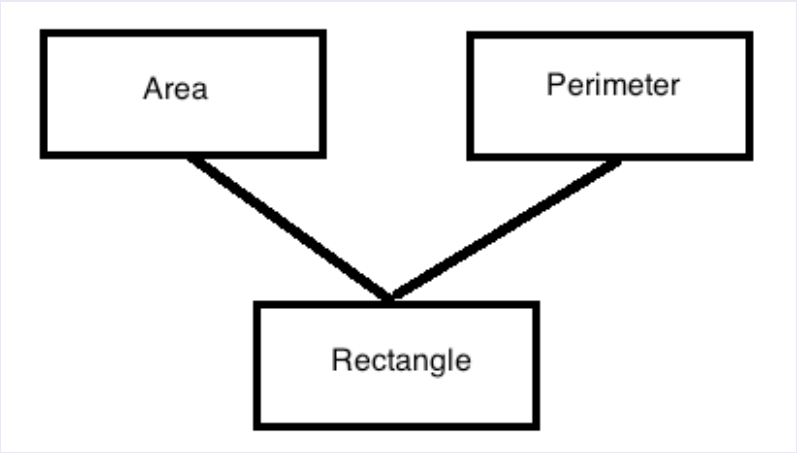
or

Multiple Inheritance

```
class A
{ .... ... .... };
class B
{ .... ... .... };
class C : public/private/.. A , public/private/.. B
{ .... ... .... };
```

Simple Multiple Inheritance Example

Shapes



A class `Rectangle` is derived from base classes `Area` and `Rectangle`.

- There is clearly no real need for Multiple Inheritance here, but this is a simple illustration

Simple Multiple Inheritance Example Cont.

multi_inherit.cpp

```
#include <iostream>
using namespace std;
class Area
{
  public:
    float area_calc(float l,float b)
    {
        return l*b;
    }
};

class Perimeter
{
  public:
    float peri_calc(float l,float b)
    {
        return 2*(l+b);
    }
};
```

Simple Multiple Inheritance Example Cont.

multi_inherit.cpp Cont.

```
/* Rectangle class is derived from classes Area and Perimeter. */
class Rectangle : private Area, private Perimeter
{
    private:
        float length, breadth;
    public:
        Rectangle() : length(0.0), breadth(0.0) { }
        void get_data( )
        {
            cout<<"Enter length: " << endl;
            cin>>length;
            cout<<"Enter breadth: " << endl;
            cin>>breadth;
        }

        float area_calc()
        {
        /* Calls area_calc() of class Area and returns it. */
            return Area::area_calc(length,breadth);
        }

        float peri_calc()
        {
        /* Calls peri_calc() function of class Perimeter and returns it. */
            return Perimeter::peri_calc(length,breadth);
        }
};
```

Simple Multiple Inheritance Example Cont.

multi_inherit.cpp Cont.

```
int main()
{
    Rectangle r;
    r.get_data();
    cout<<"Area = "<<r.area_calc() << endl;
    cout<<"\nPerimeter = "<<r.peri_calc() << endl;
    return 0;
}
```

Problems with Multiple Inheritance

While multiple inheritance seems like a **simple** extension of single inheritance, multiple inheritance:

- introduces a lot of issues that can markedly increase the complexity of programs
- make code maintenance a nightmare.

Two common problems:

- Ambiguity — when multiple base classes contain a function with the same name.
- Diamond of Doom — when a class multiply inherits from two classes which each inherit from a single base class

Problems with Multiple Inheritance: Ambiguity

Consider this code fragment:

```
class base1
{
  public:
     void some_function( )
     { .... ... .... }
};
class base2
{
    void some_function( )
     { .... ... .... }
};
class derived : public base1, public base2
{

};

int main()
{
    derived obj;

/* Error because compiler can't figure out which function to call either
     same_function( ) of base1 or base2 .*/
    obj.same_function( )
}
```

Problems with Multiple Inheritance: Ambiguity

Easy solution to Ambiguity:

Using scope resolution function to specify which function to class either `base1` or `base2`.

```
int main()
{
    obj.base1::same_function();  /* Function of class base1
        is called. */
    obj.base2::same_function();  /* Function of class base2
        is called. */
}
```

However .....

Problems with Multiple Inheritance: Ambiguity

However, while this workaround is pretty simple:

- things can get complex when your class inherits from **several base classes**
  - which may inherit from other classes themselves!
- The **potential** for naming conflicts increases **exponentially** as you **inherit** more classes
  - each of these naming conflicts needs to be resolved explicitly!

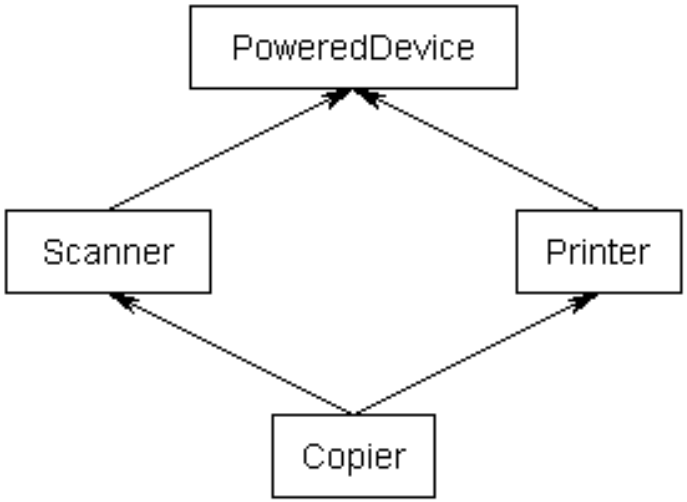
Problems with Multiple Inheritance: Diamond of Doom

Diamond of Doom

When a class multiply inherits from two classes which each inherit from a single base class.

- This leads to a diamond shaped inheritance pattern
- A bigger problem than Ambiguity.

For example:



Problems with Multiple Inheritance: Diamond of Doom

Diamond of Doom Code Fragment

```
class PoweredDevice
{
};

class Scanner: public PoweredDevice
{
};

class Printer: public PoweredDevice
{
};

class Copier: public Scanner, public Printer
{
};
```

- `Scanners` and `Printers` are both powered devices, so they derived from `PoweredDevice`.
- However, a **copy** machine incorporates the functionality of **both** `Scanners` and `Printers`.

Is multiple inheritance more trouble than it's worth?

Things to consider:

- Consider composition of features, instead of inheritance
- Be wary of the Diamond of Dread
- Consider inheritance of multiple **interfaces** instead of objects
- Sometimes, Multiple Inheritance is the right thing
  - The C++ way — you have the power, use it wisely!

However, multiple inheritance should be used extremely judiciously.

- If you cannot justify using multiple-inherited architectures then they may be a better way!

Aside: you have already been using classes written using multiple inherited without knowing it: the `iostream` library objects `cin` and `cout` are **both** implemented using **multiple inheritance**.

Multiple Inheritance in other languages

It turns out, most of the problems that can be solved using multiple inheritance can be solved using single inheritance:

- Many object-oriented languages do not even support multiple inheritance — *e.g.* **Smalltalk**, **PHP**.
- Many relatively modern languages restricts classes to single inheritance of normal classes, but allow multiple inheritance of interface classes — **Java**, **C#**.

The driving idea behind disallowing multiple inheritance in these languages is that it simply makes the language too complex, and ultimately causes more problems than it fixes!

Summary & On to the Lab Class:

After this lecture & the following lab, you should:

- Understand the purpose of constructors & destructors
- Write simple classes that use constructors & destructors for initialisation and cleanup
- Be able to use inheritance in C++ to define new classes
- Be able to override functions in subclasses
- Understand the purpose of abstract classes and pure virtual functions