Programming in C++

Session 7 – Multiple Inheritance

Dr Christos Kloukinas

City St George's, UoL
https://staff.city.ac.uk/c.kloukinas/cpp
(based on slides originally produced by Dr Ross Paterson)

Major Differences between Java and C++

These are the main pain points to understand C++ **[*]**
(why did Java "simplify" them?)

- call-by-reference (session 1 and since)
- operator overloading (session 3)
- genericity or template classes (sessions 4–6)
- memory management
  - local allocation of objects (sessions 1–2 and since)
  - pointers (sessions 5–6)
  - dynamic allocation & de-allocation (sessions 8–9)
- multiple inheritance (this session)

[*] *(and to answer in job interviews)*

Multiple Inheritance

- In many object-oriented languages, including C++ and Eiffel, a class may derive from more than one base class.
- Java supports a common special case: a class may extend only one class, but may implement any number of interfaces.
- Multiple inheritance is very useful, but raises some questions:
  - What if both happen to ***define*** the same names?
  - What if both ***derive*** from a common class?

Both these are ***implementation*** (code reuse) problems, nothing to do with the type hierarchies.

That's why interfaces (i.e., abstract classes) don't have problems...

The simple case

- A common, simple, use of multiple inheritance to combine two essentially unrelated classes:

```
class read_write : public reader, public writer {
  ...
};
```

An IS-A relation.

- We can also combine classes using sub-objects:

```
class chess_game : public window {
  protected:
    board board;
  ...
};
```

An IS-A relation.

A HAS-A relation.

Key question: should the new class be usable by clients of the old? That is, do we need an IS-A relation (yes) or a HAS-A relation (no) ? This question is about the type relation – nothing to do with code reuse.

An asymmetrical case

Often a class extends a concrete base class and an abstract one, using the concrete class to implement the undefined methods from the second class, and possibly a bit more:

```
class active_grid : public grid,
                    public button_listener {
public:
        void mouse_pressed(button_event & e) {
                // use grid stuff
        }
};
```

Java supports only this special case.

Name clashes (ambiguity)

What if two base classes define the same name?

```
class A { public: int f(); };

class B { public: int f(); };

class AB : public A, public B {
public:
        int g() {
                return f() + 1; // which one?
        }
};
```


Possible solutions

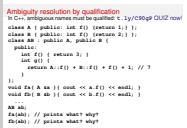
- The language chooses one, using some rule (some LISP dialects).
- The language permits the programmer to rename methods of a base class in a derived class, thus avoiding the clash (Eiffel).
- The programmer must explicitly qualify the names with the class from which they come (C++).

Renaming the methods in the original classes is often not an option, as they may be part of a library or fixed interface.

Ambiguity resolution by qualification

In C++, ambiguous names must be qualified: `t.ly/C90gP` QUIZ now!

```
class A { public: int f() {return 1;} };
class B { public: int f() {return 2;} };
class AB : public A, public B {
  public:
    int f() { return 3; }
    int g() {
      return A::f() + B::f() + f() + 1; // 7
    }
};
void fa( A &a ){ cout << a.f() << endl; }
void fb( B &b ){ cout << b.f() << endl; }
  ...
AB ab;
fa(ab); // prints what? why?
fb(ab); // prints what? why?
```



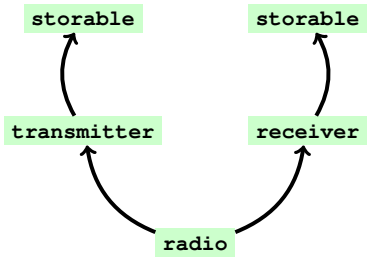
- Will print 1 & 2 respectively, because **`f`** is **NOT** virtual!
- So there's no dynamic binding – compiler chooses the appropriate **`f`** statically (at compilation time), by considering the interface of the object.
- *If* **`A`**'s **`f()`** was **`virtual`**, then **`fa()`** would print **`3`** if its argument was of class **`AB`**...
- What if **`f`** was **`virtual`** *only* inside class **`A`**?

Replicated base classes

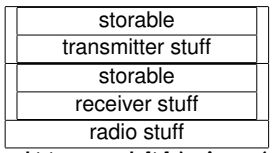
```
class storable { int width; ... };//I HAVE *width*!

class transmitter : public storable { ... };

class receiver : public storable { ... };

class radio : public transmitter,
              public receiver { ... };
```

- A **`radio`** object will contain *two* distinct **`storable`** components, and thus two versions of each member.
- All references to **`storable`** members in **`radio`** must be qualified with either **`transmitter`** or **`receiver`**.

Replicated base classes, graphically



Memory view:




```
&(rad1.transmitter::width) != &(rad1.receiver::width)
```


Virtual functions in the base class

```
class storable {
public:
        virtual void write() = 0;
};

class transmitter : public storable {
public:
        virtual void write() { ... }
};

class receiver : public storable {
public:
        virtual void write() { ... }
};
```

Overriding virtual methods

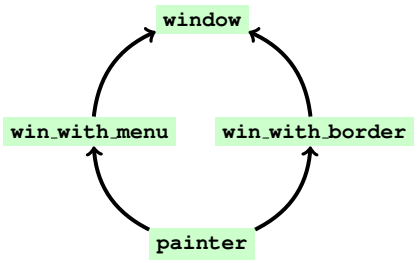
A virtual function in the replicated base class can be overridden:

```
class radio : public transmitter,
              public receiver {
public:
        virtual void write() {
                transmitter::write();
                receiver::write();
                // write extra radio stuff
        }
};
```

The use of the base class versions, plus a bit more, is common.

Virtual inheritance (sharing)

Suppose we want:



Virtual base class

If we write

```
class window { ... };

class win_with_border : public virtual window {...};
class win_with_menu   : public virtual window {...};

class painter : public win_with_border,
                public win_with_menu { ... };
```

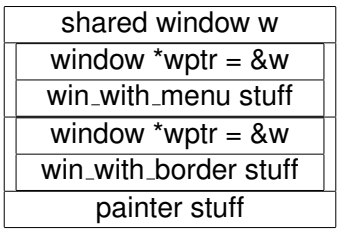
Then a `painter` object includes *a single* `window`.
Class `window` is a ***virtual base class*** of class `painter`.

- Virtual method – you have a ***pointer*** to the method.
- Pure virtual method (`= 0`) means a `nullptr` pointer (no code)

⇒ ***Virtual base class – you have a pointer to it!***

(just like "virtual memory" in OSs uses indirection to real memory)

Virtual inheritance – Memory view



Constructors

```
class window {
public:
        window(int i) { ... }
};
class win_with_border : public virtual window {
public:
        win_with_border() : window(1) { ... }
};
class win_with_menu : public virtual window {
public:
        win_with_menu() : window(2) { ... }
};
```

PROBLEM: The base classes of `painter` want to initialise the common `window` object in a different way – they don't know it's shared!
SOLUTION: Ignore them – class `painter` is the one best placed to decide how the common `window` object should be initialised.

Constructors for a virtual base class

The class in the hierarchy that *knows* that a common virtual base class is shared decides how to construct it (intermediate classes don't know if the virtual base class is shared or not).

```
class painter : public win_with_border,
                public win_with_menu {
public:
        painter(int i) : window(i),
                         win_with_border(),
                         win_with_menu() { ... }
        ...
};
```

- Avoids conflicts between intermediate class constructors
- Language ensures each constructor is called exactly once

Ensuring other methods are called only once

When the virtual base class has a method redefined by each class?

```
class window {
public:
        virtual void draw() {
                // draw window
        }
};
```


Drawing, first attempt

```
class win_with_border : public virtual window {
public:
        virtual void draw() {
                window::draw();
                // draw border
        }
};

class win_with_menu : public virtual window {
public:
        virtual void draw() {
                window::draw();
                // draw menu
        }
};
```

Disaster!!!

But then if we write:

```
class painter : public win_with_border,
                public win_with_menu {
        void draw() {
                win_with_border::draw();
                win_with_menu::draw();
                // draw painter stuff
        }
};
```

The **window** gets drawn twice!

Solution: auxiliary methods

We put the drawing of the extra stuff in a method of its own:

```
class win_with_border : public virtual window {
protected:
        void own_draw() { ... }
public:
        virtual void draw() {
                window::draw();
                own_draw();
        }
};
```

And similarly for **win_with_menu**.

Calling each method once

```
class painter : public win_with_border,
                public win_with_menu {
protected:
        void own_draw();
public:
        void draw() {
                window::draw();
                win_with_border::own_draw();
                win_with_menu::own_draw();
                own_draw();
        }
};
```

Then each part is drawn exactly once.

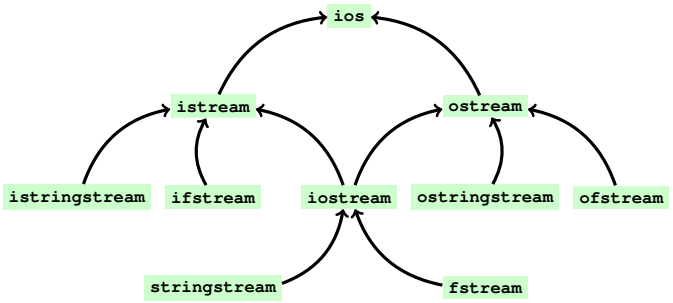
Virtual inheritance: Summary

- *Good news:* Virtual inheritance is a rare case.
- *Even better:* Language ensures constructors called exactly once.
- *Bad: Code Re-use Kills*
  If a method is defined in the virtual base class and overridden in more than one derived class (a rare case), considerable care is required to ensure that each method is called exactly once.
- If a method is pure virtual in the virtual base class, the issue does not arise (because the derived versions cannot call it).
- If the method is overridden in only one branch, the issue does not arise (because only that version need be called).

I/O stream classes

```
class ios {
    // private state
public:
    bool good() const { ... }
    bool eof() const { ... }
    bool fail() const { ... }
    bool bad() const { ... }
};

class istream : virtual public ios { ... };

class ostream : virtual public ios { ... };

class iostream : public istream, public ostream {};
```


Stream class hierarchy



The state of **ios** is *not* duplicated.

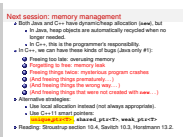
Next session: memory management

- Both Java and C++ have dynamic/heap allocation (**new**), but
  - In Java, heap objects are automatically recycled when no longer needed.
  - In C++, this is the programmer's responsibility.
- In C++, we can have these kinds of bugs (Java only #1):
  1. Freeing too late: overusing memory
  2. Forgetting to free: memory leak
  3. Freeing things twice: mysterious program crashes
  4. (And freeing things prematurely...)
  5. (And freeing things the wrong way...)
  6. (And freeing things that were not created with **new**...)
- Alternative strategies:
  - Use local allocation instead (not always appropriate).
  - Use C++11 smart pointers:
    unique_ptr<T>, **shared_ptr<T>**, **weak_ptr<T>**
- Reading: Stroustrup section 10.4, Savitch 10.3, Horstmann 13.2.

Final Notes – I:

- Multiple inheritance is a major difference between Java and C++.
  - Java doesn't allow it – inheriting fields and code from multiple classes is problematic:
    - What if multiple parent classes define the same fields or functions?
    - What if multiple parent classes inherit from a common class themselves?

    Both these problems are caused by code reuse, not by introducing a type hierarchy.
  - That's why in Java you can inherit from only one class and ... multiple interfaces (that don't have any code).
  - C++ allows multiple inheritance – it gives you all the tools you need to solve the issues (enough rope to hang yourself...).
- Sometimes you can avoid inheritance altogether – the key question to ask is:
  Should class A be usable at all settings where class B is usable?
  If so, then A should inherit from B (A Is-A B). Otherwise A can simply contain a B (A Has-A B).
- Name ambiguity is resolved by qualification:
  ClassName::MemberName()



Final Notes – II:

- Two types of multiple inheritance:
  - Replicated inheritance:

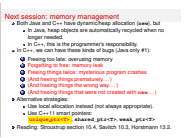
```
#include <cassert>
// struct's a class with everything public.
struct A {int x;};
class B: public A {};
class C: public A {};
class D: public B, public C {};
int main() {
  D d1;
  d1.B::x = 1; // assign d1's x from the B side
  d1.C::x = 2; // assign d1's x from the C side
  assert( &(d1.B::x) != &(d1.C::x) );
  assert( d1.B::x == 1 );
  assert( d1.C::x == 2 );
  // restricted view of d1 - B interface (B & ...)
  B & b_view_of_d1 = d1;
  // restricted view of d1 - C interface (C & ...)
  C & c_view_of_d1 = d1;
  assert( &(b_view_of_d1.x) != &(c_view_of_d1.x) );
  assert( b_view_of_d1.x == 1 );
  assert( c_view_of_d1.x == 2 );
        /* ALWAYS */
    assert( &(d1.B::x) == &(b_view_of_d1.x) );
    assert( &(d1.C::x) == &(c_view_of_d1.x) );
  return c_view_of_d1.x - b_view_of_d1.x; // 1
}
```

    D contains two copies of A – one from the B side and one from the C side (like persons having two grandfathers – one from their mother's side and one from their father's side).
  - Virtual inheritance:

```
#include <cassert>
struct A {int x;};
class B: virtual public A {}; // virtual public =
class C: public virtual A {}; // public virtual
class D: public B, public C {};
int main() {
  D d1;
  d1.B::x = 1;
    assert( &(d1.B::x) == &(d1.C::x) && d1.B::x == 1 );
  d1.C::x = 2;
    assert( &(d1.B::x) == &(d1.C::x) && d1.B::x == 2 );
  B &b_view_of_d1 = d1;
  C &c_view_of_d1 = d1;
  assert( &(b_view_of_d1.x) == &(c_view_of_d1.x) );
  assert( b_view_of_d1.x == 2 );
        /* ALWAYS */
    assert( &(d1.B::x) == &(b_view_of_d1.x) );
    assert( &(d1.C::x) == &(c_view_of_d1.x) );
  return c_view_of_d1.x - b_view_of_d1.x; // 0
}
```

    D contains only one copy of A – the B and C side have virtual A's.
    - Compiler ensures constructors work as expected (only called once).
    - You need auxiliary methods to get this version of inheritance work for other methods.

2024-11-22
Programming in C++
└Next session: memory management



Checking if two objects are the same

As we saw in slide 10, to check if two objects are the same (and not just equal to each other) we need to check their addresses in memory Here's a simple example of that using int:

```
#include <cassert>

int main() {

  int i = 3, j = 3;

  assert( i == j ); // same values

  assert( &i != &j ); // BUT different objects
                      // (cause different addresses!)

  return 0;
}
```

Comparing addresses is closer to what the `===` operator does in languages like JavaScript, which is (mainly) used to check if two objects are the same (developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_equality). In Java, the `==` operator when applied to Object references compares the references directly (i.e., the pointers), so can be used to figure out if two objects are the same. In Java we need to call **`equals()`** to compare object contents instead.

```
public class SameObj {
  public int i = 1;
  public boolean equals(SameObj other) {
    return i == other.i;
  }

  public static void main(String[] args) {
    SameObj o1 = new SameObj(), o2 = new SameObj();
        // Comparing VALUES: .equals()
    if (o1.equals(o2))
      System.out.println("o1 & o2 are equal");
    else
      System.out.println("o1 & o2 are *not* equal");
        // Comparing POINTERS: ==
    if (o1 == o2)
     System.out.println("o1 & o2 are the same");
    else
     System.out.println("o1 & o2 are *not* the same"); /**/
    if (o1 == o1) // should always be true
     System.out.println("o1 & o1 are the same"); /**/
    else
     System.out.println("o1 & o1 are *not* the same");
  }
}
```