Programming in C++

Session 6 – Inheritance in C++

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)

The most important slide of the lecture

Why use inheritance?

Reasons for Inheritance (revision)

- **Implementation Re-Use** ***Bad-ish...*** [+]
  - new classes *extend* existing classes with additional fields and methods, and can *override* the definitions of existing methods.
- **Interface/Type Hierarchies (Is-A relations [*])** ***Good!***
  - the new class is also a *subtype* of the old: its objects can be used wherever objects of the old class can (*subtype polymorphism*) with the appropriate method selected by *dynamic binding*.
  - *abstract classes* declare methods without defining them: the methods are defined in subclasses.

[*] **Is-A** vs **Has-A** relations:

- A car **Is-A** vehicle — Inheritance
- A car **Has-A** steering wheel — Composition

[+] Just have an object of that class as a field (composition) and have your methods forward calls to it.

Inheritance in C++

The basic concept is similar to Java, but

- different syntax
- **objects** of subclasses may be assigned to **object** variables of superclasses, by ***slicing*** off the extra parts.
- interactions with:
  - overloading
  - pointers
  - template classes

Inheritance syntax in Java and C++

- in Java:
  ```
  public class holiday extends date {
  ```
- in C++:
  ```
  class holiday : public date {
  ```
  we will always use **public** inheritance.
- C++ terminology: **date** is a *base class*; **holiday** is a *derived class*.
- multiple inheritance (in C++):
  ```
  class child : public parent1, public parent2 {
  ```
- there are no interfaces in C++.

A base class

Recall the class **date** from session 2:

```
class date {
        int day, month, year;

public:
        date();          // today's date
        date(int d, int m);
        date(int d, int m, int y);
        int get_day() const { return day; }
        int get_month() const { return month; }
        int get_year() const { return year; }
};
```


Inheritance and initialization

The members of base class(es) are initialized similarly to subobjects:

```
class holiday : public date {
        string name;
public:
        holiday(string n) : date(), name(n) {}

        holiday(string n, int d, int m) :
                date(d, m), name(n) {}

        string get_name() const { return name; }
};
```

Members of the base class can't be initialized directly:

use constructor `date`

(can only use it in the initialisation list!)

Order of initialization – IMPORTANT!

Initialization is done in the following order:

1. constructors for base classes
2. members (in order of declaration) – WARNING!!! BE CAREFULL!!!
3. body of constructor

Principle: ***The constructor body needs a fully initialised object!***

Danger: ***Order of initializers has *NO* effect, only declaration order matters!***

So: Initialise members with order-independent expressions

Initialization and assignment

- As in Java, we can initialize and assign from derived classes, but here **objects** are copied, not pointers:

```
holiday h("Anzac Day", 25, 4);
date d = h;
```

initializes `d` as a copy of the `date` part of `h`

```
d = h;
```

copies the `date` part of `h` into `d`

- In both cases, the object is ***sliced***

Note: *Call-by-value initialises a new variable, so it also involves copying (and slicing)*

Method overriding in Java and C++

The default in C++ is the ***opposite*** to that in Java:

- in Java:

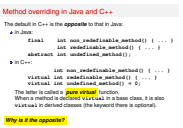
```
final    int non_redefinable_method() { ... }
         int redefinable_method() { ... }
abstract int undefined_method();
```

- in C++:

```
         int non_redefinable_method() { ... }
virtual  int redefinable_method() { ... }
virtual  int undefined_method() = 0;
```

The latter is called a ***pure virtual*** function.
When a method is declared `virtual` in a base class, it is also `virtual` in derived classes (the keyword there is optional).

Why is it the opposite?



- It's the opposite because non-redefinable member functions are faster than redefinable (virtual) ones. (C++'s #1 aim is speed!)
- Redefinable member functions are actually *pointers* to functions – at run time the code has to dereference the pointer held in the class information of the current object to figure out which code to execute.
- This also explains the bizarre syntax for abstract (pure virtual) member functions:
" `= 0`" means that the function pointer is the `nullptr`, *i.e.*, there's no respective code for it!

Method overriding

Overridable methods must be declared `virtual`:

```
class date {
        ...
        virtual string desc() const { ... }
};
```

Overriding in a derived class:

```
class holiday : public date {
        ...
        virtual string desc() const {
                return name + " " + date::desc();
        }
};
```

Note: ***Qualify*** with the class name to get the base version.

Static and dynamic binding

Given functions

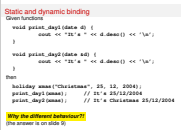
```
void print_day1(date d) {
        cout << "It's " << d.desc() << '\n';
}

void print_day2(date &d) {
        cout << "It's " << d.desc() << '\n';
}
```

then

```
holiday xmas("Christmas", 25, 12, 2004);
print_day1(xmas);      // It's 25/12/2004
print_day2(xmas);      // It's Christmas 25/12/2004
```

Why the different behaviour?!

(the answer is on slide 9)



Dynamic Binding

In order to get dynamic binding we need:

1. a type hierarchy (inheritance)
2. some virtual member functions
3. references or pointers to objects
   (so that the compiler isn't sure what the real object type is)

If you don't need Dynamic Binding, then you don't need Inheritance!
You can simply use composition and (implicit) conversion:

```
class holiday {
  date d;
 ...
 operator date() const { return d; } // convert to date
};
void print_month(date d) { // works with holiday objects to
    cout << d.get_month() << endl;
}
```


Abstract classes

A class containing a pure virtual function is *abstract*, though this is not marked in the syntax.

```
class pet {
protected:
    string _name;
public:
    pet(string name) : _name(name) {}
    virtual string sound() const = 0;
    virtual void speak() const {
        cout << _name << ": " << sound() << "!\n";
    }
};
```

As in Java, abstract classes may not be instantiated, so no variable may have type `pet`, but we can declare a reference (or a pointer).

Derived classes

```
class dog : public pet {
public:
   dog(string name) : pet(name) {}
   string sound() const { return "woof"; }
   void speak() const {      // virtual is optional
       pet::speak();
       cout << '(' << _name << " wags tail)\n";
   }
};

class cat : public pet {
public:
   cat(string name) : pet(name) {}
   virtual string sound() const { return "miao"; }
};
```

Subtype polymorphism and dynamic binding

We cannot pass `pet`s by value, but we can pass them by reference:

```
void speakTwice(const pet &a_pet) {
    a_pet.speak();
    a_pet.speak();
}
```

Then we can write

```
dog a_dog("Fido");
speakTwice(a_dog);
cat a_cat("Tiddles");
speakTwice(a_cat);
```

Why can't we pass `a_pet` by value to `speakTwice` ?

Subtype polymorphism and dynamic binding
We cannot pass pets by value, but we can pass them by reference:
void speakTwice(const pet &a_pet) {
a_pet.speak();
a_pet.speak();
}
Then we can write
dog a_dog("Fido");
speakTwice(a_dog);
cat a_cat("Tiddles");
speakTwice(a_cat);
Why can't we pass a_pet by value to speakTwice ?

Because

- call-by-value involves ***instantiating*** a new local object, which is initialised using the original parameter (see slide 9); and
- `a_pet` is an abstract class, so we cannot instantiate it. . .

Caution: inheritance and overloading

```
class A {
    virtual void f(int n, Point p) { ... }
};
```

Now suppose we intend to override `f` in a derived class, but make a mistake with the argument types:

```
class B : public A {
    void f(Point p, int n) { ... }
};
```

`f` will be accepted as a definition of a new and different member function.

Even forgetting a single `const` or changing a `*` to a `&` means it's a different function!

```
class B : public A {
    void f(Point p, int n) override { ... }
};
```


Caution: inheritance and overloading
class A {
virtual void f(int n, Point p) { ... }
};
Now suppose we intend to override f in a derived class, but make a mistake with the argument types:
class B : public A {
void f(Point p, int n) { ... }
};
f will be accepted as a definition of a new and different member function.
Even forgetting a single const or changing a * to a & means it's a different function!
class B : public A {
void f(Point p, int n) override { ... }
};

How can you protect yourself against such mistakes?

Since C++11 there's a new keyword `override` that you can use to state that you're trying to override a member function of one of your base classes:

```
class B : public A {
  void f(Point p, int n) override { ... }
  // Now the compiler catches the error
};
```

There's also a keyword `final` to state that derived classes should not be allow to further override the member function:

```
class A {
  virtual void f(int n, Point p) { ... }
  virtual int  g(Point p) const  { ... }
};
class B : public A {
  void f(int n, Point p) override { ... }
  int  g(Point p) const final { ... }
};
```

Which version is selected?

If more than one ***overloaded*** function or method matches, the best (most specific) is chosen:

```
class pet {};
class cat : public pet {};

void wash(pet &x) { ... }
void wash(cat &x) { ... }

int main() {
  cat felix;
  wash(felix);// both functions match; second is used
}
```

Overload: ***STATIC*** (*i.e.*, compile-time) decision

Override: ***DYNAMIC*** (*i.e.*, run-time) decision

Over$^{\text{Loading}}_{\text{Riding}}$ – Write fewer `if`'s with OOP!

Overloading – STATIC/COMPILATION TIME:

```
void f( pet & x ) {
if (x isA cat) {}
else if (x isA dog) {}
else if (x isA hamster) {}
else {assert(0);}//*ERROR*
```

```
void f(cat     &x) {...}
void f(dog     &x) {...}
void f(hamster &x) {...}
//*NO* (runtime) *ERROR*!!!
```

Overriding – DYNAMIC/RUN TIME: *(only need inheritance for this)*

```
void move(person &p) {
if (p isA driver) {}
else if (p isA cyclist) {}
else if (p isA pilot) {}
else { //*DEFAULT* } }
```

```
class person {//*DEFAULT*
 virtual void move(){...} }
class driver :person{
  void move(){...} }
class cyclist :person{
  void move(){...} }
class pilot  :person{
  void move(){...} }
```

Write better if/then/else's – let the compiler do it!

Pointers and subtyping

Pointers to derived classes are subtypes of pointers to base classes (*i.e.*, if I can point to a base class, I can also point to a derived class):

```
cat felix;
pet *p = &felix;
```

No slicing occurs here, because pointers are copied not objects (a memory address is the same size as another memory address):

```
p->speak();          // miao
```

The `speak` method uses the virtual method `sound`, which is defined in the `cat` class, and selected by dynamic binding (see slides 6–13).

Containers of pointers

Often a container holds pointers to a base type:

```
vector<pet *> pets;
cat felix("Felix");
dog fido("Fido");
pets.push_back(&felix);
pets.push_back(&fido);
```

When we access elements of the vector, dynamic binding is used:

```
for (std::size_t i = 0; i < pets.size(); ++i)
    pets[i]->speak(); // miao, woof
```

Introducing dynamic allocation

- Typically the number of things in the collection is unpredictable
- So allocate objects dynamically (as in Java) on the heap:

```
cat *cp = new cat("tiddles");
pets.push_back(cp);
```

  Here the pointer `cp` is local, but the object it points at is on the heap (so it outlasts the current block)
- Major difference: in C++ the programmer is responsible for deallocation, but we'll ignore that till session 8
- Better (C++11):

```
#include <memory>
vector<shared_ptr<pet>> pets;
// shared_ptr<cat> cp = make_shared<cat>("Tom");//Old
auto cp = make_shared<cat>("Tom");//New, simpler!!!
pets.push_back(cp);
```


Templates and subtyping (I)

When `cat` is a subtype of `pet`,

- `cat *` ***IS*** a subtype of `pet *`, but
- `vector<cat *>` ***IS NOT*** a subtype of `vector<pet *>`!

Why not? Consider this code fragment:

```
vector<cat *> cats;
vector<pet *> *p = &cats;  // illegal
dog fido;
p->push_back(&fido);       // would be trouble
```

See Stroustrup 13.6.3(3rd ed.)/27.2.1(4th ed.) for more.

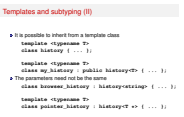
Templates and subtyping (II)

- It is possible to inherit from a template class

```
template <typename T>
class history { ... };

template <typename T>
class my_history : public history<T> { ... };
```

- The parameters need not be the same

```
class browser_history : history<string> { ... };

template <typename T>
class pointer_history : history<T *> { ... };
```

Why not

```
template <typename T>
class browser_history : history<string> { ... };
```

???
Because `borwser_history` is NOT a template class, it simply inherits from a (specialised) template class.

Next session: 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 the question of what to do when the base classes conflict.
- Reading: Stroustrup 15.2

(area left empty on purpose)

Final Notes – I

- Inheritance is used for:
  1. Code re-use (**bad**, **bad**, **bad**! That's why Java allows us to inherit from at most one class)
  2. Defining type-hierarchies through the IsA relation between types: car Is-A vehicle, cat Is-A pet (**good**, **good**, **good**! That's why Java allows us to inherit from as many interfaces as we want)
- Inheritance is required if we need ***dynamic binding***, *i.e.*, code that behaves differently at run-time depending on the real type of the objects involved.
  - For dynamic binding we also need to use references or pointers (they keep the real type of the objects and don't cause slicing to happen).
  - And of course we need some member functions to be virtual, otherwise the compiler will plug-in direct calls to the superclass member functions (static binding) instead of checking the object's real type and using dynamic binding.
- Slicing: If we try to assign an object of a derived class (like holiday) into an object of a base class (like date), then there's not enough room for all the information, so we need to slice the object of the derived class - we throw away its new members and keep just the members of the base class.
- We can initialize the base class part of a derived object by calling the constructor of the base class in the initialization list of the derived object's constructor (only there can we call it):

```
holiday(string n, int d, int m) : date(d, m), name(n) {}
```

- Initialization order:
  1. Constructors of base classes
  2. Constructors of members
  3. Body of constructor of the derived class

Principle: ***The constructor body needs a fully initialised object!***
The destruction follows the ***opposite*** order (destructor body, destructors of members, destructors of base classes).

Principle: ***The destructor body needs a fully initialised object!*** *(same principle)*

- Overriding behaviour: The base class must have declared the member function as virtual for us to be able to override it in the derived class:

```
virtual string desc() const {...}
```

- Pure virtual member functions (aka abstract methods):

```
virtual string sound() const = 0; // no code!
```

  - Virtual functions are essentially pointers (to functions).
  - Pure virtual (abstract) functions are null (`nullptr`) pointers (no code to point to). That should explain the bizarre syntax (`= 0`).
  - A class with at least one pure virtual member function is an abstract class - cannot instantiate it (but we can have references and pointers to it – for dynamic binding, see below).
  - A class with no members (fields) and all of its member functions pure virtual is equivalent to a Java interface.
  - If your class has a virtual function then it probably needs a virtual destructor.

2024-11-13
Programming in C++
└Inheritance
└Final Notes

Final Notes – II

- Static vs Dynamic binding - check out slide 12.
  - Function `print_day1` uses call-by-value (so the real object passed is copied and sliced in order to initialize the local parameter and the function always operates on a date object).
  - Function `print_day2` uses call-by-reference (so the real object is passed without copying/slicing, initializing the local reference parameter to refer to it whatever it may be, and the function operates on any kind of date object).
  - To get dynamic binding, *i.e.*, different behaviour at runtime depending on the real type of an object, one needs two things:
    - To have virtual member functions, which have been overriden in derived classes (the implementation of the different behaviour according to the type of the object)
    - To allow these virtual member functions to be selected dynamically at runtime, by passing objects either by reference or by pointer. Otherwise (*i.e.*, in pass-by-value) static binding is used.
- Java has `super(...);` to call the same method in the parent class. A C++ class may have multiple parent (base) classes, so to call one of their member functions that we've overridden, we must name the base class explicitly:

```
class dog : public pet {
  void speak() const override
 /* "override" - C++11 keyword to show that we want
    to override some base class' speak */ {
    pet::speak(); // call pet's speak
    cout << '(' << _name << " wags tail)\n";
  }
};
```

- Containers of pointers:
  - Want to have a collection of objects but your class doesn't have a default constructor?
  - Want to avoid copying objects around?
  - Want to store different sub-types of some base class and get dynamic binding when you use them (and avoid slicing them)?

  Then use a container of pointers – slide 20.
  - Beware that `vector< cat * >` isn't a sub-type of `vector< pet * >`, even though `cat *` is a sub-type of `pet *` when `cat` is a sub-type of `pet` (slides 22–23).
- Inheritance and templates: slides 22–23. Partial specialization (`PointerHistory` partially specializes the type of `History` to be a pointer to some still unknown type `T`).
  - More on template specialization (and partial specialization) www.cprogramming.com/tutorial/template_specialization.html
  - Did you notice in the template specialization article that a template parameter does not have to be a typename?
    Welcome to Template Meta-Programming
    www.codeproject.com/Articles/3743/A-gentle-introduction-to-Template-Metaprogramming
    No need to thank me.
    (DEFINETELY *NOT* IN THE SCOPE OF THE MODULE/EXAM!)

  And some interesting further reading that may help you better understand how virtual member functions work (and don't work sometimes) – not part of the exam but **highly** helpful:
  - Vee Table https://wiki.c2.com/?VeeTable
  - Fragile Binary Interface Problem https://c2.com/cgi/wiki?FragileBinaryInterfaceProblem