Inheritance and Dynamic Binding

Laurent Mathy

Object-Oriented Programming Projects

April 17, 2020

Extending our student problem

- This time, we will assume students can take the course for both undergraduate and graduate credits.
- The grad students must do extra work: they write a thesis (report).
- We'd like our previous solution to the grading problem to continue to work.

Inheritance

- The record for graduate credit is the same as for undergraduate credit, except for additional properties related to the thesis.
- When you can think of a class as being like another class but for some extensions, you have a natural place for **inheritance**.
- In fact, inheritance can also be used for expressing exceptions (see your OO design classes).
- We will design 2 classes: one to represent undergrads (`Core`), and one to represent grads (called `Grad`).
- We'll use `Student_info` to represent any kind of students.

Core class

Core is similar to `Student_info` from previous lecture.

We add a `private` utility function to read the portion of student record that all students have in common.

```cpp
class Core {
public:
    Core();
    Core(std::istream& is);
    std::string name() const;
    std::istream& read(std::istream&);
    double grade() const;
private:
    std::istream& read_common(std::istream&);
    std::string n;
    double midterm, final;
    std::vector<double> homework;
};
```

Grad class

```
class Grad: public Core {
public:
    Grad();
    Grad(std::istream&);
    double grade() const;
    std::istream& read(std::istream&);
private:
    double thesis;
};
```

- Grad **inherits** from Core, or Core is a **base class** of Grad.
- `public` in `public Core` means that the Core public interface is part of the Grad public interface.
  - `public` members of Core are also `public` members of Grad.
  - *e.g.* can call `name()` member function on a Grad.
- Grad objects will have 5 data members:
  - 4 inherited from Core + `thesis`.
- Grad will have 2 constructors and 4 member functions:
  - inherited `name()` and `read_common()`;
  - **overridden** `grade()` and `read()`.

Protection revisited

- Right now, all four data members and `read_common` in `Core` are inaccessible to member functions of `Grad`:
  - `private` members of a class are only accessible **to the class itself**, and its `friend`s.
- Use `protected` label to grant access to derived classes:
  - `protected` members are still accessible by the class itself and its friends, but also by **derived classes**.

```cpp
class Core {
public:
    Core();
    Core(std::istream& is);
    std::string name() const;
    std::istream& read(std::istream&);
    double grade() const;
protected: // Accessible to derived classes
    std::istream& read_common(std::istream&);
    std::string _name;
    double midterm, final;
    std::vector<double> homeworks;
};
```

Operations: Core

```
string Core::name() const { return _name; }

double Core::grade() const {
    return ::grade(midterm, final, homeworks);
}

istream& Core::read_common(istream& in) {
    // Read and store student's name and exam grades
    in >> _name >> midterm >> final;
    return in;
}

istream& Core::read(istream& in) {
    read_common(in);
    read_hws(in, homeworks);
    return in;
}
```

Operations: Grad

```
istream& Grad::read(istream& in) {
    read_common(in);
    in >> thesis;
    read_hws(in, homeworks);
    return in;
}

double Grad::grade() const {
    return std::min(Core::grade(), thesis);
}
```

- You **must** write `Core::grade()`, otherwise you get a recursive call to `Grad::grade()`.
- You *could* write `Core::read_common` and `Core::homework`, although these *are* members of `Grad`.

Inheritance and Constructors

Construction of derived objects:

1. Allocation of space for entire object.
2. Calling base-class constructor to initialise base-class part.
3. Initialisation of members of derived-class (via constructor initialisers).
4. Execution of derived-class constructor body, if any.

Use the constructor initialiser to specify the base-class constructor you want.

Initialiser names its base-class followed by a (possibly empty) list of arguments.

If no base-class constructor specified, then the default base-class constructor is run.

Constructors

```cpp
class Core {
public:
    // Default constructor for Core
    Core(): midterm(0), final(0) { }
    Core(std::istream& is) { read(is); }
    // ...
};

class Grad: public Core {
public:
    // Both constructors implicitly use Core::Core()
    Grad(): thesis(0) { }
    Grad(std::istream& is) { read(is); }
    //...
};
```

Note that there is no requirement that the derived-class constructor take the same arguments as the base-class constructors.

Polymorphism

We had a non-member `compare` function used by `sort` to sort student records by name:

```
bool compare(const Core& c1, const Core& c2) {
    return c1.name() < c2.name();
}
```

We can use this code to compare two `Core` objects, two `Grad` objects, or even a `Core` and a `Grad`.

```
Grad g1(in);
Grad g2(in);

Core c1(in);
Core c2(in);

compare(g1, g2);
compare(c1, c2);
compare(c1, g1);
```

The reason why this works is because every `Grad` has a `Core` part.

Polymorphism (2)

Because `Grad` **inherits** from `Core`, you can use a `Grad` where a `Core` is expected.

- A reference parameter to a `Core` will refer to the `Core` portion of a `Grad`.
- A pointer parameter to a `Core` will point to the `Core` portion of a `Grad` if a pointer to `Grad` passed instead (`Grad*` converted to `Core*` by compiler).
- Object of type `Core` corresponds to `Core` portion of `Grad` if a `Grad` object is assigned/passed instead.

We say that `Grad` is a **subtype** of `Core`, noted `Grad` $<:$ Core.

Polymorphism(3): a new `compare` function

Suppose that instead of sorting students by name, we want to sort them by final grade:

```
bool compare_grades(const Core& c1, const Core& c2) {
    return c1.grade() < c2.grade();
}
```

Polymorphism(3): a new compare function

Suppose that instead of sorting students by name, we want to sort them by final grade:

```
bool compare_grades(const Core& c1, const Core& c2) {
    return c1.grade() < c2.grade();
}
```

As `Grad` redefines the `grade` function, this `compare_grades` functions sometimes gives the wrong answer, because it **always** invokes the `Core::grade` function, as `c1` and `c2` *are* references to `Core` objects.

We need a way for the `compare_grades` function to invoke the right `grade` function, based on the **actual** type of the object that we pass:

- If `c1` or `c2` are `Grad`s, we want `Grad::grade`.
- If `c1` or `c2` are `Core`s, we want `Core::grade`.
- This must be done at **run-time**, as the **dynamic** types of the passed objects can only be determined at run-time.

Polymorphism (4): `virtual`

To support this:

```
class Core {
public:
    virtual double grade() const; // added `virtual`
    // ...
};
```

- `grade` is now a `virtual` function.
- `virtual` keyword may be used *only* in class definitions: do not repeat it in function definitions.
- `virtual` is inherited, so no need to repeat it in `Grad`, though doing it doesn't hurt.

Note to Java programmers: in Java, all member functions are `virtual` by default. In C++, you must turn this **dynamic binding** on explicitly!

Dynamic Binding

The run-time selection of the `virtual` function is relevant only for references and pointers.

If a function is called on behalf of an object, you get the version of the function corresponding to the object type. In other words, the type of an object is immutable!

```cpp
// Incorrect implementation
bool compare_grades(Core c1, Core c2) {
    return c1.grade() < c2.grade();
}
```

- `c1` and `c2` are always `Core`.
- If you pass a `Grad` to this function, only the `Core` part gets copied.
- Because we said the parameters are `Core` objects, the calls to `grade` are **statically** bound at **compile-time** to `Core::grade`.

Dynamic-binding only applies to references and pointers.

Dynamic Binding (2)

```
Core c;
Grad g;
Core* p = &c;
Core& r = g;

c.grade();  // Static binding to Core::grade()
g.grade();  // Static binding to Grad::grade()
p->grade(); // Dynamic binding to type object p points to
r.grade();  // Dynamic binding to type object r refers to
```

Polymorphic call:
The type of the reference or pointer is fixed, but the type of the object referred or pointed to can be the corresponding type or any type derived from it.

`virtual` and pure `virtual`

- Non-virtual functions can be declared, without being defined, if they are not called.
- `virtual` functions *must* be defined, whether they are called or not.
  - You'll get weird compile errors if not.
- If there is no meaningful implementation for a virtual function, make it a **pure** `virtual` function.

```cpp
class Abstract {
public:
    virtual int pure() const = 0; // Pure virtual
    // ...
};
```

Such (abstract) classes cannot be instantiated: they can only serve as base for derived classes.

A program dealing with only undergrad records

```
vector<Core> students;
Core record;
string::size_type max_len = 0;
// Read and store the student records
while (record.read(cin)) { // Core::read()
    max_len = max(max_len, record.name().size());
    students.push_back(record);
}
// Alphabetize the student records
sort(students.begin(), students.end(), compare);
// Write the names and grades
for (auto& s : students) {
    cout << s.name()
         << string(max_len + 1 - s.name().size(), ' ');
    try {
        cout << s.grade() << endl; // Core::grade
    } catch (domain_error e) {
        cerr << e.what() << endl;
    }
}
```

A program dealing with only grad records

```
vector<Grad> students; // Different type in vector
Grad record; // Different type into which to read
string::size_type max_len = 0;
// Read and store the student records
while (record.read(cin)) { // Grad::read()
    max_len = max(max_len, record.name().size());
    students.push_back(record);
}
// Alphabetize the student records
sort(students.begin(), students.end(), compare);
// Write the names and grades
for (auto& s : students) {
    cout << s.name()
         << string(max_len + 1 - s.name().size(), ' ');
    try {
        cout << s.grade() << endl; // Grad::grade
    } catch (domain_error e) {
        cerr << e.what() << endl;
    }
}
```

Towards a program that deals with both types of records

We need to eliminate the following type dependencies:

- Definition of the `vector`.
- Definition of the local variable.
- Calling the right `read` function.
- Calling the right `grade` function.

As `read` and `grade` are **`virtual`**, last two points have been solved.

The only problem is that our code makes **statically-bound** calls to these functions. We need to turn **dynamic** binding on.

We can use `Core*` where we used `Core` or `Grad`, and let our users allocate memory.

We'll also need a function to compare `Core`s identified by pointers:

```
bool compare_Core_ptrs(const Core* cp1, const Core* cp2) {
    return compare(*cp1, *cp2);
}
```

Dealing with both types of records (2)

```
vector<Core*> students; // Store pointers, not objects
Core* record; // Temporary must be a pointer as well
char ch;
string::size_type max_len = 0;

// Read and store the student records
while (cin >> ch) {
    if (ch == 'U')
        record = new Core; // Allocate a Core object
    else
        record = new Grad; // Allocate a Grad object
    record->read(cin); // Virtual call
    max_len = max(max_len, record->name().size());
                                   // ^ Dereference
    students.push_back(record);
}

// Pass the version of compare() that works on pointers
sort(students.begin(), students.end(), compare_Core_ptrs);
```

Dealing with both types of records (3)

```cpp
// Write the names and grades
for (auto s : students) {
    // s is a pointer
    cout << s->name() // Dereference to call function
         << string(max_len + 1 - s->name().size(), ' ');
    try {
        cout << s->grade() << endl; // Dereference to call

    } catch (domain_error e) {
        cerr << e.what() << endl;
    }
    delete s; // Free the object allocated when reading
}
```

Virtual destructors

Our previous program *almost* works:

- When we `delete` the records, it is always through a `Core*`.
- But these `Core*` can point to either `Core` or `Grad` objects.
- Which destructor to call and how much space to reclaim?
  - Sounds like exactly what the `virtual` mechanism handles.

`virtual` destructor:

```
class Core {
public:
    virtual ~Core() { }
    // ...
};
```

- A `virtual` destructor is needed any time a derived type object can be destroyed through a pointer to base.
- If no other reason to have destructor, then that destructor has no work to do and is empty
- No need to add destructor to `Grad`: virtual property of destructor is inherited and synthesised destructor is fine.

Another Solution

In previous solution, our users had to do memory management for us: this is messy and error prone.

We'll define a **handle** (a.k.a. **proxy**) class, based on a `Core*` that does the memory bookkeeping itself.

Student_info handle class

```
class Student_info {
public:
    // Constructors and copy control
    Student_info() { }
    Student_info(std::istream& is) { read(is); }
    Student_info(const Student_info&);
    Student_info& operator=(const Student_info&);
    // Operations
    std::istream& read(std::istream&);
    std::string name() const {
        if (cp) return cp->name();
        else throw std::runtime_error("uninitialized Student");
    }
    double grade() const {
        if (cp) return cp->grade();
        else throw std::runtime_error("uninitialized Student");
    }
    static bool compare(const Student_info& s1, const Student_info& s2) {
        return s1.name() < s2.name();
    }
private:
    std::unique_ptr<Core> cp;
};
```

Student_info handle class (2)

- As Core has a virtual destructor, the Student_info destructor will work properly, whether it represents a Core or Grad
- Student_info::read will allocate the appropriate space.
- As grade is **virtual**, we will get correct version as called through Core* pointer.
- compare has been made a **static function**, it:
  - is associated with class, not with any particular object;
  - cannot access non-static members;
  - call Student_info::compare.

Reading the handle

```
istream& Student_info::read(istream& is) {
    char ch;
    is >> ch; // Get record type

    // Assignment to `cp` will free if needed
    if (ch == 'U')
        cp = std::make_unique<Core>(is);
    else
        cp = std::make_unique<Grad>(is);

    return is;
}
```

Copying handle objects

We need a copy constructor and assignment operators to manage the `Core*`.

Question: when we copy, are we copying a `Core` or a `Grad`? There is no easy way to know!

We solve this problem by giving `Core` and its derived classes a new **`virtual`** function:

```
class Core {
    friend class Student_info;
protected:
    virtual std::unique_ptr<Clone> clone() const
        { return std::make_unique<Core>(*this); }
    // ...
};
class Grad {
protected:
    virtual std::unique_ptr<Clone> clone() const
        { return std::make_unique<Grad>(*this); }
    // ...
};
```

These **`virtual`** `clone` functions call the synthesised copy constructor for `Core` and `Grad`.

Copying Handle objects (2)

- We do not want `clone` as member of `public` interface, so it is made `protected`.
- Because `clone` is `protected`, we make `Student_info` `class` a `friend` of `Core`.
  - All member functions of `Student_info` are now `friend`s with `Core`.
- When derived class redefines a function from base class, it usually does it **exactly**: parameter list and return type are identical.
  - However, if base-class function returns a pointer or reference to the base class, then the derived class can return a pointer or reference to the derived class.
- `friend`ship is not inherited, but no need to make `Student_info` a `friend` of `Grad`.
  - `Student_info` never refers to `Grad::clone` directly,
  - only through `virtual` calls to `Core::clone`.

Copying and assignement

```
Student_info::Student_info(const Student_info& s) {
    if (s.cp)
        cp = s.cp->clone();
}

Student_info& Student_info::operator=(const Student_info& s) {
    if (&s != this) {
        if (s.cp)
            cp = s.cp->clone();
        else
            cp = nullptr;
    }

    return *this;
}
```

Using the handle class

```
vector<Student_info> students;
Student_info record;
string::size_type max_len = 0;
// Read and store the student records
while (record.read(cin)) {
    max_len = max(max_len, record.name().size());
    students.push_back(record);
}
// Alphabetize the student records
sort(students.begin(), students.end(), Student_info::compare);
// Write the names and grades
for (auto& s : students) {
    cout << s.name()
         << string(max_len + 1 - s.name().size(), ' ');
    try {
        cout << s.grade() << endl;
    } catch (domain_error e) {
        cerr << e.what() << endl;
    }
}
```

Subtleties: inheritance and containers

```
vector<Core> students;
Grad g(cin);
students.push_back(g);
```

What happens?

Subtleties: inheritance and containers

```cpp
vector<Core> students;
Grad g(cin);
students.push_back(g);
```

What happens?

We are allowed to store a `Grad` in a `vector<Core>`.
But only the `Core` part of the `Grad` will be stored!

Subtleties: functions

If base-class and derived-class have a function with the same name but different signatures, they behave as unrelated functions.

```
void Core::regrade(double d) { final = d; }
void Grad::regrade(double d1, double d2) {
    final = d1;
    thesis = d2;
}
```

If `r` is a reference to `Core`

```
r.regrade(100); // OK, call Core::regrade
r.regrade(100, 100);
// Compile error: Core::regrade takes 1 argument
```

This second call is an error even if `r` actually refers to a `Grad`!

Subtleties: functions (2)

If `r` is a reference to `Grad`

```
r.regrade(100); // Comp. error: Grad::regrade takes 2 arguments
r.regrade(100, 100); // OK, call Grad::regrade
```

Even though there is a base-class version that takes a single argument, it is effectively hidden by the existence of `regrade` in the derived class.

If you really want the base-class version, you need:
`r.Core::regrade(100);`

To use `regrade` as a **`virtual`** function, it must have the same interface in both the base and derived class:

```
virtual void Core::regrade(double d, double = 0) { final = d; }
void Grad::regrade(double d1, double d2) {
    final = d1;
    thesis = d2;
}
```