

# Inheritance and Dynamic Binding

Laurent Mathy

Object-Oriented Programming Projects

April 20, 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.

---

```
1  class Core {
2  public:
3      Core();
4      Core(std::istream& is);
5      std::string name() const;
6      std::istream& read(std::istream&);
7      double grade() const;
8  private:
9      std::istream& read_common(std::istream&);
10     std::string n;
11     double midterm, final;
12     std::vector<double> homework;
13 };
```

---

## Grad class

---

```
1 class Grad: public Core {
2 public:
3     Grad();
4     Grad(std::istream&);
5     double grade() const;
6     std::istream& read(std::istream&);
7 private:
8     double thesis;
9 };
```

---

- 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 **friends**.
- 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**.

---

```
1  class Core {
2  public:
3      Core();
4      Core(std::istream& is);
5      std::string name() const;
6      std::istream& read(std::istream&);
7      double grade() const;
8  protected: // Accessible to derived classes
9      std::istream& read_common(std::istream&);
10     std::string _name;
11     double midterm, final;
12     std::vector<double> homeworks;
13 };
```

---

## Operations: Core

```
12  string Core::name() const { return _name; }
13
14  double Core::grade() const {
15      return ::grade(midterm, final, homeworks);
16  }
17
18  istream& Core::read_common(istream& in) {
19      // Read and store student's name and exam grades
20      in >> _name >> midterm >> final;
21      return in;
22  }
23
24  istream& Core::read(istream& in) {
25      read_common(in);
26      read_hws(in, homeworks);
27      return in;
28  }
```

## Operations: Grad

---

```
11  istream& Grad::read(istream& in) {
12      read_common(in);
13      in >> thesis;
14      read_hws(in, homeworks);
15      return in;
16  }
17
18  double Grad::grade() const {
19      return std::min(Core::grade(), thesis);
20  }
```

---

- 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

---

```
1  class Core {
2  public:
3      // Default constructor for Core
4      Core(): midterm(0), final(0) { }
5      Core(std::istream& is) { read(is); }
6      // ...
7  };
8
9  class Grad: public Core {
10 public:
11     // Both constructors implicitly use Core::Core()
12     Grad(): thesis(0) { }
13     Grad(std::istream& is) { read(is); }
14     //...
15 };
```

---

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:

---

```
35 bool compare(const Core& c1, const Core& c2) {  
36     return c1.name() < c2.name();  
37 }
```

---

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

---

```
22 Grad g1(in);  
23 Grad g2(in);  
24  
25 Core c1(in);  
26 Core c2(in);  
27  
28 compare(g1, g2);  
29 compare(c1, c2);  
30 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  $\text{Grad} <: \text{Core}$ .

## Polymorphism(3): a new compare function

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

---

```
10 bool compare_grades(const Core& c1, const Core& c2) {  
11     return c1.grade() < c2.grade();  
12 }
```

---

## Polymorphism(3): a new compare function

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

```
10 bool compare_grades(const Core& c1, const Core& c2) {  
11     return c1.grade() < c2.grade();  
12 }
```

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 Grads, we want Grad::grade.
- If c1 or c2 are Cores, 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:

---

```
1 class Core {
2 public:
3     virtual double grade() const; // added `virtual`
4     // ...
5 };
```

---

- `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!

---

```
1 // Incorrect implementation
2 bool compare_grades(Core c1, Core c2) {
3     return c1.grade() < c2.grade();
4 }
```

---

- 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)

---

```
39 Core c;  
40 Grad g;  
41 Core* p = &c;  
42 Core& r = g;  
43  
44 c.grade(); // Static binding to Core::grade()  
45 g.grade(); // Static binding to Grad::grade()  
46 p->grade(); // Dynamic binding to type object p points to  
47 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.

---

```
1 class Abstract {
2     public:
3         virtual int pure() const = 0; // Pure virtual
4         // ...
5 };
```

---

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

## A program dealing with only undergrad records

```
10  vector<Core> students;
11  Core record;
12  string::size_type max_len = 0;
13  // Read and store the student records
14  while (record.read(cin)) { // Core::read()
15      max_len = max(max_len, record.name().size());
16      students.push_back(record);
17  }
18  // Alphabetize the student records
19  sort(students.begin(), students.end(), compare);
20  // Write the names and grades
21  for (auto& s : students) {
22      cout << s.name()
23           << string(max_len + 1 - s.name().size(), ' ');
24      try {
25          cout << s.grade() << endl; // Core::grade
26      } catch (domain_error e) {
27          cerr << e.what() << endl;
28      }
29  }
```

## A program dealing with only grad records

```
10 vector<Grad> students; // Different type in vector
11 Grad record; // Different type into which to read
12 string::size_type max_len = 0;
13 // Read and store the student records
14 while (record.read(cin)) { // Grad::read()
15     max_len = max(max_len, record.name().size());
16     students.push_back(record);
17 }
18 // Alphabetize the student records
19 sort(students.begin(), students.end(), compare);
20 // Write the names and grades
21 for (auto& s : students) {
22     cout << s.name()
23         << string(max_len + 1 - s.name().size(), ' ');
24     try {
25         cout << s.grade() << endl; // Grad::grade
26     } catch (domain_error e) {
27         cerr << e.what() << endl;
28     }
29 }
```

## 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 Cores identified by pointers:

---

```
10 bool compare_Core_ptrs(const Core* cp1, const Core* cp2) {
11     return compare(*cp1, *cp2);
12 }
```

---

## Dealing with both types of records (2)

```
15  vector<Core*> students; // Store pointers, not objects
16  Core* record; // Temporary must be a pointer as well
17  char ch;
18  string::size_type max_len = 0;
19
20  // Read and store the student records
21  while (cin >> ch) {
22      if (ch == 'U')
23          record = new Core; // Allocate a Core object
24      else
25          record = new Grad; // Allocate a Grad object
26      record->read(cin); // Virtual call
27      max_len = max(max_len, record->name().size());
28                  // ^ Dereference
29      students.push_back(record);
30  }
31
32  // Pass the version of compare() that works on pointers
33  sort(students.begin(), students.end(), compare_Core_ptrs);
```

## Dealing with both types of records (3)

```
35 // Write the names and grades
36 for (auto s : students) {
37     // s is a pointer
38     cout << s->name() // Dereference to call function
39         << string(max_len + 1 - s->name().size(), ' ');
40     try {
41         cout << s->grade() << endl; // Dereference to call
42
43     } catch (domain_error e) {
44         cerr << e.what() << endl;
45     }
46     delete s; // Free the object allocated when reading
47 }
```

## 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:

---

```
1 class Core {
2 public:
3     virtual ~Core() { }
4     // ...
5 };
```

---

- 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

```
10  class Student_info {
11  public:
12      // Constructors and copy control
13      Student_info() { }
14      Student_info(std::istream& is) { read(is); }
15      Student_info(const Student_info&);
16      Student_info& operator=(const Student_info&);
17      // Operations
18      std::istream& read(std::istream&);
19      std::string name() const {
20          if (cp) return cp->name();
21          else throw std::runtime_error("uninitialized Student");
22      }
23      double grade() const {
24          if (cp) return cp->grade();
25          else throw std::runtime_error("uninitialized Student");
26      }
27      static bool compare(const Student_info& s1, const Student_info& s2) {
28          return s1.name() < s2.name();
29      }
30  private:
31      std::unique_ptr<Core> cp;
32  };
```

## 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

---

```
7  istream& Student_info::read(istream& is) {
8      char ch;
9      is >> ch; // Get record type
10
11     // Assignment to `cp` will free if needed
12     if (ch == 'U')
13         cp = std::make_unique<Core>(is);
14     else
15         cp = std::make_unique<Grad>(is);
16
17     return is;
18 }
```

---

## 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:

---

```
1  class Core {
2      friend class Student_info;
3  protected:
4      virtual std::unique_ptr<Clone> clone() const
5          { return std::make_unique<Core>(*this); }
6      // ...
7  };
8  class Grad {
9  protected:
10     virtual std::unique_ptr<Clone> clone() const
11         { return std::make_unique<Grad>(*this); }
12     // ...
13 };
```

---

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 `friends` 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.
- `friendship` 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 assignment

```
20 Student_info::Student_info(const Student_info& s) {
21     if (s.cp)
22         cp = s.cp->clone();
23 }
24
25 Student_info& Student_info::operator=(const Student_info& s) {
26     if (&s != this) {
27         if (s.cp)
28             cp = s.cp->clone();
29         else
30             cp = nullptr;
31     }
32
33     return *this;
34 }
```

## Using the handle class

```
10  vector<Student_info> students;
11  Student_info record;
12  string::size_type max_len = 0;
13  // Read and store the student records
14  while (record.read(cin)) {
15      max_len = max(max_len, record.name().size());
16      students.push_back(record);
17  }
18  // Alphabetize the student records
19  sort(students.begin(), students.end(), Student_info::compare);
20  // Write the names and grades
21  for (auto& s : students) {
22      cout << s.name()
23           << string(max_len + 1 - s.name().size(), ' ');
24      try {
25          cout << s.grade() << endl;
26      } catch (domain_error e) {
27          cerr << e.what() << endl;
28      }
29  }
```



## Subtleties: inheritance and containers

---

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

---

What happens?

## Subtleties: inheritance and containers

---

```
1 vector<Core> students;  
2 Grad g(cin);  
3 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.

---

```
1 void Core::regrade(double d) { final = d; }
2 void Grad::regrade(double d1, double d2) {
3     final = d1;
4     thesis = d2;
5 }
```

---

If `r` is a reference to `Core`

---

```
1 r.regrade(100); // OK, call Core::regrade
2 r.regrade(100, 100);
3 // 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`

---

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

---

```
1 virtual void Core::regrade(double d, double = 0) { final = d; }
2 void Grad::regrade(double d1, double d2) {
3     final = d1;
4     thesis = d2;
5 }
```

---