C++ Programming with Class(es)

Laurent Michel
Computer Science & Engineering

Outline

- Classes and Objects
- Privacy
- Attributes
- Constructors
- Destructors
- Members

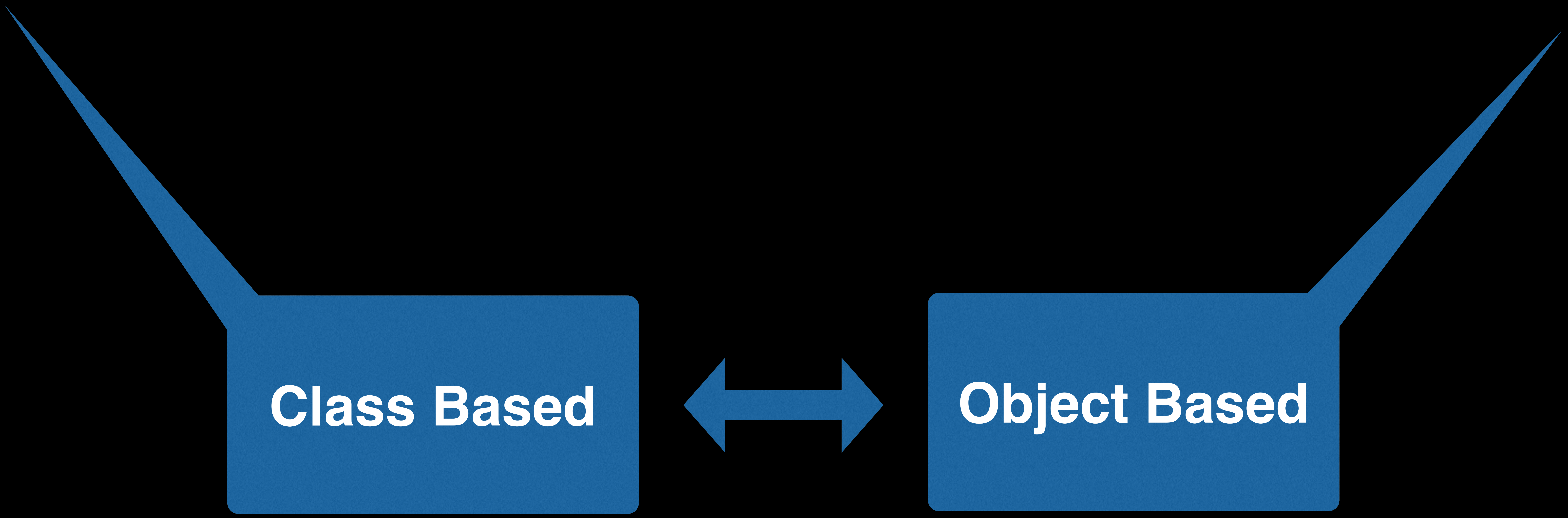
Classes

- Recall that there are two families of object-oriented languages

UCONN

Classes

- Recall that there are two families of object-oriented languages



Classes

- Recall that there are two families of object-oriented languages

C++, C#, Java

Smalltalk, Objective-C

Class Based ↔ **Object Based**

Classes

- Recall that there are two families of object-oriented languages

C++, C#, Java

Class Based

Smalltalk, Objective-C

Object Based

Class

- A class is a **TYPE**
- It describes **a set** of **objects** that
  - Share some properties
  - Meet some requirements
- Adding properties or requirements….
  - Constraints the type.
  - Fewer objects have the properties / requirements.

Objects ?

- We will call them *instances of a class*
  - Simply members of the set of objects denoted by the class
- When a program starts…
  - It has lots of classes
  - But (often) no ***instances***
  - ***Instances are created at runtime.***

Example

- Define a student!

```
#ifndef __STUDENT_H
#define __STUDENT_H

#include <string>

struct Student {
  std::string _name;
  double  _mt,_final;
  std::vector<double> _homeworks;
};

#endif
```

What is going on ?

- Reuse the “struct” concept of C
  - Attributes of structure are attributes of a class.
- A “struct” fits the definition of a class
- There is a catch though [more shortly]

Some Style

- Programmers can be picky with style….
- Convention *I* use
  - **Names** follow the **camel case** convention
  - **Classes** (and Types in general) start with an upper-case
  - **Attributes** start with an underscore
  - **Methods** (and functions in general) start with a lower-case
  - It's easier to tell what a name refers to!

https://en.wikipedia.org/wiki/CamelCase

Usage scenario

```
#include "student.H"
#include <memory>
#include <iostream>

int main()
{
  using namespace std;
  Student  s;  // creates an object on the stack
  shared_ptr<Student> sp(new Student); // creates an object on the heap
  s._name  = "Joe";
  s._mt    = 80;
  s._final = 90;
  s._homeworks.push_back(65);
  s._homeworks.push_back(42);
  s._homeworks.push_back(96);
  // initialize the stack object.
```

Usage scenario

```
  *sp = s; // copies the stack object on the heap

  cout << "S._name = " << s._name << endl;
  cout << "sp->_name = " << sp->_name << endl;

  cout << "&S._name = " << &s._name << endl;
  cout << "&sp->_name = " << &sp->_name << endl;
  s._name = "Bertie";
  cout << "S._name = " << s._name << endl;
  cout << "sp->_name = " << sp->_name << endl;

  return 0;
}
```

What’s wrong?

- The “structure” only **bundles** attributes
  - It does not enforce any access mechanisms
  - It does not provide high-level operations
  - It does not provide a **contract**
- **This is essentially structured programming**

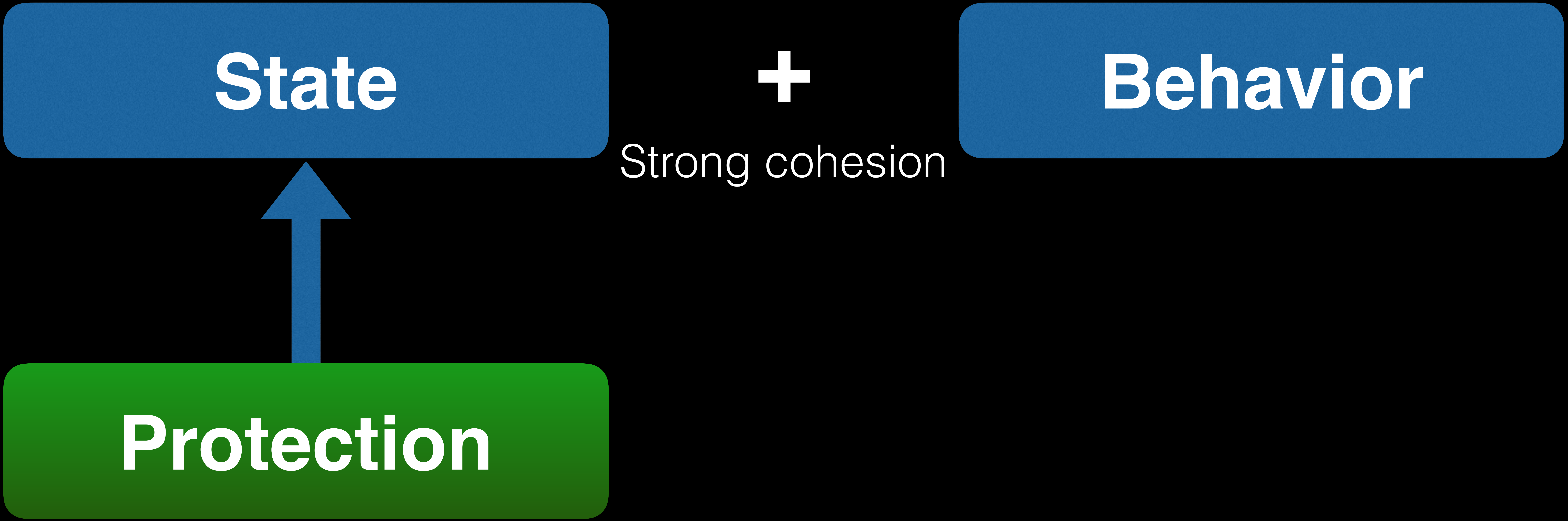
Going O.O.

State **+** **Behavior**

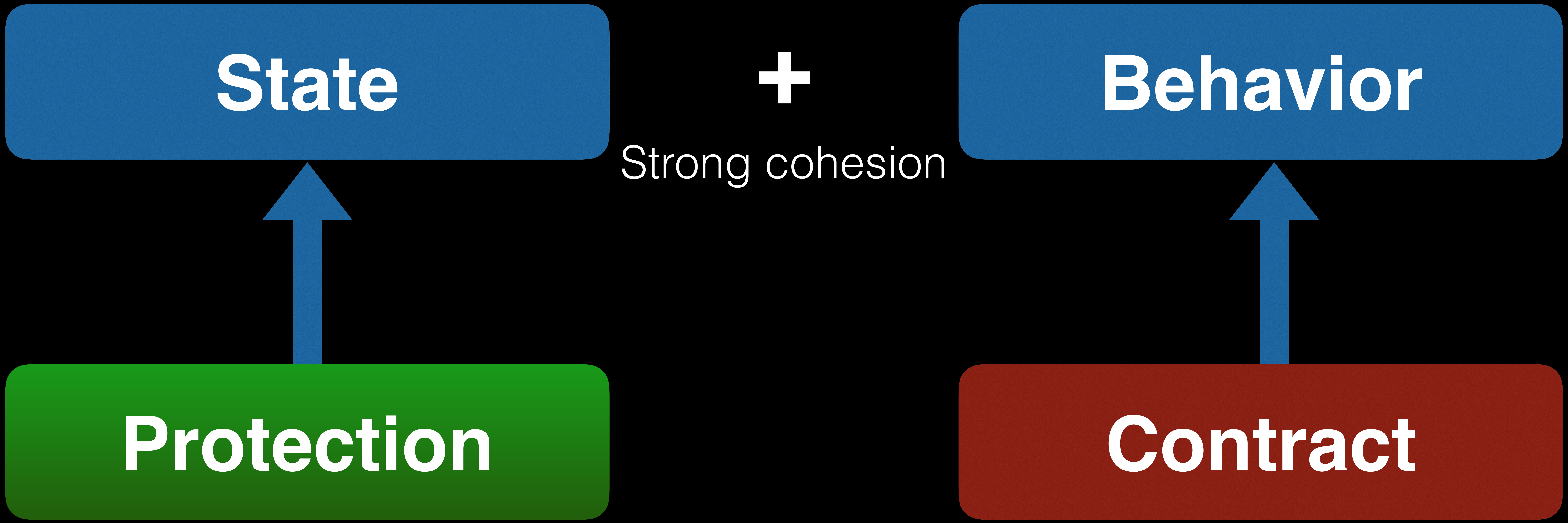
Strong cohesion

UCONN

Going O.O.



Going O.O.



What needs to change?

- Handle
  - State + Behavior
  - Lifetime
  - Privacy
  - Contracts

State + **Behavior**

student.H

```
#ifndef __STUDENT_H
#define __STUDENT_H
#include <string>
#include <vector>
#include <istream>

struct Student {
  std::string _name;
  double  _mt,_final;
  std::vector<double> _homeworks;
  void read(std::istream& is);
  void print(std::ostream& os);
};

#endif
```

UCONN

State + Behavior

student.cpp

```
#include "student.H"
void Student::read(std::istream& is) {
  is >> _name >> _mt >> _final;
  int nbH = 0;
  is >> nbH;
  for(int i=0;i<nbH;i++) {
     int v;
     is >> v;
     _homeworks.push_back(v);
  }
}
void Student::print(std::ostream& os) {
  os << "Name:" << _name << '[' << _mt << ',' << _final << ']' << std::endl;
  for(int v : _homeworks)
     os << "\t:" << v << std::endl;
}
```

Methods

- You can **overload** methods. Several definitions with:
  - Same name
  - Different # arguments
  - Different types of arguments
- C++ disambiguates the call and selects the right method.
- Do not confuse **overloading** with **overriding**

Lifetime

Concept

C++

- **Birth**
  - With constructors
- **Transfers**
  - With copy operators
- **Death**
  - With destructors

Birth

Constructor

- Simple Idea
  - Execute a method when creating an object
  - Can overload constructors too
- Multiple “kind” of constructors

Default **Custom** **Copy** **Move**

Default Constructor

- Called when no arguments provided

```
struct Student {
    std::string _name;
    double  _mt,_final;
    std::vector<double> _homeworks;

    Student()   { _mt = _final = 0;}

    void read(std::istream& is);
    void print(std::ostream& os);
};
```

Notes
1. **Not** initializing _name
2. **Not** initializing _homeworks

Custom Constructor

- Called with arguments to setup the instance

```
struct Student {
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;

   Student()                         { _mt = _final = 0;}
   Student(const std::string& s) { _name = s;_mt=_final=0;}

   void read(std::istream& is);
   void print(std::ostream& os);
};
```

Notes

1. You do **not** have to inline!
2. You can put the constructor code in the .cpp file.

Copy Constructor

Notes

1. If you do not provide a copy constructor, the compiler does for you!
2. Understand shallow vs. deep copy (as in C)

```
struct Student {
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;
   Student()                      { _mt = _final = 0;}
   Student(const std::string& s) { _name = s;_mt=_final=0;}
   Student(const Student& s2)     {
      _name = s2._name;_mt=s2._mt;_final=s2._final;
      for(double d : s2._homeworks) _homeworks.push_back(d);
   }
   void read(std::istream& is);
   void print(std::ostream& os);
};
```

Copy Take 2

```
struct Student {
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;
   Student()                          { _mt = _final = 0;}
   Student(const std::string& s) { _name = s;_mt=_final=0;}
   Student(const Student& s2)
      : _name(s2._name),
        _mt(s2._mt),
        _final(s2._final),
        _homeworks(s2._homeworks)
   {}
   void read(std::istream& is);
   void print(std::ostream& os);
};
```

```
                  Notes
1. Lighter syntax to copy attributes
2. Calls the copy constructors rec.
```

Concept

C++

Move Constructor

```
struct Student {
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;
   Student()                          { _mt = _final = 0;}
   Student(const std::string& s) { _name = s;_mt=_final=0;}
   Student(const Student& s2)
      : _name(s2._name),
        _mt(s2._mt),_final(s2._final),
        _homeworks(s2._homeworks)
   {}
   Student(Student&& s2)
      : _name(std::move(s2._name)),
        _mt(s2._mt),_final(s2._final),
        _homeworks(std::move(s2._homeworks))
   {}
   void read(std::istream& is);
   void print(std::ostream& os);
};
```

Caveats

1. **Special new syntax!!!!**
2. Move constructors “delete” implicit copy constructors
3. Do not confuse constructors and copy operators (to be seen shortly)

Transfers

Transfer operators

- Simple Idea
  - Execute a method invoking assignment (e.g., `x = y;`)
- Multiple transfer operators

Copy **Move**

Copy Operators

```
struct Student {
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;
   Student()                        { _mt = _final = 0;}
   Student(const std::string& s) { _name = s;_mt=_final=0;}
  …
   Student& operator=(const Student& s) {
      _name = s._name;
      _mt   = s._mt;
      _final = s._final;
      _homeworks = s._homeworks;
      return *this;
   }
   void read(std::istream& is);
   void print(std::ostream& os);
};
```

Notes

1. **C++ let you redefine '='**
2. Argument is **const T&**
3. return value is **T&**

Copy Operators (Take 2)

```
struct Student {
    …
   Student& operator=(const Student& s) {
      if (this == &s) return *this;
      _name = s._name;
      _mt   = s._mt;
      _final = s._final;
      _homeworks = s._homeworks;
      return *this;
   }
   void read(std::istream& is);
   void print(std::ostream& os);
};
```

Notes

1. **Protect against `x = x;`**

Move Operator

```
struct Student {
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;
   Student()                        { _mt = _final = 0;}
   Student(const std::string& s) { _name = s;_mt=_final=0;}
   …
  Student& operator=(Student&& s) {
      _name = std::move(s._name);
      _mt   = s._mt;
      _final= s._final;
      _homeworks = std::move(s._homeworks);
      return *this;
   }
   …
}
```

Note

1. A move “**steals**” from the source
2. A custom move **disables** default copy

Death

Purpose

- Release whatever resource is held by the instance
  - Memory
  - Files
  - Network connections (sockets)
  - …
- Made easier if you use `shared_ptr<T>`

Only One Destructor

```
struct Student {
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;
   Student()                        { _mt = _final = 0;}
   Student(const std::string& s) { _name = s;_mt=_final=0;}
   Student(const Student& s2)
      : _name(s2._name),
        _mt(s2._mt),_final(s2._final),
        _homeworks(s2._homeworks)
   {}
   …
   ~Student() {
      std::cout << "destroy(" << this << ")" << std::endl;
   }
}
```

But there is more!

- You may not know what is being destructed
- You need a mechanism to handle *polymorphic destruction*
- That is possible with ….

$$\boxed{\begin{array}{c}\texttt{virtual}\\ \texttt{destructors}\end{array}}$$

- More on this once we have covered polymorphism

Protection

- You wish to….
  - Hide valuable bits
  - Protect from intruders

Protection

- However
  - A C-style “**`struct`**” exposes everything
    - State
    - Behavior
  - We need to alter protection strategy

Protection

- Three strategies are available

public **protected** **private**

By default…

- C-style **`struct`** **assume** that everything is **public**!
- But you can change that!

UCONN

Making the state private

```
struct Student {
   Student();
   Student(const std::string& s);
   Student(const Student& s2);
   Student(Student&& s2);
   ~Student();
   Student& operator=(const Student& s);
   Student& operator=(Student&& s);
   void read(std::istream& is);
   void print(std::ostream& os);
   void setName(const std::string& n);
private:
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;
};
```

Notes
1. You may need setters/getters now!
2. You can switch back and forth
3. use public: or private:

Why this Default?

- It is not a good default…
- But it is backward compatible with C (where everything is public)
- If you want a default where everything is private….
  - Use a different keyword!

Student Again!

```
class Student {
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;
public:
   Student();
   Student(const std::string& s);
   Student(const Student& s2);
   Student(Student&& s2);
   ~Student();
   Student& operator=(const Student& s);
   Student& operator=(Student&& s);
   void read(std::istream& is);
   void print(std::ostream& os);
   void setName(const std::string& n);
};
```

Notes

1. class makes default private
2. you can switch to public for methods

Class Protection

- By Default everything is **private**
  - Attributes
  - Methods
- You can change (several times) the default
  - Declaration that follows a privacy change use the new privacy

Protected ?

- Only makes sense when dealing with inheritance!
- We will come back to this

Contract

- C++ supports the separation of
  - Contract
  - Implementation
- Two mechanisms
  - Header + Implementation file ✓
  - Abstract classes. ✗

Abstract Class

- Idea: A class with
  - (state +) behavior
  - no implementation! [or a partial implementation]
- Corollary
  - One **never instantiates** an abstract class
  - One **only sub-classes**   an abstract class

Example

```
class AStudent {
public:
   virtual void read(std::istream& is)  = 0;
   virtual void print(std::ostream& os) = 0;
   virtual void setName(const std::string& n) = 0;
};
```

A New keyword!

= 0 means:
⇒ Don't expect an implementation

How to Use Abstract Classes?

- Simple idea
  - Use ***inheritance*** to ***claim*** that you support the contract
  - **Provide** an ***implementation*** in the sub-class for "**pure**" methods

Inheritance

```
class Student: public AStudent {
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;
public:
   Student();
   Student(const std::string& s);
   Student(const Student& s2);
   Student(Student&& s2);
   ~Student();
   Student& operator=(const Student& s);
   Student& operator=(Student&& s);
   void read(std::istream& is);
   void print(std::ostream& os);
   void setName(const std::string& n);
};
```

UCONN

Inheritance

```
class Student: public AStudent {
   std::string _name;
   double  _mt,_final;
   std::vector<double> _homeworks;
public:
   Student();
   Student(const std::string& s);
   Student(const Student& s2);
   Student(Student&& s2);
   ~Student();
   Student& operator=(const Student& s);
   Student& operator=(Student&& s);
   void read(std::istream& is);
   void print(std::ostream& os);
   void setName(const std::string& n);
};
```

Overriden methods!

Methods and Operators

- C++ lets you
  - Define methods (prefix style)
  - Define operators (prefix, infix and postfix!)

complex.H

```
class Complex {
  double _real;
  double _imag;
public:
  Complex() { _real = _imag = 0;}
  Complex(double r) { _real = r;_imag = 0;}
  Complex(double r,double i) { _real = r;_imag = i;}
  void print(std::ostream& os);
  Complex conjugate();
  Complex add(const Complex& c2) const;
  Complex mul(const Complex& c2) const;
};
```

complex.cpp

```
#include "complex.H"
void Complex::print(std::ostream& os) {
  if (_imag) {
    if (_imag > 0)
      os << '(' <<  _real << " + " << _imag << "i" << ')';
    else
      os << '(' <<  _real << ' ' << _imag << "i" << ')';
  } else
    os << _real;
}
Complex Complex::add(const Complex& c2) const {
  return Complex(_real+c2._real,_imag + c2._imag);
}
Complex Complex::mul(const Complex& c2) const {
  const double pr = _real * c2._real;
  const double pi = - _imag * c2._imag;
  return Complex(pr + pi,_real * c2._imag + _imag * c2._real);
}
Complex Complex::conjugate() { return Complex(_real,-_imag);}
```

Usage

```
#include <memory>
#include <iostream>
#include "complex.H"

int main() {
  Complex a(1,1),b(1,2);
  Complex c = a.add(b);
  Complex d = a.mul(b);
  Complex e = d.conjugate();
  c.print(std::cout);
  std::cout << std::endl;
  d.print(std::cout);
  std::cout << std::endl;
  e.print(std::cout);
  std::cout << std::endl;
  return 0;
}
```

Ugly!

Non-member Functions

- In C++ you can
  - Write classes with methods in them
  - Write plain-old function taking objects as inputs.
- Question
  - When should you write a function vs. a method?

Revised Header

```
class Complex {
 double _real;
 double _imag;
 Complex add(const Complex& c2) const;
 Complex mul(const Complex& c2) const;
public:
 Complex() { _real = _imag = 0;}
 Complex(double r) { _real = r;_imag = 0;}
 Complex(double r,double i) { _real = r;_imag = i;}
 void print(std::ostream& os);
 Complex conjugate();
 friend Complex operator+(const Complex& a,const Complex& b)
 { return a.add(b);}
 friend Complex operator*(const Complex& a,const Complex& b)
 { return a.mul(b);}
};
```

Better Program

```
#include <memory>
#include <iostream>
#include "complex.H"
int main() {
  Complex a(1,1),b(1,2);
  Complex c = a + b;
  Complex d = a * b;
  Complex e = d.conjugate();
  c.print(std::cout);
  std::cout << std::endl;
  d.print(std::cout);
  std::cout << std::endl;
  e.print(std::cout);
  std::cout << std::endl;
  return 0;
}
```

The output case

- Consider defining an output operator (<<)
- To replace call to “print”

Header File

```
class Complex {
  double _real;
  double _imag;
  Complex add(const Complex& c2) const;
  Complex mul(const Complex& c2) const;
  void print(std::ostream& os) const;
public:
  Complex() { _real = _imag = 0;}
  Complex(double r) { _real = r;_imag = 0;}
  Complex(double r,double i) { _real = r;_imag = i;}
  Complex conjugate();
   …
  friend std::ostream& operator<<(std::ostream& os,const Complex& c) {
    c.print(os);
    return os;
  }
};
```

Usage

```
#include <memory>
#include <iostream>
#include "complex.H"

int main() {
   Complex a(1,1),b(1,2);
   Complex c = a + b;
   Complex d = a * b;
   Complex e = d.conjugate();
   using namespace std;
   cout << c << endl;
   cout << d << endl;
   cout << e << endl;
   return 0;
}
```

Looks good now!