Inheritance

Inheritance

It expresses the idea: objects of type B are of type A but have additional properties

Example 1:

- a student is a member of a university
- a PhD student is a member of the scientific staff of a university
- a post-doc is also member of the scientific staff of a university but has a PhD degree
- a professor also belongs to the scientific staff but is permanent
- etc

Example 2:

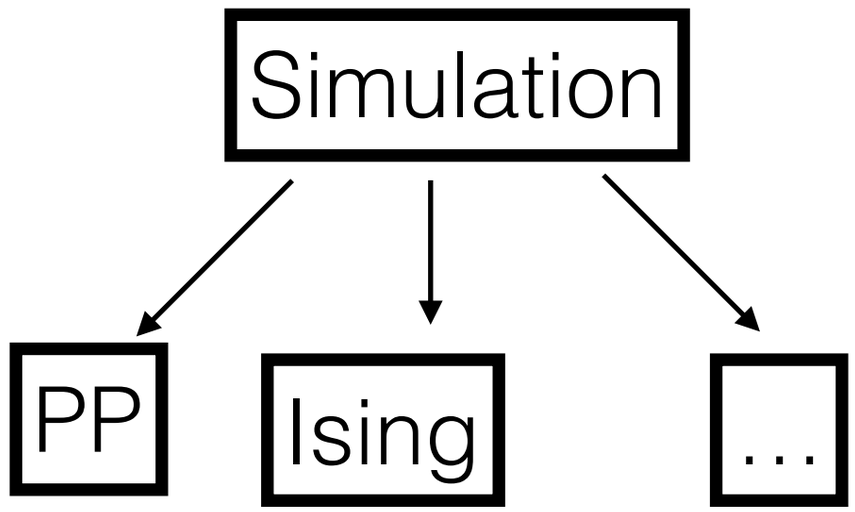
- simulations can be run in general
- a molecular dynamics simulation is a simulation but with more features
- a predator-prey simulation is a simulation but solves ODEs
- a classical Monte Carlo simulation is a simulation sharing parameters such as the duration, scheduling, … but needs to specify random number generators
- a quantum Monte Carlo simulation shares with a classical simulation the statistical evaluation but is far more complicated
- we can think of a simulation class which serves as an interface for the actual program, reading parameters, preprocessing and postprocessing the data etc

Base and derived classes

```
class Simulation {    // example of a base class
public:
  Simulation() : name("hi") {};
  void set_name(const std::string & s) {name = s;};
  void print(std::ostream & os) const { os << name << "\n";}
  std::string name;
};

class PredatorPrey : public Simulation {
public:
  void RK4() { /* ... */}
private:
  double a,b;
};


int main() {
  Simulation MySim;
  MySim.set_name("new simulation");     // OK
  MySim.print(std::cout);       // OK
  PredatorPrey NewSim;
  NewSim.print(std::cout);      // OK
  NewSim.RK4(); // OK
  return 0;
}
```



- *Simulation* is the base class, *PredatorPrey* is derived from it
- *PredatorPrey* has access to *name*, *set_name* and *print* from *Simulation*
- *Simulation* does not have access to *a* and *b* from *PredatorPrey*
- unless otherwise specified, *PredatorPrey* calls the constructor from *Simulation*
- *public Simulation* means that the members of *Simulation* are treated in the same way (not less restrictive) in the derived class; in case of protected Simulation all members of the base class would be treated more restrictively in the derived class. The default is private for classes and public for struct

The keyword protected

```
class Simulation {    // example of a base class with protected members
public:
  Simulation() : name("hi") {};
  void set_name(const std::string & s) {name = s;};
  void print(std::ostream & os) const { os << name << "\n";}
protected:
  std::string name;
};

class PredatorPrey : public Simulation {
public:
  void RK4() { std::cout << "RK4 calculation of " << name << "\n";}
private:
  double a,b;
};


int main() {
  Simulation MySim;
  MySim.set_name("new simulation");     // OK
  MySim.print(std::cout);       // OK
  PredatorPrey NewSim;
  NewSim.set_name("carnivorous");
  NewSim.print(std::cout);      // OK
  NewSim.RK4(); // OK
  return 0;
}
```

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

- *name* is a protected member of *Simulation :* it is public to derived classes but private for all other classes
- private members of the base class are not accessible to derived classes

Base and derived classes

The publicly derived class inherits all members of the base class except:

- constructors and destructors
- operator= (assignment)
- friends
- private members

the constructor of the base class can be called in different ways:

```
class Simulation {    // example of a base class
public:
  Simulation() : name("hi") {std::cout << name << "\n";};
  Simulation(const std::string& s) : name(s) {std::cout << name << "\n";};
protected:
  std::string name;
};

class PredatorPrey : public Simulation {
public:
  PredatorPrey(const std::string& s)  {name = s; std::cout << name << "\n";}
  void RK4() { /* ... */}
private:
  double a,b;
};

class Ising : public Simulation {
public:
  Ising(const std::string& s) : Simulation(s)  {name = s; std::cout << name << "\n";}
  void MonteCarloRun() { /* ... */}
private:
  int N;
};
```

calls default constructor of base class

calls specific constructor of base class

Multiple inheritance

A class may inherit from more than one base class: (example from cplusplus.com)

```
class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
};

class Output {
  public:
    static void print (int i);
};

void Output::print (int i) {
  cout << i << '\n';
}

class Rectangle: public Polygon, public Output {
  public:
    Rectangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height; }
};

class Triangle: public Polygon, public Output {
  public:
    Triangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height/2; }
};

int main () {
  Rectangle rect (4,5);
  Triangle trgl (4,5);
  rect.print (rect.area());
  Triangle::print (trgl.area());
  return 0;
}
```

design considerations

- Our class Simulation is insufficient from a design point of view
- different types of simulations require different parameters and data sets
- they require different algorithms
- they require different postprocessing tools
- we only know that every simulation should have read_params(), save(), load(), test(), run(), … functions
- only the derived class can properly implement these functions
- solution: separate the interface from implementation.
- in C++ interfaces are implemented as abstract base classes

Virtual member functions

```
class Simulation {    // example of an abstract base class and use of the keyword virtual
public:
  Simulation() {};
  virtual std::string name() const = 0;
  virtual void run() = 0;
};

class PredatorPrey : public Simulation {
public:
  std::string name() const;
  void run();
};


std::string PredatorPrey::name() const {
  return static_cast<std::string>("carnivorous");
}
```

```
void PredatorPrey::run() {
  /* ... */
  return;
}


int main() {
  //Simulation MySim;            // Error, base class is abstract
  PredatorPrey NewSim;           // OK
  std::cout << NewSim.name() << "\n";
  Simulation& Mysim = NewSim;            // OK, since it is a reference
  return 0;
}
```

- the keyword virtual indicates that the member function can be overwritten
- = 0 indicates that this must be provided for any concrete Simulation. It turns simulation into an abstract base class, of which no instances can be defined
- a reference to the base class is allowed

virtual member functions

Without the keyword virtual decisions are made at compile time (see polymorph6.cpp)

```
class Polygon {
protected:
  int width, height;
public:
  void set_values (const int a, const int b) { width=a; height=b; }
  int area() const {return width*height;}
  void print() { std::cout << " I am polygon\n";}
};
class Rectangle: public Polygon {
public:
  int area () const { return (width * height); }
  void print() { std::cout << " I am Rectangle\n";}
};

void static_print(Polygon p) {
  p.print();      // always invokes print of Polygon, never of a derived class (compile time binding)
}

void dynamic_print(Polygon* p) {
  p->print();     // always invokes print of Polygon, never of a derived class (compile time binding)
}
```

The output of this is always “I am a polygon”. Compare this to:

```
class Polygon_abstract {
protected:
  int width, height;
public:
  void set_values (const int a, const int b) { width=a; height=b; }
  int area() const {return width*height;}
  virtual void print() = 0;
};

class Rectangle_bis: public Polygon_abstract {
public:
  int area () const { return (width * height); }
  void print() { std::cout << " I am Rectangle\n";}
};
```

```
void dynamic_print(Polygon_abstract* p) {
  p->print();     // run-time binding
}
```

Virtual function table

How does the program know the concrete type of an object?

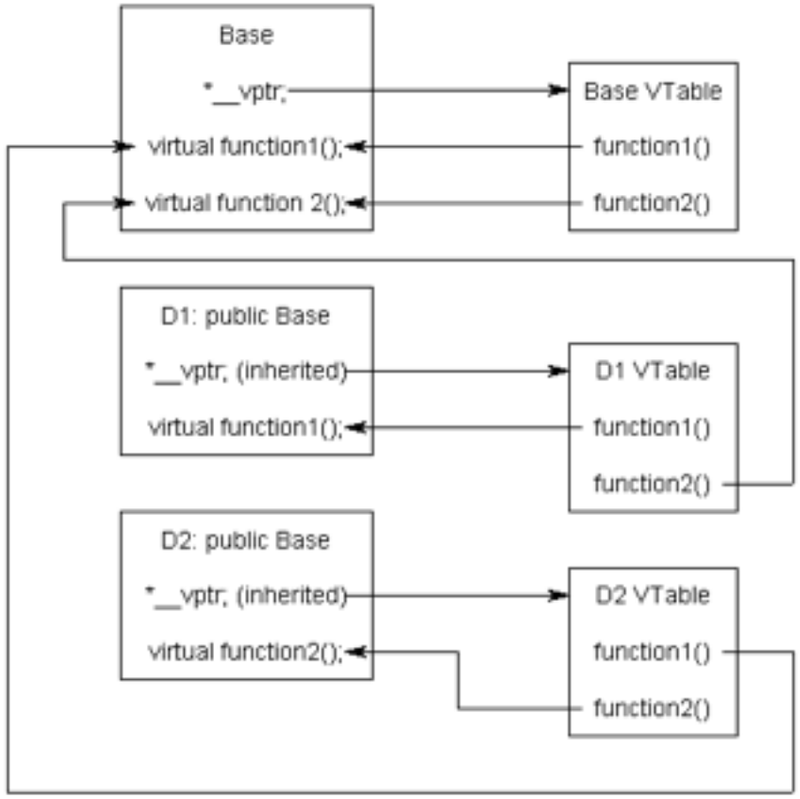
- the compiler creates a virtual function table (vtable), containing pointers to the functions
- a pointer is stored in that object, before (or after, compiler-dependent) any members
- the program checks the vtable of the object to find out which address to call (needs 2 memory accesses and cannot be inlined)
- is needed for polymorphism and late run-time binding (also exists in Python, …)

this line is added automatically by the compiler, not by the programmer

```
class Base
{
public:
    FunctionPointer *__vptr;
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    virtual void function1() {};
};

class D2: public Base
{
public:
    virtual void function2() {};
};
```



example from learncpp.com

polymorphism

A pointer to a derived class is type-compatible with one to its base class.
Polymorphism is the art of taking advantage of this feature.

```
class Simulation {
public:
  Simulation() {};
  virtual std::string name() const = 0;
  virtual void run() = 0;
};

class PredatorPrey : public Simulation {
public:
  std::string name() const;     // as before
  void run();                   // as before
};
class Ising : public Simulation {
public:
  std::string name() const;     // as before
  void run();                   // as before
};
int main() {
  PredatorPrey MyPP;
  Ising MyIsing;
  Simulation* sim1 = &MyPP;                 // polymorphism
  Simulation* sim2 = &MyIsing;              // polymorphism
  std::cout << sim1->name() << "\t" << sim2->name() << "\n";
  return 0;
}
```

The pointers are of the base class and take the address of objects of the derived class. Dereferencing of the pointers is allowed, but only members of the base class can be accessed. In case Ising had additional members, they could not be addressed by dereferencing sim2

polymorphism

Overriding/overruling without virtual may lead to counter-intuitive results:

```
class Polygon {
protected:
  int width, height;
public:
  void set_values (const int a, const int b) { width=a; height=b; }
  int area() const {return width*height;}
  void print() { std::cout << " I am polygon\n";}
  void print(const int color) { std::cout << " I am " << color << " polygon\n";}
};


class Rectangle: public Polygon {
public:
  int area () const { return (width * height); }
  void print(std::string& s) { std::cout << " I am " << s << " Rectangle\n";}
};

int main () {
  Polygon poly;
  Rectangle rect;
  poly.set_values(4,5);
  rect.set_values(5,6);
  std::string s = "beautiful";
  rect.print(s);
  // rect.print();          // error
  // rect.print(1);         // error
  poly.print(5);
  poly.print();
  return 0;
}
```

The print function in the derived class overrides all print functions of the base class, even those with different arguments!

polymorphism

Destructors are not inherited but should always be virtual: you want to invoke the correct destructor using the run-time type of the object.

```
void cleanup(Simulation* sim) {
  delete sim;
}
```

If the destructor of Simulation is not virtual, delete will be compile time bound and invoke the Simulation class destructor. However, if at runtime sim was pointing to an object of an Ising derived class, we really want to invoke the destructor of the derived class. Therefore, the destructor should be virtual — *even though the names do not quite match*, ~Simulation() vs ~Ising()

Some compilers will warn you if you define a class with virtual functions but a non-virtual destructor

After the destructor of the derived class is completed, the destructor of the base class is invoked automatically — you never invoke this explicitly

polymorphism

Assigning/copying a derived to a base "*slices*" the object.

In the Simulation class operator= takes as argument a reference to a Simulation object, but it is perfectly fine when this is a reference to an Ising object. The program will copy/assign the Simulation-part of Ising and ignore the Ising-specific part, "slicing" the Ising object and throwing the rest away. The result is a Simulation object.

The same holds for the copy constructor, which passes objects by value: it slices the object.

Note that the difference with Simulation* and Ising* , when there is no loss of information.

example: see polymorph5.cpp

Assigning/copying a base to a derived is not allowed.

rationale: I could only copy from PredatorPrey the members to Simulation that also exist in Simulation. The specific members of PredatorPrey would remain uninitialized, which is unsafe. This would inevitably happen in the copy constructor. In practice, one could create a special assignment operator= (which then only copies the relevant members) but in practice this is never needed.

dynamic allocation and polymorphism

A final example on polymorphism with dynamic memory allocation

```
// dynamic allocation and polymorphism -- from cplusplus.com
// note that Clang will produce a warning on calling delete on the abstract 'Polygon' which has a non-virtual destructor
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height/2; }
};

int main () {
  Polygon * ppoly1 = new Rectangle (4,5);
  Polygon * ppoly2 = new Triangle (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

object oriented programming

object oriented programming

(encapsulation, overloading, inheritance, polymorphism,…)

- derived class derived from a base class
- uses virtual functions
- concrete type decided at runtime
- one function for the base class -> saves space
- virtual functions are slow due to lookup in type table
- useful for big frameworks, user interfaces ,…

generic programming

- works for objects having the right members
- uses templates
- concrete type decided at compile time
- every instance has its own function -> more space
- no lookup, can be inlined: faster
- useful for small fast functions, low level constructs, generic algorithms, …

procedural programming

Let us compare procedural, generic and object-oriented programming for the integration exercise. Procedural approach:

```
#include <iostream>
#include <cmath>

using namespace std;

double fun(double x) {
  return x*exp(-x);
}

double integrate( double (*f) (double), double a, double b, unsigned int N)
  {
    double res = 0;
    double x = a;
    double dx = (b-a)/N;
    for (unsigned int i=0; i<N; ++i) {
      res += f(x);
      x += dx;
    }
    return (res*dx);
}

int main() {
  cout << integrate(fun, 0.,1., 1000) << "\n";
  return 0;
}
```

Same as in C, Fortran, …

generic programming

Same but with templates:

```
#include <iostream>
#include <cmath>

using namespace std;

class fun {
public:
  double operator() (double x) { return x*exp(-x);}
};

template <class T, class F>
T integrate(F f, T a, T b, unsigned int N) {
    T res=T(0);
    T x=a;
    T dx=(b-a)/N;
    for (unsigned int i=0; i<N; ++i) {
      res +=f(x);
      x += dx;
    }
    return (res*dx);
}


int main() {
  cout << integrate(fun(), 0. ,1., 1000) << "\n";
  return 0;
}
```

Allows inlining!
works for any type T

object-oriented programming

```
#include <iostream>
#include <cmath>

using namespace std;

class Integrator {
public:
  Integrator() {}
  double integrate(double a, double b, unsigned int n);
  virtual double f(double) = 0;
};
```

```
double Integrator::integrate(double a, double b, unsigned int N) {
  double res=0;
  double x = a;
  double dx = (b-a)/N;
  for (unsigned int i=0; i<N; ++i) {
    res += f(x);
    x += dx;
  }
  return (res*dx);
}

class Func : public Integrator {
public:
  Func() {}
  double f(double x) { return x*exp(-x);}
};


int main() {
  Func fun;
  cout << fun.integrate(0., 1., 1000) << "\n";
  return 0;
}
```

factory

A factory is a utility that creates an instance of a class from a family of derived classes

(it is a notion that simulates a virtual constructor)

it makes an abstraction of the creation of objects; the creation of objects is separated from its actual implementation

this extra layer is useful in the context of interfaces (say you run different types of simulations in a browser and use libraries for the GUI; your simulation class is best an abstract base class (interface) with an additional factory class for the creation of simulations)

see factory.cpp for an example

override and final

C++11

the keyword override ensures that a function is virtual and overrides a virtual function of the base class

```
// from cppreference.com
struct A
{
    virtual void foo();
    void bar();
};

struct B : A
{
    //void foo() const override; // Error: B::foo does not override A::foo
                                 // (signature mismatch)
    void foo() override; // OK: B::foo overrides A::foo
    //void bar() override; // Error: A::bar is not virtual
};
```

the keyword final ensures that a function is virtual and may not be overridden by a derived class

Question

- It is a very bad idea to treat arrays polymorphically. It will almost never work as you intended. Can you explain why and give an example?