An Introduction to C++

Inheritance
Exceptions
A C++ review: from modular to generic programming

Inheritance

- is another very important feature
- it models the concept:
  objects of type B are the same as A, but in addition have…
- Examples
  - A shape is a 2D figure which has an area and can be drawn, although I know neither generally
  - A triangle is a shape, but its area is … and it looks like …
  - A square is a shape, but its area is … and it looks like …
  - A complex figure is a shape and consists of an array of shapes
  - A monoid is a semigroup, but in addition contains a unit element
  - A group is a monoid, but in addition has an inverse
  - A simulation can be run but I don't know how generally
  - A Penna simulation is run this way …

Abstract base classes

- are good for expressing common ideas
- We want to have a function that can run a simulation and print some information:
  - ```
    void perform(Simulation& s) {
      std::cout << "Running the simulation "
                << s.name() << "\n";
      s.run(); // run it
    }
    ```
- This class must have an `name()` and a `run()` member function
  - ```
    class Simulation{
    public:
      Simulation () {};
      virtual std::string name() const =0;
      virtual void run() =0;
    };
    ```
  - `virtual` means that this function depends on concrete simulation
  - `=0` means that this function **must** be provided for any concrete simulation

Concrete derived classes

- PennaSim and IsingSim are both Simulations:
  - ```
    class PennaSim: public Simulation {
    public:
      std::string name() const;
      void run();
    };
    ```
  - ```
    class IsingSim: public Simulation{
    public:
      std::string name() const;
      void run()
    };
    ```
- Examples
  - `Simulation x;`
    // Error since it is abstract! `name()` and `run()` not defined
  - `PennaSim p;` // OK!
  - `IsingSim i;` // OK!
  - `Simulation& sim=p;` // also OK, since it is a reference!

Using inheritance

- recall the function `void perform(Simulation&);`
- let us call it for two simulations

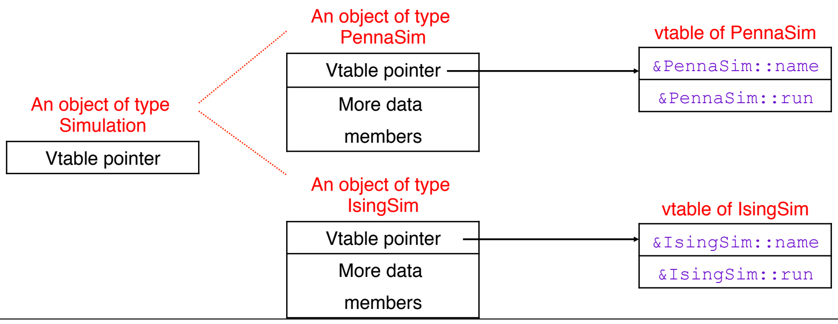
```
PennaSim p;
IsingSim i;
perform(p); // will use PennaSim::name() and PennaSim::run()
perform(i); // will use IsingSim::name() and IsingSim::run()
```

- All virtual function can be redefined by derived class
- In addition a derived class can define additional members

- There exists a third access specifier: `protected`
  - means `public` for derived classes
  - means `private` for others

The virtual function table

- How does the program know the concrete type of an object?
  - The compiler creates a virtual function table (vtable) for each class
    - The table contains pointers to the functions
  - A pointer to that table is stored in the object, before the other members
  - The program checks the virtual function table of the object for the address of the function to call
    - Needs two memory accesses and cannot be inlined



Using templates instead

- The same could be done with templates:
  - ```
    template <class SIMULATION>
    void perform(SIMULATION& s) {
      std::cout << s.name() << "\n";
      s.run();
    }
    ```
  - ```
    class PennaSim {
    public:
      std::string name() const;
      void run();
    };
    ```
  - ```
    PennaSim p;
    perform(p); // instantiates the template for PennaSim
    ```
- But type of `SIMULATION` must be known at compile time!

Comparing virtual functions and templates

- Object Oriented Programming:
  - ```
    void perform(Simulation& s)
    {
      s.run();
    }
    ```
  - Object needs to be derived from Simulation
  - Concrete type decided at runtime
- Generic programming:
  - ```
    template <class SIM> void perform(SIM& s)
    {
     s.run();
    }
    ```
  - Object needs to have a run function
  - Concrete type decided at compile time
    ```

Virtual functions versus templates

Object oriented programming

- uses virtual functions
- decision at run-time
- works for objects derived from the common base
- one function created for the base class -> saves space
- virtual function call needs lookup in type table -> slower
- extension possible using only definition of base class
- Most useful for application frameworks, user interfaces, "big" functions

Generic programming

- uses templates
- decision at compile-time
- works for objects having the right members
- a new function created for each class used -> more space
- no virtual function call, can be inlined -> faster
- extension needs definitions and implementations of all functions
- useful for small, low level constructs, small fast functions and generic algorithms

When to use which?

- Generic programming allows inlining
  - faster code
- Object oriented programming more flexible
  - how to age an Array of animals of different types?

```
void show(std::vector<Animal*> a) {
  for (int i=0; i<a.size(); ++i)
    a[i]->grow();
}
```

  - This works for array of mixed animals, e.g. fish, sheep, …

Example: random number generators

- We want to be able to switch random number generators at runtime: use virtual functions
- First attempt: rng1.h
  - Make `operator()` a virtual function
  - Problem: virtual function calls are slow
- Second attempt: rng2.h
  - Store a buffer of random numbers
  - `operator()` uses numbers from that buffer
  - Only when buffer is used up, a virtual function `fill_buffer()` is called to create many random numbers
  - This reduces the cost of inheritance since the virtual function is called only rarely

Example: Penna model with fishing

- The Penna population class was:
  - ```
    class Population {
      … constructors and more …
      void simulate(int years); // the full simulation
      void step(); // one year
    }
    ```
  - ```
    void Population::simulate(int years)
    {
      while (years--)
        step();
    }
    ```
  - ```
    void Population::step()
    {
      … lots of work to do one year …
    }
    ```
    ```

Example: Penna with fishing (part 2)

- Now we want to do a new simulation with fishing
- We want to reuse code and not copy&paste

```
class FishingPopulation : public Population
{
  … constructors …
  void step(); // one year
}
```

```
void FishingPopulation::step()
{
  Population::step(); // do the normal aging
  … then implement fishing …
}
```

Example: Penna with fishing (part 3)

- This will not do what we want since `Population::simulate` does not call the step function of `FishingPopulation`

```
void Population::simulate(int years)
{
  while (years--)
    step();
}
```

- Solution: need to make step() a virtual function

```
class Population {
  … constructors and more …
  void simulate(int years); // the full simulation
  virtual void step(); // one year
}
```

How to deal with runtime errors?

- What should our integration library do if the user passes an illegal argument?
  - Return 0?
  - Return infinity?
  - Abort?
  - Set an error flag?
- Neither of these is ideal
  - Return values of 0 or infinity cannot be distinguished from good results
  - Aborting the program is no good idea for mission critical programs
  - Error flags are rarely checked by the users
- Solution
  - C++ exception handling

C++ Exceptions

- The solution are exceptions
- The library recognizes an error or other exceptional situation.
  - It does not know how to deal with it
  - Thus it *throws* an exception
- The calling program might be able to deal with the exception
  - It can *catch* the exception and do whatever is necessary
- If an exception is not caught
  - The program terminates

How to throw an exception

- What is an exception?
  - An object of any type
- Thrown using the `throw` keyword:
  - ```
    if(n<=0)
      throw "n too small";
    ```
  - ```
    if(index >= size())
      throw std::range_error("index");
    ```
- Throwing the exception
  - causes the normal execution to terminate
  - The call stack is unwound, the functions are exited, all local objects destroyed
  - Until a catch clause is found

The standard exception base class

- Is in the header `<exception>`
  - ```
    class exception {
    public:
      exception() throw();
      exception(const exception&) throw();
      exception& operator=(const exception&) throw();
      virtual ~exception() throw();
      virtual const char* what() const throw();
    };
    ```
- The function qualifier `throw()` indicates that these functions do not throw any exceptions
    ```

The standard exceptions

- are in `<stdexcept>`, all derived from `std::exception`
- Logic errors (base class `std::logic_error`)
  - `domain_error`: value outside the domain of the variable
  - `invalid_argument`: argument is invalid
  - `length_error`: size too big
  - `out_of_range`: argument has invalid value
- Runtime errors (base class `std::runtime_error`)
  - `range_error`: an invalid value occurred as part of a calculation
  - `overflow_error`: a value got too large
  - `underflow_error`: a value got too small
- All take a string as argument in the constructor

Catching exceptions

- Statements that might throw an exception are put into a `try` block
- After it `catch()` clauses can catch some or all exceptions
- Example:
  - ```
    int main()
    {
      try {
        std::cout << integrate(sin,0,10,1000);
      }
      catch (std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
      }
      catch(...) {// catch all other exceptions
        std::cerr << "A fatal error occurred.\n";
      }
    }
    ```
    ```

Exceptions example: main.C, simpson.h, simpson.C

```
int main() {

  bool done;
  do {
    done = true;
    try {
      double a,b;
      unsigned int n;
      std::cin >> a >> b >> n;
      std::cout << simpson(sin,a,b,n);
    }
    catch (std::range_error& e) {
      // also catches derived exceptions
      std::cerr << "Range error: " << e.what() << "\n";
      done=false;
    }
 // all other exceptions go uncaught
  } while (!done);
}
```

More exception details

- Exceptions and inheritance
  - A `catch(ExceptionType& t)` clause also catches exceptions derived from `ExceptionType`
- Rethrowing excpeptions
  - If a catch() clause decides it cannot deal with the exception it can re-throw it with `throw;`
- More details in text books
  - Uncaught exceptions
  - throw() qualifiers
  - Exceptions thrown while dealing with an exception
  - Exceptions in destructors
    - Can be very bad since the destructor is not called!

C++ review

- Stack class
  - procedural
  - modular
  - object oriented
  - generic

Procedural stack implementation: stack1.C

```
void push(double*& s, double v)
{
  *s++=v;
}

double pop(double *&s)
{
  return *--s;
}
```

```
int main() {

double stack[1000];
double* p=stack;

push(p,10.);

std::cout << pop(p) << “\n”;
std::cout << pop(p) << “\n”;
// error of popping below
// beginning goes undetected!
}
```

Modular stack implementation: stack2.C

```
namespace Stack {
struct stack {
  double* s;
  double* p;
  int n;};

void init(stack& s, int l) {
  s.s=new double[l];
  s.p=s;
  s.n=l;}

void destroy(stack& s) {
  delete[] s.s;
}
```

```
void push(stack& s, double v) {
  if (s.p==s.s+s.n-1) throw
    std::runtime_error("overflow");
  *s.p++=v;
}

double pop(stack& s) {
  if (s.p==s.s) throw
    std::runtime_error("underflow");
  return *--s.p;
}

int main() {
Stack::stack s;
Stack::init(s,100); // must be called
Stack::push(s,10.);
Stack::pop(s);
Stack::pop(s); // throws error
Stack::destroy(s); // must be called
}
```

Object oriented stack implementation: stack3.C

```
namespace Stack {
class stack {
  double* s;
  double* p;
  int n;
public:
  stack(int=1000); // like init
  ~stack(); // like destroy
  void push(double);
  double pop();
};
```

```
int main() {
  Stack::stack s(100);
  // initialization done automatically
  s.push(10.);
  std::cout << s.pop();
  // destruction done automatically
}
```

Generic stack implementation: stack4.C

```
namespace Stack {
template <class T>
class stack {
  T* s;
  T* p;
  int n;
public:
  stack(int=1000); // like init
 ~stack(); // like destroy
 void push(T);
  T pop();
};
```

```
int main() {
  Stack::stack<double> s(100);
 // works for any type!
 s.push(1.3);
  cout << s.pop();
}
```

Summary of Programming Styles

- Procedural implementation
  - possible in all languages
- Modular implementation
  - allows transparent change in underlying data structure without breaking the user's program. E.g. we can add range checks
- Object oriented implementation
  - additionally makes sure that initialization and cleanup functions are called whenever needed
- Generic implementation
  - works for any data type

Review of the numerical integration exercise

- The numerical integration exercise demonstrates all four programming styles:
  - 1st part: procedural programming
  - 2nd part: modular programming
    - We built a library
  - 3rd part generic programming
    - We uses templates
  - 4th part: object oriented programming
    - We derive from a base class
- After you have coded all four versions, perform benchmarks
  - Which version is fastest?
  - Which version is the most flexible?

Procedural programming

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

```
double func(double x) {return x*sin(x);}
cout << integrate(func,0,1,100);
```

- same as in C, Fortran, etc.

Generic programming

- ```
  template <class T, class F>
  T integrate(F f, T a, T b, unsigned int N)
  {
    T result=T(0);
    T x=a;
    T dx=(b-a)/N;
    for (unsigned int i=0; i<N; ++i, x+=dx)
      result +=f(x);
    return result*dx;
  }
  ```
- ```
  struct func {
    double operator()(double x) { return x*sin(x); }
  };
  cout << integrate(func(),0.,1.,100);
  ```
- allows inlining!
- works for any type T

Object oriented programming

- ```
  Class Integrator { // base class implements integration
    public:
      Integrator() {}
      double integrate(double a, double b, unsigned int n);
      virtual double f(double)=0;
  };
  ```
- ```
  class MyFunc : public Integrator { // derived class
  public:
    MyFunc() {}
    double f(double x) {return x*sin(x);} //implements function
  };
  ```
- ```
  MyFunc f;
  f.integrate(0,1,1000);
  ```
  ```