C++
Some Loose Ends

Laurent Michel
Computer Science & Engineering

Outline

- Some C++ Loose Ends
  - Nesting typedefs
  - Nesting classes
  - Randomization & the STL
  - Priority Queue & Functional Objects
  - Templates

Nesting

- You can nest class and types definition
- Purpose
  - Make the code more readable
  - Make the types more self-contained
- Examples?

Iterators!

```
#ifndef __RANGE_H
#define __RANGE_H

#include <tuple>

template <typename T>
class Range {
   T _from,_to;
public:
   class iterator {
      friend class Range<T>;
      T _i;
   protected:
      iterator(T start) : _i(start) {}
   public:
      T operator *() const  { return _i; }
      const iterator& operator++() { ++_i; return *this; } // pre-increment
      iterator operator ++(int) { iterator copy(*this); ++_i; return copy; } // post-increment
      bool operator ==(const iterator& other) const { return _i == other._i; }
      bool operator !=(const iterator& other) const { return _i != other._i; }
   };
   Range(T f,T t) : _from(f),_to(t) {}
   iterator begin() const { return iterator(_from); }
   iterator end() const   { return iterator(_to); }
};
```

Pointers

```
class Event {
protected:
   const int _id;
   double    _at;
   virtual std::ostream& print(std::ostream& os) const;
public:
   typedef std::shared_ptr<Event> Ptr;
   Event(int id) : _id(id) {}
   Event(int id,double t) : _id(id),_at(t) {}
   virtual ~Event() {}
   double when() const { return _at;}
   virtual void simulate(Simulator* sim) = 0;
   friend std::ostream& operator<<(std::ostream& os,const Event& evt);
};
```

Pointers

```
class Event {
protected:
   const int _id;
   double    _at;
   virtual std::ostream& print(std::ostream& os) const;
public:
   typedef std::shared_ptr<Event> Ptr;
   Event(int id) : _id(id) {}
   Event(int id,double t) : _id(id),_at(t) {}
   virtual ~Event() {}
   double when() const { return _at;}
   virtual void simulate(Simulator* sim) = 0;
   friend std::ostream& operator<<(std::ostream& os,const Event& evt);
};
```

Benefit?

```
std::shared_ptr<Event> e = std::shared_ptr<Event>(new StartEvent(...));
...
```

```
Event::Ptr e = Event::Ptr(new StartEvent(...));
...
```

Use in STL ?

- Sure!
  - For instance: The **`size_type`** type!

http://en.cppreference.com/w/cpp/container/vector

vector<T>

Member types

Member type	Definition
value_type	T
allocator_type	Allocator
size_type	Unsigned integral type (usually std::size_t)
difference_type	Signed integer type (usually std::ptrdiff_t)
reference	Allocator::reference (until C++11) value_type& (since C++11)
const_reference	Allocator::const_reference (until C++11) const value_type& (since C++11)
pointer	Allocator::pointer (until C++11) std::allocator_traits<Allocator>::pointer (since C++11)
const_pointer	Allocator::const_pointer (until C++11) std::allocator_traits<Allocator>::const_pointer (since C++11)
iterator	RandomAccessIterator
const_iterator	Constant random access iterator
reverse_iterator	std::reverse_iterator<iterator>
const_reverse_iterator	std::reverse_iterator<const_iterator>

Benefits?

- Better than using “int”
- Adapts automatically to new revisions of the STL
- size_type is adapted to the container (wide enough)
- Works well with auto

Randomization

- STL provides lots of classes for supporting randomization
  - Devices
  - Generators
  - Distributions

http://en.cppreference.com/w/cpp/numeric/random

Device

- A hardware source of real randomness.
  - Can be used to provide random values in a range [min..max]
  - Beware, only so much entropy in the source!
- Typical usage
  - Provide the *seed* to initialize a pseudo-random generator.

Generators

- Object responsible for creating random numbers
  - Generate them in a given range [min..max]
  - Relies on a specific algorithm
    - e.g., a congruence: $x_{i+1} = a * x_i + b \ mod \ c$
  - It creates a sequence $(x_0, x_1, x_2, x_3, \ldots)$ seeded at x_0.
  - The values generated are distributed uniformly in the range [min .. max]
  - You should **never** use the same generator for different purposes

Distributions

- We often need numbers that follow something other than uniform
- Easy to build yourself
- Example: Discrete
- In practice: Use the STL!

Priority Queues

- Also provided by the STL
- But the declaration is not “trivial”

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

What is this **Compare** type?

Functional Objects

- Remember closures?
- Closures are ***callable*** *objects* representing *functions*
  - Namely
    - they are structures
    - they support the function call operator
  - Anything like that is called a functional object.

Example

```
class Event {
protected:
   const int _id;
   double    _at;
   virtual std::ostream& print(std::ostream& os) const;
public:
   typedef std::shared_ptr<Event> Ptr;
   Event(int id) : _id(id) {}
   Event(int id,double t) : _id(id),_at(t) {}
   virtual ~Event() {}
   double when() const { return _at;}
};

struct CompareEvent {
   bool operator() (Event::Ptr a,Event::Ptr b) {
      return a->when() > b->when();
   }
};
```

Example

```
class Event {
protected:
    const int _id;
    double    _at;
    virtual std::ostream& print(std::ostream& os) const;
public:
    typedef std::shared_ptr<Event> Ptr;
    Event(int id) : _id(id) {}
    Event(int id,double t) : _id(id),_at(t) {}
    virtual ~Event() {}
    double when() const { return _at;}
};
```

```
struct CompareEvent {
    bool operator() (Event::Ptr a,Event::Ptr b) {
        return a->when() > b->when();
    }
};
```

Usage

```
std::priority_queue<Event::Ptr,std::vector<Event::Ptr>,CompareEvent> _queue;
```

- Provide:
  - The type of element
  - The type of the backing store
  - The functional object that implements the ordering

Templates

- **Beware**
  - Templates **provide** polymorphism
  - Templates **do not** use Inheritance
- Example
  - Relationship between
    - pair<A,B>
    - pair<C,D> ?

Templates are Great for…

- Generic containers
  - vectors, list, graphs…
- Generic algorithms
  - finding, searching, querying, printing…
- If you need sub-typing…
  - Templates are NOT the answer.
  - Use inheritance instead

Specialization

- You can instantiate template with types/values to specialize them
- Example
  - Factorial meta-programming…

Factorial?

```
template <int N> int fact() {
   return N * fact<N-1>();
}
template <> inline int fact<0>() {
   return 1;
}

template <int N> struct F {
   static const int value = N * F<N-1>::value;
};
template <> struct F<0> {
   static const int value = 1;
};

int main()
{
   int x = fact<5>();
   int y = F<5>::value;
   return x+y;
}
```