CS193D
Autumn 2000

Handout 20
October 26, 2000

Section Handout 5[1]

Kudos to Erik Neuenscwander for taking care of this week's handout.

STL Orthogonality

Orthogonality, a fancy word for a simple idea in the STL: independence. The STL was designed for freedom of expression. So in the design, containers, iterators, functions, and algorithms are all “orthogonal” concepts; you can take any container and use any algorithm to call any function on any range of the elements it contains. Freedom of Choice! It’s not cool unless you can use it, though, so let’s see some examples. (All code will be available at `~erikn/public/193d/sect5` (that’s a Unix directory, not a web address) once I get it cleaned up...)

We’re going to take the `WeatherReport` class from last section, store it in a map, then perform operations on it. Here’s the class (slimmed down to just its interface)

```
class WeatherReport {
public:
   // [... types removed]

  WeatherReport() {}
  WeatherReport(int nDays) : m_dailyTemps(nDays) {}

  void AddTemp(const DailyTemp& p) { m_dailyTemps.push_back(p); }

  temp_type GetHigh(unsigned day) const;  // get high temp for a given day
  temp_type GetLow(unsigned day) const;
  temp_type GetHigh() const;              // get high temp for the report
  temp_type GetLow() const;
  temp_type GetAvgHigh() const;
  size_type NumDays() const { return m_dailyTemps.size(); }

  static temp_type GetHigh(const DailyTemp& p) { return p.second; }
  static temp_type GetLow(const DailyTemp& p)  { return p.first; }

  void Print() const;
  static void PrintDay(const DailyTemp&);

  iterator begin()             { return m_dailyTemps.begin(); }
  iterator end()               { return m_dailyTemps.end(); }
  const_iterator begin() const { return m_dailyTemps.begin(); }
  const_iterator end() const   { return m_dailyTemps.end(); }

private:
  std::vector<DailyTemp> m_dailyTemps;
};
```

[1] There was no Section Handout 4 . We're titling the handout by week.

We’re going to implement `GetHigh`, `GetLow`, `GetAvgHigh` which I’m going to give out after section. And talk about a “LowestHigh”

```
//
// Helper functions
//
bool LessHighTemp(const WeatherReport::DailyTemp& p1,
                  const WeatherReport::DailyTemp& p2)
{
  return WeatherReport::GetHigh(p1) < WeatherReport::GetHigh(p2);
}

bool LessLowTemp(const WeatherReport::DailyTemp& p1,
                 const WeatherReport::DailyTemp& p2)
{
  return WeatherReport::GetLow(p1) < WeatherReport::GetLow(p2);
}

WeatherReport::temp_type AccumHighTemp(const WeatherReport::temp_type& t,
                                       const WeatherReport::DailyTemp& p)
{
  return t + WeatherReport::GetHigh(p);
}

//
// Interface functions
//
WeatherReport::temp_type WeatherReport::GetHigh() const
{
  return GetHigh(*max_element(m_dailyTemps.begin(), m_dailyTemps.end(),
                              LessHighTemp));
}

WeatherReport::temp_type WeatherReport::GetLow() const
{
  return GetLow(*min_element(m_dailyTemps.begin(), m_dailyTemps.end(),
                             LessLowTemp));
}

WeatherReport::temp_type WeatherReport::GetAvgHigh() const
{
  return accumulate(m_dailyTemps.begin(),
                    m_dailyTemps.end(),
                    WeatherReport::temp_type(0),
                    AccumHighTemp) / m_dailyTemps.size();
}

void PrintGoodCities(const vector<WeatherReport>& cities,
                     int goodHi, int goodLo)
{
   std::vector<WeatherReport>::const_iterator iter;

   for(iter = cities.begin(); iter != cities.end(); ++iter) {
      WeatherReport::const_iterator iterw;
      const WeatherReport& wr = *iter;
      WeatherReport::temp_type sum = WeatherReport::temp_type(0);

      for(iterw = wr.begin(); iterw != wr.end(); ++witer) {
         sum += GetHigh(*witer);
      }

      double avg = sum/wr.NumDays();

      if (avg < goodHi && avg > goodLo)
         wr.Print();
   }
}
```

```
void PrintGoodCities(const vector<WeatherReport>& cities,
                     int goodHi, int goodLo)
{
  std::vector<WeatherReport>::const_iterator iter;

  for(iter = cities.begin(); iter != cities.end(); ++iter) {
    const WeatherReport& wr = *iter;

    double avg = accumulate(wr.begin(), wr.end(),
               WeatherReport::temp_type(0)) / wr.NumDays();

    if (avg < goodHi && avg > goodLo)
      wr.Print();
  }
}
```

```
void PrintGoodCities(const vector<WeatherReport>& cities,
                     int goodHi, int goodLo)
{
  std::vector<WeatherReport>::const_iterator iter;

  for(iter = cities.begin(); iter != cities.end(); ++iter) {
    const WeatherReport& wr = *iter;

    double avg = wr.GetAvgHigh();
    if (avg < goodHi && avg > goodLo)
      wr.Print();
  }
}
```

☺ *YOU DO NOT HAVE TO UNDERSTAND THIS LAST ONE* ☺

```
void PrintGoodCities(vector<WeatherReport>& cities,
                     WeatherReport::temp_type goodLo,
                     WeatherReport::temp_type goodHi)
{
  for_each(cities.begin(), cities.end(),
           compose1(ptr_fun(PP),
           compose2(makepair(),
                    identity<WeatherReport>(),
                    compose2(logical_and<bool>(),
                             compose1(
                                      bind2nd(greater<
                                              WeatherReport::temp_type>(),
                                              goodLo),

mem_fun_ref(&WeatherReport::GetAvgHigh)),
                             compose1(

bind2nd(less<WeatherReport::temp_type>(),
                                              goodHi),
mem_fun_ref(&WeatherReport::GetAvgHigh))
                             ))));
}
//
// Helper code for the final (crazy) version of PrintGoodCities
//

#include "wr.h"
#include <algorithm>
#include <functional>

//
// Prototypes
//
void PP(const pair<WeatherReport, bool>& p);
void PrintGoodCities(vector<WeatherReport>& cities,
                     WeatherReport::temp_type goodLo,
                     WeatherReport::temp_type goodHi);

//
// makepair function object
//
class makepair : public unary_function<void, pair<WeatherReport, bool> >
{
public:
  pair<WeatherReport, bool> operator()(const WeatherReport& w, bool f)
const
    { return pair<WeatherReport, bool>(w, f); }
};

//////////////////////////////////////////////////////////////////////////
int main()
{
  vector<WeatherReport> cities;
  WeatherReport wr;

  wr.AddTemp(WeatherReport::DailyTemp(65, 95));
  cities.push_back(wr);
  PrintGoodCities(cities, 98.0, 100.0);
}

//
// WR functions
//
void PP(const pair<WeatherReport, bool>& p)
{
  if (p.second)
    p.first.Print();
}
```