# CISC 3142
# Programming Paradigms in C++

## Ch5 – A Tour of C++:

Concurrency and Utilities (selected topics)

(*Stroustrup – The C++ Programming Language, 4th Ed*)

# Small Utility Components - Time

- Facilities for dealing with time are found in sub-namespace std:chrono in <chrono>

```
using namespace std::chrono; // see §35.2
auto t0 = high_resolution_clock::now(); // returns time_point (a point in time)
do_work();
auto t1 = high_resolution_clock::now();
cout << duration_cast<milliseconds>(t1−t0).count() << "msec\n";
```

- Here t1-t0 is represented in nanoseconds. To convert it to milliseconds, use duration_cast<desired_unit_of_time>
- Rather than guessing “efficiency” of code, measure it in time

# pair (<utility>)

```
template<typename Forward_iterator, typename T, typename Compare>
  pair<Forward_iterator,Forward_iterator>
  equal_range(Forward_iterator first, Forward_iterator last, const T& val, Compare cmp);
```

- Given a **sorted** sequence [first;last), equal_range() returns a pair representing the subsequence that matches the predicate cmp
- A pair's data members, first and second could be heterogeneous. It provides =, ==, < if its elements do

```
auto rec_eq = [](const Record& r1, const Record& r2) { return r1.name<r2.name;}; // compare names
void f(const vector<Record>& v) { // assume that v is sorted on its "name" field          !cmp(a, b) && !cmp(b, a)
        auto er = equal_range(v.begin(), v.end(), Record{"Reg"}, rec_eq);
        for (auto p = er.first; p!=er.second; ++p) // print all equal records
                cout << *p; //assume that << is defined for Record
}
// make a pair
void f(vector<string>& v) {
        auto pp = make_pair(v.begin(), 2); // pp is a pair<vector<string>::iterator, int>
        // ...
}
```

# Random numbers (<random>)

- Useful for testing, games, simulation and security
- A random number generator consists of two parts:
  1. An *engine* that produces a sequence of random values
  2. A *distribution* that maps those values into a math distribution in a range

```
using my_engine = default_random_engine; // type of engine
using my_distribution = uniform_int_distribution<>; // type of distribution, defType: int
my_engine re {}; // the default engine, could pass a seed
my_distribution one_to_six {1,6}; // distribution that maps to the ints [1, 6]
auto die = bind(one_to_six, re); // make a generator: one_to_six(re)
int x = die(); // roll the die: x becomes a value in [1:6]
```

- This is equivalent to (note: bind requires <functional>):

```
auto die = bind(uniform_int_distribution<>{1,6}, default_random_engine{});
```