Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 21

- Compile-Time Computation
- Unrolling Loops
- STL Overview

Integer Template Parameters and Recursive Compile-Time Computations

```
template <int n> struct Fac     // general case n! = n*(n-1)!
{
  enum { value = n * Fac<n-1>::value }; // constant
};

template <> struct Fac<0>               // base case 0! = 1
{
  enum { value = 1 };                   // constant
};

int main()
{
  // Fac<c>::value is now a compile-time constant!
  cout << Fac<5>::value << endl;        // = 5! = 120
  cout << Fac<10>::value << endl;       // = 10! = 3628800
  cout << Fac<0>::value << endl;        // = 0! = 1
}
```


Using Metaprograms to Unroll Loops

```
template <typename T> inline T dot(int dim, T *a, T *b)
{
  T res = T(); // also works for basic types! (=0)
  for (int i=0; i < dim; ++i) res += a[i] * b[i];
  return res;
}

int main() {
  int a[3] = { 1, 2, 3 };
  int b[3] = { 4, 5, 6 };
  std::cout << dot(3, a, b) << endl; // 1*4+2*5+3*6=32
}
```

Problem: speed — unrolling the loop is faster!
A smart compiler can figure that out when dim is a constant. I.e.:

`dot(3, a, b)` $\leadsto$ `a[0]*b[0]+a[1]*b[1]+a[2]*b[2]`

Metaprogram Version

```
template <int DIM, typename T> struct Dot {
  static T result(T *a, T *b) {
    return *a * *b + Dot<DIM-1,T>::result(a+1,b+1);
  }
};

template <typename T> struct Dot<1,T> {
  static T result(T *a, T *b) { return *a * *b; }
};

template <int DIM, typename T> inline T dot(T *a, T *b)
{
  return Dot<DIM,T>::result(a, b);
}
```

If compiler inlines one-line functions `dot<3>(a,b)` generates `a[0]*b[0]+a[1]*b[1]+a[2]*b[2]`

Standard Template Library Overview

Container template classes

- Sequence containers
  - elements have predefined locations
  - `vector, slist, list, deque, string, ...`
- Associative containers
  - element location depends on key
  - `set, map, hash_set, hash_map, ...`

Algorithms

- Container independent
- `sort, find, merge, random_shuffle, ...`

Iterators

- pointer-like types used for traversing STL containers
- interface between algorithms and containers

Examples

```
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  // CONTAINERS + ALGORITHMS
  vector<int> v(10); // vector of 10 ints, fill with...
  generate(v.begin(), v.end(), rand); //... random values
  v[0] = 6; // access like array

  // loop through vector using ITERATORS
  vector<int>::iterator it = v.begin(), end = v.end();
  int sum=0;
  for (; it != end; ++it) sum += *it;

  // ALGORITHM: shuffle elements randomly
  random_shuffle(v.begin(), v.end());

  // MEMBER FUNCTIONS: if non-empty, erase first element
  if (!v.empty()) v.erase(v.begin());
}
```