Lecture 21

- Specializations cont.
- Template wizardry

Template Template and Default Parameters

```
template <class T, template<class> class cont=std::vector>
struct Foo {
  void push(const T &val) { co.push_back(val); }
  T back() { return co.back(); }
  void pop(){ co.pop_back(); }
  bool empty() const { return co.empty(); }

private:
  cont<T> co;
};

Foo<int, std::stack> x;    // uses std::stack
Foo<char> y;               // uses std::vector
```

- Uninstantiated template class as template parameter
- In this example, template parameter cont gets instantiated with template type parameter T
- Purpose: adapt class to special needs, here: users can choose underlaying LIFO data structure stack<T> or vector<T>

Integer Template Parameters and Recursive Compile-Time Computations

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

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

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


Application: Type Traits (Compile-Time Type Reflection)

```
template <typename T> struct TypeTraits
{
  template <typename U> struct PointerTraits // general case
  {
    enum { result = false };                // no pointer
  };

  template <typename U> struct PointerTraits<U*> //spec. case
  {
    enum { result = true };   // is pointer
  };

  enum { is_pointer = PointerTraits<T>::result };
  // enum { isReference = ... };
  // enum { isConst = ... };
  // enum { isStdArith = ... };  // integral or floating point
  // enum { isStdFundamental = ... };  // => bit-wise copy OK
  // ...
};
```

Compile-Time Assert

```
//   generating compile-time errors depending on compile-time
//   constant similar to assert
//   idea: only provide template instantiation for true
//         using false will generate compiler error msg.

template <bool x> struct ASSERTION_FAILURE;// incomplete type
template <> struct ASSERTION_FAILURE<true> // specialization
{ enum { value = 1 }; };

// fails if x is 0 or false
#define COMPILE_ASSERT(x) \
typedef char foo[ASSERTION_FAILURE<bool(x)>::value]

int main()
{
   // generates compile time error
   COMPILE_ASSERT(TypeTraits<char>::is_pointer);
   // OK
   COMPILE_ASSERT(TypeTraits<int*>::is_pointer);
}
```


Using Metaprograms to Unroll Loops

```
// compute dot product of two vectors of size n
template <typename T> inline T dot(int n, T *a, T *b)
{
   T res = T();  // for numeric types equal to 0
   for (int i=0; i < n; ++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; // 4+10+18=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) => a[0]*b[0]+a[1]*b[1]+a[2]*b[2]

Metaprogram Version

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

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

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

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