Lecture 20

- Template functions
  - syntax
  - instantiation
- Class templates
- Specializations

Template Function Definition

- Syntax:
  template < <type-param-list> >
  <type> <func-name>**(**<func-param-list>**)**
  - followed by **;** (**forward declaration**)
  - **{ ... }** (**function definition**)
  - <type-param-list> : sequence of comma-separated ‘**typename**/**class** <type-id>”pairs
  - type-ids can only occur once in the type-par-list
  - all type-ids must appear at least once as types in the prototype of the function
- Put template definitions in header files

More Examples

```
template <typename T> T max(T a, T b);
// OK, forward declaration

template <typename T> void swap(T &a, T &b);
// OK, forward declaration

template <class U, typename V> U foo(U a, V b) {
  return a;
}
// OK, class/typename are synonyms
// U,V appear in prototype

template <typename U, V> U bar(V a);
// ERROR: no typename/class in front of V
// PROBLEM: when calling bar(x) compiler cannot
// infer type U =>  need to say bar<U>(a)
```

Function Template Instantiation

- Function templates specify how individual functions can be constructed given a set of actual types (**Instantiation**)
- This happens as **side-effect** of either **invoking** or **taking the address** of a template function
- Compiler and/or the linker has to remove multiple equal instantiations
- Template instantiation may be slow - dumb compilers repeat compilation

Type Parameter Binding

```
template <typename T> T max(const T *a, int size) { }

float *a[100], x; ... x = max(a, 100);

formal param.: const T *a    -> T *a
actual param.: float *a   => T = float
```

1. Each formal argument of the template function is examined for the presence of formal type parameters
2. If a formal type parameter is found, the type of the corresponding actual argument is determined
3. The types of the formal and actual argument are matched. Type qualifiers are ignored.
   No non-trivial type conversions take place. Safer -> Good!

Which function is called?

1. Examine all non-template instances
   - exactly one? -> **found, OK**
   - more than one? -> **ambiguous, ERROR**
2. Examine all template instances of the function
   - exactly one? -> **found, OK**
   - more than one? -> **ambiguous, ERROR**
3. Re-examine non-template instances now allowing type conversions

Quicksort revisited: vanilla template version

```
template <typename T>
void qsort(T *a, int l, int r) { // sort a[l..r]
   if (l < r) {
      int i=l-1, j=r;
      const T &v = a[r];
      for (;;) {            // partition: < v | >= v
         while (a[++i] < v);
         while (v < a[--j]) if (j <= l) break; // (*)
         if (i >= j) break;
         swap(a[i], a[j]);  // exchange misplaced elems.
      }
      swap(a[i], a[r]);
      qsort(a, l, i-1);  // recursively sort part 1
      qsort(a, i+1, r);  // recursively sort part 2
   }
}
int    ia[6] = { 1, 3, 5, 2, 8, 0 };
double da[6] = { 1.5, 0.5, 3.4, 5.2 };
qsort(ia, 2, 4);   // T=int, sort elements ia[2..4]
qsort(da, 0, 5);   // T=double, sort elements da[0..5]
(*) : if (j <= l) not necessary in Median-of-3 quicksort
```

Class Templates Overview

- Reuse type independent code
- Classes are parameterized by types
- Very useful for container types (vector, set, list...)
- Advanced application: compile time computations!
- Simplest syntax:
  - template <typename T> class X {
      ... T can be used here ...
    };
- Instantiation is explicit: E.g. vector<int> a;
- Compiler instantiates classes on demand

More Examples

```
template <class T> class
Stack {
public:
   Stack();
   ~Stack();

   void push(T v);
   T pop();
   bool empty();

private:
   ...
};

Stack<int> si;
Stack<float> sf;

si.push(5);
cout << si.pop();
if (si.empty()) exit(0);

sf.push(3.5); ...
```

```
template <class T> class Vector
{
public:

   Vector(int size);
   ~Vector();

   T &operator[](int i);
   const T &operator[](int i)
const;

private:
   int size;
   T *p;
};

Vector<int> vi(10);
Vector<char*> vs(20);

vi[0] = 10;
vs[1] = “text”;
```

Pair Class Template

```
template <typename T> class Pair {
public:

   Pair(T v1, T v2);
   void set_first(T v)  { first = v; }
   void set_second(T v) { second = v; }
   T get_first()  const { return first; }
   T get_second() const { return second;}
private:
   T first, second;
};

template <typename T> Pair<T>::Pair(T v1, T v2)
{
  first = v1; second = v2;
}
```

```
Pair<int> pi(0,0);
Pair<float> pf(0.0,2);
Pair<Pair<int> > pp(Pair<int>(0,2), Pair<int>(3,1));
// Note: > > (not >>, which produces an error
message)
pi = pp.get_first();
```

Full Template Specialization

```
template <typename T> class X { ... };
(A)

template <> class X<bool> { ... };   (B)

template <> class X<int> { ... };    (C)

X<float> x;  // instantiates (A)
X<bool> x;   // instantiates (B)
X<int> x;    // instantiates (C)
```

- Adapts class templates to special needs
  -> different code for different types

Partial Template Specialization

```
template <typename T> class List { ... };
(A)

template <typename T> class List<T*> {
(B)
  List<void*> impl; ... // infinite recursion?!
};

template <> class List<void*> { ... };
(C)

List<int> x;     // instantiates (A)
List<int**> x;   // instantiates (B)
List<void*> x;   // instantiates (C)
```

- Adapt class templates even more
- Compiler chooses the most specialized match
- Details for instance in book “C++ Templates”