Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 19

- Operator Overloading
- Generic Programming
- Template Functions

Prefix/Postfix Mysteries

```
class Complex {
public:
  ...
  Complex &operator++();    // prefix ++
  Complex operator++(int);  // postfix ++
};

// postfix: cannot return a reference to the variable
// because the returned value is different

Complex Complex::operator++(int) {
  Complex temp(*this);
  ++re;
  return temp; // returns the previous contents
}

// with prefix++ returning a reference is OK
// variable is changed and reference is returned

Complex &Complex::operator++() {
  ++re;
  return *this; // prefix operators are more efficient!
}
```


Part 4: Generic Programming

Code is often independent of actual types. E.g.

- Sorting routines (qsort)
- Containers (vectors, lists, sets)

Generic programming:
implement once and reuse code for arbitrary types

Benefit: code is easy to maintain!

C way: use `void*` as generic pointer type and pass function pointers

C++ way: template functions, class templates, and functors

Template Functions

```
int min(int a, int b) { return a < b ? a : b; }

float min(float a, float b) { return a < b ? a : b; }

...

No need for defining long list of identical functions!
The following generic definition covers (almost) all:

template <typename T> T min(T a, T b) {
  return a < b ? a : b;
}

Function min is now paramerized by type T
```

Compiler generates implementations for actual type instances when function is used

Example

```
template <typename T> T max(T a, T b)
{
  return a > b ? a : b;
}

template <typename T> void swap(T &a, T &b)
{
  T temp = a; a = b; b = temp;
}

int main()
{
  int a=10, b=5, c = min(a,b);       // min<int,int> called
  float e=2.0, f=1.0, g = min(e,f); // min<float,float>

  swap(a,b); // swap<int,int>
  swap(e,f); // swap<float,float>
}
```


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 function parameter list

Template definitions belong 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 identical 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 $\rightsquigarrow$ Good!

Which function is called?

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

Selection sort revisited: template version

```
template <typename T>
void sort(T *a, int n) { // sort a[0..n-1]
  for (int i=0; i < n; ++i) {
    int i_min = i;
    for (int j=i+1; j < n; ++j) {
      if (a[j] < a[i_min]) i_min = j;
    }
    if (i_min != i) swap(a[i], a[i_min));
  }
}

int    ia[6] = { 1, 3, 5, 2, 8, 0 };
sort(ia, sizeof(ia)/sizeof(ia[0]));  // T=int

double da[6] = { 1.5, 0.5, 3.4, 5.2 };
sort(da, sizeof(da)/sizeof(da[0]));  // T=double
```