Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 20

- Class Templates
- Template Specialization
- Type Traits
- Compile-Time Reflection

Template Functions vs. Macros

- Template functions are type-aware, macros are not
- Template functions can be specialized easily, macros cannot
- Inlined template functions are as fast as macro-generated code
- No "unwanted" side-effects with template functions

$\rightsquigarrow$ In C++ macros are not that useful anymore

Class Templates Overview

- Reuse type independent code
- Classes are parameterized by types
- 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: formal type parameters are replaced by actual parameters
- Very useful for container types (vector, set, list...)
- Advanced application: compile time type reflection and computation

Examples

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

  void push_back(T &v); // append
  T    &back();    // last element
  void pop_back(); // remove last
  bool empty() const;

private:
  ...
};

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

si.push_back(5);
cout << si.back();
if (si.empty()) exit(0);

sf.push_back(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> struct Pair
{
  Pair(T v1, T v2);
  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.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 $\rightsquigarrow$ 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
- More details in "C++ Templates" by Vandevoorde and Josuttis

Application: Type Traits (Compile-Time Type Reflection)

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

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

  enum { is_pointer = PointerTraits<T>::value };
  enum { is_reference = ... };
  enum { is_const = ... };
  enum { has_trivial_assign = ... }; // <=> bit-wise copy OK
  ...
};
```

Example: Fast Array Copy Function

Fast memcpy function can be used for types that can be copied bit-wise. Otherwise, iterate assignments.

Approach: class template that stores a flag telling the "smart" copy routine when to use memcpy

```
template <bool U> struct CopyImpl {  // default (slow)
  template <typename T> static void copy(T *dst, T *src, int n)
  { for (int i=0; i < n; ++i) dst[i] = src[i]; }
};

template <> struct CopyImpl<true> {  // specialization (fast)
  template <typename T> static void copy(T *dst, T *src, int n)
  { memcpy(dst, src, n*sizeof(T)); }
};

template <typename T> void copy(T *dst, T *src, int n)
{
  CopyImpl<has_triv_asgn<T>::value>::copy(dst, src, n);
}
```



```
// safe default: no trivial assignment
template <class T> struct has_triv_asgn { enum { value=0 }; };

// basic types can be assigned trivially
template <> struct has_triv_asgn<char> { enum { value=1 }; };
...
template <> struct has_triv_asgn<double> { enum { value=1 };};

// pointers can be assigned trivially
template <> struct has_triv_asgn<T*> { enum { value=1 }; };

struct Foo {      // can be assigned trivially
  int a, b;
};
template <> struct has_triv_asgn<Foo> { enum { value=1 }; };

struct Bar {      // can't be assigned trivially
  ... int *p;     // non-shared memory
};
template <> struct has_triv_asgn<Bar> { enum { value=0 }; };
```


Call Examples

```
const int N = 1000;

Foo a[N];
Foo b[N];
copy(b, a, N);  // uses memcpy

double a[N];
double b[N];
copy(b, a, N);  // also uses memcpy

Bar a[N];
Bar b[N];
copy(b, a, N);  // uses Bar assignment operator
```


Boost: A Collection of C++ Libraries

Testing ground for C++ libraries available at `boost.org`

Community proposes/reviews/improves libraries

Large variety of useful libraries:

> regular expressions, formatted output, type traits, portable threading, random number generators, ...

Some of them like `shared_ptr` and `type_traits` already made it into the new C++ standard (tr1)

Local copy is located at `~c201/public/boost`

Use g++ option `-I ~c201/public/boost` to access

Boost Type Traits Example

```
#include "boost/type_traits.hpp"
#include <iostream>
using namespace std;

struct X { virtual ~X() {} };

int main()
{
  cout << boost::is_pointer<int*>::value << endl;
  cout << boost::has_trivial_assign<int*>::value << endl;

  cout << boost::is_polymorphic<X>::value << endl;
  cout << boost::has_trivial_assign<X>::value << endl;
}
```