Templates

Object-Oriented Programming with C++

Function overloading

- Same function name with different argument-lists.

```
void print(char * str, int width); // #1
void print(double d, int width); // #2
void print(long l, int width); // #3
void print(int i, int width); // #4
void print(char *str); // #5

print("Pancakes", 15);
print("Syrup");
print(1999.0, 10);
print(1999, 12);
print(1999L, 15);
```

Function overloading

- Same function name with different argument-lists.

```
void print(char * str, int width); // #1
void print(double d, int width); // #2
void print(long l, int width); // #3
void print(int i, int width); // #4
void print(char *str); // #5

print("Pancakes", 15); // #1
print("Syrup");        // #5
print(1999.0, 10);     // #2
print(1999, 12);       // #4
print(1999L, 15);      // #3
```

Overload and auto-cast

```
void f(int i);
void f(double d);

f('a');
f(2);
f(2L);
f(3.2f);
```

Default arguments

- A *default argument* is a value given in the *declaration* that the compiler automatically inserts if you don't provide a value in the function call.

```
Stash(int size, int initQuantity = 0);
```

Default arguments

- To define a function with an argument list, default arguments must be added from right to left.

```
int harpo(int n, int m = 4, int j = 5);
int chico(int n, int m = 6, int j);  //illegal
int groucho(int k = 1, int m = 2, int n = 3);

beeps = harpo(2);
beeps = harpo(1,8);
beeps = harpo(8,7,6);
```

Why templates?

- Suppose you need a list of `X` and a list of `Y`
  - The lists would use similar code
  - They differ by the type stored in the list

Why templates?

- Choices
  - Clone code
    - preserves type-safety
    - hard to manage
  - Make a common base class
    - May not be desirable
  - Untyped lists
    - type unsafe

Templates

- Reuse source code
  - *generic* programming
  - use types as *parameters* in class or function definitions

Templates

- Function Template
  - Example: `sort` function
- Class Template
  - Example: containers -- `stack`, `list`, `queue`, ...
    - stack operations are *independent* of the type of items in the stack
  - template member functions

Function templates

- Similar operations on different types of data.
- Swap function for two int arguments:

```
void swap ( int& x, int& y ) {
  int temp = x;
  x = y;
  y = temp;
}
```

- What if we want to swap `floats`, `strings`, `Currency`, `Person`?

Example: `swap` function templates

```
template < class T >
void swap( T& x, T& y ){
    T temp = x;
    x = y;
    y = temp;
}
```

- The `template` keyword introduces the template
- The `class T` specifies a parameterized type name
  - class means any built-in type or UDT
- Inside the template, use T as a type name

Function templates syntax

- Type parameters represent:
  - types of arguments to the function
  - return type of the function
  - define variables within the function

Template instantiation

- Generating a definition from a *template class/function* and *template arguments*:
    - Types are substituted into template
    - New body of function or class definition is created
        - syntax errors, type checking
    - Specialization -- a version of a template for a particular argument(s)

Example: using swap

```
int i = 3; int j = 4;
swap(i, j);  // use explicit int swap

float k = 4.5; float m = 3.7;
swap(k, m);  // instantiate float swap

std::string s("Hello");
std::string t("World");
swap(s, t);  // instantiate std::string swap
```

- A *template function* is an instantiation of a function template

Template argument deduction

- Only *exact* match on types is used
- *No* conversion operations are applied

```
swap(int, int); // ok
swap(double, double); // ok
swap(int, double); // error!
```

- Even implicit conversions are ignored
- Template functions and regular functions coexist

Overloading rules

- Check first for unique function match
- Then check for unique function template match
- Then implicit conversions on regular functions

```
void f(float i, float k) { /*...*/ };
template <class T> void f(T t, T u) { /*...*/ };

f(1.0f, 2.0f);
f(1.0, 2.0);
f(1, 2);
f(1, 2.0);
```

Function instantiation

- The compiler *deduces* the template type from the actual arguments passed into the function.
- Types can also be *explicitly* provided:
  - For example, the parameter may not be in the function signature

```
template <class T>
void foo() { /* ... */ }

foo<int>();    // type T is int
foo<float>();  // type T is float
```

Class templates

- Classes parameterized by types
  - Abstract operations from the types being operated upon
  - Define potentially *infinite set of classes*
  - Another step towards reuse!

Class templates

- Typical use: container classes
  - `stack <int>`
    - is a stack that is parameterized over int
  - `list <Person*>`
  - `queue <Job>`

Example: Vector

```
template <class T>
class Vector {
public:
  Vector(int);
  ~Vector();
  Vector(const Vector&);
  Vector& operator=(const Vector&);
  T& operator[](int);
private:
  T* m_elements;
  int m_size;
}
```

Usage

```
Vector<int> v1(100);
Vector<Complex> v2(256);

v1[20] = 10;
v2[20] = v1[20];  // ok if int=>Complex is defined
```

Vector members

```
template <class T>
Vector<T>::Vector(int size): m_size(size) {
  m_elements = new T[m_size];
}

template <class T>
T& Vector<T>::operator[](int index) {
  if(index < m_size && index >= 0) {
    return m_elements[index];
  } else {
    /*...*/
  }
}
```

A simple sort function

```
// bubble sort - don't use it!
template <class T>
void sort(Vector<T>& arr) {
  const size_t last = arr.size() - 1;
  for (int i = 0; i < last; ++i)
  for (int j = last; j > i; --j) {
    if (arr[j] < arr[j-1]) {
      // which swap?
      swap(arr[j], arr[j-1]);
    }
  }
}
```

Sorting the Vector

```
Vector<int> vi(4);
vi[0] = 4; vi[1] = 3; vi[2] = 7; vi[3] = 1;
sort(vi);  // sort(Vector<int>&)

Vector<string> vs(5);
vs[0] = "Fred";
vs[1] = "Wilma";
vs[2] = "Barney";
vs[3] = "Dino";
vs[4] = "Prince";
sort(vs);  // sort(Vector<string>&);
// NOTE: sort use operator< for comparison
```

Templates

- Templates can use multiple types

```
template < class Key, class Value >
class HashTable {
  const Value& lookup (const Key&) const;
  void insert (const Key&, const Value&);
  /* ... */
}
```

Templates

- Templates can be nested – they're just new types!

```
Vector< Vector<double*> >
```

- Type arguments can be complicated

```
Vector< int (*) (Vector<double>&, int) >
```

Expression parameters

- Template arguments can be *constant* expressions
- Non-Type parameters
  - can have a default argument

```
template <class T, int bounds = 100>
class FixedVector {
public:
  FixedVector();
  T& operator[](int);
private:
  T elements[bounds];  // fixed-size array!
}
```

Non-Type parameters

```
template <class T, int bounds>
T& FixedVector<T, bounds>::operator[] (int i) {
  return elements[i];  // no error checking
}
```

Non-type parameters

- Usage

```
FixedVector<int, 50> v1;
FixedVector<int, 10*5> v2;
FixedVector<int> v3;  // uses default
```

Non-type parameters

- Summary
  - Embedding sizes not necessarily a good idea
  - Can make code faster
  - Makes code more complicated
    - size argument appears everywhere!
  - Can lead to (even more) code bloat

Member templates

- Template declarations can appear inside a member function of any class.
- A member-template constructor in `std::complex`:

```
template<typename T> class complex
{
 public:
  template<class X> complex(const complex<X>&);

 /* ... */
};
```

Templates and inheritance

- Templates can inherit from non-template classes

```
template <class A>
class Derived : public Base { /* ... */ }
```

Templates and inheritance

- Templates can inherit from class templates

```
template <class A>
class Derived : public List<A> { /* ... */ }
```

Templates and inheritance

- Non-template classes can inherit from instantiated template classes

```
class SupervisorGroup : public
  List<Employee*> { /* ... */ }
```

Recurring template pattern

- General form

```
// The Curiously Recurring Template Pattern (CRTP)
template <class T>
class Base
{
  /* ... */
};

class Derived : public Base<Derived>
{
  /* ... */
};
```

Recurring template pattern

- Simulate virtual function in generic programming

```
template <class T>
struct Base {
  void interface() {  // normal
    static_cast<T*>(this)->implementation();
  }
  static void static_func() {  // static
    T::static_sub_func();
  }
};

struct Derived : public Base<Derived> {
  void implementation();
  static void static_sub_func();
};
```

Morality

- In general, put the definition and the declaration for templates in the *header file*.
    - won't allocate storage for the function/class at that point
    - compiler/linker has mechanism for removing multiple definitions

Writing templates

- Get a non-template version working first
- Establish a good set of test cases
- Measure performance and tune
- Review implementation
  - Which types should be parameterized?
- Convert the non-parameterized version into a template
- Test against the established test cases