C++

Template functions and classes

Introduction

- Templates provide the backbone for generic programming in C++
- The flexibility provided by template is resolved at compile time providing efficiency
- C++ supports two kinds of templates: function template and class template

Function templates

- Suppose we have a set of functions which all look similar for a given type T:
  T max (const T& a, const T& b) { … }
- We can define all these functions in C++ with one function by using templates:
  template <typename T> T max (const T&a, const T& b) {
  return a > b ? a : b;
  }

Function templates

- The keyword **typename** in the previous example can be exchanged with the keyword **class**:
  template <class T> T max (const T& a, const T& b) {
  return a > b ? a : b;
  }
- This definition and the previous definition are equivalent
- Template definitions are not functions; a function template can not be called
- Instead function template are **instantiated**

Function template instantiation

- Functions are created by the compiler using the provided template
- Example:
  int a=1, b=2;
  max(a, b);
  string c=“template”, d=“function”;
  max(c, d);
- The compiler will create two functions: one for type “int” and one for the type “string” using the template function provided for “max”:
  template <typename T> T max(const T& a, const T& b);
- This is called **template instantiation**

Function template instantiation

- Template instantiation: the compiler creates functions (instances) from the template for each type it encounters
- The parameter T in the template definition is called the **formal parameter** or **formal argument**
- In the previous example, max is instantiated with the actual arguments “int” and “string”

Formal argument matching

- When the compiler instantiates a template, it determines the template parameter from the types of the actual arguments:
  max(2, 3); // T is int
  max(2.0, 3.0); // T is double
- Automatically inferring the type of a variable is called **type inference**
- Type inference is originally coming from functional languages like ML (OCaml) or Haskell

Formal argument matching

- There is no automatic type conversion in argument matching
- The following code will not compile:
  max(1, 2.0);
- It is possible to force the instantiation.
  - This example will force the compiler to instantiate the template for the type double:
    max<double>(1, 2.0);

Template function: more than one formal argument

- It is possible to have several formal arguments in the template definition (as in the example here)
- It can lead to tricky problem sometimes

```
template <typename T1, typename T2>
T1 max(const T1& a, const T2& b)
{
return a > b ? a : b;
}


// …

cout << ::max(1, 1.5); // T1: int, T2:
double
cout << ::max(1.5, 1); // T1: double T2: int

// note that: max(1, 1.5) != max(1.5, 1) !
```

Template function: size of created code

- Template function can lead to code bloat
- Example:
  int a = 1; double x = 2.0;
  max(a,a); max(a,x); max(x,a); max(x,x);
- The compiler will generate 4 functions in the above case (no code share)
- For 4 var. of different types and all possible combinations, $4^2 = 16$ functions created

Class templates

- The concept of template works similarly for classes
  - Except that the actual types have to be explicitly provided by the programmer
- Example of a generic list class:

```
template <class T>
class List {
public:
// append t to the list
void push_back(const T& t);
};

template<class T>
void List<T>::push_back(const T& t)
{ //… }

int main() {
List<int> l; // instantiates the list for int
l.push_back(1);
}
```

Class templates – other features

- Class templates can have built-in integral types as template parameters
- Example:

```
template <int dim>
class Point {
double coordinates[dim]; // cord array
// …
};

int main () {
Point<3> p; // a point in 3D
}
```

Class template and default argument

- For class templates, template arguments can have default values
- Example: a stack class built by default from a previously defined list

```
template <class T, class Container = List<T> >
class Stack {
private:
Container c;
public:
//…
};

// …
Stack<int> s; // will use List<int> as a container
Stack<int, Array<int> >; // will use Array<int> as a container
```

Differences between class and function templates

- For function templates, template arguments are inferred by the compiler at compile-time:
  max(1.0, 2.0); // the compiler will infer that T is double, and create a function max with double
- For class templates, the actual template arguments need to be explicitly specified:
  List l; // compile-time error
  List<int> l; // ok

Function templates and type inference

- There are some cases where the compiler can not infer the actual template arguments
- In these cases, the type should be explicitly specified
- Example:

```
template <typename T> T f() { //… }
template <typename T> void g() {
T a;
// …
}
```

- How to use:

```
f<int>();
g<double>();
```

- If the compiler can not infer the type, it has to be explicitly indicated

Location of template functions and classes

- Template functions and classes code and definitions should be put in header files
- These header files should be included in every file where you use an instantiation of the template

```
//…
template <class T>
void
swap(T&a, T&b)
{
T tmp = b;
b = a;
a = tmp;
}
```

utils.h

```
// …
#include <utils.h>
int main ()
{
int a=1, b=2;
swap(a,b);
}
```

main.cpp

Local types and typename

- The keyword **typename** can also be used to indicate to the compiler that a token is a type
- This is needed for example when defining template classes that uses a container and types defined inside the container

Example

```
template <class T>
class List {
typedef T value_type;
// …
};

int main() {
List<int> l;
List<int>::value_type el; // el is of type int
}
```

Example

- Suppose we define a Stack using a Container such as for example the List<T> defined before
- Suppose we need the value type T of the container
- How can we do it ?

```
template <class Container>
class Stack {
private:
Container::value_type el;
// …
};
```

It does **not work**:
the compiler has no way to figure out that value_type is a type.
By default the compiler will assume it is a member of Container

Example fixed

- Solution: indicate to the compiler that value_type is a type, by using the keyword **typename**

```
template <class Container>
class Stack {
private:
typename Container::value_type el;
// …
};
```

Example of template class

```
template <class T>
Vector_3 {
private:
T _x, _y, _z;
public:
Vector_3() : _x(0), _y(0), _z(0) {}
Vector_3(const T& x, const T& y, const T& z): _x(x), _y(y), _z(z) {}
T x() { return _x; }
T y() { return _y; }
T z() { return _z; }
T norm() { return sqrt(_x*_x + _y*_y + _z*_z); }
// …..
};

template <class T>
Vector_3<T> operator+(const Vector_3<T>& v1, const Vector_3<T> & v2)
{ return Vector_3<T>(v1.x()+v2.x(), v1.y()+v2.y(), v1.z()+v2.z()); }
```