SAPIENZA
UNIVERSITÀ DI ROMA

Robot Programming
C++ Templates

Giorgio Grisetti

Function Objects

```
float timesTree(float f){
  return 3*f;
}


…
float v=3;
v=timesThree(v);
```

```
struct TimesThree {
  int operator()(float f) {
    return 3*f;
  }
};
…
float v=3;
TimesThree timesThree; // declare object
v=timesThree(v);
```

By defining the () operator, one can construct an object that behaves syntactically similar to a function. Function Objects can be passed as parameters, or at type parameters.

Function Objects

```
struct compareInt {
  bool operator()
   (const int a, const int b) const
  {
    return a<b;
  }
};
```

```
struct compareFloat() {
  bool operator()
   (const float a, const float b)
const
  {
    return a<b;
  }
};
```

Function objects are typically used to pass “traits” about types to algorithms, think about “compare” in sort.

Generic Programming

Well known algorithms and data structures to be adapted for a compiled language require a specific instantiation that depends

- on the data types
- on algorithms

Examples:

List/Stack/Heap/Tree of (integers, float, string, classes)

Sorting algorithm (container to be sorted, comparison operator)

Matrices/Vectors (float, double)

Generic Programming

Metaprogramming is a technique that minimizes the code replication by generating stubs (templates) that get instantiated to specific instances at **compile time.**

If properly used, instantiation at compile time allows for hard optimizations (loop unrolling, simd extensions etc), that provide significant runtime improvements.

Templates

Overall sintax:

add

`template <Arguments>`

before a class or a function.

Inside the function/class, use the arguments as if they were going to be substituted (in fact they are, at compile time).

`<Arguments>` can be either “typenames” or variables “specific types”

Function Objects and Templates

```
template <typename T>
struct DefaultCompare_ {
  bool operator()
   (const T& a, const T& b)
  {
    return a<b;
  }
};
```

```
DefaultCompare_<int> compInt;

DefaultCompare_<float> compFloat;

DefaultCompare<double> compDouble;

cerr << compInt(3,4);
cerr << compFloat(0.1, 0.2);
```

We can define a generic compare class for types that support the < operator. Let’s call it DefaultCompare.

Templates, Type and Value arguments

```
struct Vec3f {
  static const int size=3;
  float values[size];
  float& at(int pos){
    return values[pos];
  }
};

struct Vec4d {
  static const int size=4;
  double values[size];
  double& at(int pos){
    return values[pos];
  }
};
Vec3f v3_1, v3_2;
Vec3f v4_1;
```

```
template <typename T, int s>
struct VecT_ {
  static const int size=s;
  T values[size];

  ValueType& at(int pos){
    return values[pos];
  }
};
…
VecT_<float, 3> v3_1, v_3_2;
VecT_<double, 4> v4_1;
```

with templates we
can define the stuff once
and use it many times

Template functions

Templates can be also be used in functions, to make them parametric w.r.t the data type

A generic min function, supporting all types that define a comparison operator is the following

```
template <typename T>

const T& myMinTemplate(const T&a,
const T&b) {

  if (a<b) return a;

  return b;

}


int main(){

  cout << myMinTemplate(3,4) <<
endl;

  cout << myMinTemplate(4.5,3.8) <<
endl;

}
```

00_template_functions

Templates: recursion

Nasty things can be done with partial specialization. Factorial is evaluated at compile time!

```
template <int i>
inline int factorial_() {
  return i*factorial_<i-1>();
}
template<>
inline int factorial_<0>() {
  return 1;
}
int main(){
  cout << factorial_<5>() << endl;
}
```

Using and Constexpr

using defines a type alias; It helps the eyes when dealing with long template expressions.

```
using <name> = <type>;
using MyInt= long int;
using V3=Vec_<float, 3>;
MyInt i; // equivalent to long_int i;
V3 v;    //equivalent to Vec_<float, 3>;
```

constexpr serves to specify that an expression is a compile time constant and forces its evaluation once. This helps the compiler and the template system.

Templates, a more complex example

Let’s rework our binary_tree class to be parametric w.r.t the type contained in the tree.

In the algorithms inside the class, the only type dependant operation is the comparison.

Not all types define a < operator. E.G. your classes.

Solution we can use a “Compare” function object to rewrite our class.

In the remainder we will put the two implementations (int and generic), side by side.

`01_binary_tree/{binary_tree_int,binary_tree_template}`

Templates, type members

By the **using** keyword you can define nested types of a template object, that presumably depend on the template parameters

You can also render the syntax much more readable

In the following example we will rework our binary tree to exploit using, and we will declare a static variable for the const compare object. This is relevant, since the compare does not need to be either stored in each instance of the tree node and never changes.

Static variables should be initialized outside the class/struct scope!

01_binary_tree/{binary_tree_int,binary_tree_template}

Templates: Vector Class

We show a step by step example on how to make a vector class parametric w.r.t. the field, and the dimension.

Each vector with a specific dimension and field will be an own type.

We will drop the heap allocation, and the dynamic memory allocation.

02_mat_and_vec/{static_vec,static_vec_test}

Templates: Matrix Class

Similar to the vectors, we will show how through templates we can construct a static fixed size matrix class, that supports traditional operators.

If the sizes of the matrix are defining its “class”, some operations such as transposition or multiplication might lead to new types.

Using the matching rules we will achieve this.

`02_mat_and_vec/{static_mat,static_mat_test}`

Templates (declaration/definition)

Similar to the “plain c++” also template functions can be defined outside the body of a class. The mechanism is the same, and the template clause should be repeated before each method definition

Examples and Exercises

- Profile the same matrix expression

  X=C*(A*B+v*v.transpose())

  With A=3*2, b=2*3, C=3*3, v=3*1

  executing the calculation 1000000 times, with and without optimizations in the static and dynamic cases

- Write a class MyItem {float f, double d, string s} with a special comparison operator

  i1<i2 if (i1.f<i2.f) || (i1.f==i2.f && i1.d<i2.d) || (i1.f==i2.f && i1.d==i2.d && i1.s<i2.s)

  instantiate a binary tree operating on it
- Add a “Remove” operation to the (template) binary tree class.