Meta-programming

Ettore Speziale

Politecnico di Milano

Contents

Contents

What is Meta-programming?

Meta-programming is the writing of programs that:

- write or manipulate other programs (or themselves) as their data

Basically it is a technique:

- many languages supports meta-programming

C++ is a pragmatic:

- meta-programming is a cool stuff
- what I can do with it?

Motivating Example
Simple Stack

Showing code on slides is both boring and error-prone, so I will use as much as possible vi and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.

"Talk is cheap, show me the code" [3]

A Java-to-C++ translation table is available at slides end.

Please refer to:

- `cpp-stack.cpp`
- `cpp-no-template-stack.cpp`

Simple Stack
Apply C-programmer Tricks

The basic stack implementation – `cpp-stack.cpp`:

- uses **void** $*$ to abstract from stack type
- stores elements out-of-place

The macro version – `cpp-no-template-stack.cpp`:

- exploit macros to generate a piece of program
- allows storing elements in-place

Simple Stack
Out-of-place vs In-place Layout

Let Foo be a **class**, and let Bar be a type:

- an instance of Bar is part of Foo state: how can it be represented?

out-of-place [1] Foo references an instance of type Bar

in-place [2] Foo holds an instance of type Bar

Out-of-place Layout

```
class Foo {
  Bar *bar;
};
```

In-place Layout

```
class Foo {
  Bar bar;
};
```

[1] External
[2] Internal

Out-of-place vs In-place Layout

Performance

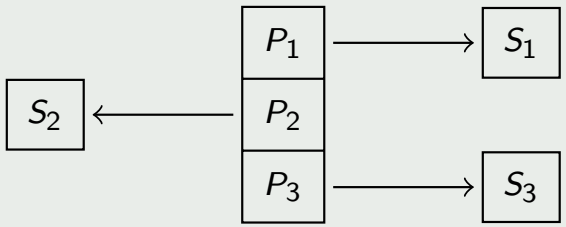
When speaking about performance:

- in-place layout is more efficient

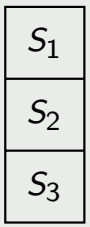
When accessing an instance of the container class – Foo – the included content – Bar – is likely to be loaded in cache together with its container

- we expect a lower amount of cache misses

Out-of-Place/External Stack



In-place/Internal Stack



Legend: P_x = pointer to x, S_x = storage for x

Comment to C-like Macro Programming

Using macros code generation is not under control

- no constraints on parameters

What we would is a similar feature:

- support code generation
- constraint parameters

The tool we need are templates:

- they allow implementing meta-programming

Contents

Hello Template World

A template is composed by:

- a set of parameters
- a parametrized construct

Parameters can be:

types a built-in type – e.g. **int**
or an user-defined class – e.g. **class** Foo

values integer constants – e.g. 11

Parametrized constructs:

class template parameter used to parametrize data member and member functions

function template parameters used to parametrize function body

Hello Template World

Syntax

The **template** keyword must be placed in front of parametrized construct:

$$TPL \rightarrow template < PARAM(, PARAM)^* > CONSTRUCT$$

Class Parametrization

```
template
<typename Ty,
   int N>
class Foo {
   Ty bar[N];
};
```

Function Parametrization

```
template
<typename Ty>
void bar(Ty baz) {
   baz.hello();
}
```

Hello Template World
Work-flow

A template is just an declaration:

- it tells the compiler there is a parametrized constructs

The programmer instantiates the template:

- e.g. Foo<**int**, 4>

This will force the compiler generating the code for the Foo<**int** ,4> class, that is an instance of the Foo template:

- if the template is instantiated twice with the same arguments in the same compilation unit, code is generated only one time
- otherwise, the same code can be generated more than one time, increasing code-segment size

Hello Template World

Using Templates

To use a class template:

1. declare a variable whose class is a template class instance – e.g. Foo<**int**, 4> foo
2. use the variable like a normal instance of a class

To use a function template:

- reference a template instance – e.g. call – by providing all parameters needed to instantiate the template – e.g. bar<Baz>(Baz())

Compiler try to infer template parameters from values used by the template instance:

- the argument of the function template bar is of type Ty, that is the template parameter
- you can call bar passing a Baz value – bar(Baz()) – this allows the compiler inferring the value of the template parameter – Baz

Meta-programming with Templates

Showing code on slides is both boring and error-prone, so I will use as much as possible `vi` *and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.*

"Talk is cheap, show me the code" [3]

A Java-to-C++ translation table is available at slides end.

Templates let C++ supporting meta-programming:

- `cpp-template-stack.cpp` shows how both type and value parameters can be used

Specialization

Given a template, it is possible specializing it:

- provide a different implementation for some type/value of parameters

Specialization can be:

full specialization is done by assigning a type/value to all template parameters

partial specialization is done by assigning a type/value to some of template parameters

Partial specialization is supported only by class templates

Class Specialization
Example

Foo Full Specialization

```
template <>
class Foo<char, 16> {
  char *bar;
};
```

Foo Partial Specialization

```
template <unsigned N>
class Foo<char, N> {
  char bar[N];
};
```

Foo Template

```
template
<typename Ty,
  int N>
class Foo {
  Ty bar[N];
};
```

When instantiating the Foo template:

- compiler looks for a full specialization using the given template parameters. If found, the full specialization is used for code generation – e.g. Foo<**char**, 16>
- otherwise a partial template definition – e.g. Foo<**char**, 4> – or the general template definition – Foo<**int**, 4> – are used

Class templates supports default parameters:

Default Parameters

```
template
<typename Ty, int N = 16>
class Foo { Ty bar[N]; };
```

- Foo<**int**> = Foo<**int**, 16>

Function Specialization
Example

The bar Template

```
template <typename Ty, int N>
Ty bar() {
  return Ty(N);
}
```

The bar Full Specialization

```
template <>
int bar<int, 4>() {
  return 4;
}
```

- see also `hello-template-world.cpp`

Templates Usage
Containers

C++ Standard Template Library heavily uses templates:

- STL containers need to store different types of elements
- using templates it is possible defining elements storage area and container-specific algorithms parametric on the container element type

STL **int** Vector

```
std::vector<int>
ints;
ints.push_back(2);
ints.push_back(3);
```

STL **float** Vector

```
std::vector<float>
floats;
floats.push_back(2.0);
floats.push_back(3.0);
```

Templates Usage
Parametric Container

Showing code on slides is both boring and error-prone, so I will use as much as possible `vi` *and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.*

"Talk is cheap, show me the code" [3]

A Java-to-C++ translation table is available at slides end.

The `cpp-template-stack.cpp` implements a simple parametric stack

Contents

Compile-time Computations
Factorial

Templates enables compile-time computations:

- toy example is factorial [3]

Factorial

```
template <unsigned N>
struct fact {
  enum { value = N * fact<N - 1> };
};

template <>
struct fact<0> {
  enum { value = 1 };
};
```

[3] See `fact.cpp`

Factorial
Comment

Factorial computation for 4:

1. referencing value using 4 starts factorial computation
2. the fact template is recursively instantiated, until template specialization for 0 is reached
3. the expression is fold at compile-time, thus computing 24

The key feature of factorial computation is exploiting **enum** to trigger a compile-time computation:

- the compiler is forced to evaluate the fact<4>::value expression because it must associate a constant integer to the value **enum**

Bare templates does not enforce parameters to implement an interface:

- e.g. be usable with a function, providing a member function, ...

Popping Elements

```
template <typename Ty, unsigned N>
Ty Stack<Ty, N>::pop() {
  if(top == 0)
    abort();

  return storage[--top];
}
```

Traits
Constraints for Templates

Traits is a programming idiom used to force a template parameter implementing some interface:

Traits for Error Handling

```
template <typename Ty>
struct ErrorTraits;

template <>
struct ErrorTraits<int> {
  enum { invalidValue = -1 };
};
```

Traits
Using Traits with Stack

Using traits it is possible returning an error value when trying to pop from an empty stack:

Popping Elements with Error Traits

```
template <typename Ty, unsigned N>
Ty Stack<Ty, N>::pop() {
  if(top == 0)
    return ErrorTraits<Ty>::invalidValue;

  return storage[--top];
}
```

Example `stack-visit.cpp` shows how traits can be used to visit a stack without caring about the actual stack implementation

Code Factorization

One common use of templates is removing unnecessary casts:

Container Cast

```
elt = *reinterpret_cast<int *>(stack.pop());
```

Container No-cast

```
elt = stack.pop();
```

However, using templates force compiler generating multiple copies of the same code:

- templates always in-lined

Code Factorization
Exploiting Inheritance

A common technique to contain memory footprint is to combine templates and class inheritance:

- a class is split into a template-dependent and a template-independent part

The template-independent part is implemented in separate class:

- only used to contains template-independent code
- not intended to be an end-user interface

The template-dependent part derives from the template-independent class:

- exploit services of base class to implement all functionalities requested by the end-user
- uses templates to provide a kind interface to end-user

Code Factorization
Example

Showing code on slides is both boring and error-prone, so I will use as much as possible `vi` *and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.*

"Talk is cheap, show me the code" [3]

A Java-to-C++ translation table is available at slides end.

The file `stack-heap-stack.cpp` implements an adaptive stack. When its size is under the half of the capacity, internal storage is used. Passing the half, stack is moved on the heap:

- StackBase contains template-independent code – e.g. stack expansion
- Stack contains template-dependent code – e.g. push

Parametric Polymorphism

C++ templates partially implements parametric polymorphism:

- polymorphism is achieved at compile time through template **typename** parameters

Full parametric polymorphism not supported by C++:

- e.g. Haskell type-classes allows to strictly require a type implementing an interface in order to be used in a parametric function

C++ approach is idiom-based:

- templates allows to perform some meta-programming
- to request a type implementing an interface, use traits

But traits are a programming technique:

- the compiler is unaware of their existence
- is up to the programmer correctly using them

Contents

Java Concepts

Java generics are not equivalent to C++ templates:

- C++ templates allows to perform computations at compile-time
- they are Turing-complete

Java generics goal is different:

- free end-user from the burden of writing tons of casts
- enforcing some constraints – e.g. using **extends** with generic classes

Contents

Bibliography

Bruce Eckel.
Thinking in C++ – Volume One: Introduction to Standard C++.
http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

Bruce Eckel and Chuck Allison.
Thinking in C++ – Volume Two: Practical Programming.
http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

Linus Torvalds.
Re: SCO: "thread creation is about a thousand times faster than onnative.
https://lkml.org/lkml/2000/8/25/132.