Templates: A Short Introduction

Christopher Pinke

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Institute for Theoretical Physics
Goethe-University Frankfurt

HIC for FAIR Helmholtz International Center

C++ User Group
FIAS, Frankfurt, December 17th, 2014

Websites visited for this talk...

- http://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part
- http://www.cprogramming.com/tutorial/template_specialization.html
- http://eli.thegreenplace.net/2014/sfinae-and-enable_if/
- http://www.cplusplus.com/reference/type_traits/enable_if/
- http://accu.org/index.php/journals/442
- http://www.cprogramming.com/c++11/c++11-compile-time-processing-with-constexpr.html

Overview

Introduction to templates

- General properties and usage
- Template specialization and partial specialization

More advanced techniques

- Type traits
- SFINAE and `std::enable_if`
- Constant expressions `constexpr`

Templates

General idea

Write generic classes and functions (independent of actual type)

- Less/No code duplication
- Flexibility ("On demand compilation")

- Categories: Function Templates and Class Templates
- Examples: standard containers (vector, pair, set . . .)

Hands on example: Function templates (1)

Suppose you have this code somewhere (hypothetically):

```
int calcDifferenceAndPrint(int a, int b)
{
  int result = a - b;
  std::cout << result <<  std::endl;
  return result;
}
double calcDifferenceAndPrint(double a, double b)
{
  double result = a - b;
  std::cout << result << std::endl;
  return result;
}
...
a = calcDifferenceAndPrint(2,3); //int
b = calcDifferenceAndPrint(1.23, 4.56); //double
```

Same code, only with different types! ⇒ **Replace with template!**

Hands on example: Function templates (2)

Templated code:

```cpp
template<typename Type>
  Type calcDifferenceAndPrint(Type a, Type b)
{
  Type result = a - b;
  std::cout << result;
  std::cout << " (Got type: \""
    << typeid(Type).name() << "\")" <<  std::endl;
  return result;
}
...
a = calcDifferenceAndPrint(2,3);
  //output: "-1 (Got type: "i")"
b = calcDifferenceAndPrint(1.23, 4.56);
  //output: "-3.33 (Got type: "d")"
```

Compiler generates appropiate functions (at compile time!)

Class Templates (Actually more relevant than function templates)

```
template < typename T >
class Subtracter
{
public:
  Subtracter(T &a, T &b)
  {
    result = a - b;
    std::cout << result << std::endl;
  }
  T result;
};
...
Subtracter<int> subtracter(2, 3);
```

→ E.g. `vector<double>, pair<int, int>, ...` (STL)
Note: Compiler cannot deduce type from arguments like in fct. templates (need explicit <...>)

Templates: Notes (1)

Arguments

- Multiple Arguments possible
- Template can have any argument, including a class template instantiation
  (E.g. `std::Pair< int, std::Pair< int, int> >`)
- Can use `const`, $\star$ and `&` in parameter specialization like in "normal" code.
- Non-type template arguments possible
  (E.g. `template<type T, int a>`)
  Restriction to integral types and compile time constants!

Templates: Notes (2)

Compiler

- Class/fct. template vs. template instance
  `template<type T> class A` vs. `A<double>`
- Generation of code only when needed ⇒ Less sourcecode ("Compilation on demand")
- Compilation errors only under certain circumstances
  (E.g. `%` operation for `int/float` or
  `Subtracter<int> subtracter(1.2, 3)`)
  ⇒ Providing appropiate methods necessary.
- `A<int>` and `A<double>`: Different types to compiler
  (E.g. comparision or assignment will not work (UDT!))

Templates: Notes (3)

Class Templates

- Function declaration inside or outside of body
  (Not as straight-forward as "normal" header/source !)
- `virtual` fct. and templates do not work together
  (runtime vs. compiletime)
- Inheritance possible

Template Specialization (1)

Recall the previous example. What about these calls?

```
a = calcDifferenceAndPrint("Hello", "World");
Subtracter<std::string> subtracter("Hello","World");
```

Perfectly fine to use strings with templates, BUT compiler objects:

```
error: no match for 'operator-'
(operand types are 'std::basic_string<char>' and
  'std::basic_string<char>')
  Type result = a - b;
```

Way out: Define operator OR specialize template

"Allows customizing the template code for a given set of template arguments."

Template Specialization (2)

```
void calcDifferenceAndPrint
  (const std::string & a, const std::string & b)
{
  std::string result = a + "\"-\"" +  b;
  std::cout << result  << std::endl;
}
template <>
class Subtracter<std::string>
{
  public:
Subtracter(const std::string a,const std::string b)
  {
    result = a + "\"-\"" +  b;
    std::cout << result << std::endl;
  }
  std::string result;
};
```

Partial Template Specialization

"Allows customizing class templates for a given category of template arguments."

```
//"Normal template"
template< typename T>
class A
{ [class declaration] }
//Partially specialized template
// (for pointer-like arguments)
template< typename T>
class A< T* >
{ [class declaration specific to pointer types] }
```

More advanced techniques

- Type traits
- SFINAE and `std::enable_if`
- Constant expressions `constexpr`

Type Traits (1)

(trait = Merkmal)

Idea: Use specialized templates to build a “switch” for different types. Example for illustration: `isVoid`

```
//define default value via template
template< typename T >
struct isVoid{
  static const bool value = false;
};
//define specialized template for actual void
template<>
struct isVoid< void >{
  static const bool value = true;
};
```

All objects will give `false` by default, only `void` objects dont.

Type Traits (2)

Example: Use an optimized algorithm for specific object type "Switcher" (just like `isVoid`) ...

```
template< typename T >
struct supportsOptimizedImplementation
{  static const bool value = false; };
template<>
struct supportsOptimizedImplementation
  < optimizedType >
{ static const bool value = true; };
```

... and algorithm:

```
template< typename T >
void algorithm( T& object ) {
  algorithmSelector
    < supportsOptimizedImplementation< T >::value >
    ::implementation(object);
}
```

Type Traits (3)

```
//default:
template< bool objectHasOptimizedImplementation >
struct algorithmSelector {
  template< typename T >
  static void implementation( T& object )
  {
    //implementation of algorithm
  }
};
//specialization for objects that have opt. impl.
template<>
struct algorithmSelector< true > {
  template< typename T >
  static void implementation( T& object )    {
    object.optimizedImplementation();
  }
};
```

SFINAE (1)

(Substitution Failure Is Not An Error)

Example:

```
//implementation for int
int negate(const int& i) { return -i; }
//more general template
template <typename T>
typename T::value_type negate(const T& t)
{ return -t(); }
```

Although valid `negate` implementation exists for `int`, compilation would fail because the template yields invalid code:

```
int::value_type negate(const int& t);
```

However, with SFINAE this does not give a compilation error.
⇒ Very important to use templates in a broader context

SFINAE (2)

Substitution Failure Is Not An Error from C++ standard:

If a substitution results in an invalid type or expression, type deduction fails. An invalid type or expression is one that would be ill-formed if written using the substituted arguments. Only invalid types and expressions in the immediate context of the function type and its template parameter types can result in a deduction failure.

"Immediate context": This variant would give compilation error

```
template <typename T>
 void negate(const T& t) {
 typename T::value_type n = -t();
}
```

⇒ Must make compiler fail deduction for invalid types right in the declaration to cause substitution failure

enable_if (1)

SFINAE can be used very effectively with `enable_if`:

```
template <bool, typename T = void>
  struct enable_if {};
template <typename T>
  struct enable_if<true, T> { typedef T type; };

template <typename T>
void do_stuff(T &t, typename enable_if
  <std::is_integral<T>::value,T>::type *_t = NULL)
  { ... }
template <typename T>
void do_stuff(T &t, typename enable_if
  <std::is_class<T>::value, T>::type *_t = NULL)
  { ... }
```

enable_if (2)

```
template <typename T>
void do_stuff(T &t, typename enable_if
  <std::is_integral<T>::value,T>::type *_t = NULL)
  { ... }
template <typename T>
void do_stuff(T &t, typename enable_if
  <std::is_class<T>::value, T>::type *_t = NULL)
  { ... }
```

Now `do_stuff(25)`: The second template is "disabled" because it gives a substitution error!

Compiler output:

```
note: template<class T> void do_stuff(T, typename enable_if
  <std::is_class<T>::value, T>::type*)
note:   template argument deduction/substitution failed
In substitution of 'template<class␣T>␣void␣do_stuff(T,␣typename␣enable_if
␣␣<std::is_class<T>::value,␣T>::type*)␣[with␣T␣=␣int]':
functionTemplates.cpp:152:16:   required from here
functionTemplates.cpp:97:6: error:
   no type named 'type' in 'struct␣enable_if<false,␣int>'
```

enable_if (3)

- std::enable_if since C++11
- More handy version since C++14:

```
template <bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type;

template <typename T>
void do_stuff(T &t, typename enable_if
  <std::is_integral<T>::value,T>::type *_t = NULL)
  { ... }

template <typename T>
void do_stuff(T &t, std::enable_if_t
  <std::is_integral<T>::value,T> *_t = NULL)
  { ... }
```

constexpr

Example: Factorial

```
constexpr int factorial (const int n)
{
  return n > 0 ? n * factorial( n - 1 ) : 1;
}
```

Calculations at compile time (C++11)

- C++14: Can consist of multiple statements
- It can call only other constexpr functions
- It can reference only constexpr global variables and fct. arguments
- Also available at runtime (normal fct.)
- Allows floating point operations! (Templates do not)

Summary & Perspectives

Introduction to templates

- General properties and usage
- Template specialization and partial specialization

More advanced techniques

- Type traits
- SFINAE and `std::enable_if`
- Constant expressions `constexpr`

Perspectives

Template metaprogramming