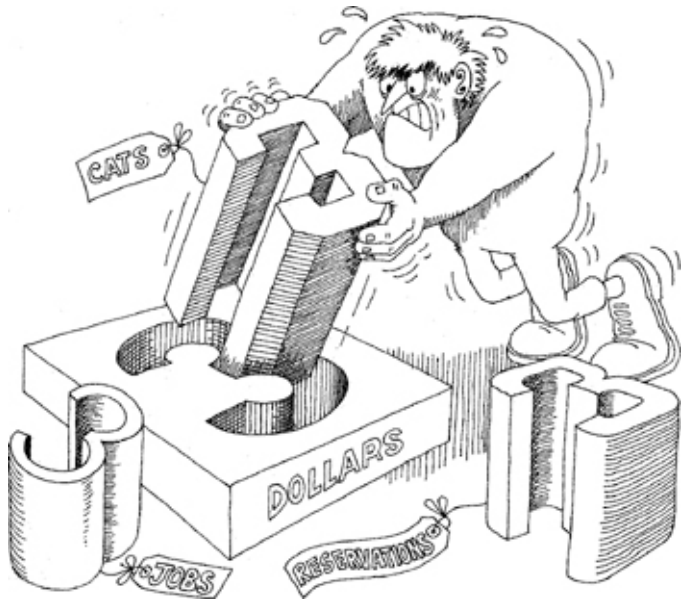
Templatemetaprogrammierung in C++

FlorianJW



Wiederholung

Verwendung

```
#include <vector>
#include <algorithm>
#include <string>

int main() {
  std::vector<int> vec{1,2,3};
  using std::string;
  auto str = std::min(string{"foo"},
                      string{"bar"});
}
```

Wiederholung

allgemeine Funktionstemplates

```
//yes, this is extremly simplified:
template<typename T>
T max(T arg1, T arg2) {
   return (arg1 > arg2) ? arg1 : arg2;
}
```

Wiederholung

spezialisierte Funktionstemplates

```
#include <cstdlib>

template<>
char* max(char* arg1, char* arg2) {
  return strcmp(arg1, arg2) < 0 ?
    arg2 : arg1;
}
```

Wiederholung

Klassentemplates und integrale Templateargumente

```
template<typename T, std::size_t Tsize>
class my_array{
  T data[Tsize];
  public:
    my_array() = default;
    T& operator[](size_t index){
      return data[index];
    }
    // ...
};
```

Wiederholung

teilweise Spezialisierung von Klassentemplates

```
template<std::size_t Tsize>
class my_array<bool, Tsize>{
  // experience with std::vector<bool>
  // showed that this is a stupid idea:
  int data[(Tsize+1) / (sizeof(int)*CHAR_BIT)];

  // ...
};
```

- Nur für Klassentemplates.
- Für Funktionstemplates lässt sich ähnliches mit Überladung erreichen.

Wiederholung

constexpr

```
#include <array>

constexpr int exp(int base, int n){
  return n > 0 ?
    base * exp(base, n-1) :
    base;
}

int main(){
  std::array<int, exp(1024,2)> arr;
}
```

Wiederholung

constexpr

Chandler Carruth: “But we are not done yet. It that hard. It is harder then variadic templates, I believe it is actually harder then lambdas. It is one of the most challenging to implement correctly features in entire C++11.”
Bjarne Stroustrup: “It is interesting to note, that it is one of these features, that was made more complicated over the dead bodies of its proposers and initial implementors during the standardization.”

Wiederholung

variadische Templates (1)

```
template<typename... T>
std::string text(T&&...args) {
  std::stringstream returnstream;
  text_helper(returnstream,
              std::forward<T>(args)...);
  return returnstream.str();
}
```

Wiederholung

variadische Templates (2)

```
template<typename T>
void text_helper(std::stringstream& stream,
                 T&& arg) {
  stream << std::forward<T>(arg);
}

template<typename T, typename... Targs>
void text_helper(std::stringstream& stream,
                 T&& arg, argT&&...args) {
  stream << std::forward<T>(arg);
  text_helper(stream,
              std::forward<Targs>(args)...);
}
```

Typerich Programming

Ein einfaches Klassentemplate...

```
template<typename T>
class point{
  T _x;
  T _y;
  public:
    point(T x, T y): _x{x}, _y{y} {}
    T get_x(){ return _x; }
    T get_y(){ return _y; }
};

int main() {
  int x{1}, y{2};
  point<int> p{y,x}; // error?
}
```

Typerich Programming

Mehr Typsicherheit (1)

```
template<typename T>
class x_coord{
  T _value;
  public:
    x_coord(T value): _value{value} {}
    T get_value(){ return _value; }
    // ...
};
```

Typerich Programming

Mehr Typsicherheit (2)

```
template<typename T>
class point{
  x_coord<T> _x;
  y_coord<T> _y;
  public:
    point(xcoord<T> x, ycoord<T> y)
      : _x{x}, _y{y} {}
    x_coord<T> get_x(){ return _x; }
    y_coord<T> get_y(){ return _y; }
};

int main() {
  x_coord<int> x{1};
  y_coord<int> y{2};
  point<int> p{x,y}; // no error
}
```

Typerich Programming

Übersicht

Vorteile

- Typen prüfen Korrektheit
- Verwendung nicht-trivialer Typen findet nicht-triviale Fehler
- Ermöglichen weitreichendere Funktionsüberladungen.
- erhöhte Semantik im Code

Nachteile

- Aufwand die Typen zu erstellen
- potentiell **viel** Codeduplikation
- Unter Umständen aufwendigere Einarbeitung in den Code

Templateprogrammierung

Einführung

- Naheliegende Lösung: Automatische Typerzeugung
- Externe Lösungen sind hässlich...
- Lösung via Präprozessor noch hässlicher und unflexibel.
- Lösung mit Templates aufwendiger aber möglich, weniger hässlich, potentiell fast beliebig flexibel.

Templateprogrammierung

Definition

Templatemetaprogramming: Programmierung im Templatesystem/Typsystem einer Programmiersprache zur Berechnung von Typen, Konstanten und Codepfaden während des Kompiliervorgangs.

Templateprogrammierung

Um zum Punkt zu kommen (1)

```
#include "basic_number.hpp"

struct x_coord_id{};
template<typename T> using x_coord =
  type_builder::basic_number<T, x_coord_id>;

struct y_coord_id{};
template<typename T> using y_coord =
  type_builder::basic_number<T, y_coord_id>;
```

Templateprogrammierung

Um zum Punkt zu kommen (2)

```
template<typename T>
class point{
  x_coord<T> _x;
  y_coord<T> _y;
  public:
    point(x_coord<T> x, y_coord<T> y)
      : _x{x}, _y{y} {}
    x_coord<T> get_x(){return _x;}
    y_coord<T> get_y(){return _y;}
};

int main(){
  x_coord<int> x{1};
  y_coord<int> y{2};
  point<int> p{x,y}; // no error
};
```

Templateprogrammierung

Grundlagen der Templateprogrammierung

- Enums als Konstanten nutzbar
- rekursive Templates mit Spezialisierungen ermöglichen Rekursion
- $\rightarrow$ Strikt funktionale Programmiersprache mit Memoisation
- Berechnungen am Ende des Kompiliervorgangs abgeschlossen
- Zahlenberechnung dank constexpr nur eingeschränkt sinnvoll

Templateprogrammierung

Fibonacci (1)

Laufzeit: $\Theta(n)$

```
template<unsigned int N> struct fib{
  enum{value = fib<N-1>::value
    + fib<N-2>::value};
};

template<> struct fib<1>{
  enum{value = 1};
};

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

Templateprogrammierung

Fibonacci (2)

Laufzeit: $\Theta(n)$

```
template<int N> struct fib :
  std::integral_constant<uint64_t,
    fib<N-1>() + fib<N-2>()> {};

template<> struct fib<1> :
  std::integral_constant<int,1> {};

template<> struct fib<0> :
  std::integral_constant<int,0> {};
```

Templateprogrammierung

Fibonacci (3)

Laufzeit: $\Theta(n)$

```
template<uint64_t N>
using Num = std::integral_constant<uint64_t, N>;

template<int N> struct fib :
  Num<fib<N-1>() + fib<N-2>()> {};

template<> struct fib<1> : Num<1> {};

template<> struct fib<0> : Num<0> {};
```

Templateprogrammierung

Fibonacci (4)

Laufzeit: $\Theta(\text{fib}(n)) = \Theta\left(\frac{1+\sqrt{5}}{2}^n\right)$, potentiell zur Programlaufzeit.

```
constexpr uint64_t fib(int n){
  return n <= 1 ? n : fib(n-1) + fib(n-2);
}
```

Templateprogrammierung

grundlegende Templates

```
namespace std{

  template<bool Tenable, typename T=void>
  struct enable_if{};

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

} // namespace std
```

Templateprogrammierung

grundlegende Templates

```
namespace std{

  template<typename T1, typename T2>
  struct is_same : std::false_type {};

  template<typename T>
  struct is_same<T,T> : std::true_type {};

} // namespace std
```

Templateprogrammierung

grundlegende Templates

```
namespace std{

  template<bool Cond, typename If_true,
    typename If_false>
  struct conditional{
    typedef If_true type;
  };

  template<typename If_true, typename If_false>
  struct is_same<false, If_true, If_false>{
    typedef If_false type;
  };

} // namespace std
```

Templateprogrammierung

SFINAE - substitution failure is not an error (1)

```
template <typename T1, typename T2>
auto fun(T1 a, T2 b) ->
  typename std::enable_if<
    std::is_same<T1, T2>{}, int>::type
{
  return static_cast<int>(a+b);
}

template <typename T1, typename T2,
  typename = typename std::enable_if<
    !std::is_same<T1, T2>{}>::type
>
double fun(T1 a, T2 b) {
  return static_cast<double>(a+b);
}
```

Templateprogrammierung

SFINAE - substitution failure is not an error (2)

```
int main() {
  // first function:
  auto f = fun(1, 1.0);
  //decltype(f) is double

  // second function:
  auto i = fun(1, 1);
  //decltype(i) is int
}
```

Templateprogrammierung

static_assert

- `static_assert` ist ein compile-time assert.
- Da Templates erst bei Verwendung instanziiert werden auch bisweilen nützlich.

```
template<int Targ>
void fun(){
  static_assert((Targ!=0)&&(Targ==0),
    "This must not be called");
}

int main(){}
```

Templateprogrammierung

Template-Templateargumente

- Auch Templates können Templateargumente sein.
- So ist etwa folgendes möglich:

```
#include <vector>
template<typename T>
using vec = std::vector<T, std::allocator<T>>;

template<typename T,
    template<typename> class Tc = vec>
class safe_container: public Tc<T> {
  public:
    safe_container(): Tc<T>(){}
    T& operator[](int index) {
      return this->at(index);
    }
};
```

Templateprogrammierung

Policy-based Design

- Wie im letzten Beispiel schon zu sehen ist, kann Vererbung auch zum injecten von Methoden und Attributen genutzt werden. $\longrightarrow$ Policy-based design
- Private Vererbung kann hier sehr nützlich sein.
- Mit öffentlicher Vererbung können Mixins realisiert werden; dies verletzt allerdings schnell das “is-a”-Kriterium.

Templateprogrammierung

Policy-based Design

- Öffentliche Vererbung zur Runtime zu benutzen, kann aber bei falscher Anwendung zu undefiniertem Verhalten führen:

```
int main(){
  // undefined behaviour:
  // (and ugly: if you really need
  // a pointer, use a smart one)
  std::vector<int> * ptr =
    new safe_container<int>{};
  delete ptr;
}
```

Typebuilder

- Bibliothek zur einfachen Erzeugung von Typen.
- Primärer Fokus sind Zahlen.
- reines C++, keine Spracherweiterungen, keine Libs außer der STL; dadurch hochportabel.
- keine Unterstützung alter Compiler vorgesehen.
- wo möglich kein Overhead gegenüber nativen Typen.
- basic_number ermöglicht die Erstellung einer Bibliothek für physikalische Einheiten (SI-System) in 160 Zeilen.

Ausblick

automatischer Returntype

```
// inferred returntype:
const auto f=[](std::string arg){
  return arg += "_f";
};

// until C++11: explicit returntype required:
std::string g(std::string arg){
  return arg += "_g";
}

// the future (C++14):
auto h(std::string arg){
  return arg += "_h";
}
```

Ausblick

Template-Constraints

```
template<Sortable Cont>
void sort(Cont& container);

template<typename Cont>
requires Sortable<Cont>()
void sort(Cont& cont)


// possible Extension:
using Container{Cont};
//////////////////////
void sort(Cont& c);
```

Ausblick

Liberaleres constexpr

- Aufhebung der meisten Restriktionen.
- wichtige Ausnahmen: kein static/thread_local, kein goto, keine uninitialisierten Variablen, keine Änderungen an nichtlokalen Variablen.

Sonstiges

gefundene Compiler-Bugs: Clang

Segfault in clang (Neuentdeckung, mittlerweile gefixt):

```
int main(){
  (1 ? throw 1 : true) ? 1 : throw 2;
}
```

Sonstiges

gefundene Compiler-Bugs: GCC

gcc $\leqslant$ 4.7: falsche Zugriffsrechte (war im trunk bereits gefixt):

```
class c {
  int f;
  public:
  template <typename A>
  decltype(f) m(A) const;
};
decltype(c{}.m(0)) i;
```

Sonstiges

Standard-Issues

http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1635:

```
#include <type_traits>

template<bool bar> struct foo{
  template<typename T,
    typename = typename std::enable_if<bar>::type>
  void fun(T){}
};

int main(){
  foo<false> var{};
}
```