7. Templates

AlDaBi Praktikum

David Weese
© 2010/11

Enrico Siragusa
WS 2011/12

Inhalt

- Templates
- Template Subclassing

- Bemerkungen zur P-Aufgabe

TEMPLATES

Templates: Motivation

- **Aufgabe**: Schreibe eine Funktion max(a, b), die das Maximum zweier Zahlen a und b ausgibt:

```
int max (int a, int b)
{
    if (a > b) return a;
    return b;
}
```

- **Problem**: Funktion wird für alle Typen benötigt

```
float x = max(1.4, y);
```

Lösung: Templates

- Templates sind Schablonen, nach denen der Compiler Code herstellt

```
template <typename T>
T max (T a, T b)
{
    if (a > b) return a;
    return b;
}

...

float x = 1, y = 2;
float z = max (x, y);
```

Template-Argumente

- Es gibt zwei Möglichkeiten, die Template-Argumente zu bestimmen:

1. Explizit:

```
template <typename T1, typename T2>
void f (T1 a, T2 b) { ... }

f<int,float>(0, 3.5);
```

2. Implizit:

```
int   x = 0;
float y = 3.5;

f(x, y);
```

Template-Argumente, gemischt

- Mischung von expliziter und impliziter Bestimmung von Template-Argumenten:

```
template <typename T1, typename T2, typename T3>
void f (T3 x, T2 y, T1 z)
{ ... }

f<int>( (char) 0, 3.5F, 7 );
```

1. Argument: `char x`
2. Argument: `float y`
3. Argument: `int z`

Implizit: Problem 1

- Folgender Code verursacht ein Problem:

```
template <typename T>
T max (T a, T b)
{
    //...
}

double x = 1.0;
double y = max(x, 0);
```

`error: template parameter 'T' is ambiguous`

- Grund:
  - x (double) und 0 (int) haben unterschiedliche Typen
  - In Funktionssignatur müssen beide Argumente denselben Typ haben

Implizit: Problem 2

- Bei folgendem Beispiel funktioniert eine implizite Bestimmung des Template-Arguments gar nicht:

```
template <typename T>
T zero ()
{
    return 0;
}

int x = zero();
```

`error: could not deduce template argument`

- Grund:
  - Implizite Bestimmung benutzt Argumente, nicht den Rückgabewert

Parameter-Deklarationen

- Statt eines Types kann auch eine Konstante als Template-Parameter spezifiziert werden:

```
template <int I>
void print ()
{
    std::cout << I;
}

print<5>();
```

- Ausgabe: **5**

Template-Klassen

- Wie Template-Funktionen lassen sich auch Template-Klassen definieren:

```
template <typename T>
struct Pair
{
    T element1;
    T element2;
};

Pair <int> p;
```

- Beispiel:
  - vector, map, list (STL Template Klassen)

Bei Template-Klassen müssen die Argumente immer explizit angegeben werden

Defaults für Template-Parameter

- Für Template-Klassen können Default-Argumente definiert werden:

```
template <typename T = int>
struct Pair
{
    T element1;
    T element2;
};

Pair < > p;
```

- Nachfolgende Parameter müssen dann auch Defaults haben.

Default nur bei Template-Klassen, nicht bei Template-Funktionen

Template Beispiele

- Frage: Wozu kann man die Argumente von Template-Klassen verwenden?
  - Beispiel: Typen für Member

```
template <typename T>
struct Pair
{
    typedef T MyType;
    T element1;
    T element2;
    T get_max();
    void set_both(T elm1, T elm2);
};
```

typename

- Hinweis für den Compiler: „hier kommt ein Type!“

```
template <typename T>
struct A
{
    typename T::Alphabet x;    // abhängig, T ist Template-Arg
    typedef int MyInteger;
};

A<char>::MyInteger y;          // char ist kein Template-Arg
```

- `typename` steht vor ...
  1. Template-Argumenten, die Typen sind (keine Konstanten)
  2. Typen, die von Template-Argumenten abhängen

"Patterns" von Template Typen

- Typen von Klassentemplates können als eine Art "Pattern" angegeben werden:

```
template <typename T1, typename T2>
struct MyClass;

template <typename T>
void function1 (T & obj);

template <typename T1, typename T2>
void function2 (MyClass<T1, T2> & obj);

template <typename T>
void function3 (MyClass<T, int> & obj);
```

Template Spezialisierung

- Templates können für bestimmte Template-Argumente spezialisiert werden.

```
template <typename T>
struct MyClass
{ int foo; };

template < >
struct MyClass <int>
{ int x; };

MyClass<int> obj;
obj.x = 17;

obj.foo = 18;            // ERROR!
```

- Grund:
  - `foo` ist in `MyClass<int>` nicht bekannt.

Quiz: Was wird ausgegeben?

```
template <typename T>
struct C
{
    static void f() { std::cout << "allgemein"; }
};

template < >
struct C <char*>
{
    static void f() { std::cout << "speziell"; }
};

template <typename T>
void call_f(T & t)
{
    C<T>::f();
}

call_f("hallo");
```

Quiz: Und jetzt?

```
...

typedef char * c_ptr;
c_ptr x;
call_f(x);
```

Partielle Spezialisierung

```
template <typename T, unsigned int I>
class Array
{
    T arr [I];
};

template <typename T>
class Array <T, 0>
{
};
```

Nur Template-Klassen können partiell spezialisiert werden

Bei Template-Funktionen gibt es keine partielle Spezialisierung

Metafunctions

- Metafunctions sind eine Anwendung für Template Spezialisierung
  - Beispiel: Eine Funktion auf Strings

```
template <typename TString>
char first_character(TString & str)
{
    return str[0];
}
```

  - Problem:
    - Was machen wir, wenn der Alphabettyp von `TString` nicht `char` ist?

Metafunctions

- Idee: Man benötigt eine Art "Funktion", die Typen zurückliefert
  - Pseudo-Code:

```
template <typename TString>
Value(TString) first_character(TString & string)
{
    return string[0];
}
```

Der angegebene Code ist natürlich kein gültiges C++

Metafunctions

- Lösung: Verwende Klassen-Templates

```
template <typename T>
struct Value
{
    typedef char Type;
};


template <typename TString>
typename Value<TString>::Type
first_character(TString & str)
{
    return str[0];
}
```

Metafunctions

- Spezialisiere Template für verschiedene Typen:

```
template <
    typename TChar,
    typename TTraits,
    typename TAlloc >
struct Value < basic_string<TChar, TTraits, TAlloc> >
{
    typedef TChar Type;
};


template <>
struct Value <char *>
{
    typedef char Type;
};
```

STL Containers und Iteratoren enthalten value_type Member, z.B vector<char>::value_type ist char

Rekursive Templates

- Beispiel: Fibonacci

```
template <int N>
struct Fib {
    static const int value = Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<0> {
    static const int value = 0;
};

template <>
struct Fib<1> {
    static const int value = 1;
};

// Output: 102334155
cout << Fib<40>::value;
```

TEMPLATE SUBCLASSING

Das Delegation-Problem

- Problem: Es sei z.B. folgendes Programm gegeben:

```
struct Car {
  void refill() { cout << "Gas Please"; }

  void drive() {
    if (tank.empty())
      this->refill();
    ...
  }
};

struct ElectricCar: Car {
  void refill() { cout << "Energy Please"; }
};
```

Das Delegation-Problem (II)

- Was passiert hier ?

```
ElectricCar e;
e.drive();
```

- Ausgabe: Gas Please (!)
- Losung: **`virtual`**

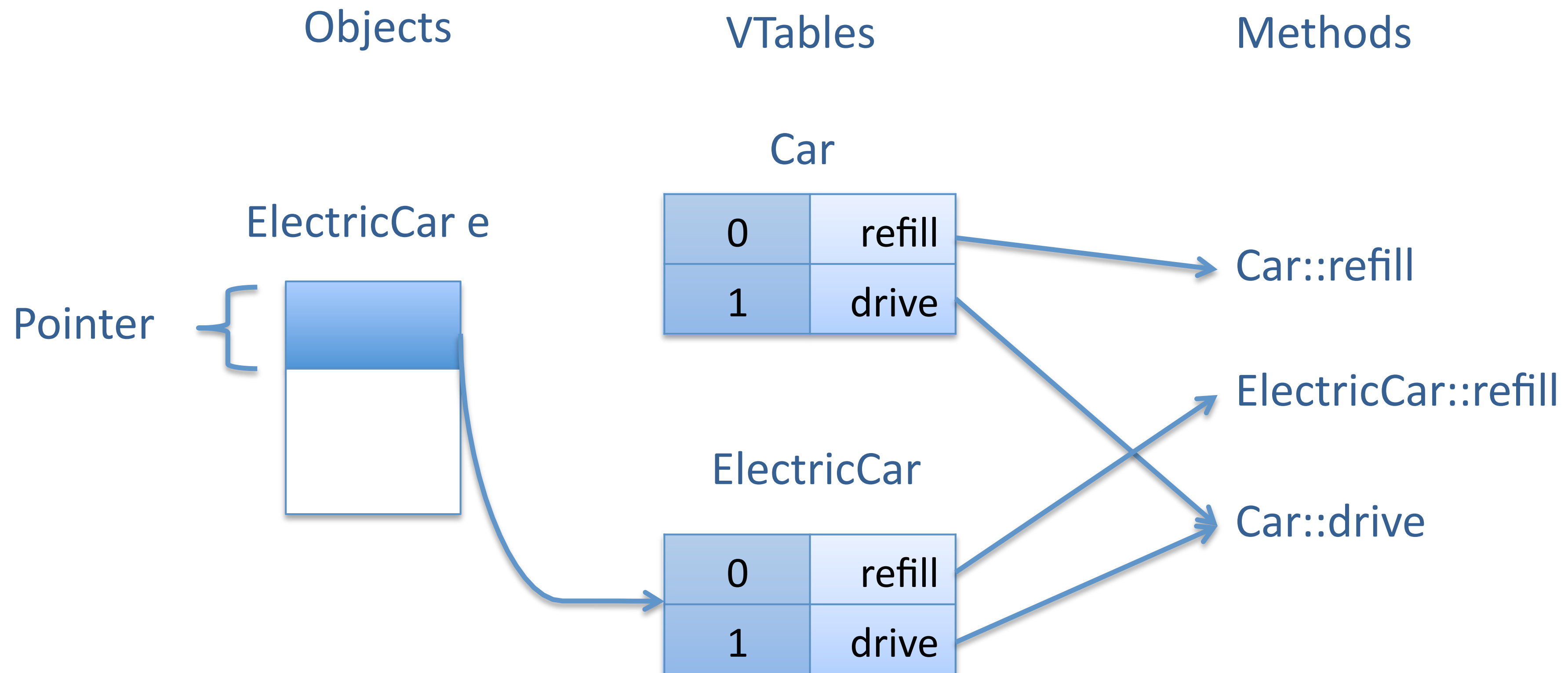
```
struct Car {
  virtual void refill() { ... }
};

...

ElectricCar e;
// Prints "Energy Please"
e.drive();
```

Wie funktioniert `virtual`?

- Prinzip: Objekt hält Zeiger auf eine Tabelle mit Zeigern auf die richtigen Funktionen



Eigenschaften virtueller Funktionen

- *dynamic binding*
  - Die tatsächlich aufgerufene Funktion wird erst zur Laufzeit bestimmt

- Nachteile des dynamic binding
  - zusätzlicher Speicherbedarf pro Objekt
  - langsamer, indirekter Sprung
  - kein Inlining

- Beobachtung
  - oft braucht man gar kein *dynamic binding*
  - Typen stehen schon zur Compile-Zeit fest

Template Subclassing

- Eine alternative Lösung des Delegation-Problems mit static binding

- Bisher (objektorientiert):

```
struct Car { /*Car*/ };
struct ElectricCar : Car { /*ElectricCar*/ };
```

Template Subclassing

- Schritt 1: “Template Spezialisierung statt Ableitung”

```
template <typename T>
struct Car { /*Car*/ };

struct ElectricCar;

template <>
struct Car <ElectricCar> { /*ElectricCar*/ };
```

ElectricCar ist ein Tag, das nur deklariert werden muss

Template Subclassing (II)

- Schritt 2: “Globale Funktionen statt Member Funktionen”

```
template <typename T>
void refill (Car<T> & obj)
{
    std::cout << “Gas Please"; ...
}

void refill (Car<ElectricCar> & obj)
{
    std::cout << ”Energy Please"; ...
}

template <typename T>
void drive (Car<T> & obj)
{
    refill(obj); ...
}
```

Template Subclassing (III)

- Template Subclassing löst das Delegation Problem:

```
Car<ElectricCar> car;
drive(car);

// Output: “Energy Please"
```

Anwendungsbeispiel

- Gesucht: Funktion, die das größte Element eines Feldes bestimmt
  - Gegeben ist einen Feld von Zeigern auf eigentliche Elemente
  - Vergleichsfunktion soll frei wählbar sein

- Lösung 1: Objektorientiert
  - Basisklasse `Comparable` mit virtueller Vergleichsfunktion `less`
  - Definiere Elementtyp als Kindklasse und überlade `less`

- Lösung 2: Templates
  - Definiere globale `less`-Funktion und templatisiere `maxArg`
  - Spezialisiere `less` für Elementtyp

Objektorientiert

```
struct Comparable
{
    virtual bool less(Comparable &right);
};

struct Pair: public Comparable
{
    int a, b;
    bool less(Comparable &right) {
        return a < static_cast<Pair&>(right).a;
    }
};

Comparable * maxArg(Comparable *arr[], int size)
{
    Comparable *max = NULL;
    for (int i = 0; i < size; ++i)
        if (max == NULL || max->less(*arr[i]))
            max = arr[i];
    return max;
}
```

Templates

```
template <typename T>
inline bool less(T &left, T &right)
{
    return left < right;          // allgemein
}

template < >
inline bool less(Pair &left, Pair &right)
{
    return left.a < right.a;      // speziell
}

template <typename T>
T * maxArg(T *arr[], int size)
{
    T *max = NULL;
    for (int i = 0; i < size; ++i)
        if (max == NULL || less(*max, *arr[i]))
            max = arr[i];
    return max;
}
```

Vergleich

- Objektorientiert mit virtueller Funktion
  - Es existiert genau eine `maxArg`-Funktion
  - Indirektion beim Lesen der Funktionsadresse von `less`
  - Sprung und Rücksprung, Stackframe auf- und abbauen
- Templates und inline
  - Für jeden benutzten Elementtyp wird eine eigene `maxArg`-Funktion erzeugt
  - `less`-Funktion wird direkt in `maxArg` eingebaut (inline)

- Laufzeitmessung mit 1 Mrd. Elementen
  - 3350 ms mit OOP
  - 2150 ms mit Templates

BEMERKUNGEN ZUR P-AUFGABE

Bemerkungen zu Aufgabe 5

- Anwendung des Schubfachprinzips
  - Alle Reads haben gleiche Länge m und Mismatches k
  - Jede Read in k+1 Teilstücke teilen, $l = \lfloor m/k+1 \rfloor$
  - q-Grammen sollen nicht zu lange sein, $q = \min\{13, \lfloor m/k+1 \rfloor\}$
  - Falls l > q, die letzten l – q Stellen wurden nicht benutzt

- q-Grammen berechnen
  - ordValue.resize(256, alphabetSize); ordValue['A'] = 0; ordValue['C'] = 1; ...
  - Kein substr benutzen und die Teilstücke nicht speichern
  - Iteratoren von Start- und Endposition benutzen

- Duplikaten filtern
  - Die Endpositionen jedes Match speichern
  - Ein std::set pro Read (logarithmische Komplexität mit großen Konstanten)
  - Alternativ std::vector benutzen (lineare Komplexität mit kleinen Konstanten)

Hinweise zu Aufgabe 8

- Pattern Basisklasse zu spezialisieren

```
template <typename S, typename A = void>
struct Pattern {
    S & sequence;
    Pattern(S & sequence) : sequence(sequence) {};
};
```

- Funktion shift zu spezialisieren

```
template <typename S, typename A, typename T>
inline void shift(Pattern<S, A> & pattern, T & textIt) {
    ++textIt;
};
```

- Funktion find ruft shift und sollte nicht spezialisiert werden!

```
...
shift(pattern, textIt);
```