# 6. Templates

AlDaBi Praktikum

David Weese
WS 2010/11

# 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)

# Defaults für Template-Parameter

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

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

Pair < > p;
```

(nur für Template-Klassen)

- Nachfolgende Parameter müssen dann auch Defaults haben.

# Template Beispiele

- Frage: Wozu kann man die Argumente von Template-Klassen verwenden?
  - Beispiel 1: 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>
{
};
```

# 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];
}
```

# 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;
};
```

# „rekursive“ Templates

- Beispiel:

```
template <int I>
struct Tupel: Tupel <I-1>
{
};

template < >
struct Tupel<1>
{
};
```

# TEMPLATE SUBCLASSING

# Das Delegation-Problem

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

```
struct Auto
{
    void tanken() { std::cout << "Benzin"; ...}
    void starten()
    {
        if (Tankfuellung == 0) tanken();
        ...
    }
};

struct Solarmobil: Auto
{
    void tanken() { std::cout << "Sonne"; ... }
};
```

# Das Delegation-Problem (II)

- Was passiert hier ?

```
...
Solarmobil mob;
mob.starten();
```

- Ausgabe: Benzin (!)

- Grund:
  - Starten ist in der Basisklasse Auto definiert
  - Auto kennt die abgeleitete Klasse Solarmobil nicht

```
void starten()
{   ...
    tanken();    // this-Pointer hat Typ Auto *
}
```

# Virtuelle Funktionen

- Lösungsmöglichkeit: **`virtual`**

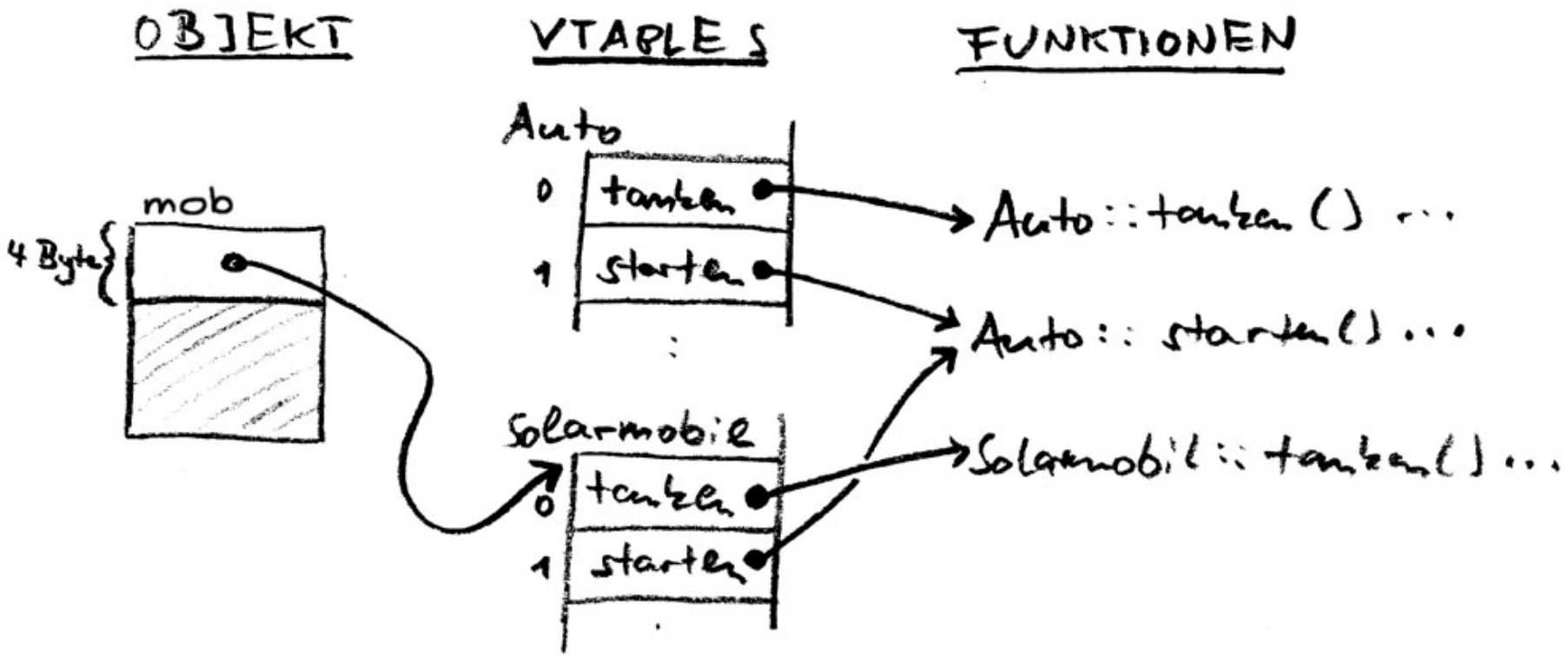
```
struct Auto
{
    virtual void tanken() { ...}
};

...

Solarmobil mob;
mob.starten();     // Ausgabe: "Sonne"
```

# Wie funktioniert `virtual`?

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



# abstrakte Basisklassen

- Abstrakte Klasse = hat pur virtuelle Funktionen

```
struct Abstract
{
    virtual void f() = 0;
};

struct Derived: Abstract
{
    void f() { /* implemented */ }
};

Abstract obj;  // Error!
Derived  obj;  // OK
```

# 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 Auto { /*Auto*/ };
struct Solarmobil: Auto { /*Solar*/ };
```

# Template Subclassing

- Schritt 1: “Template Spezialisierung statt Ableitung”

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

struct Solarmobil;

template <>
struct Auto <Solarmobil> { /*Solar*/ };
```

# Template Subclassing (II)

- Schritt 2: “Globale Funktionen statt Member Funktionen”

```
template <typename T>
void tanken (Auto<T> & obj)
{
    std::cout << "Benzin"; ...
}

void tanken (Auto<Solarmobil> & obj)
{
    std::cout << "Sonne"; ...
}

template <typename T>
void starten (Auto<T> & obj)
{
    tanken(obj); ...
}
```

# Template Subclassing (III)

- Template Subclassing löst das Delegation Problem:

```
Auto<Solarmobil> mobi;
starten(mobi);              // Ausgabe: "Sonne"
```

# 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

# Hinweise zu Aufgabe 6

- Es sollen C-optimale Schnittpositionen der Sequenzen $s_1,\ldots,s_k$ gefunden werden
- Herangehensweise:
  - Berechne Additional-Cost-Matrix für jedes Sequenzpaar mit Needleman-Wunsch in Vorwärts- und Rückwärtsrichtung
    - Rückwärts: Entweder Sequenz umdrehen (und Matrix umgedreht benutzen) oder DP-Algorithmus umdrehen
    - Nicht vergessen, von jedem Element in $D^f$ und $D^r$ den Score $c_{opt}(s,t)$ des normalen paarweisen Alignments abzuziehen
  - Aufzählen aller möglichen Schnittpositionstupel $c_1,\ldots,c_k$ mit $0<c_i<|s_k|$
    - vector<int> der Länge k zum Speichern eines Tupels
    - Rekursiv über i und Aufsummieren der Additional Costs von $s_i$ zu $s_j$ mit $j=1,\ldots,i-1$
    - Rekursionsabbruch bei Überschreiten des bisherigen Minimums
  - Parallele Rekursion
    - Threads teilen sich den Wertebereich $c_i=1,\ldots,|s_k|-1$ auf und zählen $c_2,\ldots,c_k$ rekursiv auf
    - Globale Variablen für das bisheriges Minimum der Additional Costs und zug. Schnittpositionen

# Branch and Bound

```
#include <limits>

vector<string> sequences;
int minimum = numeric_limits<int>::max();

void enumerate (vector<int> & pos, int i, int summe)
{
     if (summe >= minimum) return;                          // bound
     if (i == pos.size())
     {
          minimum = summe;                                  // neues minimum
          return;
     }
     for (pos[i] = 1; pos[i] < sequences[i].size(); ++pos[i])
     {
          int summand = ...;
          enumerate (pos, i + 1, summe + summand);          // branch
     }
}

int main(int argc, char *argp[])
{
     ...
     vector<int> p(k);
     enumerate (p, 0, 0);
     ...
}
```