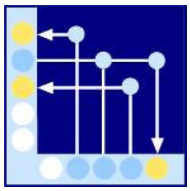
Hochschule Aalen
Fakultät Elektronik und Informatik
Studiengang Informatik



Programmieren in C++

Vorlesung im Wintersemester 2017/2018
Prof. Dr. habil. Christian Heinlein

8. Übungsblatt (11. Januar 2018)

Aufgabe 14: Untersuchung von Typen

Definieren Sie geeignete Schablonen sowie eventuell erforderliche Hilfsschablonen mit folgender Bedeutung:

- Für einen beliebigen Typ `T` ist `pointer_depth<T>` die Anzahl der „Zeigerebenen“ und `pointer_base<T>` der ggf. indirekte „Basistyp“ von `T`, zum Beispiel:

T	pointer_depth<T>	pointer_base<T>
int	0	int
int*	1	int
const int * volatile * const	2	const int
int* ()	0	int* ()

  Der Typ `int* ()` ist ein Funktionstyp mit Resultattyp `int*`. Deshalb ist `pointer_depth<int* ()>` gleich 0, obwohl der Resultattyp des Funktionstyps ein Zeigertyp ist.

- Für einen beliebigen Typ `T` ist `is_func<T>` genau dann `true`, wenn `T` ein Funktionstyp ist.
  In diesem Fall ist `result_type<T>` der Resultattyp und `param_count<T>` die Anzahl der Parameter des Funktionstyps `T`. Andernfalls sind `result_type<T>` und `param_count<T>` nicht definiert.

Beachten Sie die Erläuterung zum Schlüsselwort `typename` in §5.3.3 der Vorlesungsfolien!

✎

```
// Hilfstyp.
template <typename T>
struct pointer_help {
    // Sofern durch eine Spezialisierung nichts anderes definiert wird,
    // ist die Anzahl der Zeigerebenen von T gleich 0
    // und der Basistyp von T gleich T.
    static const int depth = 0;
    typedef T type;
};
```

```
// Spezialisierung für unqualifizierte Zeigertypen.
template <typename T>
struct pointer_help <T*> {
    // Die Anzahl der Zeigerebenen von T* ist die Anzahl der
    // Zeigerebenen von T plus 1.
    // Der Basistyp von T* ist gleich dem Basistyp von T.
    static const int depth = pointer_help<T>::depth + 1;
    typedef typename pointer_help<T>::type type;
};

// Spezialisierungen für const- und/oder volatile-qualifizierte
// Zeigertypen.

template <typename T>
struct pointer_help <T* const> : pointer_help<T*> {};

template <typename T>
struct pointer_help <T* volatile> : pointer_help<T*> {};

template <typename T>
struct pointer_help <T* const volatile> : pointer_help<T*> {};

// Schablonen zur angenehmeren Verwendung.

template <typename T>
int pointer_depth = pointer_help<T>::depth;

template <typename T>
using pointer_base = typename pointer_help<T>::type;
```

```
// Hilfstyp.
template <typename T>
struct func_help {
    // Sofern durch eine Spezialisierung nichts anderes definiert wird,
    // ist T kein Funktionstyp, und dementsprechend gibt es weder
    // Resultattyp noch Parameterzahl.
    static const bool flag = false;
};

// Spezialisierung für Funktionstypen.
template <typename R, typename ... PP>
struct func_help <R (PP ...)> {
    // R (PP ...) ist ein Funktionstyp mit Resultattyp R
    // und Parameterzahl sizeof...(PP).
    static const bool flag = true;
    typedef R type;
    static const int count = sizeof...(PP);
};

// Schablonen zur angenehmeren Verwendung.

template <typename T>
bool is_func = func_help<T>::flag;
```

```
template <typename T>
using result_type = typename func_help<T>::type;

template <typename T>
int param_count = func_help<T>::count;
```

✐

Aufgabe 15: Variadische Funktionen

Verallgemeinern Sie die Funktion `filter` aus Aufgabe 12 wie folgt:

- Anstelle einer Funktion kann auch ein Funktionsobjekt übergeben werden.
- Zusätzlich zu den bereits vorhandenen Parametern, können beliebig viele weitere Parameter mit beliebigen Typen übergeben werden, die – zusätzlich zum jeweiligen Listenelement – unverfälscht an die übergebene Funktion bzw. das Funktionsobjekt weitergegeben werden.

Definieren Sie außerdem eine Funktion `list` zur bequemeren Erzeugung von Listen, sodass `list(x1, x2, ...)` äquivalent zu `cons(x1, cons(x2, cons(...)))` ist!

Verwendungsmöglichkeit:

```
List<int> ls = list(1, 2, 3, 4, 5, 6);
bool mult_of (int x, int f) { return x % f == 0; }
filter(ls, mult_of, 3); // 3, 6
```

Hinweis: Verwenden Sie bei Bedarf `auto` als Resultattyp!

✎

```
#include <utility>
using namespace std;

// Code von Aufgabe 12.
#include "list.cxx"

// Prädikat p auf jedes Element der Liste xs anwenden und die Liste
// derjenigen Elemente liefern, für die das Prädikat erfüllt ist.
// Die optionalen Argumente aa ... werden unverfälscht an p weitergegeben.
template <typename X, typename P, typename ... AA>
List<X> filter (List<X> xs, P&& p, AA&& ... aa) {
    if (xs) {
        X x = head(xs);
        xs = filter(tail(xs), forward<P>(p), forward<AA>(aa) ...);
        return forward<P>(p)(x, forward<AA>(aa) ...) ? cons(x, xs) : xs;
    }
    else {
        return List<X>();
    }
}
```

```
// Liste mit Element x erzeugen.
template <typename T>
List<T> list (T x) {
    return cons(x);
}

// Liste mit Elementen x1, x2, xx ... erzeugen.
template <typename T1, typename T2, typename ... TT>
auto list (T1 x1, T2 x2, TT ... xx) {
    return cons(x1, list(x2, xx ...));
}
```

✎

Aufgabe 16: Funktionsobjekte

Ein Funktionsobjekt ist ein Objekt einer Klasse, die eine (oder eventuell auch mehrere) Elementfunktion(en) `operator()` besitzt, sodass das Objekt wie eine Funktion „aufgerufen“ werden kann, zum Beispiel:

```
// str als Abkürzung für const char*.
typedef const char* str;

struct StrHash {
    // Streuwert der Zeichenkette s wie bei
    // java.lang.String.hashCode berechnen.
    // (Beachte: Für vorzeichenlose Typen wie size_t
    // ist arithmetischer Überlauf wohldefiniert.)
    size_t operator() (str s) const {
        size_t h = 0;
        while (char c = *s++) h = h * 31 + c;
        return h;
    }
};

// Definition und Verwendung des Funktionsobjekts h.
StrHash h;
int i = h("abc"); // Bedeutet eigentlich: h.operator()("abc")
```

Ein Vorteil gegenüber normalen Funktionen besteht darin, dass ein Funktionsobjekt zusätzliche Daten speichern kann, die in der Elementfunktion `operator()` verwendet werden können, zum Beispiel:

```
struct StrHash {
    // Bei der Berechnung des Streuwerts verwendeter Faktor.
    size_t factor;

    // Faktor mit f initialisieren.
    explicit StrHash (size_t f = 31) : factor(f) {}
```

```
    // Streuwert der Zeichenkette s ähnlich wie
    // bei java.lang.String.hashCode berechnen.
    size_t operator() (str s) const {
        size_t h = 0;
        while (char c = *s++) h = h * factor + c;
        return h;
    }
};

// Funktionsobjekt h47 mit Faktor 47 erzeugen und verwenden.
StrHash h47 (47);
int i = h47("abc");
```

Definieren Sie geeignete Schablonen sowie eventuell erforderliche Hilfsschablonen, die wie folgt verwendet werden können:

- Für einen Wert `y` eines beliebigen Typs `Y` liefert `eq(y)` ein Funktionsobjekt `f`, sodass der Aufruf `f(x)` für einen Wert `x` eines beliebigen Typs `X` das Resultat des Vergleichs `x == y` liefert (sofern dieser Ausdruck typkorrekt ist). Analog für `ne` (not equal), `gt` (greater than), `ge` (greater than or equal), `lt` (less then) und `le` (less then or equal).

- Für einen Wert `y` eines beliebigen Typs `Y` liefert `add(y)` ein Funktionsobjekt `f`, sodass der Aufruf `f(x)` für einen Wert `x` eines beliebigen Typs `X` das Resultat der Addition `x + y` liefert (sofern dieser Ausdruck typkorrekt ist). Analog für `sub`, `mul`, `div` und `rem` (remainder).

- Für ein beliebiges Funktionsobjekt `f` liefert `neg(f)` ein Funktionsobjekt `g`, sodass der Aufruf `g(xx ...)` für beliebig viele Werte `xx ...` mit beliebigen Typen `TT ...` dasselbe Ergebnis liefert wie der Ausdruck `!f(xx ...)` (sofern dieser Ausdruck typkorrekt ist).

- Für zwei beliebige Funktionsobjekte `f1` und `f2` liefert `conj(f1, f2)` ein Funktionsobjekt `g`, sodass der Aufruf `g(xx ...)` für beliebig viele Werte `xx ...` mit beliebigen Typen `TT ...` dasselbe Ergebnis liefert wie der Ausdruck `f1(xx ...) && f2(xx ...)` (sofern dieser Ausdruck typkorrekt ist).
  Analog für `disj` (Disjunktion).

- Für zwei beliebige Funktionsobjekte `f1` und `f2` liefert `comp(f1, f2)` ein Funktionsobjekt `g`, sodass der Aufruf `g(xx ...)` für beliebig viele Werte `xx ...` mit beliebigen Typen `TT ...` dasselbe Ergebnis liefert wie der Ausdruck `f1(f2(xx ...))` (sofern dieser Ausdruck typkorrekt ist).

Verwendungsmöglichkeiten:

```
List<int> ls = list(1, 2, 3, 4, 5, 6);
filter(ls, gt(3))                       // 4, 5, 6
filter(ls, conj(ge(2), neg(gt(4))))     // 2, 3, 4
filter(ls, comp(eq(1), rem(2)))         // 1, 3, 5
```

Verallgemeinern Sie `conj` und `disj` dahingehend, dass sie auch auf mehr als zwei Funktionsobjekte angewandt werden können, zum Beispiel:

```
filter(ls, conj(ge(2), le(4), comp(eq(0), rem(3)))) // 3
```

Hinweise:

- Verwenden Sie bei Bedarf wiederum `auto` als Resultattyp!

- Lästige Codeverdopplungen können u. U. durch Makros vermieden werden.

✎

```
// Für einen Namen name (z. B. add) und ein zugehöriges Operatorsymbol
// oper (z. B. +) definiert aux(name, oper) eine Funktionsschablone
// name (z. B. add), die mit einem Wert y eines beliebigen Typs Y
// aufgerufen werden kann und als Resultat ein Objekt des Hilfstyps
// name_type<Y> (z. B. add_type<Y>) liefert, das den Wert y enthält.
// Die Elementfunktion operator() dieses Typs kann mit einem Wert x
// eines beliebigen Typs X aufgerufen werden und liefert als Resultat
// den Wert x oper y (z. B. x + y).
#define aux(name, oper) \
    template <typename Y> \
    struct name##_type { \
        Y y; \
        name##_type (Y y) : y(y) {} \
    \
        template <typename X> \
        auto operator() (X x) const { \
            return x oper y; \
        } \
    }; \
    \
    template <typename Y> \
    name##_type<Y> name (Y y) { \
        return name##_type<Y>(y); \
    }

// Verwendung von aux für Vergleichsoperatoren.
aux(eq, ==)
aux(ne, !=)
aux(gt, >)
aux(ge, >=)
aux(lt, <)
aux(le, <=)

// Verwendung von aux für arithmetische Operatoren.
aux(add, +)
aux(sub, -)
aux(mul, *)
aux(div, /)
aux(rem, %)

#undef aux
```

```
// Die Funktionsschablone neg kann mit einem Funktionsobjekt f eines
// beliebigen Typs F aufgerufen werden und liefert als Resultat ein
// Objekt des Hilfstyps neg_type<F>, das das Objekt f enthält.
// Die Elementfunktion operator() dieses Typs kann mit beliebigen
// Werten xx ... aufgerufen werden und liefert als Resultat den Wert
// !f(xx ...).
```

```
template <typename F>
struct neg_type {
    F f;
    neg_type (F f) : f(f) {}

    template <typename ... XX>
    auto operator() (XX ... xx) {
        return !f(xx ...);
    }
};

template <typename F>
auto neg (F f) {
    return neg_type<F>(f);
}

// Für einen Namen name (z. B. conj) und einen zugehörigen Ausdruck expr
// (z. B. f1(xx ...) && f2(xx ...) definiert aux(name, expr) eine
// Funktionsschablone name (z. B. conj), die mit zwei Funktionsobjekten
// f1 und f2 mit beliebigen Typen F1 und F2 aufgerufen werden kann und
// als Resultat ein Objekt des Hilfstyps name_type (z. B. conj_type)
// liefert, das die Objekte f1 und f2 enthält.
// Die Elementfunktion operator() dieses Typs kann mit beliebigen
// Werten xx ... aufgerufen werden und liefert als Resultat den Wert
// expr (z. B. f1(xx ...) && f2(xx ...)).
#define aux(name, expr) \
    template <typename F1, typename F2> \
    struct name##_type { \
        F1 f1; \
        F2 f2; \
        name##_type (F1 f1, F2 f2) : f1(f1), f2(f2) {} \
        \
        template <typename ... XX> \
        auto operator() (XX ... xx) { \
            return expr; \
        } \
    }; \
    template <typename F1, typename F2> \
    auto name (F1 f1, F2 f2) { \
        return name##_type<F1, F2>(f1, f2); \
    }

// Verwendung von aux für Konjunktion, Disjunktion und Komposition.
aux(conj, f1(xx ...) && f2(xx ...))
aux(disj, f1(xx ...) || f2(xx ...))
aux(comp, f1(f2(xx ...)))

#undef aux
```

```
// Für einen Namen name (z. B. conj) definiert aux(name) eine
// variadische Funktion name (z. B. conj) als Verallgemeinerung der
// gleichnamigen zweistelligen Funktion, die mit drei oder mehr
// Funktionsobjekten f1, f2, f3, ff ... aufgerufen werden kann.
#define aux(name) \
    template <typename F1, typename F2, typename F3, typename ... FF> \
    auto name (F1 f1, F2 f2, F3 f3, FF ... ff) { \
        return name(name(f1, f2), f3, ff ...); \
    }

// Verwendung von aux für Konjunktion und Disjunktion.
aux(conj)
aux(disj)

#undef aux
```