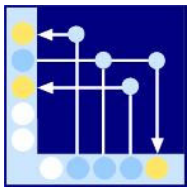
Fakultät Elektronik und Informatik
Studiengang Informatik



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!

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!

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.