Ludwig-Maximilians-Universität München
Mathematisches Institut
Gregor Kleen

Programmieren II für Studierende der Mathematik

Blatt 6 – Lösungsvorschlag

Aufgabe 6 Erstellen Sie eine Funktion `tabulate` ohne Rückgabewert mit den Parametern:

- eine Referenz auf einen Ausgabestrom,
- einen Namen vom Typ `string` und
- ein Funktionsobjekt für Funktionen $\{\bot, \top\} \times \{\bot, \top\} \to \{\bot, \top\}$, die Menge $\{\bot, \top\}$ identifizieren wir hierbei mit dem Typ `bool` (im Folgenden *zweistellige logische Verknüpfungen*).

Die Funktion soll auf dem gegebenen Ausgabestrom eine Wahrheitstafel (beschriftet oben links mit dem Namen) für die als Parameter übergegebene Funktion ausgeben.

tabulate

```
void tabulate(ostream &stream, string fn, function<BinBoolOp> f) {
  int w = max(1, static_cast<int>(fn.size()));
  stream << setw(w) << fn;
  for (bool y: {false, true})
    stream << setw(2) << y;
  stream << endl;
  for (bool x: {false, true}) {
    stream << setw(w) << x;
    for (bool y: {false, true}) {
      stream << setw(2) << f(x, y);
    }
    stream << endl;
  }
  Aufruf commutative
}
```

Rufen Sie im Hauptprogramm Ihre Funktion `tabulate` auf für die logischen Verknüpfungen $\wedge$, $\vee$, $\oplus$ und $\Rightarrow$ (logische Implikation).

Hinweis. Die Header-Datei `<functional>` stellt für $\wedge$, $\vee$ und $\oplus$ jeweils die Funktionsobjekte `bit_and<bool>()`, `bit_or<bool>()` bzw. `bit_xor<bool>()` zur Verfügung.

bool_functions.cpp

```
#include <iostream>
#include <iomanip>
#include <functional>
#include <algorithm>

using namespace std;

Gleichheitsoperator
```

commutative

tabulate

```
bool logical_implies(bool a, bool b) {
  return !a || b; // Äquivalent: a <= b
}

int main() {
  tabulate(cout, "&", bit_and<bool>());
  cout << endl;
  tabulate(cout, "|", bit_or<bool>());
  cout << endl;
  tabulate(cout, "^", bit_xor<bool>());
  cout << endl;
  tabulate(cout, "=>", logical_implies);
}
```

Überladen Sie den Gleichheitsoperator für Funktionsobjekte für zweistellige logische Verknüpfungen in mathematisch sinnvoller Weise.

Gleichheitsoperator

```
using BinBoolOp = bool(bool, bool);

bool operator==(function<BinBoolOp> f, function<BinBoolOp> g) {
for (bool x: {false, true})
    for (bool y: {false, true})
      if (f(x, y) != g(x, y))
        return false;
  return true;
}
```

Erstellen Sie eine Funktion `commutative` die für ein als Parameter übergebenes Funktionsobjekt für eine zweistellige logische Verknüpfung feststellt ob dieses kommutativ ist und geben Sie das Resultat als Rückgabewert vom Typ `bool` zurück. Verwenden Sie den von Ihnen überladenen Gleichheitsoperator für Funktionsobjekte für zweistellige logische Verknüpfungen und `bind` in geeigneter Weise.

commutative

```
bool commutative(function<BinBoolOp> f) {
  using namespace placeholders;
  return f == bind(f, _2, _1);
}
```

Erweitern Sie Ihre Funktion `tabulate` sodass diese nun auch ausgibt ob die jeweils betrachtete zweistellige logische Verknüpfung kommutativ ist.

Aufruf commutative

```
stream << "Kommutativ: " << commutative(f) << endl;
```

```
& 0 1
0 0 0
```

```
1 0 1
Kommutativ: 1

| 0 1
0 0 1
1 1 1
Kommutativ: 1

^ 0 1
0 0 1
1 1 0
Kommutativ: 1

=> 0 1
 0 1 1
 1 0 1
Kommutativ: 0
```

Aufgabe 7 In der Header-Datei `<algorithm>` ist die Funktion `for_each` definiert. Sie nimmt drei Parameter. Die ersten beiden Parametern sind Iteratoren und beschreiben ein Intervall $[i, i')$ eines Objektes eines Behälterdatentyps mit Elementen vom Typ T. Der dritte Parameter ist ein Funktionsobjekt für den Funktionstyp mit einem Parameter vom Typ T und keinem Rückgabewert, also Rückgabetyp `void`.

Lesen Sie in Ihrem Hautprogramm solange Werte vom Typ `double` von der Standardeingabe ein (mit Eingabeaufforderung) und hängen Sie diese an ein Objekt der Klasse `list<double>` hinten an, bis das Einlesen von der Standardeingabe fehlschlägt. Verwenden Sie dann die Funktion `for_each` und für dessen Parameter einen geeigneten Funktionsauruf von `bind` um das arithmetische Mittel der eingelesenen Werte zu berechnen und geben Sie diesen aus.

Hauptprogramm Eingabe

```
list<double> vals;
double x;
while (cout << "x: ", cin >> x)
  vals.push_back(x);
```

Hilfsfunktion foreach

```
void f(double& sum, double x) { sum += x; }
```

Hauptprogramm foreach

```
double sum = 0;
using namespace placeholders;
for_each(vals.begin(), vals.end(), bind(f, ref(sum), _1));
cout << "MW (for_each):   " << (sum / vals.size()) << endl;
```

In der Header-Datei `<numeric>` ist die Funktion `accumulate` definiert. Sie nimmt vier Parameter. Die ersten beiden Parameter beschreiben erneut ein Intervall $[i, i')$ eines Objektes eines Behälterdatentyps mit Elementen vom Typ T. Der dritte Parameter ist ein *Anfangswert* von beliebigem Typ T'. Der vierte Parameter ist ein Funktionsobjekt für den Funktionstyp mit zwei Parametern und T' als Rückgabetyp. Der erste Parameter des

Funktionstyps ist ebenfalls vom Typ T' und der zweite vom Typ T. Bei Aufruf der Funktion `accumulate` auf einen Behälter mit Inhalt (im gegebenen Intervall) $x_0, x_1, \ldots, x_n$, Anfangswert y_0 und Funktionsobjekt f wird als Rückgabewert von `accumulate` der Wert $f(\ldots f(f(y_0, x_1), x_2) \ldots, x_n)$ geliefert.

Erweitern Sie Ihr Hauptprogramm sodass dieses zusätzlich auch `accumulate` benutzt um erneut das arithmetische Mittel der eingelesenen Werte zu berechnen und geben Sie auch diesen Wert zur Kontrolle aus.

Hilfsfunktion accumulate

```
double g(int& count, double prev, double x) {
  int ncount = count + 1;
  double next = (prev * count + x) / ncount;
  count = ncount;
  return next;
}
```

Hauptprogramm accumulate

```
int count = 0;
double mean = accumulate(vals.begin(), vals.end(),
  0.0, bind(g, ref(count), _1, _2));
cout << "MW (accumulate): " << mean << endl;
```

foreach_mean.cpp

```
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
#include <numeric>

using namespace std;

  Hilfsfunktion foreach

  Hilfsfunktion accumulate

int main() {
    Hauptprogramm Eingabe

    Hauptprogramm foreach

    Hauptprogramm accumulate
}
```

```
x: 3
x: 1
x: 4
x: 0
```

```
x: 5
x: -1
x: 6
x: -2
x: 7
x: -3
x:
MW (for_each):   2
MW (accumulate): 2
```