Technische Universität München
Institut für Informatik I-16
Dr. Tobias Lasser
Jakob Vogel

WiSe 2012/13
Übungsblatt 3
29. Oktober 2012

Algorithmen und Datenstrukturen

Aufgabe 1 Boolsche Logik

In der Vorlesung haben Sie verschiedene logische Verknüpfungen und Wahrheitstabellen besprochen. Vergegenwärtigen Sie sich diese bitte nochmals.

a) In der Vorlesung wurde erwähnt, dass man mit NAND und NOR die Verknüpfungen NOT, OR und AND nachbauen kann. Demonstrieren Sie dies mithilfe von Wahrheitstabellen!

b) Zeigen Sie mit Hilfe von Wahrheitstabellen die Richtigkeit der De-Morgan-Gesetze!

c) Außerdem wurde in der Vorlesung eine Formel angegeben, mit der das XOR mittels NOT, OR und AND dargestellt wird. Beweisen Sie auch deren Richtigkeit mit einer Wahrheitstabelle!

Aufgabe 2 (P) Lazy Evaluation

Sie haben mittlerweile verschiedene Operatoren für logische Verknüpfungen in C/C++ kennen gelernt. Außerdem haben Sie erfahren, dass ganzzahlige Werte direkt als Wahrheitswerte interpretiert werden können, nämlich:

```
wert == true  ⟺  wert != 0  ⟺  wert
wert == false ⟺  wert == 0  ⟺  !wert
```

Dementsprechend können Sie für logische Verknüpfungen neben den eigentlichen logischen Operatoren `&&` und `||` auch die bitweisen Operatoren `&` und `|` einsetzen. Wir untersuchen nun den Unterschied zwischen beiden Varianten.

Schreiben Sie dazu zwei Funktionen, `int immerWahr()` und `int immerFalsch()`. Beide Methoden sollen auf `std::cout` ausgeben, dass sie aufgerufen wurden und dann entsprechend einen *immer wahren* oder einen *immer falschen* Ausdruck liefern.

Testen Sie nun, etwa in `if`-statements, verschiedene Kombinationen aus *wahr* und *falsch* mit den beiden UND-Operator-Varianten, also:

- `immerWahr() & immerFalsch()` im Vergleich zu `immerWahr() && immerFalsch()`
- `immerWahr() & immerWahr()` im Vergleich zu `immerWahr() && immerWahr()`
- `immerFalsch() & immerWahr()` im Vergleich zu `immerFalsch() && immerWahr()`
- `immerFalsch() & immerFalsch()` im Vergleich zu `immerFalsch() && immerFalsch()`

Führen Sie diesen Test auch mit den beiden ODER-Operatoren `|` und `||`. Was folgern Sie über die Unterschiede zwischen den jeweiligen zwei Varianten?

Aufgabe 3 (P) Objektorientierung II

Auf dem letzten Übungsblatt haben wir in zwei Aufgaben die Erzeugung von Arrays auf dem Heap mittels `new[]` und Objektorientierung über `struct` kennengelernt. Zwei besondere Funktionen bei Klassen waren Konstruktor und Destruktor, die wir nun genauer betrachten wollen. Verfassen Sie dazu eine Klasse mit folgendem Inhalt:

```
struct TestKlasse
{
    TestKlasse()
    {
        std::cout << "TestKlasse [" << this << "]: Konstruktor" << std::endl;
    }

    virtual ~TestKlasse()
    {
        std::cout << "TestKlasse [" << this << "]: Destruktor" << std::endl;
    }

    virtual void testFunktion()
    {
        std::cout << "TestKlasse [" << this << "]: testFunktion()" << std::endl;
    }
};
```

Wie Sie sehen, verfügt die Klasse über einen Konstruktor und einen Destruktor, sowie eine „normale“ Funktion. Alle drei geben nur aus, dass sie aufgerufen wurden. Die Spezial-Variable `this` gibt dabei aus, wo das Objekt selbst im Speicher liegt.

Wir schreiben nun zwei Funktionen, um den Programmablauf nachzuverfolgen. Zunächst verwenden wir eine lokale Instanz der Klasse `TestKlasse`:

```
void funktion1()
{
    TestKlasse t;
    std::cout << "Zeiger auf Objekt: " << &t << std::endl;
    t.testFunktion();
}
```

Hier wird also die lokale Variable `t` angelegt, dann ihre Speicheradresse (logischerweise auf dem Stack) ausgegeben[1] und schließlich die `testFunktion()` aufgerufen. Rufen Sie diese Funktion aus einer `main`-Funktion heraus auf, und verfolgen Sie die Ausgaben.

Vergleichbar zu einem Array können auch Objektinstanzen auf dem Heap angelegt werden. Das funktioniert dann äquivalent zur Erzeugung eines Arrays. Sie können das in der folgenden zweiten Funktion sehen:

```
void funktion2()
{
    TestKlasse* tZeiger = new TestKlasse;
    std::cout << "Zeiger auf Objekt: " << tZeiger << std::endl;
    tZeiger->testFunktion();
    delete tZeiger;
}
```

[1] Durch Vorstellen eines `&` kann immer die Speicheradresse einer Variablen erhalten werden.

Sie sehen also, dass mittels `new` eine einzelne Instanz der Klasse `TestKlasse` auf dem Heap erzeugt wird, die dann am Ende der Nutzung natürlich via `delete` wieder freigegeben werden muss. Außerdem liegt das Objekt jetzt nicht mehr direkt vor, sondern als Zeiger in den Speicher. Dementsprechend wird statt der Punkt-Syntax für Zugriffe auf Instanz-Variablen und -Funktionen ein symbolischer Pfeil verwendet, und man schreibt etwa `tZeiger->testFunktion()`. Rufen Sie nun auch diese Funktion aus der oben erstellten `main`-Funktion heraus auf, und verfolgen Sie die Ausgaben.