Praxisorientierte Einführung in C++
Lektion: "Anweisungen und Ausdrücke"

Christof Elbrechter

Neuroinformatics Group, CITEC

May 8, 2014

Table of Contents

Imperativ vs. Funktional

- Fast alle imperativen Sprachen haben das Konzept einer Anweisung
- Fast alle funktionalen Sprachen kennen nur Ausdrücke
- Das Berechnungsmodell, welches dahinter steht ist grundverschieden:
- Imperativ:
  - Der Computer ist ein Apparat mit einem Zustand (Speicherbelegung, Registerwerte). Anweisungen verändern diesen Zustand
  - Anweisungen haben oft Seiteneffekte neben dem Black-Box Modell einer mathematischen Funktion
  - Nahe an der Hardware (Maschinencode ist selber "imperativ")
- Funktional
  - Abstrahiert vom Zustandsmodell des Computers
  - Ausdrücke sind Seiteneffektfrei (meistens)
  - Weiter von der Hardware entfernt (Zustände machen einem das Leben schwer)

Anweisungen in C++

- Anweisungen sind die "Bausteine" eines imperativen Programms
- Programmfluss ist eine Folge von Anweisungen

```
int i = 1; int j = 4; int x; // Deklarationen

// Anweisungen:
x = i + j;
std::cout << i << " + " << j << " ergibt " << x << std::endl;

x = sqrt(i * i + j * j);
std::cout << "Und die Laenge des Vektors (i, j)^T ist " << x << std::endl;
```

- Grob gesprochen: Alle Zeilen (ausser Deklarationen), die mit ; enden sind Anweisungen
- Dazu kommen noch:
  - Kontrollstrukturen (`if-else`, `for`, `switch-case`, etc..)
  - Blöcke { ... }

Ausdrücke in C++

- Anweisungen sind zusammengesetzt aus Ausdrücken

```
1 + 3 + i + k + sqrt(1 + exp(3))
```

- Ausdrücke bestehen wiederum aus Operatoren und Operanden
  - Die Operanden selbst sind wiederum Ausdrücke
- Die grundlegenden (primären) Ausdrücke sind
  - *Literale*
    - Konstante Werte
  - Bezeichner
    - Variablennamen, Klassennamen, Namensräume, Funktionsnamen
  - `this`
    - Der Zeiger auf "eigene" Objektinstanz
  - `( Ausdruck )`
    - Verschachtelte Ausdrücke

r-values vs. l-values

- Ausdrücke, die ein Objekt im Speicher referenzieren sind sogenannte *l-value*-Ausdrücke
- Inspiriert ist diese Benennung durch Zuweisungsoperator. l-values können links vom Zuweisungsoperator stehen

```
int i; i = 4; // i ist hier l-value
```

- Kontraintuitiv: Auch konstante Objekte sind l-values
  - Man unterscheidet zwischen *veränderbaren* (modifiable) und *nicht veränderbaren* (non-modifiable) l-values
- Der Wert eines Ausdrucks ist sogenannte *r-value*

```
int i; int j = 4;
i = j; // j ist hier r-value
```

- Alle l-values können zu r-values konvertiert werden, aber nicht umgekehrt

Literale

- Literale sind konstante primäre Ausdrückem deren Wert direkt in ihrer Repräsentation kodiert sind
  - Ganzzahlen, Fliesskommazahlen, Buchstaben, Texte, die im Quelltext vorkommen

```
157     // Ganzzahlliteral
0xFE    // Ganzzahlliteral in hex-Schreibweise
'a'     // Characterliteral
"abc"   // Stringliteral
0.2     // Fliesskommaliteral
0.4f    // Fliesskommaliteral
```

Literale

- Literale sind konstante primäre Ausdrückem deren Wert direkt in ihrer Repräsentation kodiert sind
  - Ganzzahlen, Fliesskommazahlen, Buchstaben, Texte, die im Quelltext vorkommen

```
157     // Ganzzahlliteral
0xFE    // Ganzzahlliteral in hex-Schreibweise
'a'     // Characterliteral
"abc"   // Stringliteral
0.2     // Fliesskommaliteral
0.4f    // Fliesskommaliteral
```

- In C++-11 können sogar eigene Literale (gekennzeichnet durch ein Postfix) definiert werden
  - nur für bestimmte Typen
  - Postfix muss mit "_" (underscore) anfangen
  - Im Einzelfall schon ganz cool

```
int64_t timeMS = 6_h + 12_s + 46_ms;
```

Benutzerdefinierte Literale C++ 11

```
#include <stdint.h>
#include <iostream>
constexpr unsigned long long operator"" _ms(unsigned long long msec){
  return msec;
}
constexpr unsigned long long operator"" _s(unsigned long long seconds){
  return seconds * 1000_ms;
}
constexpr unsigned long long operator"" _m(unsigned long long minutes){
  return minutes * 60_s;
}
constexpr unsigned long long operator"" _h(unsigned long long hours){
  return hours * 60_m;
}
constexpr unsigned long long operator"" _d(unsigned long long days){
  return days * 24_h;
}
constexpr unsigned long long operator"" _y(unsigned long long years){
  return years * 356_d;
}
int main(){
  std::cout << "1 second:" << 1_s << std::endl;
  std::cout << "1 minute:" << 1_m << std::endl;
  std::cout << "1 hour:" << 1_h << std::endl;
  std::cout << "1 day:" << 1_d << std::endl;
}
```

Literale sind r-values

- Literale sind immer r-values. Sie können also höchstens rechts vom Zuweisungsoperator stehen

```
5 = 3; // Fehler: 5 ist Ganzzahlliteral
"abc"[3] = 'a'; // Fehler! "abc" ist Stringliteral
```

Literale - Ganzzahlliterale

- Zahlen *ohne* Dezimalpunkt sind Ganzzahlliterale und haben per Default den Typ `const int`
- Mit Suffix `L` oder `l` wird der Typ zu `long int`

```
long int i = 100l;
```

- In diesem Fall überflüssig, da automatisch umgewandelt wird

Literale - Alternative Schreibweisen

- Hexadezimalschreibweise mit 0x-Prefix:

```
int i = 0xAF; // Dezimal: 115
```

- Oktalschreibweise mit 0 (kein Ooooh, sondern Null)

```
int i = 010; // Dezimal: 8
```

- `unsigned` durch u-Suffix

```
unsigned int i = 10u;
```

- Das kann z.B. praktisch sein, wenn man einen `unsigned int` mit einem Wert initialisieren will, der ausserhalb des Wertebereichs eines `int` liegt

Literale - Fliesskommazahlen

- Wenn in einer Zahl ein Dezimalpunkt vorkommt, handelt es sich um ein Fliesskommaliteral

```
double x = 0.2;
```

- Der Default-Typ ist `const double`
- Durch Suffix `f` oder `F` kann `const float` Typ erzwungen werden

```
float x = 0.3f;
```

- Nützlich, wenn man Speicher sparen will (Embedded Applikationen, Mikrocontrollerprogrammierung etc. Da zählt manchmal jedes Byte)

Literale - Buchstaben (Character)

- Einzelne Zeichen werden durch Typ `const char` repräsentiert (Zeichensatz in der Regel ASCII oder eine Locale)
- Markiert durch "Single Ticks"

```
char x = 'x';
```

- Es gibt im ASCII Code auch sogenannte non-printable Characters
  - Carriage Return, Backspace, Newline, etc..
- Für diese existieren einige *Escape-Sequenzen*

```
\n  \t  \0  \'  \"
```

Literale - Buchstabenketten (Strings)

- String-Literale haben Typ `const char *const`
- Werden durch "Double-Ticks" markiert

```
const char *text = "Dies ist ein Text";
```

- Ende des Strings wird angezeigt durch ein *Null* Byte
- Wie der Typ schon sagt, kann man diese Strings nicht verändern:

```
const char *someString = "Dies ist ein String";

*someString = 'x';
someString[0] = 'x'; // Compiler-Error
```

Stringliterale - Alternative Schreibweise

- Alternative Schreibweise

```
const char *someString = "Hallo " "Welt";
```

- Dies ist äquivalent zu

```
const char *someString = "Hallo Welt";
```

- Nützlich z.B. für zusammengesetze Pfade

```
const char *globalConf = PREFIX "/share/myP/P.cnf";
```

Arrays und Strings

- Nahe Verwandte: Arrays

```
char someString[] = "Hallo";
```

- Hier wird Array mit Charactern initialisiert
- Array ist `char *const`, nicht `const char*`
- Kurzschreibweise für:

```
char someString[] = {'H','a','l','l','o','\0'};
char someString[] = {'H','a','l','l','o',0};
```

Literale - Wahrheitswerte

- `true` und `false` bezeichenen wahr und falsch
- Type ist `const bool`

```
bool x = true;
bool y = false;
```

Bezeichner

- Variablen-Bezeichner evaluieren entweder zu
  - Dem Speicher, den das bezeichnete Objekt belegt (l-value) oder
  - Dem Wert den das Objekt repräsentiert

```
int x, y;

x = 1; // x ist hier l-value, 1 ist r-value
y = x; // y ist l-value und x r-value
```

- Ähnlich für Funktionsbezeichner

Operatoren

- Stellen Möglichkeit dar, aus einfachen Ausdrücken komplexe zusammenzusetzen

```
foo(bar(x), 1+3)
```

Operatoren

- C++ kennt eine Vielzahl von Operatoren
  - Aritmethische, logische und bitweise Operatoren
  - Adreßoperator, Derefenzierungsoperator
  - Zuweisungsoperatoren
  - Operatoren für Speichermanagement
  - Funktionsaufrufoperator
  - Operatoren um Fehler anzuzeigen
  - Typumwandlungen
  - ...
- Wir besprechen (wahrscheinlich) nicht alle

Operatorpräzedenz und Klammern

- Operatoren haben verschiedene *Bindungsstärken* a.k.a. *Präzedenz*
- C++ kennt mehr als 10 verschiedene Präzedenzstufen
- Bei Unklarheiten klammern
- Klammern binden immer am stärksten

```
if (x == 3 && y == 5) {
// Code
}
```

- Hat hier == oder && Vorrang?

```
if ((x == 3) && (y == 5)) {
// Code
}
```

- In diesem Fall irrelevant, aber "guter Stil", da explizit

Arithmetische Operatoren

- Die üblichen binären Operatoren stehen zur Verfügung + - * /
- Punkt- vor Strichrechnung, wie gewohnt
- Unäre arithmetische Operatoren: + -

Pre- und Postinkrement und dekrementoperatoren

- `++ --`
- Operieren auf Ganzzahlwerten

```
int x = 0;
int y;

y = ++x;
y = x++;
x = --y;
x = y--;
```

- Der Unterschied liegt darin, wann der Ausdruck ausgewertet wird:
  - Vor der Zuweisung (`++x`)
  - Nach der Zuweisung (`x--`)

Implizite Typumwandlung

- Bei gemischten arithmetischen Ausdrücken werden Operanden in den Typ des Operanden mit höchster Präzision umgewandelt

```
int x = 2;
float y = 3.3;

float z = x * y; // x wird erst nach float konvertiert
                 // und dann wird 2.0 * 3.3 ausgewertet
```

- Aber Vorsicht:

```
int x = 2;
int y = 3.3;

float z = x * y; // Das Ergebnis wird nicht das
                 // sein, was man erwartet
```

- Zum Glück warnt der Compiler (`-Wall`)

Logische Operatoren

- Operieren auf boolschen Ausdrücken
- Logisches und, oder, nicht: `&&`, `||`, `!`
- Werden zu Wert vom Typ bool ausgewertet
- Können z.B. mit `if-else`-Abfrage getestet werden (später mehr)

```
bool x = true;
bool y = false;

if (x && y) {
// wird niemals augefuehrt
}
```

Vergleichsoperatoren

- Test auf Gleichheit ==
  - Nicht =. Das ist eine beliebte Fehlerquelle

```
int x = 1, y = 2;

if (x == y) {
// Code
}
```

- Liefert bool zurück
- Vorsicht bei Zeigern: Überprüft, ob die Zeiger gleich sind und nicht, ob die Werte gleich sind
- Expertentipp: Zu vergleichenden Wert immer nach links:

```
bool x = something();
if (x = true) { ... } // x wird auf true gesetzt und dann getestet
if (true = x) { ... } // Compiler-Fehler
// oder in diesem Fall eigentlich besser gleich
if(x) { ... }
```

Bitweise Operatoren

- Arbeiten direkt auf der Bitrepräsentation des Werts
- Bitweises und, oder, nicht, eklusives oder:

```
&   |   ~   ^
```

- Bitverschiebeoperatoren: << >>

```
int x = 1; // Bit 0 gesetzt, da 2^0 = 1

x = x | 6; // setzt Bits 1 und 2, da 2^1 + 2^2 = 6
           // x hat nun den Wert 7

x = x & 8; // 8 = 2^3, also nur Bit 3 gesetzt.
           // Daher hat x nun Wert 0
```

?: - Operator

- Ähnliche Funktionalität wie `if-else` (später)
- Wenn Bedingung `true`: `Ausdruck1`
- Sonst: `Ausdruck2`

```
Bedingung ? Ausdruck1 : Ausdruck2
```

- Das ist die einzige Möglichkeit, dynamisch Referenzen zu initialisieren

```
int kurzeHose, langeHose;
int &hose = (schoenesWetter() ? kurzeHose : langeHose);
```

- `Ausdruck1` und `Ausdruck2` müssen gleichen Typ haben
- Der Gesamtausdruck hat dann wiederum den gleichen Typ

Zuweisungsoperator

- Zuweisungen werden mit = ausgeführt

```
Ausdruck1 = Ausdruck2
```

- `Ausdruck1` muss veränderbare l-value sein
- Zuweisungsausdruck wird i.d.R. zu Referenz auf den linken Operanden ausgewertet

```
int x;

x = 1; // Ausdruck hat Wert 1, also koennen wir
       // direkt eine weitere Zuweisung machen:

int y;

y = x = 1; // Explizit geklammert: y = (x = 1)
```

- Erlaubt Verkettung von Zuweisungen

Zuweisung mit gleichzeitiger Operation

- Zuweisung mit arithmetischer/bitweiser Operation:

```
*=  /=  %=  +=  -=  >>=  <<=  &=  ^=  |=
```

- Erlaubt kürzeren Code:

```
int x = 0;
x += 1;
x |= 0xFF;

// aequivalent zu:

x = x + 1;
x = x | 0xFF;
```

Zuweisungsoperatoren - Anmerkung

- Empfohlene-Semantik für Zuweisungs-operator ist tiefe Kopie
- Kann für benutzerdefinierte Klassen und Strukturen frei definiert werden
- Abweichung von der empfohlenen Semantik ist aber immer gefährlich (...)
  - Muss immer gut dokumentiert werden!

Funktionsaufrufsoperator

Syntax

```
Ausdruck(Argument1, ..., ArgumentN)
```

- `Ausdruck` muss Typ einer Funktion haben (oder eines Funktors - später mehr)
- Argumente müssen passende Typen haben (oder konvertiert werden können)
- Beispiel eines Funktionsaufrufs:

```
atan2(x, y)
```

- Zu Funktionen später mehr

Operatoren auf benutzerdefinierten Typen

- C++ erlaubt, die meisten Operatoren zu überladen
- Beispiel: +-Operator für Matrizen
- Semantik kann sich dadurch ändern
- Sehr mächtiges Werkzeug
- Kann Lesbarkeit des Codes erheblich verbessern
- Kann aber auch alles sehr schlimm machen
- Später mehr dazu

Blöcke

- Blöcke werden durch {} begrenzt
- Bilden einen eigenen Namensraum
- Lokale Variablen werden am Ende des Blocks zerstört

```
int x;

int main() { // Blockanfang
  int y;

  { // "freier" Block
    int z;
    // hier sind x und y und z sichtbar
  } // hier verschwindet z

  // hier sind nur noch x und y sichtbar
}
```

- Das kann sehr nützlich sein, z.B. in `switch-case`-Anweisungen (später)