Praxisorientierte Einführung in C++
Lektion: "Funktionen (aber keine Methoden)"

Christof Elbrechter

Neuroinformatics Group, CITEC

April 24, 2014

Table of Contents

Allgemeines

- Funktionen erlauben Kapselung von Code
- Erhöht...
  - ...die Wiederverwendbarkeit von Code
  - ...die Lesbarkeit (wenn gut benamst)

Deklaration

- Gibt dem Compiler den Typ der Funktion bekannt
- Meistens in Header-Datei
- Manchmal aber auch im Source

```
int add(int n1, int n2);
```

- Allgemein:

```
Typ1 Bezeichner(Typ2 [name1], Typ3 [name2], ... );
```

- Argumentnamen `name1`, `name2`, etc. sind optional
  - Schlechter Stil, da Argumentnamen im besten Fall auch Dokumentation sind

Definition

- Statt ; folgt Funktionsrumpf

```
int add(int n1, int n2) {
  return n1 + n2;
}
```

- Für jede benutzte Funktion muss *genau eine* Definition vorliegen und übersetzt werden (ansonsten gibt es Linker-Fehler)
- Argumente, die in Definition benutzt werden sollen, *müssen* benamst werden

```
int add(int, int) {
  return n1 + n2;
}
```

- Das gibt einen Compilerfehler, denn `n1` und `n2` sind dem Compiler hier nicht bekannt

Pass by Value

- In C++ werden Argumente per Default als Wert übergeben, nicht als Referenz
- Das heißt i.d.R., dass die Argumente *kopiert* werden

```
struct Image { /* Megabytes an Daten */ };

void inspect_image(Image im) {
  // Inspiziere Daten
}

int main() {
  Image image;

  // Hier wird Kopie(!) von image erstellt
  inspect_image(image);
}
```

- Auswege: Zeiger oder Referenzen

Ein Beispiel: swap()

- Gewünscht: Funktion, die den Inhalt zweier Variablen austauscht

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 3;
  int b = 5;
  swap(a, b);

  // Hier soll nun in a 5 stehen und in b 3
}
```

- Funktioniert offensichtlich nicht

Pass by Reference - Zeiger

- Swap mit Zeigern

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 3;
  int b = 5;
  swap(&a, &b);
}
```

- Funktioniert, ist aber fehlerträchtig (`swap(0,0)`, oder `swap(&a + 1, &b +1)`

Pass by Reference - Referenzen

- Swap mit Referenzen

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 3;
  int b = 5;
  swap(a,b);
}
```

- Hinweis auf Codeduplikation wenn man `swap()` für verschiedene Typen schreibt - motiviert Funktionstemplates

Pass by Reference - Diskussion

- An Funktionen übergebene Daten müssen nicht "tief" kopiert werden
- An Funktionen übergebene Daten können verändert werden
  - Falls nicht, so können auch konstante Referenzen verwendet werden
- Anmerkung: Vieles ist auch durch Pointer möglich

Faustregel Referenzen vs. Pointer

- Gute Frage: Wann nimmt man Pointer? – wann nimmt man Referenzen?

Faustregel

Bevorzuge Referenzen !

- In einigen Fällen benötigt man optionale Funktionsargumente
- In diesem Fall bieten sich Pointer an (NULL-Pointer bedeutet: Argument nicht verwendet)
- Das geht auch irgendwie mit Referenzen; das ist aber meist sehr viel Komplizierter

Anmerkung zur Komplexität

- Referenzen werden intern auch als Pointer übergeben
- Effizienz ist also equivalent zu Pointern
- Für kleine Standardtypen (z.B. `int` oder `float`) ist i.d.R. die Übergabe per Value nicht weniger effizient

Rückgabewerte

- Funktionen geben Ergebnis zurück mithilfe der `return` Anweisung

```
return Ausdruck;
```

- `Ausdruck` kann beliebiger Ausdruck sein, der den richtigen Typ hat

```
int add(int n1, int n2) {
  return n1 + n2;
}
```

- `n1 + n2` ist hier rvalue vom Typ `int`

Kein Rückgabewert (`void`)

- Wenn eine Funktion kein Resultat zurückgibt hat sie als Typ für den Rückgabewert `void`

```
void print_array(int array[], unsigned int length) {
  for (unsigned int i = 0; i < length; ++i)
    std::cout << array[i] << " ";

  std::cout << std::endl;
}
```

- Nützlich für Funktionen, die *nur* Seiteneffekte haben
- Seiteneffekte sollten *immer* gut dokumentiert werden, sonst wird Code schwer verständlich

Funktionsüberladung

- In C mussten sich zwei verschiedene Funktionen durch Namen unterscheiden
  - Dadurch mussten sich Programmierer oft unterschiedliche Namen für ansonsten ähnliche Funktionen ausdenken
- In C++ reicht es, wenn sie verschiedene Argumentlisten haben

```
int f();                // Eine Funktion
int f(int i);           // Eine andere
int f(float i);         // Eine dritte
int f(int i, float j); // Eine vierte
```

- Ein unterschiedlicher Rückgabetyp reicht nicht

```
int x();
float x(); // Fehler! x schon deklariert
```

Funktionsüberladung

- Der Compiler findet anhand der Typen der übergebenen Argumente heraus, welche Funktion aufgerufen werden soll

```
#include <cmath>
#include <iostream>

float logarithm(float val, float base){
  return log(val)/log(base);
}
float logarithm(float val){
  return log(val)
}

int main() {
  std::cout << "log(10) = " << logarithm(10) << std::endl;

  std::cout << "log(10) zur Basis 2 = " << logarithm(10,2) << std::endl;
}
```

Funktionsüberladung

- Wenn Mehrdeutigkeiten entstehen meckert der Compiler

```
int add(int a, int b){
  return a+b;
}
float add(float a, float b){
  return a+b;
}

int main(){
  int a = add(4,5);
  int b = add(4.2,8.1); // mehrdeutig (doubles)!
  int c = add(4.2f,8.1f); //ok
}
```

Standardargumente

- Im `logarithm`-Beispiel haben wir zwei Funktionen gleichen Namens
- `logarithem(float,float)` ist eine Verallgemeinerung von `logarithm(float)`
- Code-Verdopplung (fehlerträchtig, arbeitsintensiv insb. bei Änderungen)

```
float logarithm(float val, float base){
  return log(val)/log(base);
}
float logarithm(float val){
  return log(val) // basis e
}
```

Standardargumente

- C++ erlaubt eleganteren Weg: Standardargumente

```
#include <cmath>
float logarithm(float val,double base=M_E){
  return log(val)/log(base);
}
int main(){
  float ld8 = logarithm(8,2);
  float ln8 = logarithm(8);     // Basis e wird
                                // automatisch
                                // eingesetzt
}
```

Standardargumente - Syntax

```
RueckgabeTyp Bezeichner(T1 Bez1,
                        ...
                        TN BezN = Def1,
                        ...
                        TM BezM = DefK); //K = M-N
```

- Funktionsargumente können nur von hinten an kontinuierlich mit Standardwerten belegt werden
- Ermöglicht eindeutige Zuordnung
- Standardargumente werden bei *Deklaration* angegeben
- Kann zu Mehrdeutigkeiten bei Funktionsaufruf von überladenen Funktionen führen

```
void f();
void f(int x = 0);
```

Standardargumente

- Kann evtl. zu Verlangsamungen führen (im Vergleich zu zwei separaten Funktionen), wenn der Compiler nicht das Standardargument "wegoptimieren" kann
  - Es ist ja eine Konstante
- Das `logarithm`-Beispiel könnte *deutlich* langsamer sein, da der Compiler nicht unbedingt weiss, dass `log(M_E)` immer das gleiche Ergebnis (1) liefert
- Auch hier könnte der Compiler einiges optimieren:

```
#include <iostream>
int sum(int a,int b=0,int c=0,int d=0,int e=0){
  return a+b+c+d+e;
}

int main() { std::cout << sum(1,2,3) << std::endl; }
```

- Profile first, optimize later

Sichtbarkeit von Variablen

- Funktionen bilden lokalen Namensraum
- Variablen im "innersten Scope" haben Vorrang

```
const char *msg = "Hallo Welt!";

void greet_world() {
  const char *msg = "Hello World";
  std::cout << msg << std::endl;
}

int main() {
  const char *msg = "Meeh!!";
  greet_world();
}
```

- Dieses Beispiel schreibt `Hello World` auf die Konsole

Lebenszeit von Variablen

- Nicht `static`-Variablen (kommt gleich) leben bis zum Ende der Funktion

```
void f() {
  int i = 0;
  Image img(640, 480);

  // Hier wird irgendwas mit dem Image gemacht
  // ...

} // Hier wird automatisch aufgeraeumt
```

- Destruktor (später) von zusammengesetzten Typen wird aufgerufen
- Platz auf dem Stack wird freigegeben

Einschub: Lebenszeit von Variablen in Blöcken

- Das gleiche gilt auch für lokale Blöcke

```
int main() {
  int i = 0;                      // int 0 auf Stack schieben

  {
    int i = 1;                    // int 1 auf Stack schieben
    std::cout << i << std::endl; // 1 ausgeben
  }                               // Platz fuer 1 freigeben
}                                 // Platz fuer 0 freigeben
```

`static` Variablen in Funktionen

- `static` hat im Funktionenblock andere Bedeutung als für globale Variablen
- Bedeutet:
  - Variable wird nicht am Ende der Funktion gelöscht
  - Wird nur einmal (beim ersten *Durchlauf*) initialisiert
  - Behält den Wert zwischen Funktionsaufrufen bei
- Vorsicht bei multithreading

static Beispiel

- Kann z.B. genutzt werden, um Funktionsaufrufe zu zählen

```
void foo(){
  static int call_counter = 0;
  std::cout << "Function " << __FUNCTION__
            << " was called " << call_counter++
            << " times" << std::endl;
  // Implementation ...
}
```

extern

- `extern` ist Default bei Funktionsdeklarationen
  - D.h. Wenn Definition nicht zur Verfügung steht wird auch in anderen Translation-Units geschaut (Link Time)
- `static` verhindert dass Funktion in anderen Translation-Units sichtbar ist
  - Erlaubt Hilfsfunktionen mit gleichen Namen, aber unterschiedlichen Funktionalitäten in unterschiedlichen Source-Files zu haben

inline

- Funktionen können `inline` deklariert werden

```
inline double power(double base, unsigned int exponent) {
  double result = 1;
  for (unsigned int i = 0; i < exponent; ++i){
    result *= base;
  }
  return result;
}
```

- Der Kompiler kann dann Funktionsaufrufe direkt durch den Rumpf der Funktion ersetzen
  - Muss er aber nicht :-)
- Dazu muss (natürlich) immer der Rumpf verfügbar sein
- Deshalb werden `inline` Funktionen i.d.R direkt im Header definiert

`inline` - Diskussion

- Kann Code erheblich beschleunigen, da Spungbefehle und Übergabe der Funktionsargumente entfallen
- Führen aber zur Effektiven Code-vervielfachung (Code-Bloat), was die Übersetzungszeit erhöht
- In Headern definierte Funktionen müssen immer wieder übersetzt werden
- Gut geeignet für
  - "Getter" und "Setter" - Methoden von Klassen
  - Sehr kurze Hilfsfunktionen
- Aber
  - Ob eine Funktion dann wirklich *geinlined* wird, entscheidet der Kompiler abhängig von Optimierungsflags und Codeanalyse
  - Auch nicht-`inline`-Funktionen, deren Implementation verfügbar ist, können *geinlined* werden

inline - Beispiel aus Bildverarbeitung

```
#define W 640
#define H 480
typedef int Image[W*H];

inline int &pix(Image &a, int x, int y){
   return a[x+W*y];
}
```

- inline kann Pixelzugriff deutlich beschleunigen (ca. 25%)
- Tatsächlicher Geschwindigkeitsvorteil ist stark compiler- und optimierungsabhäng.
  - Und auch von der Anwendung

Wann wird Was `inline` übersetzt

- Falls nur Funktionsdeklaration zur Verfügung steht: niemals `inline`
- Falls Funktionsdeklaration zur Verfügung steht: Compiler entscheidet
- `inline`-Deklaration ohne direkte Definition: Fehler
- Compiler achtet nicht zwangsweise auf das `inline` Schlüsselwort für die `inline`-heit
- `inline`-Funktionen erzeugen aber nie ein extern sichtbares Symbol
- Compiler verwendet Heuristiken um zu entscheiden, ob Funktion ge-*inlined* wird
- GCC: `__attribute__ ((always_inline))` erzwingt `inline`-heit[1]

```
__attribute__ ((always_inline)) inline int sign (float x) {
  return x >= 0 ? 1 : -1;
}
```

[1] Es gibt noch weitere Attribute, die wir später besprechen werden

Funktionszeiger

- Es können auch Zeiger auf Funktionen definiert werden
- Interessant:
  - Welchen Typ hat ein Funktionszeiger
  - Relativ kompliziert
- Kann aber in Einzelfällen sehr elegante Lösungen ermöglichen

Beispiel

```
int sub(int x, int y){
  return x - y;
}

int add(int x, int y){
  return x + y;
}

int main() {
  int (*fp)(int, int) = add;
  int a = fp(1,2);
  fp = sub;
  a = fp(1,-2);
}
```

Funktionszeiger und typedef

- Funktionszeigertypen können auch mittels typedef definiert werden
- Syntax (eher unintuitiv):

```
typedef Rueckgabetyp (*NeuerBez)(Parameterliste);
```

- Hierbei werden i.d.R. nur die Typen der Parameterliste mit angegeben

```
typedef int (*BinaryIntFunc)(int,int);
typedef int (*UnaryIntFunc)(int i);
typedef void (*UnaryInplaceFunc)(int&);
```

Funktionszeiger als Argumente

- Funktionen können auch als Argumente übergeben werden
- Dabei explizite Typnamen mittels `typedef`

```
typedef void (*UnaryInplaceFunc)(int&);
void for_each(int *array, int len, UnaryInplaceFunc f){
  ...
}
```

- Oder implizite Typedefinition

```
void for_each(int *array, int len, void (*f)(int&)){
  ...
}
```

Funktionsargumente - Beispiel

```
typedef void (*UnaryInplaceFunc)(int&);

void for_each(int array[], int len, UnaryInplaceFunc f){
  for(int i=0;i<len;++i){
    f(array[i]);
  }
}
void set2(int &x){ x=2; }
void mul2(int &x){ x*=2; }
void add5(int &x){ x+=5; }
void print(int &x){
std::cout << x << " ";
}

int main(){
  int array[100];
  for_each(array,100,set2);
  for_each(array,100,mul2);
  for_each(array,100,add5);
  for_each(array,100,print);
}
```

Funktionszeiger und Optimierungen

- Wenig Optimierungsmöglichkeiten für den Compiler
- In jedem Schritt Dereferenzierung des Funktionszeigers notwendig (also kein `inline`)
- Im Beispiel: Relativ speziell
- Besser: Templates (später)

Funktionszeiger und Optimierungen

- Wenig Optimierungsmöglichkeiten für den Compiler
- In jedem Schritt Dereferenzierung des Funktionszeigers notwendig (also kein `inline`)
- Im Beispiel: Relativ speziell
- Besser: Templates (später)
- C++-11: Anonyme Funktionen mit Lambda-Expressions
  - in Kombination mit Templates: **super-mächtig**
  - kombiniert Freiheiten funktionaler Programmierung mit effizienter Compile-Time Optimierung
  - das ist eigentlich **das** Feature von C++

Vorausschau C++-11: lambda

```
#include <algorithm>
#include <iostream>
int main(){
  int is[5] = { 1, 2 ,3, 4, 5};
  std::for_each(is,is+5,[](int &x){x*=2;});
  std::for_each(is,is+5,[](int x){std::cout << x << " ";});
}
```