Herzlich willkommen!

Dozent: Dipl.-Ing. Jürgen Wemheuer

Teil 5: Funktionen

Mail: wemheuer@ewla.de
Online: http://cpp.ewla.de/

Funktionen

- Eine Funktion in C++ ist ein Unterprogramm - zu einer (mehrfach wiederholten) Nutzung im Hauptprogramm main()
- Ein Unterprogramm bekommt bei seinem Aufruf optional Eingangsdaten (Parameter) übergeben, verarbeitet die Daten und gibt optional einen Wert zurück
- Ein jedes C++-Programm hat mindestens immer die eine Funktion:

main()

Funktionsaufruf

- Jede Funktion hat einen eigenen Namen, unter dem sie und nur sie aufrufbar ist
- Bei Aufruf einer Funktion verzweigt das Programm in seiner Ausführung zum Rumpf der entsprechenden Funktion
- Bei Rückkehr aus der Funktion wird das Programm mit der nächsten Anweisung nach dem Funktionsaufruf fortgesetzt

Funktionsaufruf

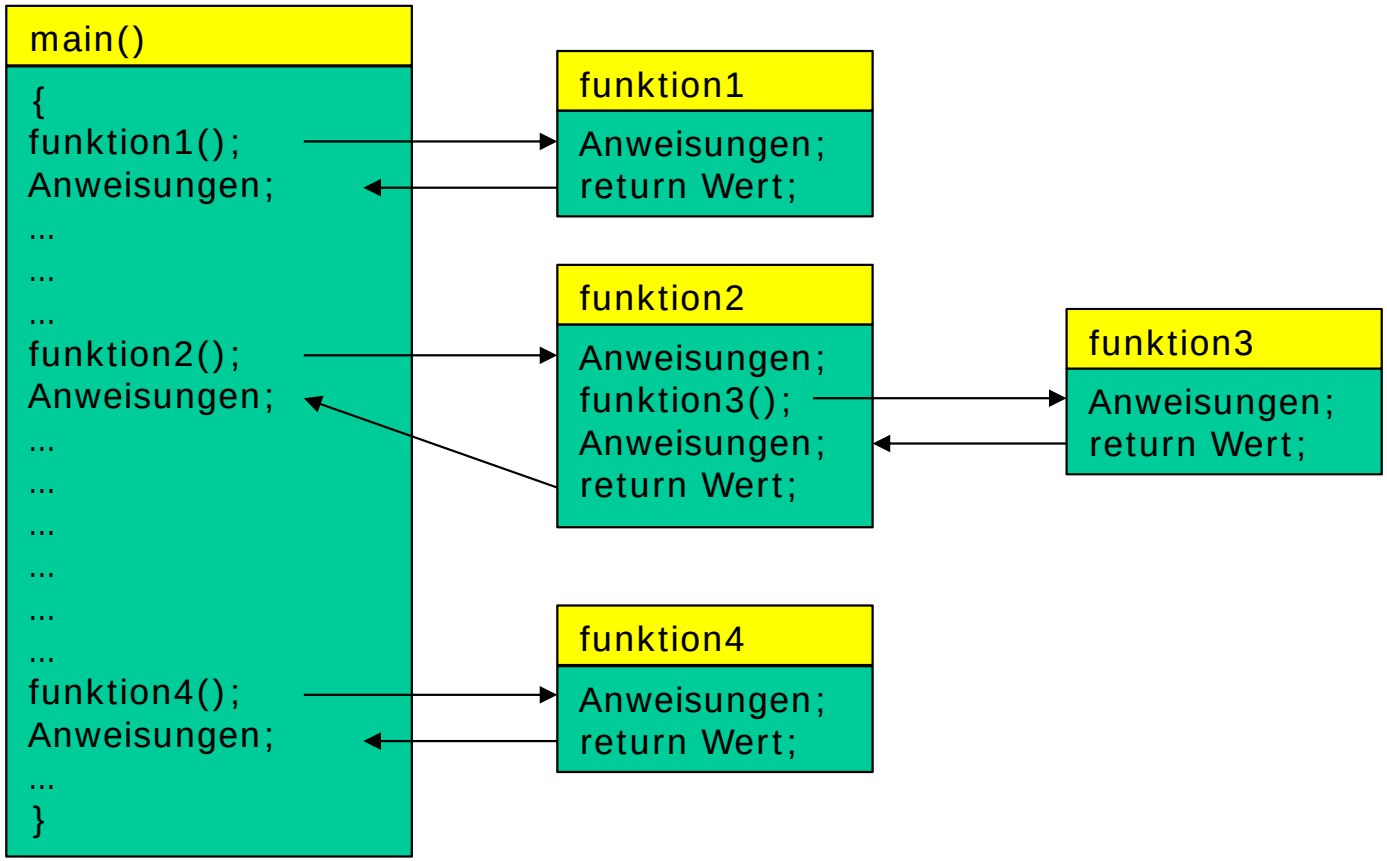

Bild:
Grafische Darstellung der Funktion in der Funktion

Sinn von Funktionen

- Mehrfache Verwendung eines Quelltextes für wiederkehrende Aufgaben in einem Programm
- Gliederung des Programms in einer Struktur
  - Eine Funktion sollte nur einen (klar definierten und abgegrenzten) Leistungsumfang haben
  - Ein komplexer Funktionsumfang sollte immer in einfache Aufgaben zerlegt und jede Aufgabe sollte als getrennte Funktion programmiert werden

Funktions-Deklaration / -Definition

- Eine Funktion muss *deklariert* und *definiert* werden und kann erst dann *aufgerufen* werden
- ***Deklaration*** einer Funktion:
  Sie teilt dem Compiler den *Namen*, den *Rückgabetyp* und die *Parameter* mit
- ***Definition*** einer Funktion:
  Sie teilt dem Computer die Arbeitsweise mit, sie ist also der Quellcode dieser Funktion
- Eine Funktion ist nur dann aufrufbar, wenn sie **vorher** deklariert worden ist

Funktionsdeklaration mit Prototyp

- *Prototyp* einer Funktion:
  Er ist die *Deklaration* einer Funktion.
  Er ist eine Anweisung, die mit einem Semikolon endet! ~> Mit der Angabe des Prototyps wird eine Funktion so deklariert:

```
Rückgabetyp   Name          (Parameter)      ;
long          MeineFunktion (int Par1, int Par2);
```

-> muss VOR der main()-Funktion stehen!

Funktionsdefinition

- Das Definieren der Funktion bedeutet:
  Editieren von:
  - Funktionskopf und
  - Rumpf = Programmcode der Funktion,
    also das, was die Funktion eigentlich „machen soll“
- Der **Kopf** der Funktion ist gleich ihrem Prototyp, wobei die Parameter hier bekannt sein müssen, er wird nicht mit einem Semikolon abgeschlossen
- Ein Block von Anweisungen, in geschweifte Klammern gesetzt, bildet den **Rumpf** der Funktion

Funktionsdefinition - Beispiel

Rückgabetyp Name Parameter

```
int MeineBilanz(int Haben,int Soll)
{     // öffnende geschweifte Klammer
      // Anweisungen
      // Schlüsselwort / Rückgabewert
      return (Haben – Soll);
}     // schließende geschweifte Klammer
```

(Die Anweisungen innerhalb der geschweiften Klammern bilden den Funktionsrumpf)

Zusammenfassung bisher

- Der **Funktionsprototyp** übergibt dem Compiler:
  1. den Namen,
  2. den Datentyp des Rückgabewerts und
  3. die Parameter der Funktion,
- Die **Funktionsdefinition** teilt als Quellcode dem Compiler mit, wie die Funktion arbeitet
- Der Prototyp und die Definition einer Funktion kommen nur einmal in einem Programm vor – der Funktionsaufruf kann aber so oft wie erforderlich vorkommen

Rückgabewerte / void

- Der Rückgabetyp und die Parameterliste des Prototyps müssen genau mit denen der Definition übereinstimmen
- Jede Funktion hat einen Rückgabetyp:
  Brauchen wir keinen Rückgabewert,
  setzen wir den Rückgabetyp auf **void**
- Liefert die Funktion einen Wert zurück, wird dieser mit **return** an die aufrufende Stelle übergeben
- Mit der return-Anweisung sollte eine Funktion also enden, obwohl diese Anweisung überall im Rumpf einer Funktion stehen darf

Beispiel einer Funktion

1. Der Prototyp:

```
unsigned long int AnzahlBytes(int Bit, long int RAM);
```

2. Die Definition:

```
unsigned long int AnzahlBytes(int Bit, long int RAM)
{
        return Bit x RAM;
}
```

Definition und Deklaration zugleich

Es ist zulässig, bei der Deklaration auch gleich den Rumpf mit zu notieren:

```
void Muenzeinwurf(float Betrag) {
    Credit = Credit + Betrag;
}
int main()...
```

Beachte: kein Semikolon!
Funktion a() ruft Funktion b() auf
Funktion b() ruft Funktion c() auf
Funktion c() ruft Funktion a() auf – Was nun???

Lokale und globale Variablen

- Die Definition von Variablen innerhalb des Rumpfes oder im Kopf einer Funktion erzeugt sogenannte Lokale Variablen
- Mit diesen lokalen Variablen kann nur innerhalb der Funktion gerechnet werden
  -> im Quellcode der Funktionsdefinition
- Von „außen" kann auf die lokalen Variablen nicht zugegriffen werden ~> beim Verlassen der Funktion sind sie nicht mehr verfügbar!!!

Lokale und globale Variablen

- Werden Variable außerhalb aller Funktionen definiert, so sind sie für alle Funktionen verfügbar ~> sie haben einen globalen Gültigkeitsbereich und heißen deshalb *Globale Variable*
- Globale Variablen gelten daher in allen Funktionen, und damit auch in der main-Funktion

Beispiel mit globalen Variablen


```
//Verwendung von globalen Variablen
#include <iostream>
int x = 4, y = 5;          // Definition der globalen Variablen x, y
int ADD (int, int);        // Prototyp der Funktion ADD

int main() {               // Definition der Funktion main
   cout << ADD(x,y);       // Ausgabe der Summe x + y
   return 0;
}

int ADD (int x, int y) {          // Definition der Funktion ADD
   return x + y;
}
```

Lokal oder global?

- Anmerkung:
  Die Definition und somit eine Benutzung globaler Variablen ist in C++ nicht **unumstritten** ~> besser also: **keine** globalen Variablen verwenden...
- Variablen sollten immer in der Funktion definiert (vereinbart) werden, wo sie ihre Anwendung haben
- Schnittstellen zwischen den Funktionen sind die (Übergabe-)Parameter ~> so werden die Werte von Funktion zu Funktion weitergereicht

Beispiel mit lokalen Variablen


```
// Lösung mit lokalen Variablen
#include <iostream>
int ADD (int);                  // Prototyp der Funktion ADD

int main() {                    // Definition der Funktion main
   int z = 6;
   cout << ADD(z);              // Aufruf der Funktion ADD mit Param.
   return 0;
}

int ADD (int z) {               // Definition der Funktion ADD: Kopf
   int x = 4, y = 5;            // Definition der lokalen Variablen x, y
   return x + y + z;            // Rückgabe x + y + z (hier: 15)
}
```

Argumente, Parameter, Rückgabewerte

- Die Argumente [x, y] einer Funktion [f] sind ihre Parameter und die Funktion selbst bestimmt den Rückgabewert [z]: z= f(x, y)
- Die Parameter und der Rückgabewert können von einem jeweils unterschiedlichen Typ sein, z.B.; long int z; int x; char y;
- Jeder gültige C++-Ausdruck ist auch als Funktionsausdruck gültig, also Konstante, mathematische und logische Ausdrücke und wieder eine neue Funktion in der Funktion:

```
double Divi (float a, float b)          // Definition der Funktion Divi
{
        double c, d=100, p;
        c = a/b;
        p = Multi(c, d);                // Aufruf einer Funktion Multi in Divi
        return p;                       // Rückgabe von p als Ergebnis von Divi
}
```

- Die übergebenen Argumente (Parameter) [a und b] sind in der Funktion *Divi* lokale Größen: Ihre möglichen Veränderungen in *Divi* haben keinerlei Einfluss auf die Variablen a und b in der aufrufenden Funktion
- Eine Funktion gibt einen Wert zurück oder keinen, dann *void*
  ~> *void* sagt dem Compiler, dass es keinen Wert für diese Funktion gibt
- Die Rückgabe erfolgt über *return (Null, Wert oder/und Ausdruck);*
  ~> return 9; return (y); return (v < 9); return 0;

Standardparameter

Beim Aufruf einer Funktion müssen die angegeben aktuellen Parameter in Reihenfolge, Anzahl und Datentyp den bei der Funktion definierten formalen Parametern entsprechen

In C++ ist es erlaubt, **die letzten** Parameter beim Aufruf wegzulassen!

Voraussetzung:

Bei der Angabe der formalen Parameter werden **in der Prototypenanweisung** Standardwerte angegeben.

Standardparameter

- Prototyp der Funktion: long DefinFkt **(int = 100);** // Deklaration
- Definition der Funktion:

```
long   DefinFkt (int x)
{
  long y;
  y = x * x;
  return y;
  }
```

1. Aufruf der Funktion: Def_F = DefinFkt(); // mit **Standardwert**
2. Aufruf der Funktion: Def_F = DefinFkt(50); // mit **Zuweisung**

- ~> Mit der Vorgabe eines Standardwerts in der Funktionsdeklaration gibt es **immer** einen Übergabewert
- Wird beim Aufruf der Funktion kein Parameter übergeben, nimmt die Funktion automatisch den Standardwert aus dem Prototyp

Standardparameter

- Bei Eintragung eines Werts für den Parameter im Funktionsaufruf rechnet die Funktion mit diesem Übergabewert:

1. Prototyp: long Fkt (int P1, int P2, int P3);
2. Prototyp: long Fkt (int P1, int P2, int P3=10);
3. Prototyp: long Fkt (int P1, int P2=20, int P3=10);
4. Prototyp: long Fkt (int P1, int P2=20, int P3);
   ~> nicht erlaubt!
5. Prototyp: long Fkt (int P1=5, int P2=20, int P3=10);

- Definition: long Fkt (int PI, int P2, int P3) {...

Verwendung von Standardparametern

- Mit der Verwendung von Standardparametern kann der Programmierer in C++ variieren zwischen aktuellen, direkt übergebenen Parameterwerten und den Standardwerten:

```
...
long Fkt (int P1, int P2 = 20, int P3 = 10);        // Prototyp mit Standardwerten
int main() {
   int P1 = 50, P2 = 25, P3 = 5;
   long Best_Vol;
   Best_Vol = Fkt (P1, P2, P3);                     // mit P1= 50, P2= 25, P3= 5
   cout << "Erstes Quadervolumen= " << Best_Vol << "\n";        // = 6250
   Best_Vol= Fkt (P1, P2);                          // mit P1= 50, P2= 25, P3= 10
   cout << "Zweites Quadervolumen= " << Best_Vol << "\n";       // = 12500
   Best_Vol= Fkt (P1);                              // mit P1= 50, P2= 20, P3= 10
   cout << "Drittes Quadervolumen= " << Best_Vol << "\n";        // = 10000
return 0; }
long Fkt (int P1, int P2, int P3)  {                // Definition der Funktion Fkt
return (PI * P2 * P3); }
```

Leere Parameterlisten

Sowohl in C als auch in C++ kann man explizit angeben, dass eine Funktion gar keine Parameter bekommt: **f(void)** , oder mit einer variablen Anzahl von Parametern aufgerufen werden kann: beispielsweise **f(...)** .

Die Angabe einer leeren Parameterliste **f()** unterscheidet sich in C und C++:

In C bedeutet eine leere Parameterliste in einer Funktionsdeklaration, dass die Funktion eine unbekannte Anzahl irgendwelcher Parameter erwartet, das entspricht **f(...)** .

In C++ dagegen drückt man damit aus, dass die Funktion keine Parameter erwartet. Damit hat eine leere Parameterliste die Wirkung von **f(void)** .
int getchar(); ist in C++ also gleichbedeutend mit **int getchar(void);**

Übergabeverfahren

In C++ gibt es zwei Möglichkeiten der Parameterübergabe:

- Wertübergabe (call by value)
- Referenzübergabe (call by reference)

Bei der **Wertübergabe** wird der Wert des Parameters an die Funktion übergeben.

- In der Funktion wird eine **Kopie** dieses Parameters angelegt, mit dieser Kopie wird gerechnet.
- Änderungen an der Kopie des Parameters ändert nichts an dem Parameter der **aufrufenden** Funktion.
- Ist die Funktion beendet, verliert die Kopie des Parameters seine Gültigkeit, sie wird „zerstört“.

Bei der **Referenzübergabe** wird die Referenz (die Adresse) des Parameters der aufrufenden Funktion an die aufgerufene Funktion übergeben.

- In der aufgerufenen Funktion wird keine Kopie des Parameters angelegt.
- Die Funktion rechnet und verändert den Parameter der aufrufenden Funktion.
- Wird der Parameter in der aufgerufenen Funktion verändert, wird auch der Parameter in der aufrufenden Funktion verändert!

Beispiel Referenzübergabe


```
#include <iostream>
using namespace std;

void manip(int & zahl1, int & zahl2)  {              // & heißt: nimm die Referenzen (Originalwerte!)
 int hilf;
 zahl1 = 4;
 zahl2 = 9;
 hilf = zahl1 + zahl2;
 cout << "Ergebnis: " << hilf << endl;
}

int main()  {
  int wert1 = 3, wert2 = 7;
  cout << "vor  Aufruf: " << wert1 << ",  " << wert2 << endl;
  manip(wert1, wert2);
  cout << "nach Aufruf: " << wert1 << ",  " << wert2 << endl;
  getchar();
}
```

Ausgabe:

```
vor Aufruf: 3,  7
Ergebnis: 13
nach Aufruf: 4,  9
```

Vor- / Nachteile der Referenzübergabe

Vorteile der Referenzübergabe:

- Dadurch, dass in der aufgerufenen Funktion keine Kopie des Parameters angelegt wird, benötigt call by reference weniger Speicherplatz als call by value
- die Parameterübergabe wird schneller abgearbeitet, da die Zeit für Anlegen der Kopie und Löschen der Kopie eingespart wird.

Nachteile der Referenzübergabe:

- es besteht die Gefahr, dass Parameter in der aufrufenden Funktion unbeabsichtigt verändert werden, da die aufgerufene Funktion Zugriff auf die ursprünglichen Parameter hat.
- Eine unbeabsichtigte Veränderung der Parameter kann durch Verwendung des Schlüsselworts const verhindert werden. Hierdurch wird der Parameter als Konstant (unveränderbar) vereinbart!

Referenzübergabe


```
#include <iostream>

void test(int & a, int b)  {
   a = 20;
}

int main()  {
   int var1=10;
   int var2=20;

   cout << "VOR  Test-Aufruf: var1:" << var1 << "  var2:" << var2 << endl;

   test(var1, var2);

   cout << "NACH Test-Aufruf: var1:" << var1 << "  var2:" << var2 << endl;

   getchar();
}
```

➔

VOR Test-Aufruf: var1:**10** var2:20

NACH Test-Aufruf: var1:**20** var2:20

Referenzen fehlerhaft verwenden

Referenzen kann man sich für viele Anwendungsfälle als Zeiger mit vereinfachter Schreibweise vorstellen. Damit kann man aber die Fehler, die man mit Zeigern gerne und oft praktiziert, auch eleganter mit Referenzen ausdrücken. Ebenso, wie man mit Zeigern auf Speicher zugreifen kann, auf den man gar nicht zugreifen darf, kann man mit Referenzen schöne Fehler produzieren:

- Zeiger p noch nicht initialisiert, aber schon eine Referenz mit *p initialisiert: **Falsch!**
- Speicher allokiert, Referenz darauf initialisiert, dann den Speicher wieder freigegeben, und die Referenz munter weiterverwenden: **Falsch!**
- Speicher allokiert, Referenz darauf initialisiert, dann den Speicher mit realloc() an eine andere Stelle verschoben, und die Referenz auf den inzwischen ungültigen Speicher weiter verwenden: **Falsch!**
- eine Referenz auf eine automatische Variable anlegen, nach dem Verlassen des umgebenden Blocks der automatischen Variable die Referenz weiterverwenden: **Falsch!**

Der Programmierer muss darauf achten, dass eine Referenz nur auf Speicher verweist, auf den man auch noch zugreifen darf. Beispielsweise ist es ziemlich debil, wenn eine Funktion eine Referenz auf eine eigene lokale Variable zurückgibt: nach dem Rücksprung aus dieser Funktion kann mit der Referenz niemand etwas anfangen, weil die zugehörige Variable ja gar nicht mehr existiert.

Inline-Funktionen

In C++ kann man eine Funktion als inline deklarieren. Das bedeutet, dass der Compiler - wenn er will - dafür gar keine eigene Funktion erzeugt, sondern den entsprechenden Maschinencode direkt an der Aufrufstelle einfügt.

Dadurch kann ein Programm schneller werden, aber unter Umständen auch größer (wenn die „Funktion“ an vielen Stellen aufgerufen wird). Der Compiler wird deshalb nicht beliebig komplexe und lange Funktionen tatsächlich inline kompilieren, sondern nur kurze, einfache. Es steht jedem Compiler frei, selbst zu entscheiden, welche Funktionen er so behandelt.

Die Tatsache, dass unter Umständen gar keine echte Funktion existiert, hat natürlich zur Folge, dass man die Adresse einer inline-Funktionen nicht bestimmen kann.

Außerdem muss der Compiler in jedem betroffenen Modul die vollständige Definition einer solchen Funktion sehen können, weil er sie ja sonst nicht expandieren kann. Daher kommt - bei einer Aufteilung in eine Deklarationsdatei und eine Definitionsdatei - eine normale Funktionsdefinition zwar in die Definitionsdatei, die Definition einer inline-Funktion dagegen in die Deklarationsdatei, weil die ja von allen Anwendern der Funktion eingebunden wird.

Überladen von Funktionen

Bei der objektorientierten Programmierung (OOP) ist es erlaubt, dass die Namen von verschiedenen Funktionen gleich sind, wenn sie sich in der **Anzahl** bzw. **Typ** der Übergabeparameter unterscheiden.

Der Datentyp des Rückgabeparameters ist irrelevant.

- es können verschiedene Funktionen mit **gleichem Namen** programmiert werden
- die Funktionen müssen sich in **Anzahl** und/oder **Datentyp** der formalen Parameter unterscheiden
- der Rückgabeparameter ist für die Unterscheidung der Funktionen nicht von Bedeutung

Überladen von Funktionen

Beim Aufruf der Funktion kann der Compiler anhand der verschiedenen Parameter unterscheiden, welche Funktion gemeint ist.

```
void fkt_1(int, int);
void fkt_1(int, char);
void fkt_1(char, char, float);
```

nicht zulässig:

```
void fkt_1(int, int);
int fkt_1(int, int);
char fkt_1(int, int);
```

Polymorphie mit Standardwerten?


```
void fkt_1(int, int, char = 'A');
void fkt_1(int, int);
// Aufruf:
int main()  {
        // ...
        fkt_1(20, 12);
}
```

Beim Aufruf erkennt der Compiler, dass die Funktion mit zwei Integer-Parametern aufgerufen werden soll → bei der ersten Version der Funktion kommt der Standardwert nicht zum Tragen → überflüssig, die erste Funktion muss immer mit 3 Parametern aufgerufen werden!

Funktionsschablonen (Templates)

- In der Praxis tritt oft das Problem auf, eine Funktion mehrfach programmieren zu müssen, um ein und denselben Algorithmus auf verschiedene Typen von Argumenten anzuwenden.
- Ein typisches Beispiel ist das Aussuchen des kleineren Werts aus zwei Argumenten. Gegebenenfalls muss man zu diesem Zweck mehrere Funktionen definieren, die jeweils int-, unsigned int-, double-Argumente usw. erhalten und entsprechende Rückgabewerte liefern.
- Neben dem Aufwand, alle Versionen auszuformulieren, hat man dann noch das Problem, die jeweils passende Funktion mit dem richtigen Namen aufzurufen. (Letzteres ließe sich noch durch Überladen lösen.)
- Wir schreiben deshalb eine „generische“ Funktion als „Schablone“ (engl. „Template“) ohne Rücksicht auf die Datentypen der Übergabeparameter:

```
template<class T> T funktionsname( T a, T b )
```

Templates: Beispiel


```
#include <iostream>
using namespace std;

template<class T> T min( T a, T b )
{
  return a<b ? a : b;
}

int main()
{
  int
    i = 5,
    j = 7;
  double
    d = 3.14,
    e = 2.7182818;

  cout << "min( i, j ) ist " << min( i, j ) << "\n";
  cout << "min( d, e ) ist " << min( d, e ) << "\n";

  return 0;
}
```