Funktionen (prozedural) bzw. Methoden (OOP)

Eine Funktion erledigt eine abgeschlossene Teilaufgabe. Die notwendigen Daten werden der Funktion mitgegeben. Das Ergebnis der erledigten Aufgabe gibt sie an den Aufrufer zurück. Eine Funktion muss nur einmal definiert werden. Anschließend kann sie beliebig oft nur durch Nennung ihres Namens aufgerufen werden, um die ihr zugewiesene Teilaufgabe abzuarbeiten. Bereits vorhandene Standardlösungen von Teilaufgaben können aus Funktionsbibliotheken abgerufen werden – ebenso wie neu entwickelte Funktionen in Bibliotheken aufgenommen werden können.

Quelle : (gekürzt) Der C++ Pogrammierer / Ulrich Breymann / Hanser- Verlag / 5.Auflage

1. Aufbau und Prototyp

Auch in C++ wird zwischen **Deklaration (Prototyp)** (Zeile ①) und **Definition** (Zeile ②) einer Funktion unterschieden. Die Deklaration erfolgt stets vor der Definition. Damit kann der Funktionsaufruf (Zeile ❶) schon nach der Deklaration erfolgen.

	Funktionen
	// Prototyp der Funktion
①	**[inline] Rückgabetyp** Bezeichner([**Typ** [,**Typ**]]);
	// Aufruf der Funktion
❶	[Variable =] Bezeichner([Wert1$_1$ [, Wert$_2$ ...]]);
	// Definition der Funktion
②	**Rückgabetyp** Bezeichner([**Typ** Arg$_1$ [,**Typ** Arg$_2$ ...]]) {
③	[ggf. Deklaration lokaler Variablen]
	Anweisung(en);
④	[**return** Rückgabewert;]
	}

Hinweise:

- Wenn die Definition vor dem Aufruf erfolgt, kann der Prototyp auch entfallen.
- Funktionen ohne Parameter werden durch leere Klammern dargestellt.
- Wenn die Funktion keine Rückgabe hat, wird dies durch das Schlüsselwort **void** als Rückgabetyp gekennzeichnet.
- **inline** Funktionen werden vom Compiler meist direkt im Quelltext (Aufruf-Stelle) ersetzt.
- Funktions-Definitionen können in eine externe Datei ausgelagert werden.
- Das Schlüsselwort **return** beendet die Abarbeitung der Funktion und kehrt an die Aufrufstelle zurück.

2. Default Parameter

Es ist auch möglich einzelnen Parametern einer Funktion einen Vorgabewert (Defaultwert) zuzuordnen. Den Parameterwert muss man dann nicht zwingend bei einem Funktionsaufruf angeben. Sie müssen (mehrere sind möglich) am Ende der Parameterliste eingefügt werden.

	Kennzeichnung im Prototyp ① oder in der Definition ②
①	**Typ** Bezeichner(**Typ**, **Typ**, **Typ** = Wert$_3$, **Typ** = Wert$_4$);
②	**Typ** Bezeichner(**Typ** Arg$_1$, **Typ** Arg$_2$, **Typ** Arg$_3$, **Typ** Arg$_4$) { // Funktionsrumpf }

3. Variable Argumentenliste

Wenn die Zahl der Argumente nicht von vornherein begrenzt ist, wird als Parameterliste die sog. *Ellipse* . . . angegeben. Der Funktion werden die Argumente dann in Form einer Liste übergeben, auf die mit Hilfe der (in der Headerdatei **cstdarg** definierten) **va**-Makros zugegriffen werden kann.

	Variable Argumentenliste
①	**Typ** Bezeichner(**Typ** Arg, ...) { // Funktionsrumpf }

„Obwohl unspezifizierte Argumente manchmal verlockend aussehen, sollten Sie sie nach Möglichkeit vermeiden. Das hat zwei Gründe: Die **va**-Befehle können nicht erkennen, wann die Argumentenliste zu Ende ist. Es muss immer mit einer expliziten Abbruchbedingung gearbeitet werden. Die **va**-Befehle sind Makros, d.h. eine strenge Typüberprüfung findet nicht statt. Fehler werden - wenn überhaupt - erst zur Laufzeit bemerkt."

Quelle: (letzter Aufruf 17.12.2018) https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Weitere_Grundelemente/_Prozeduren_und_Funktionen#Funktionen_mit_beliebig_vielen_Argumenten

4. Funktionen überladen

In C++ werden Funktionen nicht nur anhand ihres Namens, sondern auch an der Parametersignatur unterschieden. Es kann also Funktionen gleichen Namens geben, wenn sie sich in Art und/oder Anzahl der Parameter unterscheiden (der Rückgabewert gehört **nicht** zur Parametersignatur).

	Funktionsaufruf
①	`gibAus( );`
②	`gibAus( 20 );`
③	`gibAus( 20L );`
④	`gibAus(20L, 4.5);`
	:

Aufruf ① ruft Funktion ❶ auf ☑

da Funktion ❷ oder ❺ passen

Aufruf ③ ruft Funktion ❸ auf ☑

Aufruf ④ castet **long** in **int**

und ruft Funktion ❹ auf ☑

	überladene Funktionen
❶	`void gibAus() { // Funktionsrumpf }`
❷	`void gibAus( int a ) {...}`
❸	`void gibAus( long a ) {...}`
❹	`void gibAus( int a, float b ) {...}`
❺	`int gibAus( int a) {...}`

Hinweis: Variante ④ -> ❹ bedeutet, dass der Compiler die „spezifischste Methode“ sucht. Das kann zu unerwünschten Effekten (z.B.: Mehrdeutigkeiten) führen und sollte vermieden werden.

5. Referenz-Variablen in Funktionen

Bei allen Funktionsaufrufen wurden bisher Variablen als Wert (also eine Kopie des Wertes der Variablen) an die Funktion übergeben. Diese Art wird als **`call by Value`** bezeichnet. In C++ kommt mit der Übergabe per Referenz (**`call by Refrences`**) eine weitere Möglichkeit hinzu. Eine Referenz ist ein Alias für eine Variable. Es wird keine Kopie des Wertes der Variablen erzeugt, sondern die gleiche Variable kann über mehrere Namen angesprochen werden.

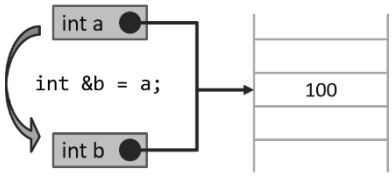
Einschub Refernz-Variablen

```
   :
①  int a = 100 ;
②  int &b = a ;

❶  cout << “a=“ << a << “ b=“ << b ;
                    // Ausgabe :  a=100 b=100
③  a = a * 2 ;
❶  cout << “a=“ << a << “ b=“ << b ;
                    // Ausgabe :  a=200 b=200
④  b = b * 2 ;
❶  cout << “a=“ << a << “ b=“ << b ;
                    // Ausgabe :  a=400 b=400
```

„Bildlich“ vereinfacht dargestellt weisen zwei Variablen auf die gleiche Speicheradresse (vergleichbar mit einer Datei auf die zwei Verknüpfungen unter Windows verweisen).

Erzeugung einer Referenz-Variable **b** die auf **a** verweist.



Somit wirken sich alle Änderungen (egal ob an a oder b) auf das gleiche Original aus.

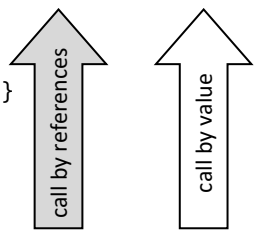
Achtung: Eine Referenz ist **kein** Zeiger (Pointer)!

Wird in der Signatur einer Funktion eine Referenz-Variable verwendet, so wird keine Kopie erzeugt, sondern ein Verweis auf die Variable im Hauptprogramm übergeben. Manipulationen an der Referenz-Variable wirken sich somit an der „Original-Variable“ im Hauptprogramm aus.

Funktions-Signatur mit Refernzparameter und Wertparameter

```
①  void shiftRight(            int &refPara , int valPara){
       while(valPara > 0) {
❸         valPara = valPara - 1 ;
❹         refPara = refPara / 10; } }

   main() {
      int value = 987654321 ;
      int steps = 4 ;
②                          shiftRight( value   ,  steps );
      cout << "value nach" << steps << "steps =" << value ;
      return 0; }
```



In Zeile ② wird die Funktion aufgerufen, wobei von **`value`** durch den &-Operator in der Funktions-Signatur ① eine Referenz in **`refPar`** übergeben wird (Referenzparameter). Vom Inhalt der Variable **`steps`** wird eine Kopie erzeugt und in **`refPara`** gespeichert (Wertparameter).
Das Dekrement in Zeile ❸ wirkt sich also nicht auf **`steps`** aus. Die Division in Zeile ❹ wird direkt an der Variable **`value`** durchgeführt. Eine Rückgabe ist deshalb nicht notwendig.