Strukturierte Programmierung und Funktionen

Um ein Programm übersichtlicher zu gestalten, zerlegt man seine Funktionalität meist in mehrere (oder auch ganz viele) Teile. Dadurch lässt sich das Programm übersichtlicher darstellen und manchmal lassen sich die Einzelteile auch in anderen Programmen sinnvoll einsetzen (wiederverwertbarer Code). C besitzt für diese Aufteilung nur die Funktion, während objektorientierte Sprachen auch Objekte (Klassen) dafür einsetzen können. Klassen sind selbst definierte Datentypen und eine Menge von dazugehörenden Methoden.

Wichtig: C (und C++)-Programme bestehen aus einer oder mehrerer Funktionen. Es muss mindestens die `main()`-Funktion vorhanden sein. Mit dieser startet und endet das Programm im Normalfall. Jede Funktionsdefinition muss vollständig in einer Quellcode-Datei enthalten sein. Man kann ein Programm auf verschiedene Quellcode-Dateien aufteilen, falls es für eine Datei zu groß und damit unübersichtlich wird. Jede Quellcode-Datei (Translation Unit) sollte sich einzeln kompilieren lassen.

Sobald der C++ Compiler eine Funktion kennt (weil er ihre Definition oder Deklaration vorher verarbeitet hat) können Sie im Programm diese Funktion aufrufen. Sie können also niemals eine erst später definierte Funktion aufrufen. Der Aufruf geschieht durch die Angabe des Funktionsnamens gefolgt von einem runden Klammerpaar `()` = **Funktionsaufruf-Operator**. Innerhalb der Klammern stehen die Argumente, die Sie an die Funktion übergeben möchten:

```
f()             // Aufruf der Funktion f  ohne Argumente
f(1, 2.)        // Aufruf der Funktion f  mit Argumenten 1 , 2.0.
f               // KEIN Aufruf der Funktion f
```

Wenn ein Funktionsaufruf erfolgen soll, springt das Programm (nach der Übergabe der Argumente) an den Anfang des Programmcodes der Funktion. Ist dieser Code abgearbeitet, springt die Programmausführung zurück zum Aufrufer und ersetzt praktisch den Funktionsaufruf durch den Rückgabewert der Funktion:

```
double quadrat(double x) // die Funktion wird definiert
{                        // damit beginnt der Codeblock der Funktion
     return x*x;         // Ergebnis der Funktion und Ende der Ausführung
}                        // damit endet der Codeblock der Funktion

...
quadrat;                          // kein Aufruf der Funktion: Klammern fehlen
quadrat(1.0);                     // Aufruf der Funktion: Ergebnis wird ignoriert
double u{quadrat(4.0)};           // 1 Aufruf -> u = 16.;
double v{quadrat(u) + quadrat(3.0)}; // 2 Aufrufe :  v = 256. + 9.;
```

Eine aufgerufene Funktion kann natürlich selbst weitere Funktionen aufrufen: `main()` rufe die Funktion `f()` auf, diese die Funktion `g()` und diese wiederum `h()`. Zu diesem Zeitpunkt sind alle 4 Funktionen gleichzeitig aktiv, d.h. sie wurden gestartet und sind noch nicht beendet. Die Menge der aktiven Funktionen zu einem Zeitpunkt nennt man den **Call-Stack,** der immer von „unten nach oben" wächst. An der untersten Stelle steht daher immer `main()`. Wenn in diesem Beispiel `h()` endet, springt das Programm zu `g()` zurück. Endet diese Funktion, geht's zurück zu `f()` und schließlich zu `main`(). Ein Funktionsaufruf vergrößert daher den Call-Stack um diese Funktion, ein Funktions-Rücksprung verringert den Call-Stack um diese Funktion.

Man spricht von einer **rekursiven** Funktion, wenn sie mehr als einmal im Call-Stack vorkommt (dann wurde sie von sich selbst oder von einer anderen Funktion aufgerufen, bevor sie selbst zu Ende war. In C/C++ (und Python) sind rekursive Funktionsaufrufe erlaubt:

```
int factorial(int n)      // Berechne n!
{    return  n <= 1 ? 1 : n*factorial(n-1);  }
```

Der Ausdruck `n*factorial(n-1)` bewirkt die Rekursion. Sie modelliert die mathematischen Beziehung `n! = n * (n-1)!` Steht etwa in `main()` die Programmzeile `auto x{factorial(5)};` wird zuerst `factorial(5)` aufgerufen, dieses ruft `factorial(4)` auf, dieses `factorial(3)` usw. Erst der Aufruf von `factorial(1)` beendet die Rekursion und alle aktiven Funktionen werden der Reihe nach beendet. Der Call-Stack sieht nach dem Aufruf von `factorial(1)` so aus:

```
main()              x = ?
  factorial(5)        5 * ?
    factorial(4)        4 * ?
      factorial(3)          3 * ?
        factorial(2)          2 * ?
          factorial(1)         1
```

Nun endet die Rekursion, denn `factorial(1)` kehrt zurück. Damit kann auch `factorial(2)` enden und dann `factorial(3)`. Der Call-Stack jetzt:

```
main()              x = ?
  factorial(5)        5 * ?
    factorial(4)        4 * ?
      factorial(3)          3 * 2 = 6
```

Alle weiteren `factorial()`−Aufrufe beenden sich analog. Am Ende bleibt `x{120}` in `main()` übrig. Sie können den Call-Stack im Debugger ansehen (`gdb`: Kommando `bt` für Back-Trace).