
Funktionen (Methoden)

Datenstrukturen und Algorithmen entwerfen und anwenden

Inhaltsverzeichnis

1	Einleitung.....	2
1.1	Unterschied zwischen einer Funktion und einer Methode.....	2
2	Namensgebung	2
2.1	Regeln der Namensgebung für Funktionen.....	2
2.2	Hinweise zur Namensgebung.....	3
2.3	Beispiele von gültigen und ungültigen Funktionsnamen	4
3	Bibliotheksfunktionen der Programmiersprache ANSI C.....	4
3.1	Hinzufügen von Bibliotheken in ein Programm unter ANSI C	4
3.2	Häufig verwendete Header-Dateien bei der Programmerstellung in ANSI C	4
4	Definition einer Funktion.....	5
4.1	Vorteile bei der Verwendung von Funktionen.....	6
5	Allgemeine Form einer Funktion.....	7
5.1	Hinweise zur Funktionsdefinition	8
5.2	Beispiel einer Funktionsanwendung ohne Funktionsdeklaration.....	8
6	Forward Deklaration	9
6.1	Funktionsdeklaration (Prototypen).....	9
7	Parameterübergabe.....	11
7.1	Übergabe der Parameter mittels "Call by Value"	11
7.2	Beispiel einer Parameterübergabe mittels "Call by Value"	12
7.3	Übergabe der Parameter mittels "Call by Reference"	14
8	Rückgabewert einer Funktion.....	15
8.1	Return-Anweisung	15
8.2	Funktionen ohne Rückgabewert ("Prozedure")	16
8.3	Funktionen mit Rückgabewert (return-Anweisung)	16
9	Gültigkeit und Lebensdauer von Variablen.....	17
9.1	Lokale Variablen.....	17
9.2	Globale Variablen	18
9.3	Schlüsselwörter unter ANSI C die das Verhalten von Variablen beeinflussen.....	18
10	Generelle Hinweise zu den Funktionen.....	19

1 Einleitung

Bis jetzt haben wir unsere Programme immer so gestaltet, dass nur eine einzelne Funktion mit dem Namen `main()` geschrieben wurde, die den gesamten Programmcode beinhaltet. Dies war bis zum jetzigen Zeitpunkt auch kein Problem, da der Programmcode im Normalfall sehr kurz, übersichtlich und auch nicht aus mehreren gleichen Programmsequenzen aufgebaut war.

Da wir jetzt aber schon einiges über die Programmiersprache ANSI C gelernt haben und nun soweit sind, auch grössere Projekte in Angriff zu nehmen, wollen wir uns mit dem modularen Aufbau von Programmen beschäftigen. Damit auch grosse Projekte richtig aufgebaut werden können und dadurch gut lesbarer, änderbarer und wartbarer Code entsteht, benötigen wir Unterprogramme die in ANSI C mittels Funktionen aufgebaut werden. Funktionen sind somit elementare Bausteine eines jeden grösseren C-Programms und haben dadurch eine grosse Bedeutung bei der Programmerstellung. Erst dank dieser Möglichkeit kann ein grösseres Programm in strukturelle Einheiten aufgeteilt werden.

1.1 Unterschied zwischen einer Funktion und einer Methode

Merke: !	<i>Funktionen gehören zu strukturierten (prozeduralen) Programmiersprachen wie z.B. ANSI C oder Pascal.</i> <i>Methoden gehören zu objektorientierten Programmiersprachen wie z.B. C# oder Java.</i>
-------------------------------	---

Eine Funktion kann sehr oft mit einer Methode in den Grundzügen verglichen werden. Beide haben die Aufgabe ein grösseres Programm in kleinere, überschaubare Unterprogramme aufzuteilen und somit lesbarer zu machen. Wenn wir also in Zukunft von einem Unterprogramm sprechen, so kann dies sowohl eine Funktion wie auch eine Methode sein.

Eine Methode gehört immer zu einer Klasse und diese lässt im Normalfall zu, dass diese überschrieben und vererbt werden kann. Bei Funktionen ist dies nicht möglich. Sie können also weder vererbt noch überschrieben werden.

Im Modul M411, wo wir die strukturierte Programmierung erlernen, setzen wir die Programmiersprache ANSI C ein. Dort werden wir als mit dem Begriff der Funktion arbeiten. In den Modulen M226A, M226B und folgende schauen wir die objektorientierte Sprache C# an. Dort werden wir mit den Methoden arbeiten.

Um nicht immer beide Begriffe im Dokument hinschreiben zu müssen, werden wir stellvertretend für den Begriff der Methode das Wort Funktion verwenden. Sollten gewissen Aussagen im Text nur auf Methoden beziehen, so wird dies klar zum Ausdruck gebracht.

2 Namensgebung

2.1 Regeln der Namensgebung für Funktionen

- Jeder Name sollte nur aus Buchstaben (inklusive Unterstrich: "_") und Ziffern aufgebaut sein. *Achtung: Durch diese Regeln sind somit keine Umlaute (ä, ü und ö) erlaubt!*
- Das erste Zeichen muss immer ein Buchstabe sein.
Da der Unterstrich ("_") ebenfalls als Buchstabe gilt, darf auch dieser als erstes Zeichen verwendet werden.

- Generell darf ein Funktionsname beliebig lang sein. Es muss aber darauf geachtet werden, dass in ANSI C interne Funktionsnamen nur innerhalb der ersten 31 Zeichen voneinander unterschieden werden können. In C# gilt diese Einschränkung nicht mehr.
- Da ANSI C auf die Gross- und Kleinschreibung von Namen achtet (case-sensitive), sind Bezeichnungen wie "BERECHNUNG()", "Berechnung()" und "berechnung()" verschiedenen Funktionen zuzuordnen!
- Die Bezeichnungen der reservierten Wörter der Programmiersprache ANSI C und C# dürfen nicht für die Namensgebung von Funktionen verwendet werden.
(Siehe Dokument: „Variablen und Konstanten von ANSI C und C# (Teil 1)“, Abschnitt: „Reservierte Schlüsselwörter der Programmiersprache ANSI C und C#“)

2.2 Hinweise zur Namensgebung

- Der ANSI C-Standard verlangt, dass ein C-Compiler die ersten 31 Zeichen bei intern verwendeten Namen unterscheidet; aufgrund der Beschränkungen einiger Linker werden bei externen Namen (Namen, die in anderen ANSI C-Quelldateien definiert wurden) allerdings nur 6 Zeichen verlangt. Das bedeutet in der Praxis, dass beim Bezug auf externe Namen Probleme auftreten können, wenn nur die ersten 6 Zeichen als signifikant gelten. Die meisten modernen C-Entwicklungssysteme gehen jedoch weit über dieses von ANSI C sehr niedrig angesetzte Minimum hinaus und entschärfen somit dieses Problem.
- Bei der Vergabe von Namen für selbstdefinierte Funktionen können beliebige Namen gewählt werden. Die Namen verschiedener Funktionen sollten sich aber nicht nur durch die Schreibweise des Namens unterscheiden.
- Methodennamen sollen ausdrücken was die Methode leistet. Der Name ist jeweils so zu wählen, dass dieser die auszuführende Tätigkeit des aufgerufenen Objektes beschreibt. Da es sich um eine Tätigkeit handelt, sollte der Name immer ein Verb enthalten.
- Verwenden Sie nur dann Abkürzungen für Namen, wenn diese allgemein bekannt sind. Ist dem nicht so, so verwenden Sie immer den vollständigen Begriff. Programmzeilen ohne Abkürzungen sind leichter zu verstehen.
- Vermeiden Sie wenn immer möglich den Unterstrich (_) bei Bezeichnern.
- Alle Standardfunktionen von ANSI C werden nur mit Kleinbuchstaben geschrieben. Ein Aufruf einer solchen Funktion mit einer Gross-Kleinschreibung führt somit zu keinem Erfolg. Beim Aufruf von Standard-Bibliotheksfunktionen muss also die Schreibweise genau eingehalten werden.
- In C# werden die Methodennamen hingegen mit einem Grossbuchstaben begonnen und dann mit Kleinbuchstaben weitergeführt. Sollte der Name aus mehreren Einzelwörter bestehen, so wird jedes neue Wort wieder mit einem Grossbuchstaben begonnen.
- **Der Name einer eigen erstellten Funktion oder Methode wollen wir in unserem Unterricht immer mit einem Grossbuchstaben beginnen.** Es wird die sogenannte **UpperCamelCase-Notation** verwendet.

Hinweis:

Da Sie im Verlauf Ihrer Ausbildung die Programmiersprache C# kennenlernen werden, setzen wir von Anfang an die C#-Coderichtlinien für die Bezeichnung von Funktionen und Methoden ein. Dies macht auch Sinn, da diese neue Sprache moderner ist und somit auch von Ihnen mehr zur Anwendung kommen wird.

2.3 Beispiele von gültigen und ungültigen Funktionsnamen

Richtige Namensgebung	Falsche Namensgebung
<ul style="list-style-type: none">- BerechneSumme()- WriteNameToConsole()- BerechneMaximalWert()- BerechneMinMaxWert()	<ul style="list-style-type: none">- berechneSumme()- Write-Name-To-Console()- Berechne Maximal Wert()- Berechne_Min/Max_Wert()

3 Bibliotheksfunktionen der Programmiersprache ANSI C

In ANSI C werden bereits viele Funktionen zur Verfügung gestellt, die in verschiedenen Bibliotheken zusammengefasst werden. Jede dieser Bibliotheken ist im Normalfall auf eine bestimmte Richtung ihrer Funktionen und Konstanten ausgelegt. So beinhaltet z.B. die Bibliothek: `stdio` mit der Prototypendatei `stdio.h` alle Funktionen und Konstanten die etwas mit der Ein- und Ausgabe von Werten zu tun haben. Die Bibliothek: `math` mit der Prototypendatei `math.h` beinhaltet hingegen Funktionen und Konstanten im Bereich der Mathematik. So ist z.B. die Funktion: `sin()` in dieser Bibliothek zu finden.

3.1 Hinzufügen von Bibliotheken in ein Programm unter ANSI C

Damit Funktionen verwendet werden können, muss die Schnittstelle dieser Funktion bekannt gemacht werden. Dies wird mit den Headerdateien, in welchen die Funktionsprototypen, Konstanten usw. deklariert sind, erreicht. Damit also eine bestimmte Bibliotheksfunktion z.B. `sin()` verwendet werden kann, muss die entsprechende Headerdatei – hier `math.h` – mit `#include` eingebunden werden (*Siehe dazu das Dokument: "Präprozessor (Precompiler)"*). Selbstverständlich muss vorausgesetzt werden, dass die Bibliotheksfunktionen vom Compiler auch gefunden werden können. Wurde eine Standardinstallation für das Entwicklungssystem (z.B. Visual Studio) verwendet, so ist dies meist schon gewährleistet. Ansonsten muss kontrolliert werden, ob die Pfadeinstellungen im Programm richtig angegeben wurden.

```
#include <Bibliotheksname.h> // Einbindung der Bibliothek
```

Codeumsetzung 1: Einbindung einer Bibliothek in ANSI C

3.1.1 Beispiel zum Einbinden einer Bibliothek in ANSI C


```
#include <stdio.h> // Einbindung der Standardbibliothek
```

Codebeispiel 1: Einbindung einer Bibliothek in ANSI C


3.2 Häufig verwendete Header-Dateien bei der Programmerstellung in ANSI C

stdio.h	<ul style="list-style-type: none">• Funktionen und Konstanten der Standard Ein- und Ausgabe.• Eine der meist verwendeten Bibliotheken und wird fast immer benötigt• Funktionsbeispiele: <code>puts()</code>, <code>printf()</code>, <code>scanf()</code>, ...
stdlib.h	<ul style="list-style-type: none">• Allgemein wichtige Standardfunktion.• Bei einigen Entwicklungsumgebungen wird diese automatisch immer eingebunden.• Funktionsbeispiele: <code>exit()</code>, <code>div()</code>, <code>abs()</code>, ...

math.h	<ul style="list-style-type: none"> • Wichtige Funktionen und Konstanten der Mathematik. • Funktionsbeispiele: <code>sin()</code>, <code>cos()</code>, <code>tan()</code>, ...
conio.h	<ul style="list-style-type: none"> • Spezifische Konsolen Ein- & Ausgabefunktionen für DOS-/Windows-Systeme die nicht mehr zum ANSI C Standard gehören. • Bildschirm kann mit diesen Funktionen komfortabler manipuliert werden. • <i>Achtung: Programm läuft nur auf einem entsprechenden DOS- / Windows-System. Unter UNIX stehen andere Header-Dateien zur Verfügung, die einen entsprechenden Funktionsumfang beinhalten.</i> • Funktionsbeispiele: <code>getch()</code>, <code>getche()</code>, ...

Merke: 	Vermeiden Sie die Erstellung eigener Funktionen, falls für den bestimmten Zweck eine ANSI C konforme Bibliotheksfunktion zur Verfügung steht. Dies reduziert die Entwicklungszeit eines Programms und erhöht zudem dessen Portierbarkeit.
--	--

4 Definition einer Funktion

Definition: 	Eine Funktion ist ein benanntes, unabhängiges C-Codefragment, das eine bestimmte Aufgabe ausführt und optional einen Wert an das aufrufende Programm zurückgibt.
---	---

Aus obiger Definition können somit folgende Rückschlüsse gezogen werden:

- **Eine Funktion hat immer einen eindeutigen Namen.** Wenn dieser Name in einem anderen Teil des Programms verwendet wird, können die Anweisungen, die sich hinter dieser benannten Funktion verbergen, ausgeführt werden. Man bezeichnet dies als Aufruf der Funktion. Eine Funktion lässt sich auch aus einer anderen Funktion heraus aufrufen.
- **Eine Funktion ist unabhängig.** Eine Funktion kann eine Aufgabe ausführen, ohne dass davon andere Teile des Programms betroffen sind oder diese einen Einfluss auf die Funktion nehmen.
- **Eine Funktion führt eine bestimmte Aufgabe aus.** Eine Aufgabe ist ein bestimmter, klar definierter Job, den das Programm im Rahmen seines Gesamtziels ausführen muss. Dabei kann es sich z.B. um das Versenden einer Textzeile an den Drucker, das Sortieren eines Arrays in numerischer Reihenfolge oder die Berechnung einer Quadratwurzel handeln.
- **Eine Funktion kann einen Wert an das aufrufende Programm zurückgeben.** Wenn das Programm eine Funktion aufruft, führt es die in der Funktion enthaltenen Anweisungen aus. Beim Rücksprung aus der Funktion kann es mit entsprechenden Anweisungen Informationen an das aufrufende Programm übermitteln.

4.1 Vorteile bei der Verwendung von Funktionen

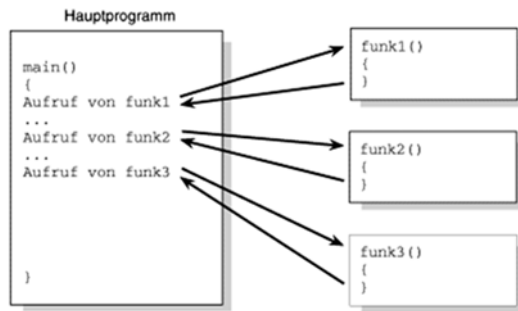


Abbildung 1: Darstellung von Funktionsaufrufen

Bekanntlich muss jedes C-Programm zu-
mindest über eine Funktion verfügen,
nämlich der Funktion: `main()`. Jedes Pro-
gramm beginnt mit der Ausführung dieser
Hauptfunktion und ruft unter Umständen
weitere Funktionen direkt oder indirekt
aus dieser heraus auf.

Wie wir in der Einleitung schon gelesen haben, sollte ein grösseres Programm unbedingt in mehrere Funktionen aufgeteilt werden um die Lesbarkeit und Wartbarkeit eines solchen Programms zu verbessern. Professionelle Programme wie Betriebssysteme, Textverarbeitungen, Datenbanken oder Tabellenkalkulationen umfassen ohne weiteres mehrere hunderttausend Zeilen Quellcode.

Die Entwicklung eines so grossen Programms ohne Unterteilung in viele überschaubare Funktionen wäre kaum durchführbar und seine Pflege extrem aufwendig. Im Weiteren ergeben sich durch die Verwendung von Funktionen die folgenden Vorteile:

- Ein Programm kann durch die Aufteilung in Module **von verschiedenen Entwicklern realisiert werden**.
- Aufgrund der Arbeitsteilung zwischen den einzelnen Funktionen können **unnötige Code-Redundanzen vermieden werden**, weil die gleiche Funktion bei Bedarf beliebig oft aufgerufen werden kann.
- Einzelne Funktionen können aufgrund ihrer Abgeschlossenheit und Eigenständigkeit für die Erstellung anderer Programme **wieder verwendet werden**.
- Häufig benutzte Funktionen können übersetzt und **in Bibliotheken abgelegt werden**. Dies spart bei grossen Projekten Kompilierzeit. Im Weiteren ist zu bemerken, dass die Fehlersuche durch die Verwendung von Bibliotheksfunktionen sehr vereinfacht wird, da nur solche Funktionen in Bibliotheken abgelegt werden, die einwandfrei funktionieren und dies durch Testverfahren auch bestätigt wurde. Auch der Zeitaufwand für die Dokumentierung des Programms verringert sich, da Bibliotheksfunktionen nicht mehr erneut in ihrer Funktion beschrieben werden müssen. Eine einmalige Beschreibung einer Bibliotheksfunktion reicht aus und kann somit an diversen Stellen in anderen Programmen durch einen Hinweis (Link) auf diese Beschreibung eingesetzt werden.
- Die relative "Privatheit" von Code und Dateien einer Funktion bietet guten **Schutz vor versehentlicher Manipulation** fremder Speicherbereiche. Diese Eigenheit ist auch bei objektorientierten Programmiersprachen (C#, C++, Java, usw.) von zentraler Bedeutung und wird dort noch intensiver in der Programmiersprache umgesetzt.

5 Allgemeine Form einer Funktion

Die Definition einer Funktion hat die allgemeine Form:

```
Datentyp Funktionsname ([Parameterliste]) {      // Angabe zur Funktion
    [Lokale Variablendefinition]
    Funktionsanweisungen;
    ...
    [return (Ausdruck);]
}
```

Codeumsetzung 2: Allgemeine Form einer Funktion

Dabei haben die einzelnen Bereiche der Funktionsdefinition die folgende Bedeutung:

Datentyp:	<p>Der Datentyp bezieht sich auf den Rückgabewert der Funktion. Fehlt diese Angabe, dann geht der Compiler automatisch vom Typ: <code>int</code> aus. Der ANSI-C-Standard empfiehlt allerdings, den Typ des Rückgabewerts immer zu spezifizieren.</p> <p>Sollte die Funktion keinen Rückgabewert aufweisen, so muss dies mit dem Begriff: <code>void</code> mitgeteilt werden.</p> <p>Der Begriff: <code>void</code> drückt dabei explizit aus, dass kein Rückgabewert für diese Funktion existiert.</p>
Funktionsname:	<p>Für die Wahl des Funktionsnamen gelten die Regeln die wir am Anfang dieses Dokumentes besprochen haben. Achten Sie aber bitte speziell darauf, dass Sie einen möglichst aussagekräftige Namen finden, der das Verhalten der Funktion beschreibt.</p>
Parameterliste:	<p>Die Parameterliste besteht aus den einzelnen Variablennamen und deren Datentypen. Die einzelnen Parameter der Parameterliste werden durch Kommas voneinander getrennt. Die runden Klammern um die Parameterliste sind immer notwendig, auch dann, wenn die Funktion keine Parameter verwendet. In ANSI C ist es für diesen Fall erlaubt, nur die runden Klammern ohne Parameter zu verwenden.</p>
Lokale Variablendefinition:	<p>Werden Variablen innerhalb der Funktion definiert, so spricht man von so genannten "lokalen Variablen". Diese werden automatisch beim Aufruf der Funktion erstellt und am Ende wieder zerstört. Durch spezielle Schlüsselwörter kann auf das oben beschriebene Verhalten der Variablen Einfluss genommen werden.</p> <p><i>(Siehe dazu den Abschnitt: "Gültigkeit und Lebensdauer von Variablen")</i></p>
Funktionsanweisungen:	<p>Hier werden alle Anweisungen angegeben, die beim Aufruf der Funktion ausgeführt werden sollen.</p>
return (Ausdruck):	<p>Mit der Anweisung <code>return (Ausdruck);</code> wird das Unterprogramm beendet und dem Funktionsnamen den Wert: <code>Ausdruck</code> zugewiesen. Dieser Wert gilt dann als sogenannter Rückgabewert der Funktion.</p>

5.1 Hinweise zur Funktionsdefinition

- Der **Funktionskörper bildet immer einen Anweisungsblock**, deshalb muss er von geschweiften Klammern umschlossen sein. Ein Funktionskörper beinhaltet die lokalen Variablen (sofern vorhanden) und die Funktionsanweisungen.
- Es ist in ANSI C und C# nicht möglich, andere Funktionen innerhalb von Funktionen zu definieren, d.h. Funktionen zu verschachteln!
- Funktionen können in beliebiger Reihenfolge innerhalb des Programms definiert werden.
- Soll eine Funktion in ANSI C vor ihrer Definition im Quellcode aufgerufen werden, so muss diese zuerst dem Compiler mittels einer "Forward Deklaration" bekannt gemacht werden. (Siehe dazu den Abschnitt: "Forward Deklaration")

5.2 Beispiel einer Funktionsanwendung ohne Funktionsdeklaration

Das folgende Programm definiert eine eigene Funktion: `CalculateFaculty(value)`, die einen `int`-Wert als Resultat dieser Funktion zurückliefert. Als Argument wird ein `int`-Wert für den Parameter: `value` übergeben, von welchem die Fakultät berechnet wird.

Der Funktionsaufruf: `CalculateFaculty(4)` liefert dann den Wert 24. ($24 = 4 * 3 * 2 * 1$)

```
// function definition
int CalculateFaculty(int value)    // calculate faculty
{
    int faculty = 1;

    while (value > 1)
    {
        faculty *= value;
        value--;
    }
    return faculty;
}

// main function
int _tmain(int argc, _TCHAR* argv[])
{
    int number = 4;    //
    printf("The faculty of the number %d is %d.", number, CalculateFaculty(number));
    return 0;
}
```

Codebeispiel 2: Funktion: `CalculateFaculty()` ohne Funktionsdeklaration in ANSI C

Programmausgabe:


The faculty of the number 4 is 24.

6 Forward Deklaration

Wie Sie schon wissen, kann eine ANSI C Funktion überall im Quellcode definiert werden. Wichtig ist aber zu wissen, dass ein Aufruf dieser Funktion erst dann geschehen kann, wenn diese dem Compiler bekannt gemacht wurde.

Eine Bekanntmachung kann auf zwei Arten geschehen:

1. Durch die Definition der Funktion selbst vor dem ersten Aufruf oder
2. durch die Angabe eines Prototypen der weiter unten definierten Funktion (Forward-Deklaration).


Merke: 	Die Deklaration einer Funktion ist immer dann notwendig, wenn sie im Quellcode bereits vor ihrer Definition aufgerufen wird. Dabei wird wie bei der Deklaration üblich, noch kein Speicherplatz für die Funktion vergeben.
--	---

Die Deklarationen benutzt der Compiler dazu, beim Aufruf einer Funktion zu überprüfen, ob die Anzahl und der Datentyp der Argumente korrekt sind. Bei Funktionen ohne Rückgabewert muss der Compiler zusätzlich sicherstellen, dass sie nicht innerhalb eines Ausdrucks aufgerufen werden kann.

6.1 Funktionsdeklaration (Prototypen)

Gegenüber Kernighan und Ritchie (Urväter der Programmiersprache C) erlaubt ANSI C eine strengere Typenprüfung durch Verwendung von Prototypen. Generell sind aber die folgenden Deklarationen einer Funktion möglich:

- `int CalculateFaculty();` ursprüngliche Deklarationsart (von K. & R.)
- `int CalculateFaculty(int);` einfacher Funktionsprototyp (ANSI C)
- `int CalculateFaculty(int value);` erweiterter Funktionsprototyp (ANSI C)

Definition: 	Ein Prototyp dient dazu, dem Compiler Typ und Anzahl der Argumente sowie den Rückgabewert einer Funktion bekannt zu machen, sofern die Funktion im Quellcode vor ihrer Definition aufgerufen wird.
---	---

Der ANSI-Standard empfiehlt dringend die Verwendung von Prototypen. Diese enthalten neben dem Typ des Rückgabewerts noch die Datentypen aller Argumente. Wenn eine Funktion keine Argumente nimmt, dann kann für den Typ des Arguments das Schlüsselwort: `void` verwendet werden.

Eine Funktionsdeklaration sollte folgendermassen aufgebaut sein:

Datentyp Funktionsname ([Parameterliste]); // Angabe zur Funktion

Codeumsetzung 3: Funktionsdeklaration in ANSI C

Die Parameterliste sollte so aufgebaut sein, dass diese jeweils für jeden Parameter den Datentyp und den Namen des Parameters angibt (erweiterter Funktionsprototyp).

6.1.1 Beispiel zur Erstellung eines Funktionsprototypen in ANSI C

```
// function declaration
int CalculateFaculy(int number);    // calculate faculty
```

Codebeispiel 3: Funktionsprototyp: CalculateFaculy() in ANSI C

6.1.2 Hinweise zur Funktionsdeklaration

- Die zusätzliche Angabe des Parameternamens ist optional. Sie bietet aber den Vorteil, dass der Compiler bei eventuellen Fehlermeldungen den Namen des Arguments angeben kann, bei dem er einen Fehler entdeckt hat.
- Beachten Sie bitte, dass eine Funktionsdeklaration immer mit einem Semikolon abgeschlossen werden muss. Die Funktionsdefinition hingegen darf nach der Parameterliste kein Semikolon erhalten, da der Funktionsblock mittels öffnender, geschweifter Klammer ({} eingeleitet wird.

Merke:

!

Um sich einen guten Programmierstil anzueignen, sollten Funktionen in ANSI C am Anfang des Programms (zwischen #include-Anweisungen und main()) mittels Prototypen deklariert werden.

6.1.3 Beispiel einer Funktionsanwendung mit Funktionsdeklaration

Das folgende Programm definiert eine eigene Funktion: CalculateFaculy(value), die einen int-Wert als Resultat dieser Funktion zurückliefert. Als Argument wird ein int-Wert für den Parameter: value übergeben, von welchem die Fakultät berechnet wird.

Der Funktionsaufruf: CalculateFaculy(4) liefert dann den Wert 24. ($24 = 4 * 3 * 2 * 1$)

```
// function declaration
int CalculateFaculy(int number);    // calculate faculty

// main function
int _tmain(int argc, _TCHAR* argv[])
{
    int number = 4;    //
    printf("The faculty of the number %d is %d.", number, CalculateFaculy(number));
    return 0;
}

// function definition
int CalculateFaculy(int value)    // calculate faculty
{
    int faculty = 1;

    while (value > 1)
    {
        faculty *= value;
        value--;
    }
    return faculty;
}
```

Codebeispiel 4: Funktion: CalculateFaculy() mit Funktionsdeklaration in ANSI C

Programmausgabe:

The faculty of the number 4 is 24.

7 Parameterübergabe

Wie schon mitgeteilt, besteht ein besonderer Vorteil von Funktionen darin, dass ihr Code und ihre Daten nur für sie selbst zugänglich sind. Nun stellt sich aber die Frage, wie Funktionen untereinander Daten austauschen sollen? Eine erste Möglichkeit bestünde darin, globale Variablen zu verwenden, die ja stets für alle Funktionen verfügbar sind. Wir haben aber die Gründe gegen den häufigen Gebrauch globaler Variablen hinreichend dargestellt und darauf hingewiesen, dass dies nicht mit dem Konzept modularer Programmierung verträglich ist. (Siehe dazu das Kapitel: "Variablen und Konstanten von ANSI C und C# (Teil1)")

Nachdem der direkte Zugriff auf die Variablen einer Funktion von aussen nicht möglich ist, muss sich der Datenaustausch über den einzig legitimen Zugriff auf den Code einer Funktion vollziehen: dem Funktionsaufruf.

Merke:

!

Wenn eine Funktion eine andere aufruft, dann kann sie ihr bei dieser Gelegenheit Variablen als Argumente übergeben. Dabei ist es zulässig, mehrere Argumente auf einmal zu übergeben.

Damit eine Funktion überhaupt Argumente entgegennehmen kann, muss sie ihrerseits Variablen definieren, die den Wert dieser Argumente aufnehmen. Diese Variablen sind die Parameter einer Funktion. Sie haben die gleichen Eigenschaften wie alle anderen lokalen Variablen, werden aber nicht im Funktionskörper, sondern in der Parameterliste definiert.

Es ist der Normalfall, dass an Funktionen Werte (Argumente) übergeben werden. Je nach Verwendungszweck dieser Argumente dürfen (müssen) diese abgeändert werden oder nicht. Aus diesem Grund werden zwei Arten der Parameterübergabe in ANSI C zur Verfügung gestellt.

Diese heissen:

- **Call by Value** → Aufruf mit Wert
- **Call by Reference** → Aufruf mit Adressangabe

7.1 Übergabe der Parameter mittels "Call by Value"

Soll einer Funktion ein Wert nur zur Verfügung gestellt werden, ohne dass dieser durch die Funktion selbst abgeändert werden kann, so wird die "Call by Value" - Übergabe von Parametern verwendet.

Merke:

!

Die Wertübergabe ("Call by Value") ist der Normalfall bei der Weitergabe von Argumenten an eine Funktion. Dabei erhält die aufgerufene Funktion Kopien der Argument-Werte; sie kann daher keine Veränderungen an den Argument-Variablen des Aufrufers vornehmen.

7.1.1 Hinweise zu Arrays, Pointer und Objekten

- Werden Datentypen wie Arrays oder Pointer an eine Funktion übergeben, so erhält die Funktion auch diesmal nur eine Kopie des Wertes. Beim Wert handelt es sich diesmal aber nicht um einen einfachen Datenwert sondern um die Adresse, wo sich dieser Datenwert (Datenwerte) befindet.
- In der objektorientierten Sprachen wie z.B. C# werden sehr oft Objekte an eine Methode übergeben. Hier muss darauf geachtet werden, dass die Übergabe des Objektes mittels einer Referenz geschieht. Dies bedeutet, dass nicht die Inhalte (Attributwerte)

des Objektes weitergegeben werden, sondern die Adresse wo sich dieses Objekt im Heap-Speicher befindet. Es handelt sich hier also um eine „Call by Reference“ Parameterübergabe.

7.2 Beispiel einer Parameterübergabe mittels "Call by Value"

Das Programm: CalculateMaxValue.c soll von zwei übergebenen Zahlen die grössere wieder an den Aufrufer zurückgeben.

```
// function declaration
int CalculateMaxValue(int Zahl1, int Zahl2);    // calculate maximum value

int _tmain(int argc, _TCHAR* argv[])
{
    int number1, number2;                      // input values

    printf("Please enter the first integer value: ");
    scanf_s("%d", &number1);
    printf("Please enter the second integer value: ");
    scanf_s("%d", &number2);

    printf("The maximum value of the numbers: %d and %d is %d.",
        number1, number2,
        CalculateMaxValue(number1, number2));

    return (0);
}

// function definition
int CalculateMaxValue(int value1, int value2) {

    int maxValue;                             // result of calculation
    maxValue = (value1 >= value2) ? value1 : value2;
    return (maxValue);                        // returns the maximum of two values
}
```

Codebeispiel 5: Funktion: CalculateMaxValue() mit Funktionsdeklaration in ANSI C

Programmausgabe:

```
Please enter the first integer value: 5
Please enter the second integer value: 12
The maximum value of the numbers: 5 and 12 is 12.

Please enter the first integer value: 12
Please enter the second integer value: 5
The maximum value of the numbers: 12 and 5 is 12.
```

Hinweis:

Am Resultat der Programmausgabe kann man erkennen, dass es keine Rolle spielt ob zuerst die kleinere oder die grössere Ganzzahl eingegeben wird. Das Programm wird immer die grössere Zahl herausfinden.

Möchte man oben stehendes Programm so abändern, dass die Funktion **keinen Rückgabewert aufweist sondern die Variable: number1 nach dem Funktionsaufruf immer den Maximalwert beinhaltet**, so muss das Programm in einigen Bereichen angepasst werden.

Die erste Version des neuen Programms wird ein falsches Resultat beim Aufruf der Funktion berechnen! Wieso?

```
// function declaration
void CalculateMaxValue(int Zahl1, int Zahl2);    // calculate maximum value

int _tmain(int argc, _TCHAR* argv[])
{
    int number1, number2;                        // input values

    printf("Please enter the first integer value: ");
    scanf_s("%d", &number1);
    printf("Please enter the second integer value: ");
    scanf_s("%d", &number2);

    printf("The maximum value of the numbers: %d and %d is ", number1, number2);
    CalculateMaxValue(number1, number2);
    printf("%d.", number1);                      // output: maximum value

    printf("\n\n");
    return (0);
}

// function definition
void CalculateMaxValue(int value1, int value2) {

    value1 = (value1 >= value2) ? value1 : value2;

    return;                                     // no return value
}
```

Codebeispiel 6: Falsche Umsetzung der Funktion: CalculateMaxValue() in ANSI C

Programmausgabe:

```
Please enter the first integer value: 5
Please enter the second integer value: 12
The maximum value of the numbers: 5 and 12 is 5.
```

Lösung:

Das die erste Version des neuen Programms nicht richtig funktioniert liegt in der Art und Weise wie die Programmiersprache ANSI C Argumente an die Funktion übergibt. In diesem Fall wird nur eine Kopie des Wertes an die Funktion übergeben. Sobald aber die Funktion: CalculateMaxValue() abgeschlossen ist, werden die lokalen Variablen zerstört ohne dass der neue Wert gerettet wird. Somit gehen alle Änderungen die an den Parameter-Variablen vorgenommen wurden, verloren!

Wenn wir wollen, dass die aufgerufene Funktion Zugriff auf die Original-Variablen hat und deren Werte verändern kann, dann müssen wir hier die Speicheradressen dieser Variablen übergeben. Dieser Vorgang heisst mit dem englischen Fachausdruck "Call by Reference".

7.3 Übergabe der Parameter mittels "Call by Reference"

Wie schon angesprochen, wird bei einer Argumentübergabe mittels "Call by Reference" nicht der Wert selbst sondern nur die Adresse der Variablen übergeben. Dadurch wird gewährleistet, dass ein direkter Bezug zur Variablen möglich wird und diese somit auch verändert werden kann.

Merke: !	Bei der Adressübergabe des Datenobjekts ("Call by Reference") kann die aufgerufene Funktion die Argument-Variablen des Aufrufers verändern. Bei "normalen" Variablen müssen Sie dazu die entsprechende Adresse mit Hilfe des Adressoperators (&) ermitteln.
-------------------------------	--

Hinweis:

Array-Namen, Pointer und Objekte repräsentieren schon von Haus aus Adresswerte. Eine Umrechnung der Variablen in einen Adresswert entfällt also bei solchen Variablen.

Die zweite Version des neuen Programms berechnet das richtige Resultat beim Aufruf der Funktion.

```
// function declaration
void CalculateMaxValue(int *Zahl1, int Zahl2); // calculate maximum value

int _tmain(int argc, _TCHAR* argv[])
{
    int number1, number2; // input values

    printf("Please enter the first integer value: ");
    scanf_s("%d", &number1);
    printf("Please enter the second integer value: ");
    scanf_s("%d", &number2);

    printf("The maximum value of the numbers: %d and %d is ", number1, number2);
    CalculateMaxValue(&number1, number2);
    printf("%d.", number1);

    printf("\n\n");
    return (0);
}

// function definition
void CalculateMaxValue(int *value1, int value2) {



    *value1 = (*value1 >= value2) ? *value1 : value2;

    return; // no return value
}
```

Codebeispiel 7: Richtige Umsetzung von: CalculateMaxValue() mittels „Call by Value“ in ANSI C

Programmausgabe:

```
Please enter the first integer value: 5
Please enter the second integer value: 12
The maximum value of the numbers: 5 and 12 is 12.
```

Merke: 	Der Adressoperator (&) ermittelt die Adresse einer Variablen. Mit dem "*" wird eine Zeigervariable gekennzeichnet, welche eine Adresse einer Variablen aufnehmen kann. Wird eine Zeigervariable ohne diesen Stern benützt, so wird die Adresse manipuliert, welche ja der Inhalt der Variablen ist. Mit dem Stern wird aber der Speicherplatz, auf welche die Adresse zeigt, angesprochen.
Merke: 	Grosse Datenstrukturen sollten immer mittels "Call by Reference" übermittle werden, auch dann, wenn die Daten nur zur Funktion müssen, da sonst infolge der Datenstruktur-Kopie ein Stack-Überlauf (Stack-Overflow) riskiert wird.

8 Rückgabewert einer Funktion


Es gibt verschiedene Programmiersprachen, die Funktionen danach unterscheiden, ob diese einen Wert zurückliefern oder nicht. So hat z.B. die Programmiersprache: Pascal zwei verschiedene Arten von Unterprogrammen. Bei der so genannten "Prozedure" liefert das Unterprogramm keinen Rückgabewert. Bei der "Funktion" hingegen, wird ein Rückgabewert erzeugt.

ANSI C unterscheidet nicht zwischen zwei verschiedenen Unterprogrammarten und stellt deswegen für die Unterprogrammentwicklung nur die "Funktion" zur Verfügung. Je nachdem wie diese definiert wird, kann aber die Funktion einen Wert zurückliefern oder nicht.

8.1 Return-Anweisung

Die return-Anweisung bewirkt die unmittelbare Beendigung einer Funktion und die Rückkehr zum aufrufenden Programmteil. Im Gegensatz zu den Argumenten, von denen eine ganze Liste an die Funktion übergeben werden kann, akzeptiert die return-Anweisung nur einen Ausdruck. Sollen von einer Funktion mehrere Daten zurückgegeben werden, dann muss dies mit zusammengesetzten Datentypen, wie z.B. Arrays oder Strukturen geschehen.

(Siehe Dokumente: „Variablen und Konstanten von ANSI C und C# (Teil 2 & 3)“)

Merke: 	Obwohl die return-Anweisung mehrmals innerhalb der Funktion vorkommen kann, sollte diese aus der Sicht einer guten Softwarestruktur immer nur am Schluss der Funktion eingesetzt werden.
--	---

8.1.1 Verwendungsarten der return-Anweisung

return (Ausdruck);	Beendet die Funktion und gibt den Wert: Ausdruck an die aufrufende Funktion zurück.
	Wird return(Ausdruck) verwendet, so handelt es sich um einen Funktionsaufruf mit Argument: Ausdruck.
return Ausdruck;	Beendet die Funktion und gibt den Wert: Ausdruck an die aufrufende Funktion zurück.
	Wird return Ausdruck verwendet, so handelt es sich bei return um eine Anweisung.

return; Beendet die Funktion ohne Rückgabewert.
Der Datentyp der Funktion wird bei dieser Verwendung der `return`-Anweisung auf `void` gesetzt.

8.2 Funktionen ohne Rückgabewert ("Prozedure")

Soll ein Unterprogramm entwickelt werden, welches keinen Rückgabewert erzeugt, so spricht man sehr oft von einer so genannten "Prozedure". ANSI C kennt aber diesen Ausdruck nicht.

Eine Funktion ohne Rückgabewert wird wie folgt definiert:

```
void Funktionsname ([Parameterliste]) {    // Angabe zur Funktion
    [Lokale Variablendefinition]
    Funktionsanweisungen
    ...
    [return;]
}
```

Codeumsetzung 4: Funktionen ohne Rückgabewert in ANSI C

Da die Funktion keinen Wert zurückgibt, wird dies mittels `void` für den Datentyp der Funktion signalisiert. Der Befehl `return` beendet die aktuelle Funktion und bewirkt die Rückkehr zur aufrufenden Funktion (ohne Rückgabe eines Wertes). Man kann in diesem Fall aber auch vollständig auf `return` verzichten. Es gilt nämlich, dass das Erreichen des Funktionsendes (abschliessende Funktionsblockklammer: `}`) immer automatisch zu einer Rückkehr (ohne Rückgabe eines Wertes) aus der Funktion führt.

8.3 Funktionen mit Rückgabewert (return-Anweisung)

Um einen Wert aus der Funktion zurückzugeben, benötigt man die `return`-Anweisung. Wie unter dem Abschnitt: "*Return-Anweisung*" beschrieben, stehen zwei Möglichkeiten der Verwendung von `return` zur Verfügung. Welche der folgenden zwei Arten dabei ausgewählt wird, spielt aber keine Rolle:

- `return (Ausdruck);`
- `return Ausdruck;`

Eine Funktion mit Rückgabewert wird wie folgt definiert:

```
Datentyp Funktionsname ([Parameterliste]) {    // Angabe zur Funktion
    [Lokale Variablendefinition]
    Funktionsanweisungen
    ...
    return (Ausdruck);    // oder: return Ausdruck;
}
```

Codeumsetzung 5: Funktionen mit Rückgabewert in ANSI C

Wichtig ist, dass der Datentyp der Funktion angegeben wird. Wird dieser weggelassen (vergessen), so verwendet ANSI C automatisch einen Integer-Wert und dies kann unter Umständen zu sehr schwierig zu findenden Fehlern führen!

Die Hauptfunktion eines C-Programms (`main()`) gibt per ANSI C Definition immer einen Wert vom Typ `int` zurück. Ihr Rückgabewert geht wie bei allen Funktionen an den Aufrufer zurück – in diesem Fall ans Betriebssystem. Dort kann der Rückgabewert in Batch-Dateien oder Shell-Scripts abgefragt werden, um Aufschluss über die erfolgreiche oder irreguläre Beendigung des Programms zu erhalten.

Wenn der Ausdruck (Rückgabewert) einen Datentyp hat, der sich von dem der zugehörigen Funktion unterscheidet, dann wird der Wert dieses Ausdrucks in diesen Rückgabetyt konvertiert. Die Umwandlung erfolgt dabei, als ob der Ausdruckswert einer Variablen von diesem Datentyp zugewiesen würde.

(Siehe Dokumente: „Variablen und Konstanten von ANSI C und C# (Teil 1)“, Abschnitt: „Implizite Datentypenumwandlung“.)

9 Gültigkeit und Lebensdauer von Variablen

Im Dokument: "Variablen und Konstanten von ANSI C und C# (Teil 1)" haben wir zum ersten Mal etwas über die Gültigkeit und die Lebensdauer von Variablen gehört. Wir wollen nun diese beiden Begriffe im Zusammenhang mit Funktionen noch einmal näher betrachten. Speziell soll in diesem Abschnitt gezeigt werden, dass die Position der Variablendefinition einen enormen Einfluss auf die Sichtweise der Variablen hat.

9.1 Lokale Variablen


Merke: !	Alle Variablen einer Funktion sind "privat" und sind nur verfügbar, während die betreffende Funktion gerade ausgeführt wird. Solche Variablen heissen deswegen auch "lokale Variablen".
---------------------------	--

Merke: !	Die Gültigkeit lokaler Variablen ist auf eine Funktion oder einen Anweisungsblock beschränkt.
---------------------------	--

9.1.1 Hinweise für lokale Variablen

- Lokale Variablen werden auch als **"automatische Variablen"** bezeichnet, da diese automatisch beim Start der Funktion definiert werden.
- Im Normalfall werden alle lokalen Variablen **direkt am Anfang des Funktionskörpers definiert** - gleich nach der öffnenden, geschweiften Klammer (`{`).
- Lokale Variablen werden **beim Aufruf der Funktion erzeugt und am Ende dieser wird zerstört**. Ein erneuter Aufruf dieser Funktion hat wohl wieder die gleichen lokalen Variablen, die Inhalte entsprechen aber nicht mehr den Werten des vorhergehenden Aufrufs der Funktion! Dieses Verhalten kann aber über zusätzliche Angaben bei den Variablen verändert werden. (Siehe dazu den Abschnitt: "Schlüsselwörter die einen Einfluss auf das Verhalten der Variablen haben")
- Lokale Variablen werden **nicht automatisch initialisiert** wie dies bei den globalen Variablen üblich ist.
- Lokale Variablen **gehen wirtschaftlich mit dem Speicher um**, da diese nur während der Funktionsausführung den Speicher belegen.

9.2 Globale Variablen

Merke: 	<p>Alle Variablen die ausserhalb einer Funktion definiert werden, werden als "globale Variablen" bezeichnet.</p> <p>Diese sind während der ganzen Programm Ausführung verfügbar und können von allen Funktionen verwendet werden.</p>
--	---

9.2.1 Hinweise für globale Variablen

- Globale Variablen werden auch als **"statische Variablen"** bezeichnet, da diese beim Start des Programms definiert werden und dann während der ganzen Programmausführung zur Verfügung stehen.
- Im Normalfall werden alle globalen Variablen noch **vor main()** definiert.
- Existiert eine globale und eine lokale Variable gleichen Namens, so **überdeckt die lokale die globale Variable**. Das heisst, die globale Variable ist in diesem Fall nicht mehr innerhalb dieser Funktion ansprechbar.
- Alle globalen Variablen werden nach ANSI C mit **dem Wert 0 initialisiert**. Aus didaktischen Gründen rate ich Ihnen aber an, alle globalen und lokalen Variablen selber zu initialisieren, falls dies für einen korrekten Programmablauf gefordert wird. Durch dieses Vorgehen müssen Sie sich stärker mit der Funktionsweise des Programms auseinandersetzen und Fehler, die bei der Nichtinitialisierung bei lokalen Variablen auftreten können, werden dadurch minimiert.
- Verwenden Sie globale Variablen möglichst selten**, da diese dem Konzept der modularen Programmierung widersprechen. Sind Variablen global, so kann keine Abschottung der Daten gegenüber unerwünschten Zugriffen gewährleistet werden. Sie stellen somit meist einen Unsicherheitsfaktor des Systems dar! Durch spezielle Schlüsselwörter kann aber auch dieses Verhalten der Variablen noch weiter beeinflusst werden. (Siehe dazu den Abschnitt: "Schlüsselwörter die einen Einfluss auf das Verhalten der Variablen haben")
- Globale Variablen **gehen nicht wirtschaftlich mit dem Speicher um**, da Sie den Speicher die ganze Zeit fest belegen.

9.3 Schlüsselwörter unter ANSI C die das Verhalten von Variablen beeinflussen

Schlüsselwort:	Lokale Variablen:	Globale Variablen:
extern		<p>Das Schlüsselwort: extern gibt an, dass diese Variable schon extern definiert wurde und deshalb keine neue Definition benötigt. Mit dieser Variablen-Deklaration wird eine anderswo schon definierte Variable zur Verfügung gestellt.</p> <p>Beispiel:</p> <pre>extern int objektanzahl;</pre>

static	<p>Mit dem Schlüsselwort: static wird eine lokale Variable zu einer statischen Variable. Dies bedeutet, dass diese Variable ihren Wert beim Verlassen der Funktion nicht verliert und beim erneuten Aufruf der Funktion weiter verwenden kann.</p> <p>Diese Art von Variablen erhält genauso wie die globale Variablen automatisch bei der Definition den Anfangswert 0.</p> <p>Beispiel: <code>static int werte;</code></p>	<p>Bei globalen Variablen wird durch dieses Schlüsselwort der Gültigkeitsbereich eingeschränkt. Dies bedeutet, dass diese globale Variable nur denjenigen Funktionen zur Verfügung steht, die in der gleichen Quelldatei vorhanden sind ("modulglobal").</p> <p>Beispiel: <code>static int werte;</code></p>
---------------	---	--

10 Generelle Hinweise zu den Funktionen

- Jede Funktion sollte eine klar definierte Aufgabe erfüllen. Der Funktionsname sollte unmissverständlich Aufschluss über den Zweck einer Funktion geben. Dies erhöht die Wiederverwendbarkeit vom Programmcode.
- Als Faustregel für die Länge einer Funktion sollte gelten, dass sie nicht mehr als eine Seite umfasst. Überlange Funktionen oder endlose Parameterlisten deuten darauf hin, dass ihre Aufgaben nicht klar eingegrenzt sind. Sie sollten daher besser in mehrere kleine Funktionen zerlegt werden.
- Deklarieren Sie alle benutzten Funktionen mit Hilfe von Prototypen. Wenn Sie Bibliotheksfunktionen in Anspruch nehmen, dann sollten Sie mit einer `#include`-Anweisung immer die dazugehörige Header-Datei einfügen. Sie enthält den Prototyp dieser Funktion.
- Der Prototyp, die Definition und der Aufruf einer Funktion müssen in Anzahl, Typ und Reihenfolge der Argumente und Parameter übereinstimmen und zudem den gleichen Rückgabewert voraussetzen.
- Beenden Sie alle Funktionen, die nicht vom Typ: `void` sind, stets mit einer `return()`-Anweisung. Andernfalls ist das Ergebnis der Funktion undefiniert, was bei Funktionsaufrufen innerhalb von Ausdrücken zu „mysteriösen“ Fehlern führen kann.
- Sämtliche Anweisungen nach einem `return`-Statement werden ignoriert. Der Compiler meldet sich dann mit einer Warnung wie: "Code has no effect in function xy".
- Funktionsdeklarationen müssen mit einem Semikolon abgeschlossen werden; hingegen darf bei Funktionsdefinitionen kein Semikolon nach der Parameterliste stehen.
- Wenn Sie bei der Verwendung von Bibliotheksfunktionen das Einfügen der entsprechenden Header-Datei vergessen, führt dies bei den meisten Compilern zumindest zu einer Warnung. Leider gibt es aber auch Situationen, bei denen keine Warnung ausgegeben wird und das Programm dann ein merkwürdiges Verhalten aufweist. Dieser Effekt ist meistens im Zusammenhang mit selbst geschriebenen Funktionsbibliotheken anzutreffen.

Abbildungen:

Abbildung 1: Darstellung von Funktionsaufrufen	6
--	---

Codeumsetzung:

Codeumsetzung 1: Einbindung einer Bibliothek in ANSI C	4
Codeumsetzung 2: Allgemeine Form einer Funktion	7
Codeumsetzung 3: Funktionsdeklaration in ANSI C	9
Codeumsetzung 4: Funktionen ohne Rückgabewert in ANSI C	16
Codeumsetzung 5: Funktionen mit Rückgabewert in ANSI C	16

Codebeispiele:

Codebeispiel 1: Einbindung einer Bibliothek in ANSI C	4
Codebeispiel 2: Funktion: CalculateFaculty() ohne Funktionsdeklaration in ANSI C	8
Codebeispiel 3: Funktionsprototyp: CalculateFaculty() in ANSI C	10
Codebeispiel 2: Funktion: CalculateFaculty() mit Funktionsdeklaration in ANSI C	10
Codebeispiel 5: Funktion: CalculateMaxValue() mit Funktionsdeklaration in ANSI C	12
Codebeispiel 6: Falsche Umsetzung der Funktion: CalculateMaxValue() in ANSI C	13
Codebeispiel 6: Richtige Umsetzung von: CalculateMaxValue() mittels „Call by Value“ in ANSI C	14

Historie

Vers.	Bemerkungen	Verantwortl.	Datum
0.1	- erstes Zusammenführen der Informationen	W. Odermatt	28.07.2005
0.2	- erster Entwurf fertig gestellt	W. Odermatt	15.08.2005
0.3	- Fehlerkorrekturen	W. Odermatt	18.08.2005
1.0	- letzte Layoutanpassungen gemacht	W. Odermatt	11.09.2005
1.1	- Textüberarbeitung und Korrekturen	W. Odermatt	11.09.2006
1.2	- Programmkorrekturen	W. Odermatt	27.08.2009
2.0	- Anpassungen und Erweiterungen an neue Programmiersprache C#	W. Odermatt	21.02.2016

Referenzunterlagen

LfNr.	Titel / Autor / File / Verlag / ISBN	Dokument-Typ	Ausgabe
1	„Programmiersprache C“, Implementierung C, Grundkurs / H.U. Steck	Theorieunterlagen	2003
2	„Programmieren in C“ / W. Sommergut / Beck EDV-Berater im dtv / 3-423-50158-8	Fachbuch	1997
3	„C Kompakt Referenz“ / H. Herold / Addison Wesley / 3-8273-1984-6	Fachbuch	2002
4	„C in 21 Tagen“ / P. Aitken, B. L. Jones / Markt + Technik / 3-8272-5727-1	Fachbuch	2000
5	„C Programmieren von Anfang an“ / H. Erlenkötter / rororo / 3-499-60074-9	Fachbuch	2001
6	„Programmieren in C“ / B.W. Kernighan, D.M. Ritchie / Hanser / 3-446-25497-3	Fachbuch	1990
7	„Einstieg in Visual C# 2013“ / Thomas Theis / Galileo Computing / 978-3-8362-2814-5	Fachbuch	2014
8	„Visual C# 2012“ / Andreas Kühnel / Rheinwerk Computing / 978-3-8362-1997-6	Fachbuch	2013
9	„Objektorientiertes Programmieren in Visual C#“ / Peter Loos / Microsoft Press / 3-86645-406-6	Fachbuch	2006