C-Cheatsheet

Das vorliegende Cheatsheet zeigt die Syntax der Programmiersprache C mit Beispielen, gefolgt von jeweils einem Quiz / Selbsttest, um das Verständnis zu prüfen. Zum schnellen Nachschlagen können Sie auch die **C-Kurzreferenz** verwenden, eine tabellarische und sortierbare Übersicht der wichtigsten C-Befehle. Als Entwicklungsumgebung verwenden wir **Visual Studio Community Edition**, diese unterstützt C-Programmierung als Teilmenge der C++-Entwicklung und man kann relativ unkompliziert von C zu C++ übergehen. Für das Testen kleiner Codefragmente kann der Online-Compiler **OnlineGDB** verwendet werden, dort im Dropdown rechts oben als Programmiersprache C auswählen.

Vorab: Infos rund um die C-Programmierung

C ist eine **prozedurale Programmiersprache**, d.h. Programme werden mit Hilfe von Funktionen / Prozeduren strukturiert, die Daten (Variablen, Arrays, Strukturen) verarbeiten. Mittels **Sequenzen** von Einzelanweisungen (Variablen und Arrays deklarieren, Eingabe, Ausgabe, ...), **Verzweigungen**, **Schleifen** und **Funktionen** kann man Lösungsalgorithmen für eine Vielzahl von Aufgabenstellungen in C implementieren.

Mit C werden vor allem **Konsolenprogramme** geschrieben, Betriebssysteme und Mikrocontroller programmiert. Die Stärke der Programmiersprache C liegt darin, dass sie **hardware-nah** ist, mit Hilfe von Zeigervariablen kann man z.B. direkt auf Adressbereiche zugreifen. Für die Entwicklung grafischer Benutzeroberflächen bieten objektorientierte Sprachen wie C++, Java oder C# eine bessere Unterstützung.

Vor der Programmierung ist es hilfreich, den Algorithmus bzw. Programmablauf mit Hilfe eines **Flussdiagramms** oder eines **Struktogramms** zu visualisieren. Flussdiagramme und Struktogramme sind Diagramme mit standardisierten Elementen, deren Erstellung durch verschiedene Tools unterstützt wird, z.B. yEd Graph Editor.

Übersicht

1. Erste Schritte mit C

C-Quellcode | Header-Datei | main-Funktion | include | define | Kommentar | Compiler | IDE

Top

Neue Quellcode-Datei anlegen

Ein minimales C-Programm besteht aus einer einzelnen **Quellcode-Datei mit der Endung *.c**, zum Beispiel myprog.c, die die **Anweisungen** des Programms sowie **Kommentare** enthält. Größere C-Programme können noch weitere benutzerdefinierte **Header-Dateien** (Endung *.h) und **Quellcode-Dateien** (Endung *.c) enthalten.

Mit Hilfe der **#include-Direktiven** werden benötigte Programm-bibliotheken über ihre Headerdateien importiert, danach können die darin enthaltenen Funktionen verwendet werden. **#define**-Direktiven werden ebenfalls an den Anfang des Programms gestellt, mit ihrer Hilfe werden Konstante und sogenannte Makros definiert. Jedes C-Programm enthält genau eine **main-Funktion**, diese ist der Einstiegspunkt des Programms. Die Anweisungen des Programms werden in die **main()-Funktion** geschrieben, ggf. auch in weitere selbststefinierte Funktionen. Jede Anweisung wird mit einem Semikolon beendet! Mehrzeilige **Kommentare** werden mit /* und */ umrahmt, einzeilige **Kommentare** mit // eingeleitet.

```
1. /* myprog.c  */
2. // TODO: Hier include-Direktiven einfügen
3. #include <stdio.h>
4. #include <stdlib.h>
5. // TODO: Hier define-Direktiven einfügen
6. //
7. int main(void){
8.   // TODO: Fügen Sie Ihre Anweisungen hier ein!

9.   printf("Hallo!\n"); // Ausgabe des Textes Hal

10. }
```

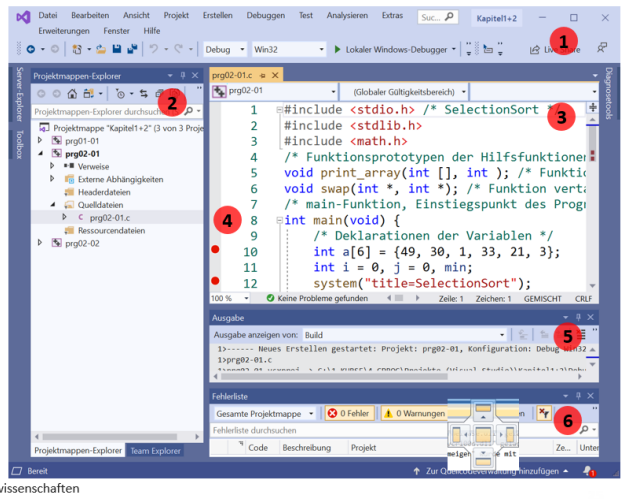
Programm erstellen und ausführen

Um aus C-Quellcode ein ausführbares Programm zu erstellen, benötigt man einen **C-Compiler**. Integrierte Entwicklungsumgebungen (engl. **Integrated Development Environment, IDE**) für C-Programmierung sind z.B. **Eclipse IDE for C/C++**, **NetBeans**, **Code::Blocks** oder **Microsofts Visual Studio**, diese enthalten einen C-Compiler und unterstützen darüber hinaus die Programmierung mit Syntaxhighlighting, Fehlersuche etc.

Der Screenshot zeigt die Anordnung der Fenster in **Visual Studio Community Edition**: Toolbar, Projekte, Source Code Editor, Ausgabefenster und Fehlerliste. Alle Entwicklungsumgebungen (IDEs) haben ähnliche Default-Fensterkonfigurationen, die den Entwicklungsprozess unterstützen.

1. Toolbar
2. Projekte
3. Source Code Editor
4. Haltepunkte
5. Ausgabefenster
6. Fehlerliste

Der Screenshot zeigt links (2) die Projekte der Projektmappe Kapitel1+2. Das ausgewählte Projekt ist prg02-01 und enthält die Quelltext-Datei prg02-01.c.
Rechts oben (3) wird der Quelltext angezeigt.
Die Größe und Anordnung der Fenster kann durch Ziehen verändert werden.



Hochschule Kaiserslautern University of Applied Sciences — Angewandte Ingenieurwissenschaften, Prof. Dr. E. Kiss

2. Variablen und Konstante

Variable | Konstante | Datentyp | Typumwandlung | Operatoren | sizeof | enum

Top

2-1 Variablen deklarieren

Variable sind benannte Speicherplätze, in denen die Daten des Programms gespeichert werden, z.B. a, b, c, text. Den Wert einer Variablen kann man verändern und überschreiben.

Syntax

Bei der Deklaration einer Variablen wird mit einem **Datentyp**(int, long long, float, double, char, ...) festgelegt, welche Art von Werten in dieser Variablen gespeichert werden kann. Dem Datentyp kann auch das Schlüsselwort **unsigned** vorangestellt werden, dies bedeutet, dass vorzeichenlose Zahlenwerte verwendet werden sollen. Deklariert man mehrere Variablen desselben Datentyps, werden sie mit Komma getrennt.

2-2 Konstante deklarieren

Im Unterschied zu Variablen bleibt der Wert einer **Konstanten** immer gleich. Konstanten kann man mit der Präprozessor-Direktive **#define** als Makro oder mit dem Schlüsselwort **const** deklarieren. **#define**-Makros werden ohne Zuweisung (=) und ohne Semikolon am Ende definiert!

Syntax

```
datentyp variablen_name;
datentyp varname1, varname2;
```

Beispiel

Variablen deklarieren

```
1. int a = 0, b; // ganze Zahlen
2. unsigned int c = 100; // vorzeichenlose ganze Z

3. float x; double y; // Zahlen mit Nachkommastell

4. char z; char * text; // Zeichen und Zeichenkett
```

Der **sizeof** -Operator gibt die Größe des Speicherbedarfs einer Variablen / eines Objektes in Bytes an.

Beispiel

sizeof-Operator

In diesem Beispiel finden wir den Speicherverbrauch der Variablen a und x heraus und stellen fest: Variablen des Datentyps int werden mit 4 Byte, Variablen des Datentyps double werden mit 8 Byte gespeichert.

```
1. // Ganze Zahl
2. int a = -100;
3. // Fließkommazahl mit doppelter Genauigkeit
4. double y = 0.999;
5. // groesse_von_a = 4 Byte = 32 Bit
6. int groesse_von_a = sizeof(a);
7. // groesse_von_y = 8 Byte = 64 Bit
8. int groesse_von_y = sizeof(y);
9. printf("%d %d\n", groesse_von_a, groesse_von_y)
```

Der **sizeof** -Operator wird z.B. bei der dynamischen Speicherallokation verwendet, in Kombination mit den Funktionen malloc und malloc.

Starte das Quiz "C-Grundlagen"

```
#include <stdio.h>
#define name wert // (1)
int main(void){
    const datentyp name; // (2)
}
```

Beispiel

Konstante mit define oder const

Hier wird die Konstante PI einmal als Makro mit #define und einmal mit const deklariert. Ihr Wert kann durch Zuweisung nachher nicht mehr veränder werden.

```
1. #define PI 3.14159 // Konstante mit #define
2. int main(void){
3.   const double PI2  = 3.14159; // Konstante mit

4.   PI  = 3.14; // FEHLER!
5.   PI2  = 3.141; // FEHLER!
6. }
```

In C können auch **Aufzählungen** verwendet werden, um Listen ganzzahliger Konstanten zu verwalten, dies mit Hilfe des Schlüsselworts **enum**.

Syntax

Eine **Aufzählung** wird durch das Schlüsselwort **enum** eingeleitet, gefolgt von einem selbstdefinierten Aufzählung-Namen und einer Liste von Elementen, die in geschweifte Klammern gesetzt werden. Den Elementen einer Aufzählung werden defaultmäßig ganzzahlige Werte zugewiesen, die bei 0 beginnen, es können jedoch auch eigene Werte vergeben werden.

```
enum ENUM_NAME {elem_1, ..., elem_n}; // (1)
int main(void){
    enum ENUM_NAME var = elem_i; // (2)
}
```

Beispiel

Konstanten mit enum deklarieren

Die Konstante **antw** hat den Datentyp **enum ANTWORT** und kann nur einen der festgelegten Werte: **ja, nein, vielleicht** annehmen.

```
1. enum ANTWORT { ja=10, nein=20, vielleicht=15 };

2. int main(void) {
3.     enum ANTWORT antw = vielleicht;
4.     printf("Antwort = %d\n", antw); // Ausgabe:

5. }
```

Starte das Quiz "C-Grundlagen"

2-3 Zuweisung und Typumwandlung

Einer zuvor deklarierten Variablen kann man mit Hilfe des = Zeichens Werte zuweisen. Dabei ist der Datentyp zu beachten, einer int-Variablen weist man ganzzahlige Werte zu, einer float-Variablen Kommazahlen. Hat der zugewiesene Wert einen anderen Datentyp, wird er implizit in den Datentyp der Variablen umgewandelt.

2-4 Berechnungen

Berechnungen werden durchgeführt, indem Variablen über **Operatoren** (+, -, *, /, %, ...) zu Ausdrücken verknüpft werden, dabei ist die Priorität der Operatoren zu berücksichtigen. C verfügt über Operatoren für **abkürzende Zuweisung** (+=, *= etc.), **Inkrementierung**und Dekrementierung (++, --),

Typumwandlung kann auch auch explizit erfolgen, indem man den Namen eines Datentyps vor den Namen des zugewiesenen Wertes schreibt. Die Zuweisung **double x = (double)10;** bewirkt, dass 10 als Fließkommazahl mit doppelter Genauigkeit gespeichert wird.

Syntax

```
variablen_name = variablen_wert;
variablen_name = (datentyp)variablen_wert;
```

Beispiel

Zuweisung

```
1. int a = 0; double b = 0.0; char z = ' ';
2. // Zuweisung des Wertes 10 an die Variable a
3. a = 10;
4. // Zuweisung des Wertes 20.5 an die Variable b
5. b = 20.5;
6. // Zuweisung des Zeichens a an die Variable z
7. z = 'a';
8. // Nachkommastellen werden abgeschnitten, a = 3

9. a = 3.99;
```

Beispiel

Typumwandlung

```
1. int a = 10; float b = 1.2345;
2. double c = 1.9;
3. // Implizite Typumwandlung a = 10
4. a = 10.2;
5. // Explizite Typumwandlung, a = 1
6. a = (int)b;
7. // Explizite Typumwandlung, a = 1
8. a = (int)c;
9. // Explizite Typumwandlung, b = 20.00(...)
10. b = (double)20;
11.
```

Starte das Quiz "C-Grundlagen"

Vergleichsoperatoren, deren Ergebnis WAHR oder FALSCH ist (==, !=, >, <, >=, <=) und **logische Operatoren** (&&, ||, !), deren Ergebnis WAHR oder FALSCH ist. Die Auswertung eines Ausdrucks erfolgt entsprechend einer festgelegten Priorität der Operatoren, die durch Klammern beeinflusst werden kann: a * b + c ist wie (a * b) + c, jedoch anders als a * (b + c).

Beispiel

Operatoren in C

```
1. int a = 25, b = 3, c = 0;
2. // Inkrementierung
3. a += 1; // a = 26
4. // Abkürzende Zuweisung
5. b *= 2; // b = 6
6. // Modulo-Operator: Rest der Teilung oh
7. c = 17 % 5; // c = 2
8. // Fließkommadivision: Typumwandlung erford

9. double res1 = (double) a / b; // res1 = 4.3

10. // Ganzzahldivision
11. int res2 = a / b; // res2 = 4
12. // Ausgabe: 26, 6, 4.33, 4
13. printf("%d, %d, %.2f, %d\n", a, b, res1, re

14. // Vergleichs-Operatoren: ==, !=, <, >, <=,
```

Beispiel

Berechne Flächeninhalt des Dreiecks mit Seiten a, b, c

```
1. #include<math.h>
2. int main(void){
3.     double a = 10, b = 20, c = 20;
4.     double s = (a + b + c ) / 2;
5.     double F = sqrt(s * (s-a) * (s-b) * (s-c));
6. }
```

Starte das Quiz "C-Grundlagen"

3. Ausgabe und Eingabe

printf scanf Formatzeichen

Top

3-1 Die printf-Funktion

In C erfolgt die Ausgabe auf die Konsole mit Hilfe der **printf** -Funktion. Die Werte von Variablen können mit Hilfe von **Formatzeichen** in eine formatierende Zeichenkette eingefügt werden. Zu jedem Datentyp gehören passende **Formatzeichen**: %d oder %i für int (ganze Zahlen), %f für float (Fließkommazahlen mit einfacher Genauigkeit), %lf für double (Fließkommazahlen mit doppelter Genauigkeit).

Syntax

Ausgabe einer Zeichenkette

```
printf(zeichenkette);
```

Formatierte Ausgabe mit Platzhaltern für Variablen

3-2 Die scanf-Funktion

In C werden Werte von der Konsole mit Hilfe der Funktion **scanf** eingelesen Die Funktion scanf erhält als ersten Parameter eine formatierende Zeichenkette, z.B. "%d %lf", gefolgt von einer Liste von Variablen, deren Anzahl und Datentyp zu den Formatbeschreibern passen muss. Jeder Variablen muss ein &-Zeichen vorangestellt werden, das die Adresse der Variablen bezeichnet. In Visual Studio wird scanf_s anstelle von scanf verwendet!

Syntax

```
scanf("formatzeichen", &variablenname);
// In Visual Studio:
scanf_s("formatzeichen", &variablenname);
```

```
printf(formatierende_zeichenkette,
variablen_liste);
```

Beispiel

Formatierte Ausgaben

```
int a = 0; double b = 3.33; char z = 'a';
printf("Hallo zusammen!\n");
printf("a = %d, b = %5.2f, z = %c\n", a, b, z);
```

Beispiel

Hier werden drei Variablen eingelesen. Vor jedem Einlesen wird eine Eingabe-Aufforderung für den Anwender ausgegeben. Wichtig: in scanf sollten die Formatzeichen %d, %f, %lf, %c ohne weitere Texte oder Steuerzeichen verwendet werden.

```
1. int a = 0; double b = 0.0; char z = '';
2. printf("Eingabe a: ");  scanf("%d", &a);
3. printf("Eingabe b: ");  scanf("%lf", &b);
4. printf("Eingabe c: ");  scanf("%c", &z);
```

4. Bedingte Verzweigungen und Fallunterscheidungen

if | else | else if | switch-case

Top

4-1 if-else-Anweisung

Eine bedingte Verzweigung ist eine Kontrollstruktur, die festlegt, welcher von zwei (oder mehr) Anweisungsblöcken, abhängig von einer (oder mehreren) Bedingungen, ausgeführt wird. Sie wird in C, wie in fast allen Programmiersprachen, mit der if-else-Anweisung abgebildet.

Syntax

if-else-Anweisung

Es kann keinen, einen oder mehrere else-if-Teile geben. Die geschweiften Klammern werden dann benötigt, wenn es mehrere Anweisungen in einem Block gibt.

```
1. if (bedingung_1) {
2.     anweisungen_1
3. }
4. else if (bedingung_2){
5.     anweisungen_2
6. }
7. else {
8.     anweisungen_default
9. }
```

Beispiel

Berechne Zinssatz abhängig von Betrag

Welcher Zinssatz wird für betrag = 20000 berechnet?

```
1. float betrag = 10000.0, zinssatz = 0.0;
2. if (betrag > 50000)
3.   zinssatz = 1.0;
4. else if (betrag > 10000)
5.   zinssatz = 0.5;
6. else if (betrag > 0)
7.   zinssatz = 0.2;
8. else
9.   zinssatz = -0.2;
```

4-2 switch-case-Anweisung

Die **switch-case**-Anweisung ist eine spezielle bedingte Verzweigung, die verwendet wird, wenn es viele Fallunterscheidungen gibt.

Syntax

switch-case-Anweisung

Ein Ausdruck wird mit verschiedenen Werten verglichen und bei Übereinstimmung wird die entsprechende Anweisung ausgeführt. Die **break**-Anweisung bewirkt, dass der case-Zweig sofort verlassen wird. Wenn break in einem Zweig weggelassen wird, werden auch die folgenden case-Zweige ausgeführt, bis ein break gefunden wird, oder die switch-Anweisung zu Ende ist.

```
1. switch(ausdr){
2.     case ausdr_1:
3.       anweisungen_1
4.       break;
5.     case ausdr_2:
6.       anweisungen_2
7.       break;
8.     ...
9.     default:
10.      anweisungen_default
11. }
```

Beispiel

Ausgabe abhängig von Wahl

```
1. int wahl;
2. printf("Eine der Zahlen 1, 2, 3 eingeben: ");
3. scanf("%d",&wahl);
4. switch(wahl){
5.   case 1:
6.     printf("Erste Wahl\n");break;
7.   case 2:
8.     printf("Zweite Wahl\n");break;
9.   case 3:
10.    printf("Dritte Wahl\n");break;
11.  default:
12.    printf("Fehler!\n");
13. }
```

5. Schleifen

while **do-while** **for** **break** **continue**

Top

C unterstützt drei Arten von Schleifen, die sich darin unterscheiden, wo die Schleifenbedingung geprüft wird: die **while-Schleife** verwendet eine Ausführungsbedingung, die vor dem Ausführen des Schleifenrumpfs geprüft wird, die **do while**-Schleife funktioniert über eine Abbruchbedingung, die nach dem Ausführen des Schleifenrumpfs geprüft wird, und die **for-Schleife** verwendet eine Zählvariable, für die Startwert, Endwert und Schrittweite festgelegt werden. Eine Schleife kann mit **break** sofort verlassen werden, einzelne Schleifenschritte können mit **continue** übersprungen werden.

5-1 while- und do-while-Schleife

Eine **while-Schleife** ermöglicht es, Anweisungen wiederholt auszuführen, und zwar so lange, wie eine Ausführungsbedingung erfüllt ist. Dabei wird die Variable, die in der Bedingung abgefragt wird, nicht automatisch heraufgesetzt, sondern muss im Schleifenrumpf explizit inkrementiert werden. Die **do-while-Schleife** funktioniert ähnlich, allerdings wird die Bedingung ans Ende gestellt, d.h. die Anweisungen werden auf jeden Fall mindestens einmal durchgeführt.

Syntax

```
while (bedingung){
  anweisungen
}
do {
  anweisungen
} while (bedingung);
```

Beispiel

Berechne Summe 1 + 2 + ... + 5

```
1. double sum = 0.0; int i = 1;
2. while (i <= 5) {
3.    sum += i;
4.    i += 1;
5. }
6. printf("Summe: %.2f, i: %d\n", sum, i);
7. sum = 0.0; i = 1;
8. do {
9.    sum += i;
10.   i += 1;
11. } while (i <= 5);
12. printf("Summe: %.2f, i: %d\n", sum, i);
```

5-2 for-Schleife

Eine **for-Schleife** ist eine Zählschleife, die für eine Zählvariable eine Start- und Endbedingung sowie eine Schrittweite festlegt. Die Anweisungen werden für eine vorgegebene Anzahl an Schleifen-Durchläufen wiederholt. Die Angabe der Anzahl wird über die Zählvariable umgesetzt, die automatisch nach jedem Schleifendurchlauf um 1 (bzw. eine andere Schrittweite) erhöht wird.

Syntax

```
for (count = start;count <= end;count +=
schritt){
  anweisungen
}
```

Beispiel

Berechne Summe 1 + 2 + ... + 5

```
1. double sum = 0.0;
2. for(int i=1;i<=5;i++){
3.    sum += i;
4. }
5. printf("Summe:\n");
6. printf("%.2f\n", sum);
```

Im folgenden Beispiel wird der Befehl **break** verwendet, um eine Schleife bei Erfüllung einer Bedingung zu verlassen.

Beispiel

Finde ungeraden Teiler von n

```
1. int n = 20;
2. for (int i = 3; i <= n / 2; i += 2) {
3.    if (n % i == 0) {
4.       printf("Ungerader Teiler gefunden: %d !", i
5.       break;
6.    }
7. }
```

6 Funktionen

Parameter **Referenzparameter** **Rückgabewert** **void** **return**

Top

Eine **Funktion** ist ein benannter Codeblock, der nur ausgeführt wird, wenn er in einer anderen Funktion verwendet wird, dies nennt man den **Funktionsaufruf**. Funktionen werden einmal definiert und können dann beliebig oft aufgerufen werden.

Man kann Daten oder sogenannte **Parameter** an eine Funktion übergeben, entweder **als Wert oder als Referenz**. **Referenzparameter** werden als Zeigervariablen bzw. Adressen übergeben, sie haben die Besonderheit, dass ihr Wert in der aufrufenden Funktion weiterverwendet werden kann. Eine Funktion kann auch einen **Rückgabewert** haben, d.h. einen Wert zurückgeben.

6-1 Funktionen ohne Rückgabewert

Eine **Funktion ohne Rückgabewert** bzw. mit Rückgabetyp **"void"** ist eine benannte Gruppierung von Anweisungen. Innerhalb der Funktion können neue Werte berechnet und direkt ausgegeben werden.

Syntax

Die Platzhalter typ_i stehen für Datentypen, param_i sind Parameter-namen, und arg_i die tatsächlichen Argumente. Anzahl, Reihenfolge und Datentyp muss bei den Parametern und den tatsächlichen Argumenten übereinstimmen.

```
1. /* (2) Funktionsprototyp */
2. void myfunc(typ_1, typ_2, ... typ_n);
3. int main(void){
4. /* (3) Funktionsaufruf */
5.   myfunc(arg_1, arg_2, ... arg_n);
6. }
7.
8. /* (1) Funktionsdefinition */
9. void myfunc(typ_1 param_1, ... typ_n param_n){
10.   // Funktionsrumpf mit Anweisungen
11.   // . . .
12. }
```

Beispiel

Formatierungsfunktion AusgabeInEuro()

```
1. #include <stdio.h>
2. void AusgabeInEuro(double);
3. int main(void){
4.   double x = 49.99;
5.   // Erster Funktionsaufruf
6.   AusgabeInEuro(x);
7.   // Zweiter Funktionsaufruf
8.   AusgabeInEuro(2.5);
9. }
10. /* Funktionsdefinition */
11. void AusgabeInEuro(double betrag){
12.     printf("Betrag: %.2f Euro\n", betrag);
13. }
```

Beispiel

Funktion mit Referenzparametern

Der Referenzparameter m speichert den Mittelwert der Wert-Parameter a und b und soll in der main-Funktion weiter verwendet werden. Beachte: m wird als Zeigervariable übergeben, daher *m in Zeile 1 und 2 und &m in Zeile 6.

6-2 Funktionen mit Rückgabewert

Eine Funktion kann auch Daten/Parameter als **Rückgabewert** zurückgeben, diese können in der aufrufenden Funktion in Berechnungen oder Ausgaben weiter verwendet werden. Damit eine Funktion einen Wert zurückgeben kann, benutzt man das Schlüsselwort **"return"**.

Syntax

Der Parameter r_typ bezeichnet den Datentyp des Rückgabewertes. Wichtig: Eine Funktion kann nur einen Rückgabewert haben!

```
1. /* (2) Funktionsprototyp */
2. r_typ myfunc(typ_1, typ_2, ... typ_n);
3. int main(void){
4.   /* (3) Funktionsaufrufe */
5.   r_wert_a = myfunc(arg_1a,... arg_na);
6.   r_wert_b = myfunc(arg_1b,... arg_nb);
7. }
8. /* (1) Funktionsdefinition */
9. r_typ myfunc(typ_1 param_1, ... typ_n param_n){

10.   r_typ r_wert;
11.   // Anweisungen, die den r_wert berechnen
12.   // Rückgabewert mit return zurückgeben
13.   return r_wert;
14.   }
```

Beispiel

Funktion brutto_aus_netto()

Die Funktion **brutto_aus_netto()** berechnet für einen gegebenen Netto-Betrag und eine gegebene Mehrwertsteuer den Bruttobetrag. Sie hat zwei Parameter, netto (Datentyp double) und mwst (Datentyp int) und einen Rückgabewert vom Datentyp double.

```c
#include <stdio.h>
void Mittelwert(double a, double b, double *m){

    *m = (a + b ) / 2;
}
int main(void){
    double x = 2.0,y = 4.0, m = 0;
    Mittelwert(x, y, &m);
    printf("Mittelwert= %d\n", m);
}
```

Starte das Quiz "C-Funktionen"

```c
double brutto_aus_netto(double, int);
int main(void){
  // Erster Funktionsaufruf
  double betrag = 1000.95; int mwst = 19;
  printf("Brutto = %.2f\n",
         brutto_aus_netto(betrag, mwst) );
  // Zweiter Funktionsaufruf
  printf("Brutto = %.2f\n",
         brutto_aus_netto(1234.55, 19) );
}
/* Funktionsdefinition */
double brutto_aus_netto(double netto, int mwst)

  double brutto = netto + netto * mwst / 100;
  return brutto;
}
```

Die Argumente im Funktionsaufruf müssen in Anzahl und Datentyp mit den Parametern in der Definition übereinstimmen, jedoch nicht im Namen! D.h. der Name des übergebenen Argumentes (hier: betrag) in der aufrufenden Funktion muss nicht mit dem Namen des Parameters (hier: netto) übereinstimmen.

Starte das Quiz "C-Funktionen"

7 Arrays

Element Index Zufallszahlen rand

Top

Ein **Array** ist eine **statische Datenstruktur**, die **Elemente desselben Datentyps** in einem zusammenhängenden Speicherbereich ablegt. "Statische Datenstruktur" bedeutet, dass die Größe des Speicherbereichs vorab fest reserviert ist und nicht mehr verändert werden kann. Der Zugriff auf die Elemente des Arrays erfolgt über einen **Index**, der in C bei 0 beginnt.

Eindimensionale Arrays

Eindimensionale Arrays werden verwendet, um Listen zu speichern. Die maximale Größe N des Arrays, d.h. die Anzahl an Elementen, die darin gespeichert werden können, muss zuvor als Konstante definiert werden.

Syntax

```c
const int N = 10;
datentyp array_name[N];
```

Beispiel

```c
double list[100]; // double-Array mit 100 Eleme

int a[10], b[10]; // Zwei int-Arrays
char text[1000];  // Array des Datentyps char
```

Beispiel

Array mit Zufallszahlen befüllen

Zweidimensionale Arrays

Zweidimensionale Arrays werden verwendet, um Tabellen und Matrizen zu speichern. Sie werden deklariert, indem nach dem Arraynamen in eckigen Klammern zuerst wird die Zeilen-Dimension M und danach die Spalten-Dimension N angegeben wird.

Syntax

```c
#define M 10 // Zeilen-Dimension
#define N 20 // Spalten-Dimension
int main(void){
   datentyp array_name[M][N];
}
```

Beispiel

```c
// 2x2 Einheitsmatrix
int a[2][2] = { {1, 0}, {0, 1} };
// double-Array mit 2 Zeilen und 3 Spalten
double a[2][3] = {0.0};
```

Beispiel: Array mit Zufallszahlen befüllen

Die Zuweisung von Werten an die Elemente eines Arrays erfolgt über Schleifen, ebenso die Ausgabe. Die Verwendung selbstdefinierter Funktionen für Ein- oder Ausgabe (hier: print_array) ist nützlich, dabei ist auf die korrekte Verwendung des **Arrays als Funktionsparameter** zu achten: Bei der Funktionsdefinition schreibt man den Namen des Arrays, gefolgt von eckigen Klammern, beim Funktionsaufruf den Namen des Arrays ohne eckige Klammern.
Funktionsdefinition mit Array-Parameter: void print_array(double **x[]**, int n){...}
Funktionsufruf mit Array-Parameter: print_array(**x**, n), print_array(**a**, n) etc.

In C können Pseudo-Zufallszahlen mit Hilfe der Funktion **rand()** erstellt werden.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #define N 10
4. void print_array(double[], int);
5. int main(void) {
6.     double a[N] = { 0 }, b[N] = {0}; int n

7.     printf("Anz. Elemente: "); scanf("%d", &

8.     srand(1);
9.     for (int i = 0; i < n; i++)
10.      a[i] = (rand() % 100) / 10.0;
11.    print_array(a, n);
12.    print_array(b, n);
13. }
14. void print_array(double x[], int n) {
15.   for (int i = 0; i < n; i++)
```

Starte das Quiz "C-Arrays"

Ein zweidimensionales Array wird zeilenweise befüllt bzw. ausgelesen, dafür benötigt man zwei for-Schleifen: die äußere mit Index i durchläuft die Zeilen, die innere mit Index j durchläuft die Spalten. Um eine tabellarische Ausgabe zu erreichen, werden die Elemente einer Zeile mit Leerzeichen getrennt und nach jeder Zeile wird ein Zeilenumbruch eingefügt.

```
1. #include <stdio.h>
2. #define M 10 // Anzahl Zeilen
3. #define N 10 // Anzahl Spalten
4.
5. int main(void) {
6.  double a[M][N];
7.  for(int i=0;i<M;i++){
8.      for(int j=0;j<N;j++){
9.        a[i][j] = rand()%100;
10.       printf("%5.2f  ", a[i][j]);
11.     }
12.     printf("\n"); // Zeilenumbruch
13.   }
14. }
```

Starte das Quiz "C-Arrays"

8 Adressen und Zeiger

Adresse | Zeiger | Referenzparameter | Zeichenketten | Dynamische Speicherallokation | malloc | calloc

Top

Eine **Zeigervariable** (engl. pointer) ist eine Variable, die die **Adresse** einer anderen Variablen speichern kann. Die Operationen, die für "normale" Variablen definiert sind, sind für Zeigervariablen nicht definiert, bis auf eine Ausnahme: man kann Zeigervariablen inkrementieren oder dekrementieren und damit einen Speicherbereich durchlaufen.

Zeiger deklarieren

Um eine **Zeigervariable** zu definieren, wird dem Namen ein Stern * vorangestellt. Um einer Zeigervariablen die Adresse einer anderen Variablen zuzuweisen, wird der **Adress-Operator &** verwendet. Zeigervariablen müssen denselben Datentyp haben wie die Variablen, auf die sie zeigen.

Syntax

```
datentyp variable;
datentyp *zeiger;
zeiger = &variable;
```

Beispiel

Anwendungsgebiete von Zeigern

Zeiger haben drei wichtige Anwendungsbereiche: (1) Arrays und Zeiger können in vielen Ausdrücken austauschbar verwendet werden, dies wird vor allem bei **Zeichenketten** verwendet. (2) Zeiger als Funktionsparameter ermöglichen Parameterübergabe als Referenz. (3) Zeiger ermöglichen **dynamische Speicherverwaltung**. Mit Hilfe der Funktionen **malloc, calloc und free** kann Speicher im Heap-Bereich allokiert und freigegeben werden.

Beispiel

Berechne Länge einer Zeichenkette

```
1. int a = 10, *z;
2. z = &a;
3. *z = 20;
4. printf("Wert von a = %d\n",*z);
5.
6. double *p = NULL; // p mit NULL initialisieren
7. double list[] = {1.2, 2.3, 3.4};
8. p = &list[0]; // p zeigt auf den Anfang von lis

9. p++; // Zeiger p inkrementieren
10. p zeigt jetzt auf list[1]
11. printf("%.2lf\n",*p); // Ausgabe: 2.3
```

Der **NULL-Zeiger** ist ein spezieller Zeiger, der für die Zeigerinitialisierung verwendet wird und im Fehlerfall zurückgegeben wird, wenn z.B. kein Speicher reserviert werden kann.

```
1. // Dynamische Speicherallokation
2. char *s = (char *)malloc(20*sizeof(char));
3. // falls kein Speicher reserviert werden kann
4. if (s == NULL)
5.     return; // beende das Programm
6. fgets(s, 20, stdin); // Zeichenkette einlesen
7. int length = 0;
8. while(*s != '\0'){ // Solange das Endzeichen ni

9.   length++; // wird die Länge hochgezählt
10.  s++; // und die Zeigervariable inkrementiert
11. }
```

9 Strukturen

struct typedef

Top

Eine **Struktur** ist eine Zusammenfassung mehrerer zusammengehöriger Variablen verschiedenen Datentyps unter einem gemeinsamen Namen. Strukturen werden verwendet, um komplexe Objekte der Realität abzubilden, die durch zusammengesetzte Informationen beschrieben werden. Durch die Deklaration einer Struktur wird zunächst ein neuer Datentyp mit benutzerdefiniertem Namen festgelegt, anschließend kann man neue Variablen deklarieren, die genau diesen Datentyp haben.

Strukturen verwenden

Eine Struktur wird definiert, indem man nach dem Schlüsselwort **struct** den Namen der Struktur angibt, gefolgt von geschweiften Klammern, in die man die zugehörigen Variablen-Deklarationen setzt. Eine Struktur wird verwendet, indem man neue Variablen deklariert, die den Datentyp struct struct_name haben.

Syntax

```
struct struct_name {
    datentyp1 var_1;
    ...
    datentyp_n var_n;
}
struct struct_name s1, s2;
```

Beispiel

Struktur für Studenten

Die Struktur struct Student fasst die Variablen zusammen, die die Attribute eines Studenten speichern können.

Strukturen mit typedef

In C besteht die Möglichkeit, mit Hilfe des Schlüsselwortes **typedef** symbolische Namen für Datentypen zu definieren und dadurch das Programm verständlicher gestalten. Die Verwendung von typedef ist besonders nützlich, wenn man Strukturen verwendet, da man sonst das Schlüsselwort struct stets mit angeben müsste.

Beispiel

In diesem Beispiel wird mit **typedef struct Student STUDENT;** festgelegt, dass man bei späteren Deklarationen anstelle von **struct Student** einfach nur **STUDENT** schreiben kann. D.h. um Variablen vom Datentyp STUDENT zu deklarieren, schreibt man **STUDENT st1, st2;**

```c
#include <stdio.h>
int main(void){
  struct Student {
    char* name;
    char* vorname;
    int matnr;
  };
  struct Student st = { "Muster", "Max", 12345

  printf("%s %s, %d\n", st.name, st.vorname, st

  st.name = "Test"; st.vorname = "Anna"; st.mat

  printf("%s %s, %d\n", st.name, st.vorname, st

}
```

```c
#include <stdio.h>
struct Student {
    char* name;
    char* vorname;
    int matnr;
};
typedef struct Student STUDENT;

int main(void) {
  STUDENT st1, st2;
  st1.name = "Muster"; st1.vorname = "Max";

  st1.matnr = 12345;
  printf("%s %s, %d\n",
    st1.name, st1.vorname, st1.matnr);
  st2.name = "Test"; st2.vorname = "Anna";

  st2.matnr = 12346;
  printf("%s %s, %d\n",
```

10 Standardbibliotheken

Top

C verfügt über eine überschaubare Anzahl von Standardbibliotheken mit vordefinierten Funktionen, die über ihre Header-Dateien eingebunden werden. Eine Auswahl häufig benötigter Standardbibliotheken finden Sie hier zusammengestellt.

Eingabe- und Ausgabe: stdio.h printf scanf fgets fopen fclose fprintf fscanf

Die Funktionen für Ein- und Ausgabe werden über die stdio.h inkludiert. Ein- und Ausgabe erfolgt über Streams, ein Stream kann die Konsole, eine Datei oder eine Zeichenkette sein. Hier findet man Funktionen für die Eingabe von der Konsole (scanf, fgets, getchar) und in Dateien (fscanf, fread), für die Ausgabe auf die Konsole (printf, putchar), in Dateien (fprintf, fwrite) und Strings (sprintf), für Fehlermeldungen [...].

Mathematische Funktionen: math.h sin pow sqrt fabs trunc ceil

Die Headerdatei math.h deklariert mathematische Funktionen: trigonometrische Funktionen (sin, cos, sinh, cosh, tan ...), Exponentialfunktion (exp), Logarithmen (log, log10, log2), Potenzfunktion (pow), Wurzelfunktion (sqrt), Absolutwert (fabs), Rundungsfunktionen (floor, ceil, trunc), Minimum- und Maximum-Funktionen(fmin, fmax). Hier werden auch diverse mathematische Konstanten deklariert, abhängig von der Compiler-Version, auch die Konstante M_PI. Um in Visual Studio die Konstante M_PI verwenden zu können, muss vor dem Inkludieren der math.h der Befehl **#define _USE_MATH_DEFINES** verwendet werden.

String-Manipulation: string.h strlen memcpy strcpy strcat strcmp strchr

Die Headerdatei string.h deklariert Funktionen für die Bearbeitung von Zeichenketten: Länge einer Zeichenkette bestimmen (strlen), Zeichenketten kopieren (strcpy), vergleichen (strcmp), aneinanderfügen (strcat). Weiterhin: Zeichen in einer Zeichenkette suchen (strchr) [...].

Diverse Hilfsfunktionen: stdlib.h atoi rand srand malloc calloc free

Die Headerdatei stdlib.h deklariert Funktionen für Typkonvertierung (atoi, atof,...), Zufallszahlen (srand, rand), dynamische Speicherallokation (malloc, calloc, free) [...]. Hier wurden Hilfsfunktionen für unterschiedliche Anwendungsbereiche zusammengepackt.

Tools & Quellen

1. Visual Studio 2022 Community Edition: Für die Entwicklung von C-Programmen. Visual Studio unterstützt C-Programmierung indirekt über C++ und die Vollversion wird in vielen Unternehmen als Entwicklungsumgebung eingesetzt.
2. yED Graph Editor: Für die Entwicklung von Flussdiagrammen bzw. Programmablaufplänen.
3. Jürgen Wolf, C von A bis Z: openbook.rheinwerk-verlag.de/c_von_a_bis_z/, 2020.
4. Thomas Theis: Einstieg in C. Für Programmiereinsteiger geeignet, Galileo Press, 2014.
5. Manfred Daussman, C als erste Programmiersprache: Vom Einsteiger zum Fortgeschrittenen. Vieweg, 2010.
6. Axel Böttcher, Franz Kneißl. Informatik für Ingenieure: Grundlagen und Programmierung in C. Oldenbourg Verlag, 1999.
7. Brian Kernighan, Dennis Ritchie, The C programming language. Prentice-Hall, 2010.
8. Visual Studio C Language Reference, docs.microsoft.com/en-us/cpp/c-language/