BERGISCHE UNIVERSITÄT WUPPERTAL
FACHBEREICH C - PHYSIK
ELEKTRONIKPRAKTIKUM

Einstieg in die Programmierung mit C

(als Vorbereitung auf den Versuch EP11 Mikrocontroller)

pk, Ver. 22.12.2010

Das folgende Tutorial soll eine Einführung in die Programmiersprache C (C++) geben.

Dazu werden einfache Problemstellungen und deren Lösungen vorgestellt.

Die C-Programmierung erfolgt dabei in der Programmierumgebung
Microsoft Visual Studio 2010 / Microsoft Visual C++ 2010 Express,
als Download auch unter:

`http://www.microsoft.com/germany/express/download/default.aspx`

(und dort auf den Link Visual C++ 2010 Express).

Dieses Programm ermöglicht eine Entwicklung von C-Programmen, und zwar im Hinblick auf die **Sprache**, und *nicht* mit den hardwaretypischen Einschränkungen der Mikrocontrollerprogramme.

Auf den PCs im Elektronikpraktikum ist das Programm Visual C++ 2010 Express bereits installiert. **Für die Programmierung der Mikrocontroller wird eine andere Programmierumgebung verwendet, die speziell auf die Mikrocontroller zugeschnitten ist und im Versuchsskript EP11 vorgestellt wird.**

1. Programmstart

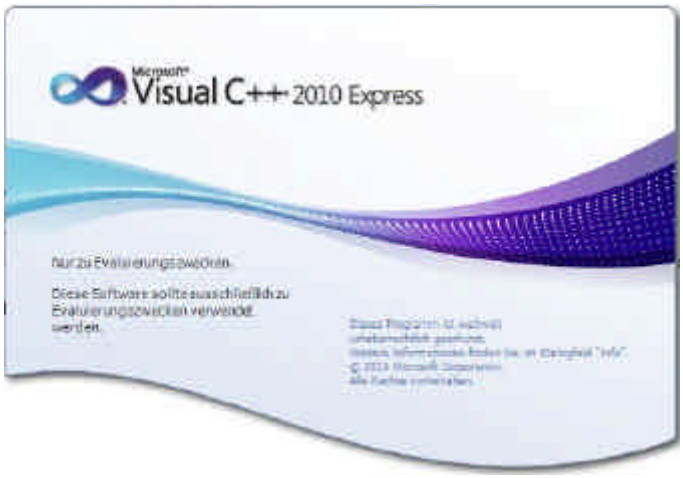
1.1. Start

Starten Sie Visual C++ 2010 Express wie folgt:

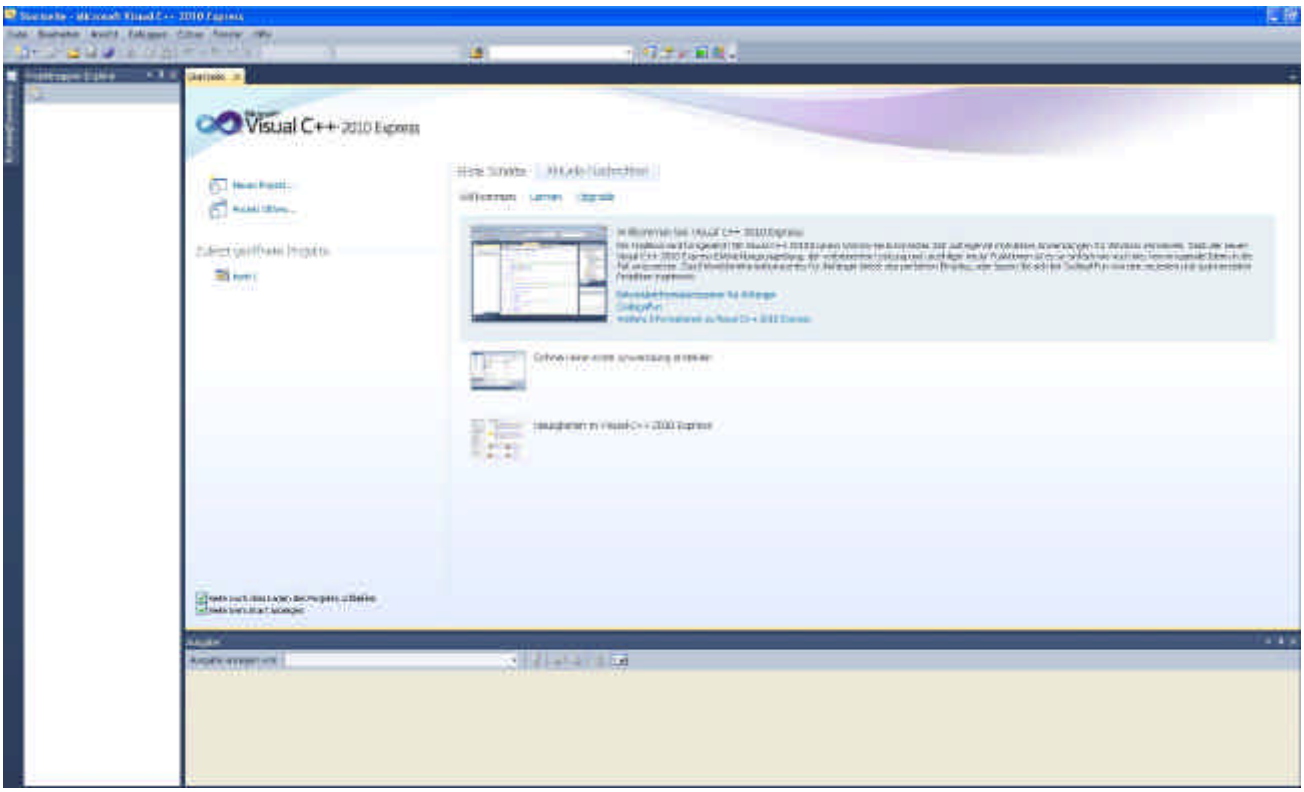
entweder über das Symbol (die Verknüpfung) auf dem Desktop (Ordner öffnet sich, darin Microsoft Visual C++ 2010 Express anklicken)

oder über:
Start > Alle Programme >
Microsoft Visual Studio 2010 Express > Microsoft Visual C++ 2010 Express

Es öffnet sich erst ein Begrüßungsfenster:



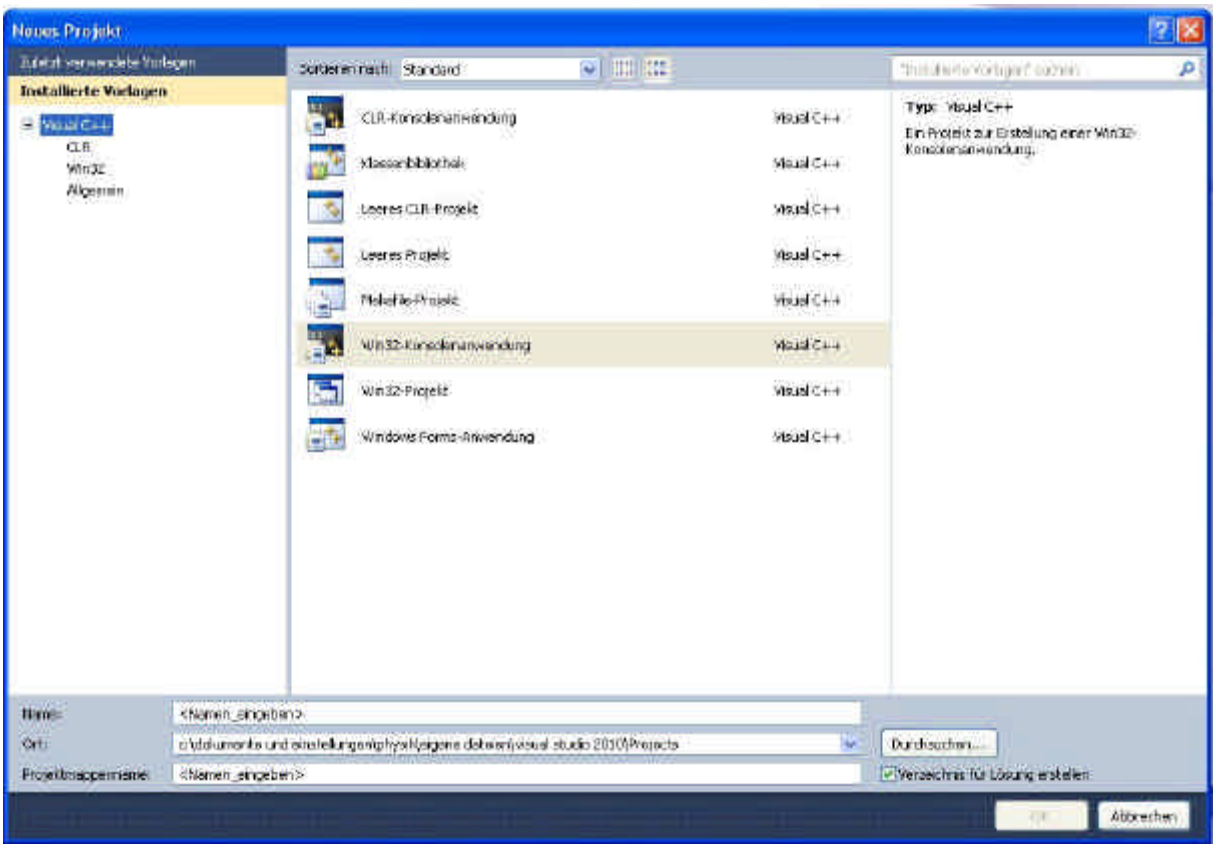
und danach die Startseite (Startfenster):



1.2 Neues Projekt anlegen

Gehen Sie im Startfenster links oben auf:
Datei > Neu > Projekt …

Es öffnet sich das Fenster „Neues Projekt“:



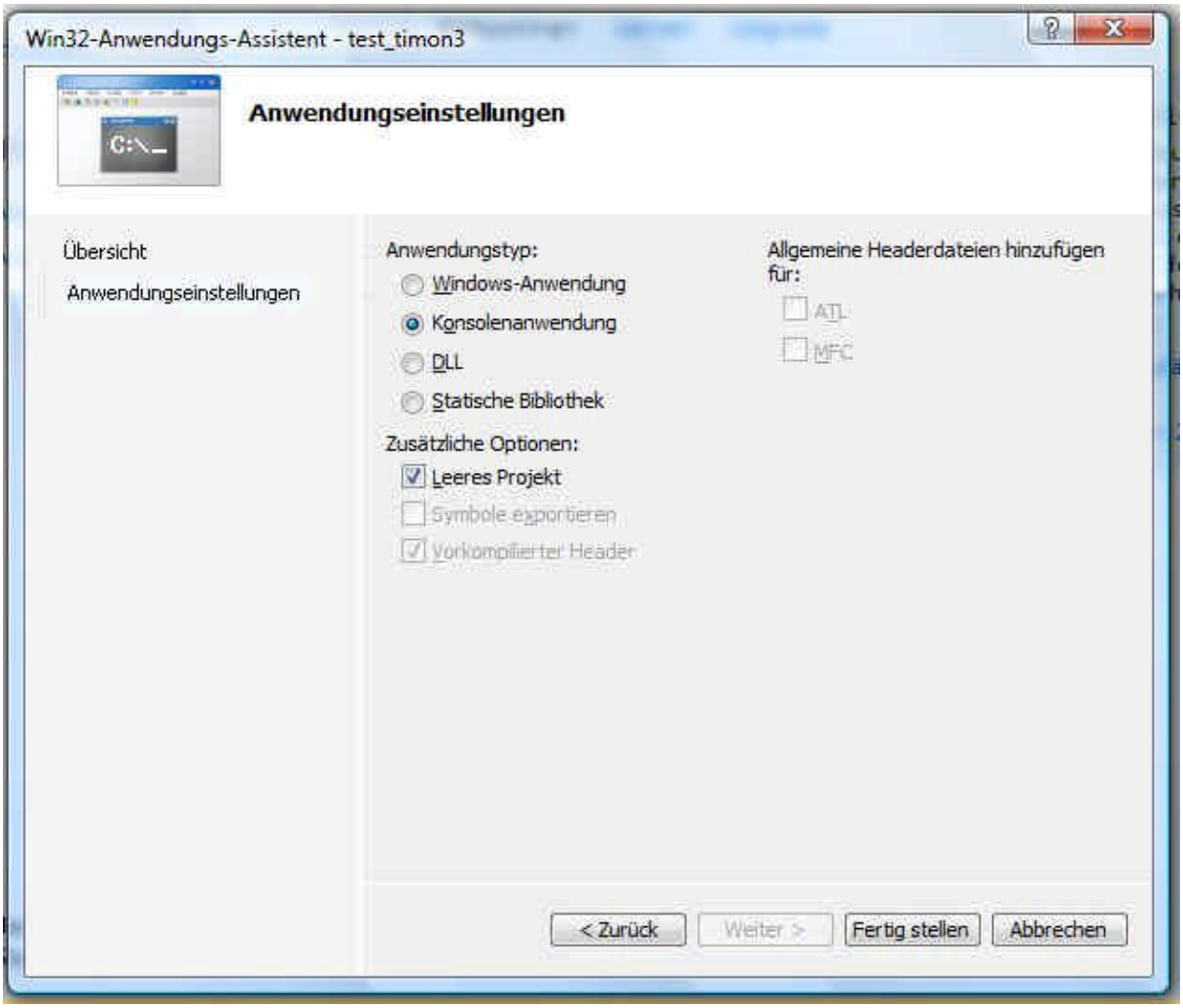
Wählen (klicken) Sie den Eintrag „WIN32-Konsolenanwendung“ und geben Sie unten im Feld „Name“ einen Projektnamen ein (z.B. Tutorial_2010). Dieser Name wird automatisch als „Projektmappenname“ übernommen (der aber auch noch geändert werden könnte).

Unter „Ort“ wird vorgeschlagen, wo die Projektdateien gespeichert werden.
(Evtl. wird am Praktikumsnachmittag ein anderer Ort empfohlen)

Klicken Sie dann auf OK.

Es öffnet sich ein Fenster „Win32-Anwendungs-Assistent“. Klicken Sie dort unten auf **Weiter** und *nicht* auf „Fertig stellen“.

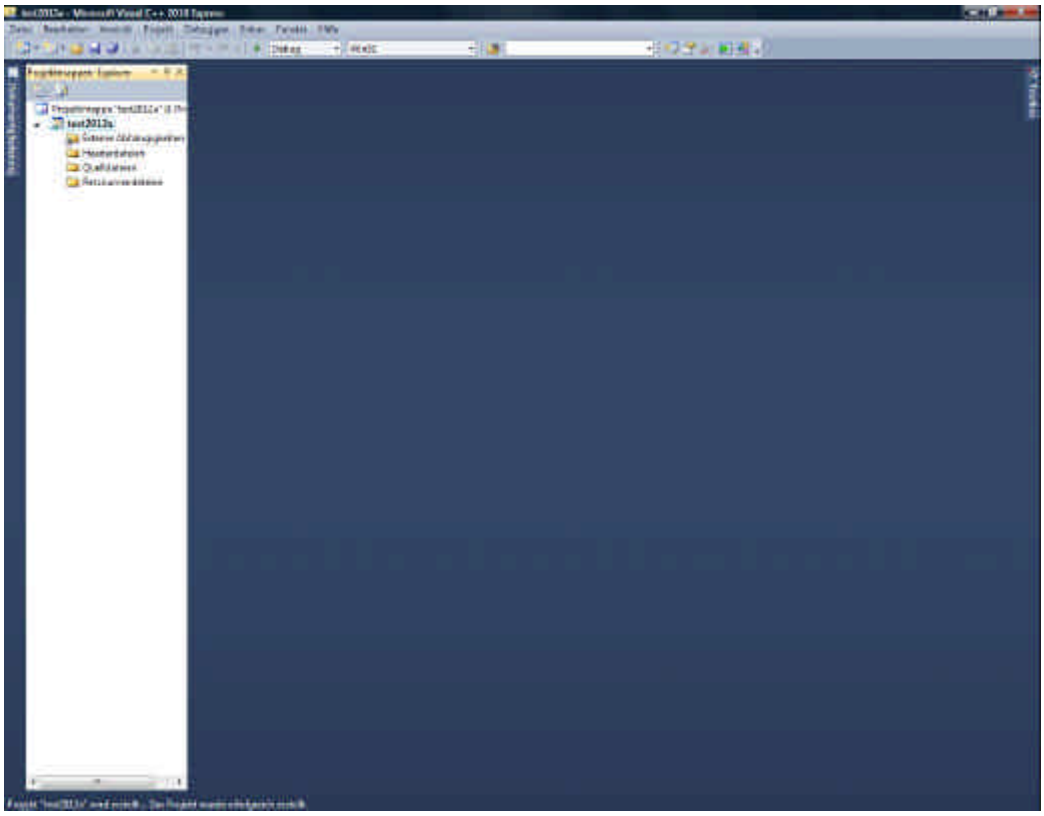
Es öffnet sich ein weiteres Fenster, siehe unten. Stellen Sie bei „Zusätzliche Optionen“ ein: „Leeres Projekt“. Das sollte dann so aussehen. Klicken Sie dann auf „Fertig stellen“.



Nach einigen Sekunden öffnet sich die Programmierumgebung.

1.3 Ein allererstes Programm

Die Programmierumgebung sieht zunächst so aus (der konkrete Projekt- und Dateiname links oben ist so, wie von Ihnen festgelegt):

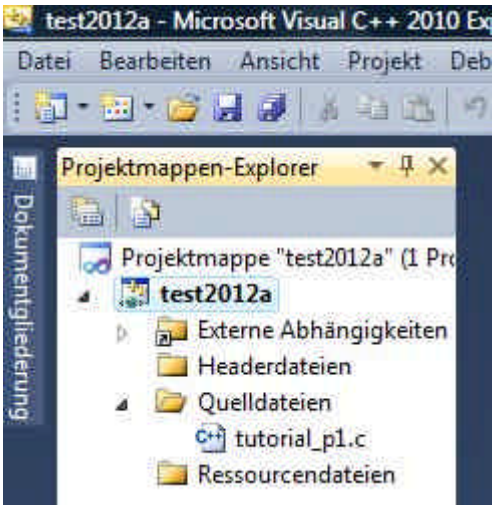


Zunächst fehlt uns die Möglichkeit, eigene Programme zu editieren. Daher wollen wir uns eine (bereits vorhandene) Programmdatei öffnen. Gehen Sie dazu auf:

Projekt > Vorhandenes Element hinzufügen

und wählen folgende Datei aus:
(Pfad wird noch bekannt gegeben) tutorial_p1_00.c und klicken auf „Hinzufügen“

Diese Datei erscheint jetzt in der linken Spalte im „Projektmappen-Explorer“ unter dem Ordnersymbol „Quelldateien“:



Klicken Sie die Datei dort an, um sie zu öffnen.

Offenbar ist bereits ein kleines Programm vorbereitet:

```
// Preprocessor commands
// (Libraries, Custom Files)
#include <stdio.h>

// Main function
int main() {
	// Print "Hello World" to the console
	printf("Hello World\n");
	// Wait for any key
	system("PAUSE");
	// End main() and return 0 (everything was ok)
	return 0;
}
```

Wir wollen das Programm einmal laufen lassen und dann genauer betrachten, was es macht.

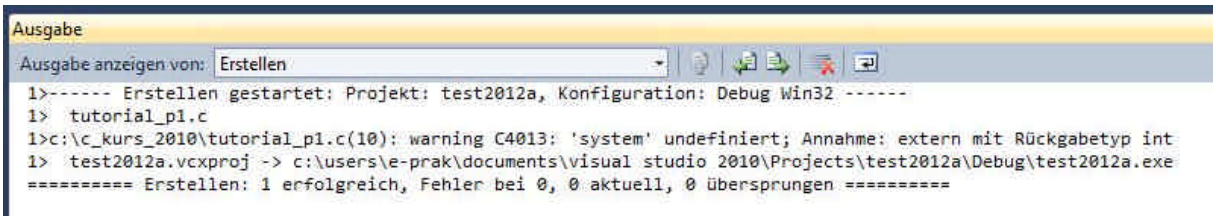
1.4 Programm compilieren und starten

Um ein Programm auf einem PC (oder Mikrocontroller oder irgendeiner anderen Hardware) laufen zu lassen, muß es erst in deren Maschinensprache übersetzt (compile) werden. Dabei wird gleichzeitig eine Prüfung auf Programmierfehler vorgenommen (Debuggen).
Dieser Vorgang dauert einen Moment.

Mit dem folgenden Befehl wird kompiliert:

Debuggen > Projektmappe erstellen (oder F7-Taste)

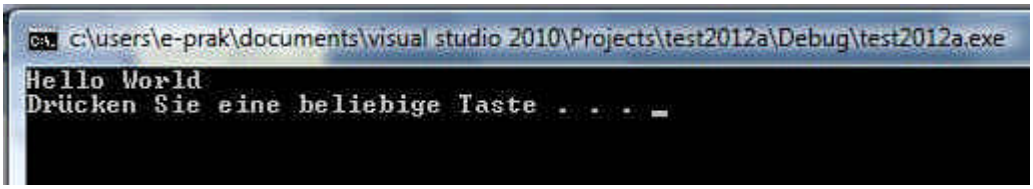
Nun erscheinen im Fenster „Ausgabe“ unter der Programmierumgebung einige Meldungen.



Um das Programm nun auszuführen, gehen Sie auf Debuggen > Debuggen starten.

(oder F5-Taste oder auf den kleinen grünen Pfeil  oben in der Menüleiste klicken)

Danach wird das Programm gestartet: Es öffnet sich ein schwarzes Fenster (die Console):



1.5 Betrachten wir nun den Programmcode:

```
// Preprocessor commands
// (Libraries, Custom Files)
```

Die ersten Zeilen beginnen mit zwei Schrägstrichen. Alles, was danach kommt, ist ein Kommentar, d.h. dieser Text wird vom Programmübersetzer (Compiler) *nicht* ausgewertet. Hier kann man beliebige erklärende Texte schreiben.

```
#include <stdio.h>
```

Diese Zeile besagt, daß eine weitere Datei mitverwendet (eingebunden, include) wurde, die für bestimmte Funktionen (z.B. Ein- und Ausgabe) wichtig ist. Das müssen wir im Moment nicht weiter betrachten.

```
// Main function
int main() {
```

Hier beginnt nun das eigentliche Programm. Es steht immer in einer Konstruktion wie:

main (Variablen) { *der Programmcode* }

main bedeutet dabei, daß es sich um das Hauptprogramm handelt, das immer als erstes ausgeführt wird. Die Variablen in der Klammer (in unserem Beispiel gibt es keine Variablen) müssen uns im Moment auch nicht interessieren. Wichtig ist, daß der Programmcode zwischen einem Paar geschweifter Klammern { } steht.

```
    // Print "Hello World" to the console
    printf("Hello World\n");
```

Dieses ist die wichtigste Programmzeile. Sie besagt, dass auf der Console (im „schwarzen Fenster“) Text geschrieben werden soll. Was geschrieben werden soll, folgt in runden Klammern. Es ist der Text *Hello World*. Er muß zusätzlich in Anführungszeichen stehen.

Jeder Programmbefehl mit einem Semikolon beendet. („main“ ist kein Befehl)

Das Zeichen „\n“ ist das Steuerzeichen für Zeilenvorschub, d.h. alles, was danach kommt, beginnt in einer neuen Zeile.

Damit das Programm danach nicht schon beendet und die Console sofort geschlossen wird, fügen wir noch folgende Anweisung ein:

```
    // Wait for any key
    system("PAUSE");
```

Durch diese Anweisung wird auf eine Eingabe (eine beliebige Eingabetaste) gewartet. Um das jedem klarzumachen, erscheint automatisch der Text „Drücken Sie eine beliebige Taste …“

```
    // End main() and return 0 (everything was ok)
    return 0;
```

Als letztes wird noch dafür gesorgt, dass das Programm den Wert „0“ zurückgibt, und zwar an das PC-Betriebssystem. Die Null besagt, dass kein Fehler aufgetreten ist. Weil es sich um eine Zahl (eine Ganzzahl, integer) handelt, wurde ganz oben vor „main“ der Variablentypname „int“ angegeben. Auf Variablentypen gehen wir aber später noch ein.

```
}
```

Der gesamte Code endet mit der schließenden geschweiften Klammer.

2. Eingabe von Werten in ein Programm - Variablen

2.1 Texteingabe

Unser letztes Programm wartete zwar auf eine Eingabe, konnte aber damit nichts anfangen, weil es sie nicht abspeicherte. Wir wollen das Programm nun so erweitern, dass Texteingaben (also von der Tastatur) möglich sind.

Dazu müssen wir einen Speicherplatz definieren, um eingegebene Texte zwischenzuspeichern, man spricht von Variablen.

Erweitern Sie Ihr Programm, damit es wie folgt aussieht. Die Kommentare müssen Sie hier nicht unbedingt abschreiben. Sie erleichtern nur das Verständnis der Programmteile.

```
// Preprocessor commands
// (Libraries, Custom Files)
#include <stdio.h>

// Main function
int main() {

	char eingabetext[100];   // char-string ("Textvariable") der Länge 100

	printf("Text eingeben:\n");    // Aufforderung zur Texteingabe
	scanf("%s", &eingabetext);     // Einlesen des Textes
	printf("Dein Text ist: %s\n", eingabetext);  // Ausgabe des gespeicherten Textes

	system("PAUSE");
	return 0;
}
```

Führen Sie das Programm wie im vorherigen Abschnitt beschrieben aus: Erst compile (F7) dann F5.

Wieder öffnet sich das Consolenfenster.

Sie werden aufgefordert, Text einzugeben (mit Enter/Return-Taste abschließen!).

Der Text wind danach ausgegeben. Offenbar wird der Text aber nur bis zum ersten Leerzeichen eingelesen.

2.2 Programmerklärung

Sehen wir uns die neuen Zeilen genauer an:

```
char eingabetext[100];   // char-string ("Textvariable") der Länge 100
```

Hier wird eine Variable mit dem Namen „eingabetext“ definiert. Sie kann bis zu 100 Zeichen aufnehmen. Man spricht auch von einem „Array“. Mit „char“ wird der Variablentyp definiert, hier handelt es sich um Textzeichen (Buchstaben, Ziffern, Satzzeichen). Weitere Variablentypen lernen wir im nächsten Abschnitt kennen.

```
printf("Text eingeben:\n");    // Aufforderung zur Texteingabe
```

Diese Zeile entspricht der Hello-World-Programmzeile von oben. Fester Text wird mit Zeilenvorschub ausgeben.

```
scanf("%s", &eingabetext);     // Einlesen des Textes
```

Hier wird nun von der Eingabe (in unserem Fall von der Tastatur) Text eingelesen. Dazu dient die Funktion scanf. Mit dem sogenannten Formatierungsbefehl "%s" wird dem Programm mitgeteilt, dass die Eingabe als Text zu interpretieren ist. Sie wird in die Variable &eingabetext abgespeichert. Beachten Sie das Zeichen & vor dem Variablennamen!

```
printf("Dein Text ist: %s\n", eingabetext);  // Ausgabe des gespeicherten Textes
```

Der gespeicherte Text wird ausgegeben. Dieser Zeile wollen wir uns ganz genau ansehen:

`printf( . . . );` Der Befehl für Ausgabe ist ja bereits bekannt.

`"Dein Text ist: %s\n"` Was soll ausgeben werden? Das wird der Reihe nach zwischen den Anführungszeichen genannt, nämlich:

`Dein Text ist:`	Ein fester Text
`%s`	Ein Textelement (string).
`\n`	Und noch ein Zeilenvorschub.

Das allein genügt aber noch nicht, denn das Programm muß ja noch wissen, durch *welche Variable* das Textelement %s bestimmt ist.

Deshalb folgt nun noch hinter einem Komma die Angabe der Variablen:

`, eingabetext`

3 Variablentypen

Wir haben bereits den Variablentyp char für Texte kennengelernt.

Mit Texten kann man aber nicht rechnen. Selbst wenn wir nur Ziffern eingeben würden, würden diese in einer Weise abgespeichert, die keine Rechenoperationen erlaubt.

Daher gibt es einige weitere Variablentypen, wie z.B.:

int	Ganze Zahlen, also *ohne Nachkommastellen*, aber mit Vorzeichen
float	Fließkommazahlen, also *mit beliebigen Nachkommastellen* und Vorzeichen

Die Unterscheidung ob mit oder ohne Kommastellen hat etwas mit dem Speicherbedarf und der Rechengenauigkeit zu tun (bei Mikrocontrollern ist besonders der Speicher begrenzt und ein unnötig kompliziertes Rechenverfahren verlangsamt die Programmausführung).

Ergänzen Sie Ihr Programm wie folgt. Wie wollen eine Teileliste mit Anzahl und Preisen aufnehmen:

```
// Preprocessor commands
// (Libraries, Custom Files)
#include <stdio.h>

// Main function
int main() {

	char  eingabetext[100];   // char-string ("Textvariable") der Länge 100
	int   anzahl;             // eine ganze Zahl
	float preis;              // eine Zahl mit Nachkommastellen

	printf("Artikel eingeben:\n");    // Aufforderung zur Texteingabe
	scanf("%s", &eingabetext);     // Einlesen des Textes
	printf("Artikel ist: %s\n", eingabetext);  // Ausgabe des gespeicherten Textes

	printf("Anzahl eingeben:\n");    // Aufforderung zur Eingabe
	scanf("%d", &anzahl);           // Einlesen der Ganzzahl
	printf("Anzahl ist: %d\n", anzahl);  // Ausgabe

	printf("Preis eingeben:\n");    // Aufforderung zur Texteingabe
	scanf("%f", &preis);     // Einlesen des Textes
	printf("Preis ist: %f\n", preis);  // Ausgabe des gespeicherten Textes

	system("PAUSE");
	return 0;
}
```

Beachten Sie:

- Bei den Zahlen ist der Platzhalter %d bzw. %f.
- In der scanf-Zeile steht vor dem Variablenname ein &

Starten Sie Ihr Programm (F7, F5). Beachten Sie: Als Dezimalkomma bei „Preis“ muß ein **Punkt** eingegeben werden, also 123.456 und *nicht* 123,456

Probieren Sie auch aus, was passiert wenn Sie:

- bei Anzahl oder Preis Buchstaben eingeben
- Bei Anzahl Nachkommastellen eingeben (werden diese gespeichert?)
- Bei Preis sehr viele Vor- oder Nachkommastellen eingeben. Ist die letzte Nachkommastelle korrekt?

4 Variablenzuweisungen im Programm

4.1 Ganzzahlen und Fließkommazahlen (int und float)

Oft werden bestimmte Zahlenwerte oder Texte an mehreren Stellen im Programm gebraucht. Es ist für die Programmstruktur übersichtlicher und bei einer Änderung des Wertes einfacher, wenn man die Werte im Programm (d.h. nicht wie oben durch Tastatureingabe) einer Variablen zuweist.

Ergänzen Sie Ihr Programm wie folgt (gezeigt ist nur der main-Bereich):

```
int main() {

	char  eingabetext[100];   // char-string ("Textvariable") der Länge 100
	int   anzahl, festezahl;          // eine ganze Zahl#
	float preis, sonderpreis;           // eine Zahl mit Nachkommastellen

	festezahl = 3;
	sonderpreis = 0.99;

	printf("Artikel eingeben:\n");    // Aufforderung zur Texteingabe
	scanf("%s", &eingabetext);     // Einlesen des Textes
	printf("Artikel ist: %s\n", eingabetext);  // Ausgabe des gespeicherten Textes

	printf("Anzahl eingeben:\n");    // Aufforderung zur Eingabe
	scanf("%d", &anzahl);          // Einlesen der Ganzzahl
	printf("Anzahl ist: %d\n", anzahl);  // Ausgabe

	printf("Preis eingeben:\n");    // Aufforderung zur Texteingabe
	scanf("%f", &preis);     // Einlesen des Textes
	printf("Preis ist: %f\n", preis);  // Ausgabe des gespeicherten Textes

	printf("Sonderposten kosten ab %d Stück nur %f! \n", festezahl, sonderpreis);

	system("PAUSE");
	return 0;
}
```

Starten Sie das Programm (F7, F5).

Geändert bzw. neu hinzugekommen sind folgende Zeilen:

```
	int   anzahl, festezahl;          // eine ganze Zahl#
	float preis, sonderpreis;           // eine Zahl mit Nachkommastellen
```

Man kann weitere Variablen desselben Typs definieren, indem man sie durch Komma getrennt auflistet. Am Ende jeder Zeile steht aber ein Semikolon.

```
	festezahl = 3;
	sonderpreis = 0.99;
```

So werden den Variablen feste Zahlen zugewiesen.

```
	printf("Sonderposten kosten ab %d Stück nur %f! \n", festezahl, sonderpreis);
```

Hier sehen wir die Ausgabe mehrere Variablen in einer Zeile. An den Stellen, wo sie im Text erscheinen sollen, stehen die Formatierungsplatzhalter %d oder %f. Danach werden die Variablen mit ihrem Namen genannt (in der richtigen Reihenfolge, wie sie im Text erscheinen).

4.2 Buchstaben und Arrays

Einzelne Buchstaben werden an Variablen des Typs char übergeben:

```
char  nur_ein_buchstabe;   // Definition
nur_ein_buchstabe = 'B';   // Zuweisung

printf("Ein Buchstabe: %c \n", nur_ein_buchstabe);
```

Achtung: Einzelne Buchstaben werden mit %c statt %s behandelt! **(CZ 21/12)**

Manchmal ist es sinnvoll, ähnliche Variablen zusammenzufassen. Sie erhalten Indizes und lassen sich später über diese Indizes leichter aufrufen. Man spricht von einem Array.

Schreiben Sie folgendes Programm: (Wenn wenig Zeit ist: laden Sie tutorial_p2_00.c)

Wie immer der Anfang:

```
// Preprocessor commands
// (Libraries, Custom Files)
#include <stdio.h>

// Main function
int main() {
```

:

Definieren Sie eine neue Variable und das Array

```
int indexnummer;
int zahlenfeld [5];      // Array definieren
```

Füllen Sie das Array mit Werten, z.B. Quadratzahlen:

```
zahlenfeld[0] = 0;  // Die Indizes beginnen immer bei 0
zahlenfeld[1] = 1;  // Wie weisen als Beispiel Quadratzahlen zu
zahlenfeld[2] = 4;
zahlenfeld[3] = 9;
zahlenfeld[4] = 16;
zahlenfeld[5] = 25;
```

Fragen Sie nach einer Indexnummer:

```
printf("Index eingeben:\n");        // Aufforderung zur Eingabe
scanf("%d", &indexnummer);          // Einlesen der Indexnummer
```

Geben Sie Index und Inhalt der entsprechenden Variable aus:

```
printf("Inhalt von Variable Nr. %d ist %d \n", indexnummer, zahlenfeld[indexnummer]);
```

Natürlich kann man auch einen festen Index im Programm angeben:

```
printf("Inhalt von Variable Nr. 2 ist %d \n", zahlenfeld[2]);
```

Und das bekannte Ende der Programmdatei

```
system("PAUSE");
return 0;
}
```

Was passiert, wenn Sie einen falschen Index eingeben, z.B. 6 oder -1?

Man erhält falsche Variablenwerte. Solche unerlaubten Eingaben müssen also „abgefangen“ werden. Möglichkeiten dazu lernen Sie später kennen.

5. Arithmetik

5.1 Ein Programm mit den grundlegenden Operationen

Die Hauptaufgabe eines Computerprogramms ist in der Regel, Berechnungen mit Zahlen, also arithmetische Operationen auszuführen.

In dem folgenden Programm lernen Sie die wichtigsten Operationen kennen. Dabei ist zu beachten, ob mit Ganzzahlen (integer, int) oder Fließkommazahlen (float) gerechnet wird. Insbesondere bei Divisionen spielt das eine erhebliche Rolle.

Öffnen Sie das Programm tutorial_p3_00.c. Es sollte wie folgt aussehen.

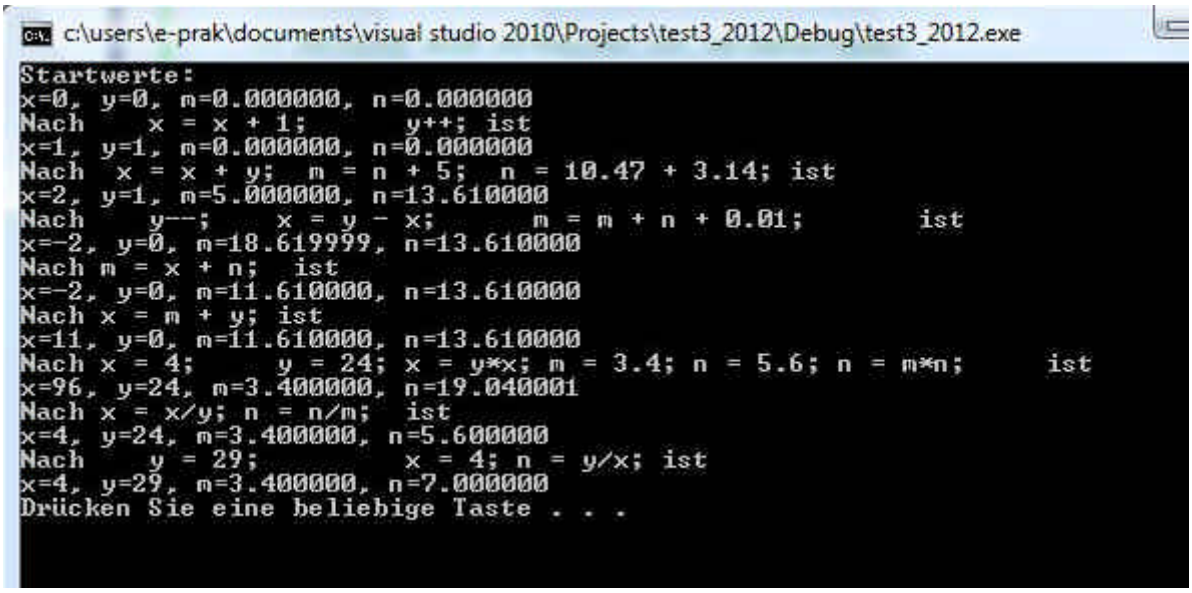
Führen Sie das Programm aus und schauen Sie sich genau die Rechenergebnisse an.

Die z.T. unerwünschten Effekte wollen wir danach genauer betrachten.

```
// Preprocessor commands
// (Libraries, Custom Files)
#include <stdio.h>

// Main function
int main() {

	int   x, y;
	float m, n;

	x = 0;
	y = 0;
	m = 0;
	n = 0;

	printf("Startwerte: \n");
	printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);

	// Addition, Subtraktion
	x = x + 1;
	y++;
	printf("Nach            x = x + 1;  y++; ist \n");
	printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);

	x = x + y;
	m = n + 5;
	n = 10.47 + 3.14;
	printf("Nach  x = x + y;  m = n + 5;  n = 10.47 + 3.14; ist \n");
	printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);

	y--;
	x = y - x;
	m = m + n + 0.01;
	printf("Nach            y--;          x = y - x;   m = m + n + 0.01;    ist \n");
	printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);

	m = x + n;
	printf("Nach m = x + n;              ist \n");
	printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);

	// Vorsicht beim Vermischen von Variablentypen!
	x = m + y;
	printf("Nach x = m + y;            ist \n");
	printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);

	// Multiplikation
	x = 4;
```

```
        y = 24;
        x = y*x;
        m = 3.4;
        n = 5.6;
        n = m*n;
        printf("Nach x = 4;    y = 24; x = y*x; m = 3.4; n = 5.6; n = m*n; ist \n");
        printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);

        // Division
        x = x/y;
        n = n/m;
        printf("Nach x = x/y; n = n/m;    ist \n");
        printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);

        // Vorsicht bei der Division mit Ganzzahlen (integer)!
        y = 29;
        x = 4;
        n = y/x;
        printf("Nach            y = 29;     x = 4;       ist \n");
        printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);

        system("PAUSE");

        // End main() and return 0 (everything was ok)
        return 0;
}
```

Die Ausgabe (Console) sollte folgendes liefern:



5.2 Grundlegende Operationen – was man beachten sollte

Gehen wir das Programm nun schrittweise durch und betrachten auch die Ergebnisse:

```
int main() {

        int   x, y;
        float m, n;

        x = 0;
        y = 0;
        m = 0;
        n = 0;

        printf("Startwerte: \n");
        printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);
```

Es werden die beiden Ganzzahlen (Integervariablen) x und y und die beiden Fließkommavariablen m und n definiert. Alle werden zunächst auf 0 gesetzt, was auch als Startwerte ausgegeben wird.
Beachte: m und n werden immer mit 6 Nachkommastellen angezeigt.

```
        // Addition, Subtraktion
        x = x + 1;
        y++;
        printf("Nach           x = x + 1;  y++; ist \n");
        printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);
```

Die Addition mit einer festen Zahl, z.B. x = x + 1. Beachte: Dies ist *keine* mathematische Gleichung (x = x+1 ist niemals lösbar), sondern bedeutet: die *rechte* Seite (hier: x + 1) wird in die *linke* Seite (hier: x) gespeichert. Am Ende steht der um 1 erhöhte Wert in x.
Eine besondere Kurzschreibweise gibt es für das „Erhöhen um 1“, das *Inkrementieren*.
Anstelle von. Z.B. y = y+1 läßt sich auch schreiben: y++.
Entsprechend hätte man statt x = x+1 auch schreiben können x++.

```
        x = x + y;
        m = n + 5;
        n = 10.47 + 3.14;
        printf("Nach  x = x + y;  m = n + 5;  n = 10.47 + 3.14;          ist \n");
        printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);
```

Zwei Ganzzahlen werden addiert: x = x + y, auch das Ergebnis ist eine Ganzzahl.
Auch hier ist das nicht als mathematische Gleichung zu verstehen (y müsste dann ja Null sein), sondern in dem Sinne: Nimm x und y, bilde die Summe x+y und speichere diese in x.
Darunter die Addition von Fließkommazahlen. Beachten Sie: Dezimal**punkt**, nicht Komma!

```
        y--;
        x = y - x;
        m = m + n + 0.01;
        printf("Nach           y--;        x = y - x;   m = m + n + 0.01;    ist \n");
        printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);
```

Auch für das „Vermindern um 1“, das *Dekrementieren* gibt es eine Kurzschreibweise.
Anstelle von z.B. y = y -1 geht auch y--.
Ganzzahlen haben Vorzeichen, also x = -2 gibt es.
Es können gleichzeitig mehrere Summanden verarbeitet werden, egal ob Variable oder feste zahl: m = m + n +0.1

```
        m = x + n;
        printf("Nach m = x + n;          ist \n");
        printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);
```

Die Verknüpfung (hier: Addition) einer Fließkommazahl mit einer Ganzzahl ist eine Fließkommazahl.

```
	// Vorsicht beim Vermischen von Variablentypen!
	x = m + y;
	printf("Nach x = m + y;		ist \n");
	printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);
```

Probleme gibt es, wenn eine Fließkommazahl an eine Ganzzahl übergeben wird. Dann gehen die Nachkommastellen verloren und es kommt zu falschen Ergebnissen!

```
	// Multiplikation
	x = 4;
	y = 24;
	x = y*x;
	m = 3.4;
	n = 5.6;
	n = m*n;
	printf("Nach x = 4;    y = 24; x = y*x; m = 3.4; n = 5.6; n = m*n; ist \n");
	printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);
```

Die Multiplikation verwendet das Zeichen * für beide Variablentypen.

```
	// Division
	x = x/y;
	n = n/m;
	printf("Nach x = x/y; n = n/m;    ist \n");
	printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);
```

Die Division verwendet das Zeichen /
Achtung auch hier bei Ganzzahlen. Nur wenn es glatt aufgeht (96/24 = 4), ist das Ergebnis richtig

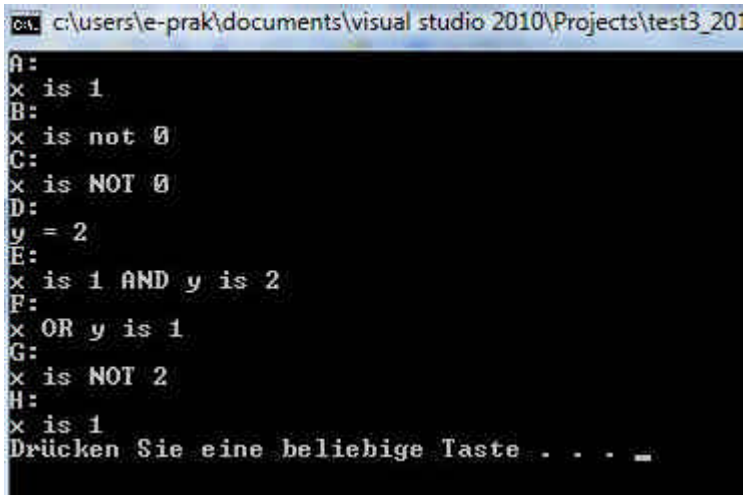
```
	// Vorsicht bei der Division mit Ganzzahlen (integer)!
	y = 29;
	x = 4;
	n = y/x;
	printf("Nach		y = 29;	x = 4;	ist \n");
	printf("x=%d, y=%d, m=%f, n=%f \n", x,y,m,n);
```

Wenn aber ein Divisionsrest bleibt (Nachkommestellen), hilft es auch nicht, das Ergebnis einer Fließkommavariablen (hier n) zuzuweisen. Das Ergebnis ist nämlich immer erstmal eine Ganzzahl. Dieses wird erst beim Abspeichern in eine Fließkommazahl (für die Fließkommavariable) umgewandelt, und dann sind die Nachkommastellen längst verloren.

```
	system("PAUSE");

	// End main() and return 0 (everything was ok)
	return 0;
}
```

6. Logische Vergleiche, Bedingungen, bedingte Sprünge (if, else, case)

Öffnen Sie die Datei tutorial_p4_00.c. und starten Sie das Programm.
Das Programm sollte folgende Ausgabe (Console) produzieren:



Betrachten wir das Programm wieder Schritt für Schritt:

```
// Preprocessor commands
// (Libraries, Custom Files)
#include <stdio.h>

// Main function
int main() {
```

Der übliche Progammvorspann

```
        int x, y;
        x = 1;
        y = 2;
```

Zwei Ganzzahlvariablen x und y werden mit 1 bzw. 2 belegt.

Beispiel A: if-else

```
        // Basic If(){}else{}
        printf("A:\n");
        if ( x == 1) {
                printf("x is 1\n");
        } else {
                printf("x is not 1\n");
        }
```

Mit **if-else** werden logische Entscheidungen getroffen.
Die Syntax ist **if (*Bedingung*) {*Befehle für „if“*} else {*Befehle für „else“*}**
Die Bedingung ist eine logische Aussage, die *wahr* (true) oder *falsch* (false) sein kann.
Oft handelt es sich um einen Vergleich. Hier wird z.B. geprüft, ob x den Wert 1 hat: (x == 1).
Beachten Sie das doppelte Gleichheitszeichen, das soviel wie arithmetische Gleichheit bedeutet (im Gegensatz zum einfachen Gleichheitszeichen x = 1 bei der Wertezuweisung an eine Variable). Mit „x == 1“ wird der Inhalt von x *nicht* verändert. Vielmehr liefert „x == 1“ den Wert „wahr“, wenn x eben den Wert 1 enthält. Andernfalls hat „x == 1“ den Wert „falsch“.

Die Konstruktion `if (x == 1)`, d.h. „wenn x mit 1 übereinstimmt“, führt nun die „Befehle für if“ aus, wenn die Bedingung „wahr“ ist, hier also den Text „x is 1“.
Andernfalls („else“) wird ausgegeben „x is not 1“.
Da bei uns aber x den Wert 1 hat, erscheint unter A) auf der Console „x is 1“.

Beispiel B: Gleichheit ==

```
// EQUAL ==
printf("B:\n");
if (x == 0) {
        printf("x is 0\n");
} else {
        printf("x is not 0\n");
}
```

Hier das gleiche Beispiel, aber mit der Prüfung, ob x den Wert 0 hat. Diesmal werden die befehle unter „else“ ausgeführt und unter B) erscheint der Text „x is not 0“.

Beispiel C: Ungleichheit !=

```
// NOT EQUAL !=
printf("C:\n");
if (x != 0) {
        printf("x is NOT 0\n");
}
```

Man kann auch auf *Ungleichheit* prüfen, und zwar mit !=. Das Ausrufezeichen ist hier als Zeichen für „nicht“, also die logische Invertierung zu verstehen.
Im Beispiel C) wird also gefragt, ob „x nicht-gleich 0“ sei. Da x den Wert 1 hat, ist das wahr und die if-Befehle werden ausgeführt; der Text „x is not 0“ erscheint.

Beispiel D: Verschalteltes if-else

```
// Combination of conditions
printf("D:\n");
if (y == 1) {
        printf("y = 1\n");
} else if ( y == 2) {
        printf("y = 2\n");
} else {
        printf("y is not 1 or 2\n");
}
```

Die if-else-Verzweigungen lassen sich kombinieren. Hier wird geprüft, ob y den wert 1 hat.
Falls ja (wahr), wird „y = 1“ ausgegeben.
Falls nein (andernfalls, else) wird geprüft, ob denn y den Wert 2 hat.
Falls dieses wahr ist, wird „y = 2“ ausgegeben.
Falls nein (andernfalls, else) wird ausgegeben „y is not 1 or 2“.

Beispiel E: Logisches UND &&

```
// AND &&
printf("E:\n");
if (x == 1 && y == 2) {
        printf("x is 1 AND y is 2\n");
}
```

Mit dem Operator && werden zwei Bedingungen logisch UND-verknüpft. Die Gesamtbedingung in den runden Klammern (x == 1 && y == 2) ist nur dann wahr, wenn die eine UND die andere Teilbedingung wahr sind (vgl. das UND-Gatter der Digitaltechnik).
In unserem Beispiel E) ist die Und-Bedingung erfüllt.

Beispiel F: Logisches OR ||

```
// OR ||
printf("F:\n");
if (x == 1 || y == 1) {
        printf("x OR y is 1\n");
}
```

Ebenfalls gibt es den Operator || für die logische ODER-verknüpfung. Hier reicht es, wenn mindestens eine Teilbedingung erfüllt ist. In unserem Beispiel F) ist die ODER-Bedingung erfüllt, weil ja x den Wert 1 hat.

Beispiel G: Logische Invertierung !(…)

```
// NOT !
printf("G:\n");
if (!(x == 2)) {
        printf("x is NOT 2\n");
}
```

Auch die INVERTIERUNG gibt es. Wir haben das NICHT ja schon in Beispiel C) kennengelernt, als gefragt wurde, ob x nicht-gleich 0 sei (x != 0).
Man kann aber ebenso fragen, ob z.B. die Bedingung (x == 2) NICHT erfüllt sei.
Dann wird der NICHT-Operator anstelle vor dem Gleichheitszeichen (d.h. !=) vor die ganze Bedingung gesetzt (d.h. !(x==2)).
Wenn die Bedingung „wahr“ ist, ist ihre Invertierung, also das „nicht-wahr“ falsch.
Wenn die Bedingung „falsch“ ist, ist ihre Invertierung, also das „nicht-falsch“ wahr.
Im unserem Beispiel G) lautet die Bedingung zunächst (x == 2). Das x = 1, ist das falsch.
Da aber für das if auf die Invertierung geprüft wird `if (!(x == 2))`, ist „nicht-(x=2)“ folglich „nicht-falsch“, also wahr. Daher wird „x is NOT 2“ ausgegeben.

Beispiel H: switch-case

```
// Switch Case statements
printf("H:\n");
switch (x) {
        case 0:
                printf("x is 0\n");
                break;
        case 1:
                printf("x is 1\n");
                break;
        case 2:
                printf("x is 2\n");
                break;
        case 3:
                printf("x is 3\n");
                break;
        default:
                break;
}
```

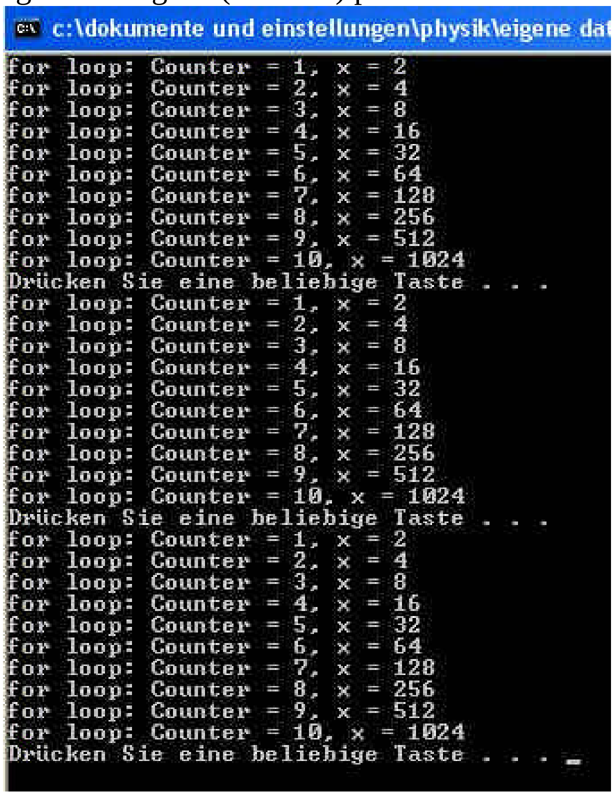
Wenn zwischen vielen verschiedenen Möglichkeiten unterschieden werden soll, ist die **switch-case**-Anweisung sehr komfortabel.
Zunächst wird mit **switch(*Variable*)** definiert, welche Variable die Auswahl, also sozusagen den Wahlschalter (switch) steuern soll.
Danach folgen mehrere Zeilen **case(*Vergleichswert*) Befehle; break;**
Es werden nur die Befehle hinter dem case ausgeführt, dessen Vergleichswert mit x übereinstimmt. Wegen des **break** wird danach die switch-case-Anweisung verlassen (andernfalls würden weitere case-Zeilen durchlaufen, was aber in der Regel keinen Sinn macht).
Für den Fall, dass x *keinem* der aufgeführten Vergleichswerte entspricht, werden die Befehle (sofern angegeben) ausgeführt, die dem **„default:“** folgen.
Weil unser x den Wert 1 hat, wird im Beispiel H) der Bereich von case(1) ausgeführt.

```
system("PAUSE");
// End main() and return 0 (everything was ok)
return 0;   }
```

Das übliche Programmende.

7. Schleifen: for / while / do-while

Öffnen Sie die Datei tutorial_p5_00.c. und starten Sie das Programm.
Das Programm sollte folgende Ausgabe (Console) produzieren:



(Hier sind an drei Stellen die Wartebefehle eingefügt, Sie müssen also dreimal eine Taste drücken, um die gesamte Ausgabe zu sehen.)

Dieses Programm soll Ihnen die drei grundlegenden Möglichkeiten zeigen, wie man Schleifen (loops) programmiert. Schleifen werden immer dann gebraucht, wenn bestimmte Programmteile mehrfach wiederholt werden sollen. Möchte man sich z.B. eine Liste der ersten 10 Zweierpotenzen ausdrucken, so braucht man nicht zehnmal dieselben Befehle zu schreiben. Vielmehr packt man alles in eine Schleife, die – von einer Zählvariable gesteuert – zehnmal wiederholt wird.

Betrachten wir das Programm wieder Schritt für Schritt:

```
// Preprocessor commands
// (Libraries, Custom Files)
#include <stdio.h>

// Main function
int main() {
        int i, counter, x;
```

Der übliche Progammvorspann. Es werden noch drei Ganzzahlvariablen definiert.

7.1 Die for-Schleife

Vor der Schleife weisen wir den Variablen counter und x den Wert 0 bzw. 1 zu:

```
        counter = 0;
        x = 1;
```

Bei der for-Schleife wird eine Zählvariable (hier i) verwendet, die als Ganzzahl (int) definiert wurde. Die eigentliche for-Anweisung hat folgende Syntax:

For (*Variable = Startwert; Schleifenbedingung; Variablen-Änderungsbefehl*) {Schleife}

```
// Basic for() loop
// - condition is tested at the beginning
// - fixed intialization
// - fixed iteration
for (i=0; i<10; i++) {
        counter = counter + 1; // or counter++;
        x = x * 2;
printf("for loop: Counter = %d, x = %d \n", counter, x);
}
```

In unserer Schleife wird also i auf den Startwert 0 gesetzt (i=0) und geprüft, ob i noch kleiner als 10 ist (i<10).
Am Ende jedes Schleifendurchlaufs wird i um 1 erhöht (i++, also i = i+1)
Diese drei Eigenschaften von i werden am *Anfang* der Schleife *einmal* fest definiert.
In unserer Schleife wird dann zur Kontrolle ein Zähler „counter“ jedes Mal um 1 erhöht und die Variable x mit 2 multipliziert. Schließlich werden die aktuellen Werte von counter und x ausgegeben.
Am Ende des 10. Schleifendurchlaufs führt i++ dazu, dass i den Wert 10 annimmt. Das wiederum setzt die Schleifenbedingung i<10 auf „falsch“ und die Schleife wird beendet.
Auf der Console sehen wir die 10 Zeilen mit den Werten 1 bis 10 für „counter“ und den fortlaufenden Zweierpotenzen in x.

```
system("PAUSE");
```

Danach wartet das Programm auf eine Taste.

7.2 Die while-Schleife

```
i = 0;
counter = 0;
x = 1;
```

Am Beginn setzen wir die Variablen wieder auf Anfangswerte. Für die while Schleife ist es wichtig, dass i vor Beginn der Schleife einen Anfangswert hat.
Die eigentliche while-Anweisung hat folgende Syntax:
while (*Schleifenbedingung*) {Schleife}

```
// Basic while() loop
// - condition is tested at the beginning
// - no fixed intialization
// - no fixed iteration
while (i<10) {
        counter ++;
        i++;  // Increment condition
        x = x * 2;
printf("for loop: Counter = %d, x = %d \n", counter, x);
}

system("PAUSE");
```

Die Schleifenbedingung wird *immer wieder am Beginn jedes Schleifendurchlaufs* überprüft. Ist sie nicht mehr wahr (z.B. weil i den Wert 10 angenommen hat), wird die Schleife nicht mehr durchlaufen.
Im Gegensatz zur for-Schleife können (und müssen) die Variablen der Schleifenbedingung innerhalb der Schleife verändert werden. Die Anzahl der Schleifendurchläufe ist daher je nach Programmstruktur veränderlich. In unserem Fall ist sie aber auch 10.
Eine wichtige Variante der while-Schleife ist die **Endlosschleife**, die – zumindest bei Mikrocontrollern – gern im Rahmen des main-Programms verwendet wird, weil es kein übergeordnetes Betriebssystem gibt. Eine Schleife, die mit **while(1)** beginnt, also „solange(immer wahr)“ wird halt immer wieder durchlaufen.

7.3 Die do-while-Schleife

```
        i = 0;
        counter = 0;
        x = 1;
```

Am Beginn setzen wir die Variablen wieder auf Anfangswerte. Auch für die do-while Schleife ist es wichtig, dass i vor Beginn der Schleife einen Anfangswert hat, denn innerhalb der Schleife würde ja immer wieder der „Anfangswert“ zugewiesen.
Die eigentliche do-while-Anweisung hat folgende Syntax:
do {Schleife} while (*Schleifenbedingung*)
Die do-while-Schleife unterscheidet sich von der while-Schleife nur dadurch, dass die Schleifenbedingung **am Ende** jedes Schleifendurchlaufs geprüft wird.
Auch hier können (und müssen) die Variablen der Schleifenbedingung innerhalb der Schleife verändert werden. Die Anzahl der Schleifendurchläufe ist daher je nach Programmstruktur veränderlich. In unserem Fall ist sie aber auch 10.

```
        //Basic do{}while() loop
        // - condition is tested at the end
        // - no fixed intialization
        // - no fixed iteration
        // - runs atleast ones
        do {
                // Do something
                counter ++;
                // Increment condition
                i++;
                x = x * 2;
        printf("for loop: Counter = %d, x = %d \n", counter, x);
        } while (i<10);



        system("PAUSE");
        // End main() and return 0 (everything was ok)
        return 0;
}
```

8 Unterprogramme (Funktionen)

Oft werden bestimmte Programmteile immer wieder benötigt, allerdings nicht in einer monotonen Wiederholung wie bei den Schleifen. Dann ist es praktisch, diese Programmteile als Unterprogramm anzulegen und über den Namen des Unterprogramms aufzurufen.

Ein wichtiger weiterer Vorteil ist, dass durch Unterprogramme der Programmablauf bedeutend übersichtlicher gestaltet werden kann.

Öffnen Sie das Programm tutorial_p6_00.c. Es zeigt Ihnen, wie Funktionen angelegt und aufgerufen werden. Das Programm ist unten abgedruckt.

Die wichtigsten Strukturen sind folgende:

Vor der main-Funktion muß jede (Unter-)Funktion entweder komplett definiert werden. Sie kann aber auch (wie in unserem Fall, was übersichtlicher ist) *nach* der main-Funktion definiert werden, dann muß sie aber vor der main-Funktion wenigstens „als Prototyp erwähnt werden“:

```
// Prototype functions definition
float umfang_kreis(float);
```

Es muß angegeben, wie die Funktion heißt, und von welchem Variablentyp die Ein- und Ausgabewerte sind.

Auch mehrere Eingabewerte sind möglich, aber immer nur *ein* Funktionsausgabewert.

```
float volumen_quader(float,float,float);
```

In der main-Funktion wird die Unterfunktion einfach mit ihrem Namen aufgerufen, sie wird dort „wie eine Variable“ verwendet (die allerdings von anderen Variablen bestimmt wird):

```
        printf("Kreisumfang bei Radius %f ist: %f \n", r, umfang_kreis(r) );
        printf("Quadervolumen x=%f, y=%f, z=%f: %f \n", x, y, z, volumen_quader(x,y,z) );
```

Die Funktion wird folgendermaßen definiert:

```
float umfang_kreis(float radius){
        float umfang;
        umfang = 2 * radius * 3.1415926;
  return umfang;
}
```

Zunächst ist der Ausgabewert-Typ definiert,
es folgen der Funktionsname und eine Liste der Eingabevariablen.
Die Namen der Eingabenvariablen (hier: radius) sind dabei beliebige Namen, die nur innerhalb der Funktion existieren. Man sollte der Übersichlichkeit wegen neue Namen verwenden, die nicht in main vorkommen (muß es aber nicht; die Variablen existieren nur innerhalb der jeweiligen Funktion).
Dann werden üblicherweise weitere funktionsinterne Variablen (hier: umfang) definiert, soweit sie für Berechnungen gebraucht werden.
Die letzte Zeile der Funktion lautet **return *Variable***. Der Wert der hier stehenden Variablen ist es, der als Funktionswert dann an das Hauptprogramm übergeben wird.

Das folgende Beispiel zeigt eine Funktion mit mehreren Eingangsvariablen:

```
float volumen_quader(float x, float y, float z){
        float volumen;
        volumen = x * y * z;
  return volumen;
}
```

Das Gesamtprogramm zu 8.

```
// Preprocessor commands
// (Libraries, Custom Files)
#include <stdio.h>

// Prototype functions definition
float volumen_kugel(float);
float flaeche_kreis(float);
float umfang_kreis(float);
float flaeche_quadrat(float);
float flaeche_rechteck(float,float);
float volumen_quader(float,float,float);


// Main function

int main() {

	float x, y, z, r;

	printf("Radius eingeben:\n");		// Aufforderung zur Eingabe
	scanf("%f", &r);		// Einlesen
	printf("Kreisumfang bei Radius %f ist: %f \n", r, umfang_kreis(r) );
	printf("Kreisflaeche bei Radius %f ist: %f \n", r, flaeche_kreis(r) );
	printf("Kugelvolumen bei Radius %f ist: %f \n", r, volumen_kugel(r) );


	printf("Laenge x eingeben:\n");		// Aufforderung zur Eingabe
	scanf("%f", &x);		// Einlesen
	printf("Breite y eingeben:\n");		// Aufforderung zur Eingabe
	scanf("%f", &y);		// Einlesen
	printf("Hoehe z eingeben:\n");		// Aufforderung zur Eingabe
	scanf("%f", &z);		// Einlesen

	printf("Quadratflaeche  bei Kantenlaenge  %f ist: %f  \n", x, flaeche_quadrat(x) );
	printf("Rechteckflaeche bei x = %f und y = %f ist: %f \n", x, y, flaeche_rechteck(x,y) );
	printf("Quadervolumen  bei x = %f, y = %f und z = %f ist: %f \n", x, y, z, volumen_quader(x,y,z) );

	system("PAUSE");
	// End main() and return 0 (everything was ok)
	return 0;
}

float umfang_kreis(float radius){
	float umfang;
	umfang = 2 * radius * 3.1415926;
   return umfang;
}

float flaeche_kreis(float radius){
	float flaeche;
	flaeche = radius * radius * 3.1415926;
   return flaeche;
}

float volumen_kugel(float radius){
	float volumen;
	volumen = radius * radius * radius * 4/3 * 3.1415926;
   return volumen;
}
```

```
float flaeche_quadrat(float x){
        float flaeche;
        flaeche = x * x;
   return flaeche;
}


float flaeche_rechteck(float x, float y){
        float flaeche;
        flaeche = x * y;
   return flaeche;
}


float volumen_quader(float x, float y, float z){
        float volumen;
        volumen = x * y * z;
   return volumen;
}
```

9 Einige Besonderheiten bei Mikrocontrollern

Die bisherigen C-Programme haben Sie auf einem PC laufen lassen.
Die Eingabe der Werte erfolgte von der Tastatur, die Ausgabe auf dem Monitor.
Beides fehlt zunächst bei Mikrocontrollern.

Die Eingabe ist meist die Abfrage von Spannungswerten, die als logische 0 oder 1 interpretiert werden. Zum Beispiel kann ein Taster gedrückt sein oder nicht und dann eine Spannung von 5 Volt (= logisch 1) oder 0 Volt (= logisch 0) liefern.

Die Ausgabe ist meist die Ausgabe von Spannungswerten, die als logische 0 oder 1 interpretiert werden müssen. Zum Beispiel kann ein Pin des Mikrocontrollers eine Spannung von 5 Volt (= logisch 1) oder 0 Volt (= logisch 0) liefern, wodurch z.B. eine LED leuchtet oder nicht.

Komfortablere Lösungen kommunizieren mit anderen Systemen (z.B PCs) über USB oder andere Schnittstellen. Auch können Informationen z.B. an LCD-Displays anzeigt werden. Das aber verlangt nach z.T. komplizierten Programmen.

Die einfachste Art der Ausgabe wird im Versuch EP11 demonstriert:

```
LATA = 0b000001;
```

An z. B. die Variable LATA wird ein binärer Muster aus Nullen und Einsen gegeben.
„0b“ bedeutet, dass es sich um eine Binärzahl handelt. Die Variable LATA bedeutet den A-Port, eine Anordnung von bis zu 8 Ausgangspins des Mikrocontrollers.

Die einfachste Art der Eingabe ist, die Variable LATB (oder einen anderen Port) zu lesen und zu prüfen:

```
if(PORTB == 0b00000001) {mache etwas}
```

Es handelt sich hier um eine if-Bedingung. Wenn an Port B das Bitmuster 0000 0001 anliegt (z.B. weil ein Taster gedrückt wurde) wird etwas gemacht.

Ob ein bestimmter Pin ein Eingang oder ein Ausgang ist, wird durch andere Variablen bestimmt. Man spricht auch von *Registern*, die in der Regel immer 8 Bit umfassen:

```
TRISA= 0b000000; // alle 6 Pins (Bits) von Port A sollen Ausgänge sein
TRISB= 0b11111111; // alle 8 Pins (Bits) von Port B sollen Eingänge sein
```

Weitere Einzelheiten finden Sie im Skript zum Versuch EP11.

ANHANG

Zusammenfassung der wichtigsten Programmstrukturen

A. 1 Hauptprogramm

```
int main() {
  Befehle;
        return 0;
}
```

Ausgabe

```
        printf("Hello World\n");
```

Warten auf Eingabetaste

```
        system("PAUSE");
```

A. 2 Texteingabe

```
        char eingabetext[100];   // char-string ("Textvariable") der Länge 100
        scanf("%s", &eingabetext);     // Einlesen des Textes
```

Ausgabe von Variableninhalten (Text)

```
        printf("Dein Text ist: %s\n", eingabetext);  // Ausgabe des gespeicherten Textes
```

A. 3 Variablentypen und Ausgabe

```
        char  eingabetext[100];   // char-string ("Textvariable") der Länge 100
        int   anzahl;             // eine ganze Zahl
        float preis;              // eine Zahl mit Nachkommastellen

        printf("Artikel ist: %s\n", eingabetext);  // Ausgabe des gespeicherten Textes
        printf("Anzahl ist: %d\n", anzahl);  // Ausgabe
        printf("Preis ist: %f\n", preis);  // Ausgabe des gespeicherten Textes
```

A. 4 Variablenzuweisungen im Programm

```
        festezahl = 3;
        sonderpreis = 0.99;
```

Buchstaben und Arrays

```
        char  nur_ein_buchstabe;   // Definition
        nur_ein_buchstabe = 'B';   // Zuweisung
        printf("Ein Buchstabe: %c \n", nur_ein_buchstabe);

        int zahlenfeld [5];       // Array definieren
        zahlenfeld[0] = 0;  // Die Indizes beginnen immer bei 0
  ( ... )
        zahlenfeld[5] = 25;

printf("Inhalt von Variable Nr. %d ist %d \n", indexnummer, zahlenfeld[indexnummer]);
printf("Inhalt von Variable Nr. 2 ist %d \n", zahlenfeld[2]);
```

A.5 Arithmetik

```
int   x, y;
float m, n;

x = 0;
y = 0;
m = 0;
n = 0;

// Addition, Subtraktion
x = x + 1;
y++;
x = x + y;
m = n + 5;
n = 10.47 + 3.14;

y--;
x = y - x;
m = m + n + 0.01;
m = x + n;

// Vorsicht beim Vermischen von Variablentypen!
//  Speichern in Ganzzahlen = Verlust der Nachkommastellen!
x = m + y;

// Multiplikation
x = 4;
y = 24;
x = y*x;
m = 3.4;
n = 5.6;
n = m*n;

// Division
x = x/y;
n = n/m;

// Vorsicht bei der Division mit Ganzzahlen (integer)!
// Es gehen die Nachkommastellen verloren, auch beim Abspeichern in float
y = 29;
x = 4;
n = y/x;
```

A.6 Logische Vergleiche, Bedingungen, bedingte Sprünge (if, else, case)

Beispiel A: if-else

```
if ( x == 1) {
        printf("x is 1\n");
} else {
        printf("x is not 1\n");
}
```

Beispiel B: Gleichheit ==

```
if (x == 0)
```

Beispiel C: Ungleichheit !=

```
if (x != 0)
```

Beispiel D: Verschalteltes if-else

```
printf("D:\n");
if (y == 1) {
        printf("y = 1\n");
} else if ( y == 2) {
        printf("y = 2\n");
} else {
        printf("y is not 1 or 2\n");
}
```

Beispiel E: Logisches UND &&

```
if (x == 1 && y == 2)
```

Beispiel F: Logisches OR ||

```
if (x == 1 || y == 1) {
```

Beispiel G: Logische Invertierung !(…)

```
if (!(x == 2)) {
```

Beispiel H: switch-case

```
switch (x) {
        case 0:
                printf("x is 0\n");
                break;
        case 1:
                printf("x is 1\n");
                break;
        case 2:
                printf("x is 2\n");
                break;
        case 3:
                printf("x is 3\n");
                break;
        default:
                break;
}
```

A.7 Schleifen: for / while / do-while

7.1 Die for-Schleife

Syntax:
For (*Variable = Startwert; Schleifenbedingung; Variablen-Änderungsbefehl*) {Schleife}

```
for (i=0; i<10; i++) {
        Befehle;
}
```

7.2 Die while-Schleife

Syntax:
while (*Schleifenbedingung*) {Schleife}

```
i = 0;

while (i<10) {
        Befehle;
}
```

Endlosschleife:
Eine Schleife, die mit **while(1)** beginnt, wird immer wieder durchlaufen.

7.3 Die do-while-Schleife

Syntax:
do {Schleife} while (*Schleifenbedingung*)

```
i = 0;
        do {
        Befehle;
        } while (i<10);
```

A.8 Unterprogramme (Funktionen)

```
// Prototype functions definition
float umfang_kreis(float);
float volumen_quader(float,float,float);
```

In der main-Funktion:

```
        printf("Kreisumfang bei Radius %f ist: %f \n", r, umfang_kreis(r) );
        printf("Quadervolumen x=%f, y=%f, z=%f: %f \n", x, y, z, volumen_quader(x,y,z) );
```

Die Funktion wird folgendermaßen definiert:

```
float volumen_quader(float x, float y, float z){
        float volumen;
        volumen = x * y * z;
   return volumen;
}
```

A.9 Einige Besonderheiten bei Mikrocontrollern

Die einfachste Art der Ausgabe wird im Versuch EP11 demonstriert:

```
LATA = 0b000001;
```

Die einfachste Art der Eingabe ist, die Variable LATB (oder einen anderen Port) zu lesen und zu prüfen:

```
if(PORTB == 0b00000001) {mache etwas}
```

Definition, ob ein bestimmter Pin ein Eingang oder ein Ausgang ist:

```
TRISA= 0b000000; // alle 6 Pins (Bits) von Port A sollen Ausgänge sein
TRISB= 0b11111111; // alle 8 Pins (Bits) von Port B sollen Eingänge sein
```