Inhaltsverzeichnis

So, auf vielfachen Wunsch, hier nun ein kleines Inhaltsverzeichnis, damit ihr schneller findet, über was ihr euch informieren wollt:

Kapitel I

(Start-Up)

Eine Einführung...

Hi an alle Programmierer da draußen und die, die es gerne noch werden möchten!
Dies ist mein erstes Tutorial was ich schreibe, also vergebt mir kleinere und größere Fehler in ihm. Wenn ihr etwas gefunden habt, dann meldet euch bitte
Bei mir. Auch für Fragen/Lob/Kritik bin ich natürlich für euch erreichbar!
E-Mail: mereep@gmx.net
So, nun aber erstmal genug der Vorrede und kommen wir nun zu dem Thema
Der Überschrift zurück „Eine Einführung". Also, was ist PureBasic eigentlich?
PureBasic ist eine Programmiersprache, aber ich glaube das weißt du ja.
Mit Programmiersprachen kann man Programme schreiben, die das Betriebssystem dann direkt ausführen kann. Es gibt davon eine ganze Menge. Die bekanntesten sind wohl C, C++, Pascal, Java, Basic, Delphi, Oberon, (Assembler) mittlerweile auch Sprachen wie zum Beispiel C#, OBERON.NET, VBasic7.NET (.NET konforme Sprachen) oder Java All diese Sprachen haben natürlich ihre Vor - und Nachteile. Als Anfänger steht man dann vor einer schwierigen Entscheidung. Es gibt so viele Dinge, die man in die Entscheidung einfließen lassen sollte. Doch Ende ist es eigentlich relativ egal für was du dich entscheidest. Als Anfänger solltest du dich aber vielleicht doch erstmal für eine Basic-Sprache entscheiden, da diese meist recht leicht erlernbar sind. So zum Beispiel die Sprache PureBasic. Diese hat aber noch einige Vorteile mehr, als „nur" leicht erlernbar zu sein. Sie ist schneller als die meisten anderen Basic-Sprachen, unterstützt Inline-Assembler, unterstützt direkte API-Calls, produziert relativ kleine Exe-Dateien, ist für mehrere Betriebssysteme verfügbar, hat einen großen eingebauten Befehlssatz, ist für jeden bezahlbar und hat eine nette Community, was echt äußerst wichtig sein, gerade am Anfang. Dies sollten die wichtigsten Highlights von PureBasic sein. Falls du noch richtiger Anfänger bist, sagen dir die meisten Sachen von da oben sicher wenig, doch das werden wir schon irgendwie hinbekommen, hoffe ich :-)
Eine kostenlose Version von PureBasic kann man unter http://www.purebasic.de downloaden. Dort kann es auch gekauft werden. PureBasic wird oft aktualisiert, weshalb man dort auch immer die neusten Versionen laden kann. Die Community befindet sich auf www.pure-board.de.

Die kostenlose Version von PureBasic ist wie folgt eingeschränkt:
- Es können keine DLL-Dateien erstellt werden.
- Die Win32 API kann nicht (direkt ;-) genutzt werden
- es gibt kein Entwicklerpaket für externe Bibliotheken
- Man kann nur ca. 800 Zeilen Sourcecode schreiben
- Der Debugger lässt sich nicht deaktivieren

Ok, nun mal weiter. In diesem Tutorial werde ich bei 0 anfangen, also ich setze keinerlei Erfahrung in irgendeiner Programmiersprache voraus. Allerdings kann ich nicht versprechen, das du immer alles gleich raffst, aber ich hoffe ich krieg das irgendwie rüber…Also fangen wir mal an…

Die Entwicklungsumgebung in PureBasic

Ok, wirf' jetzt erstmal PureBasic an.

Es gibt ein Reiter, wo <Neu> stehen sollte. So heißt die aktuelle Quelldatei. Sobald du die Datei das erste Mal gespeichert hast, wird sich das „New" zum angegebenen Dateinamen ändern. In diesem Abschnitt kann man einfach zwischen mehreren Quelldateien hin und her springen. Unten Links steht, an welcher Position sich die Schreibmarke gerade befindet (Zeichen : Zeile). Dort wird, wenn du einen Befehl eingibst, eine kurze Beschreibung des Befehls und die erwarteten *Parameter* (Übergabewerte) angegeben. Doch auch dazu später. Wenn du den Editor das erste Mal verwendest, solltest du gleich mal die Sprache auf Deutsch umstellen. Dazu klickst du auf File->Preferences. Es sollte sich ein Fenster öffnen, wo du nach dem Begriff „Language" suchst. Dort kannst du die Sprache des Editors verändern. Übrigens kann man in diesem Fenster auch die Farben für den Editor verändern. Dies ist der Standardeditor von PureBasic, doch es gibt natürlich auch Alternativen. Die wohl bekannteste heißt jaPBe. Dieser Editor bietet einige Funktionen mehr und besseren Komfort als der Standardeditor. Ich verwende ihn auch, da es wirklich einfacher geht damit zu entwickeln, doch für Anfänger sind die zusätzlichen Funktionen sicher eher verwirrend als nützlich, also benutz erstmal den Editor. Ich werde auch alles an diesem Editor erklären. Wenn du dann mit PureBasic umgehen kannst, empfehle ich dir aber dann zu wechseln, da jaPBe Freeware ist.

Die ersten Schritte mit PureBasic

Jetzt ist es soweit, wir schreiben unser erstes kleines „HelloWorld"- Programm mit PureBasic! Gebt mal folgendes in das Fenster von PureBasic ein:

```
OpenConsole()
  ConsoleTitle("Hallo Welt")
  PrintN("Hallo Welt!")
  MessageRequester("Hallo", "Es funktioniert :-)",
#PB_MESSAGEREQUESTER_OK)
CloseConsole()
End
```

Und jetzt drücke F5! (Oder Compiler -> Kompilieren/Starten) Das Programm wird jetzt ausgeführt. Doch wieso passiert das hier eigentlich? Was bedeutet das? Falls du noch NIE programmiert hast, dann mach dir jetzt ein Bierchen auf und trink es bevor du weiter liest! Ok, jetzt weiter. Also, in der ersten Zeile steht ***OpenConsole()***. Dieser *Befehl* öffnet dieses schwarze „Fenster". Er benötigt keine *Parameter*, weshalb in den Klammern nichts steht. Der nächste Befehl nennt sich ***ConsoleTitle()***. Er ändert die Titelzeile der geöffneten Console.

Wie ihr oben seht, braucht dieser Befehl genau einen *Parameter.* Dieser Parameter ist vom Typ „String" (nein, nicht DER ;-). In diesem Fall heißt das Zeichenkette. Also einfach einen Text. Strings müssen immer in Anführungszeichen übergeben werden, sonst gibt's ne Fehlermeldung, dass der falsche Typ übergeben wurde. Warum das so ist, klären wir dann. Momentan ist es eben einfach so! In unserem Fall ändert er den Titel der Console also zu „Hallo Welt". (Die Einrückung die ich vor diesem Befehl gemacht habe ist eigentlich unwesentlich, sie dient nur zur Übersichtlichkeit!).
Als nächster Befehl steht ***PrintN()***. Dieser Befehl schreibt den übergebenen Text in die Console. Außerdem „schreibt" er noch ein NewLine dahinter. Dieser Befehl setzt die Schreibmarke für weitere Schreib-Befehle also auf die nächste Zeile. Genau dies macht der Befehl ***Print()*** nicht, obwohl er ansonsten dieselbe Arbeit verrichtet. Als nächstes steht nun der Befehl ***MessageRequester()***. Dieser scheint etwas komplizierter zu sein. Er öffnet dieses kleine Fenster mit der OK-Taste. Er benötigt 3 Parameter. Parameter werden immer mit KOMMA voneinander getrennt. Die ersten 2 Parameter sind wieder Zeichenketten. Der Erste ist der Titel des kleinen Fensters, der andere ist der Text des Fensters. Beim nächsten wird es wieder interessant. Er ist nicht vom Typ String. Er erwartet eigentlich eine ZAHL. Aber da steht ja gar keine, komisch nicht ;-). Ok, schlürf jetzt den letzten Schluck…dieses *#PB_MessageRequester_Ok* ist eine *Konstante*. Sie ist in DIESEM Fall wie ein Synonym für eine Zahl. Du könntest stattdessen auch eine 0 einsetzen, und es würde das gleiche rauskommen. Dieser Parameter sagt dem Befehl einfach, dass der MessageRequester einen OK-Knopf haben soll. Das PB am Anfang der Konstanten steht übrigens einfach nur für **P**ure**B**asic. Dieser Befehl akzeptiert folgende Konstanten:

```
#PB_MessageRequester_YesNo ;Um die 'Yes' (Ja) oder 'No' (Nein) Schalter zu
erhalten
#PB_MessageRequester_YesNoCancel ;Um die 'Yes' (Ja), 'No' (Nein) und
'Cancel' (Abbruch) Schalter zu erhalten
#PB_MessageRequester_Ok ;Um nur einen 'Ok' Schalter zu erhalten
```

Das könnt ihr übrigens auch in der Hilfe nachlesen. Dieser MessageRequester hält übrigens das Programm solange an, bis er geschlossen wird. Der nächste Befehl ist ***CloseConsole()***. Er schließt, wie der Name schon sagt, die geöffnete Console wieder. Er benötigt auch keine Parameter. Der nächste Befehl, oder besser das nächste Schlüsselwort heißt **END.** Dieses Schlüsselwort wird NICHT mit Klammern geschrieben. Es beendet das Programm und gibt alle verwendeten Ressourcen wieder frei. Er würde zum Beispiel auch sowieso die Console schließen, wenn wir das nicht schon vorher getan hätten. (Ich hätte hier eigentlich nicht End schreiben müssen, da das Programm da sowieso zu ende wäre, PureBasic registriert dies und führt End eigentlich automatisch aus, aber einfach zur Vollständigkeit habe ich es trotzdem mit hingeschrieben, da ihr diesen Befehl sowieso brauchen werdet.)
So, jetzt is eigentlich nur noch eine Kleinigkeit zu erklären, und zwar dieses Fenster, wo PureBasic Debugger draufsteht. Das Teil geht automatisch auf, wenn ihr ein Programm startet und den DEBUGGER aktiviert habt. In diesem Programm kann man den Programmablauf beobachten, sich *Variablen anzeigen lassen,* Register auslesen und ändern, das Programm Stoppen, Schritt für Schritt abarbeiten oder komplett beenden. Auch kann man sich auf diesen Ding im Programmverlauf Infos anzeigen lassen. Dieses Tool(dt.: Werkzeug) ist somit dazu da um Bugs -> Fehler(dt.: Käfer) im Programm zu finden und das Programm im Notfall zu beenden. Allerdings hat der

Debugger auch seinen Tribut. Durch die ständige Fehlerüberwachung wird das Programm langsamer. Man kann ihn unter Compiler->Debugger an- und ausschalten. Allerdings ist er in der kostenlosen Demoversion nicht deaktivierbar.

Einführung in Variablen

Nachdem wir gerade ein kleines Programm geschrieben haben, obwohl du noch gar keine Ahnung von PureBasic hattest, kommen wir nun zu einem ersten wirklich wichtigen Kapitel. Wenn ihr vorhin nicht ganz alles gerafft habt, macht nichts, konzentriert euch lieber jetzt: Dieses Kapitel wird nur GRUNDLEGENDES zu Variablen verkünden, wir werden später noch einmal darauf zurückkommen, wenn wir etwas weiter sind. Variablen gibt es in JEDER Programmiersprache, was ihr hier lest werdet ihr auch in anderen Programmiersprachen anwenden können. Also, was sind jetzt diese verdammten *Variablen*? Mhm … ich habe euch vorhin schon mal kurz diese Konstante #PB_MESSAGEREQUESTER_OK verwendet und habe gesagt, dass dies ein Synonym für eine Zahl, nämlich in diesem Fall 0 ist, richt? Ja, richtig, wenn ich das sage dann stimmt das auch ;-) Ok, das ist so etwas ÄHNLICHES wie eine Variable, aber eben nicht ganz. In eine Variable kann man einfach Daten abspeichern. Eine Variable kann einen beliebigen Namen haben. Eine Variable kann aber nur eine Art von Daten speichern, was ihr bevor ihr sie benutzt, festlegen solltet!
Wie sieht jetzt so eine Variable aus?

```
IchBinEineVariable.w  ;Word-variable
IchAuch.l             ;Long-Variable
HalloDu.s             ;String-Variable
Huhu.b                ;Byte-Variable
UndNochEine.f         ;Float-Variable
```

Hab' ich´s jetzt geschafft? Siehst du noch durch? Wenn du Anfänger bist, sicher nicht. Das bekommen wir gleich hin. Ihr könnt das „Programm" spaßeshalber mal ausprobieren! Ihr werdet sehen: Es passiert nichts. Ist auch ganz klar. (Die Dinge hinter einem Semikolon werden beim *kompilieren* (dt.: übersetzen des Programmes) einfach ignoriert). Wir *deklarieren* ja auch nur ein paar Variablen. Wir haben danach fünf Variablen, mit fünf verschiedenen Typen. Dies sind die Standardtypen, die von PureBasic unterstützt werden. Den Typ der Variable gibt man, wie oben gezeigt, bei der Variable an, indem man einen Punkt und dann den Typ hinter den Variablennamen schreibt. Das braucht man aber nur beim Ersten mal, wenn man die Variable benutzt, machen. Man kann diesen Variablen verschiedene Daten zuweisen. Bis auf den Stringtyp sind alle anderen Standardtypen prinzipiell ZAHLEN-TYPEN. Man kann in ihnen also KEINE Zeichen direkt speichern(Naja, es geht alles über Umwege, aber das klären wir auch später). Man weißt ihnen einfach mithilfe des Gleichheitszeichen einen Wert zu.

```
IchBinEineVariable.w    ;Word-variable
IchBinEineVariable = 10 ;Hier wird dieser Variable der Wert 10 zugewiesen
```

Nun hat die Variable mit dem Namen „IchBinEineVariable" den Zahlenwert 10 zugewiesen bekommen. Man kann diese Variable nun anstelle von zehn in Rechnungen und sonstigem Verwenden. Wie der Name schon sagt, sind Variablen *variabel*, was soviel wie veränderlich heißt. Man kann ihnen also jederzeit einen

anderen Wert zuweisen(Aber niemals den Typ ändern!). Doch was bedeuten nun diese verschiedenen Typen von Variablen?
Der Typ **.w,** also WORD, kann Zahlen von -32768 bis +32767 speichern. Falls du diesen Wertebereich überschreitest, bekommt die Variable automatisch ihren GERINGSTEN Wert zugewiesen. Das nennt sich *Overflow*. Das ganze geht natürlich auch umgedreht und nennt sich dann *Underflow.* Dies ist durch die Binäre Arbeitsweise des Computers bedingt. Merk dir voerst erstmal, dass es einfach so ist. Der Typ **.l**, also LONG, kann Zahlen von -2147483648 bis +2147483647 speichern. Auch hier gilt natürlich der Spaß mit dem Under und dem Overflow! Dieser Typ sollte in der Regel bevorzugt werden, da er von den 32 Bit Prozessoren, die wir zur Zeit noch fast alle haben, am schnellsten verarbeitet werden können. 64Bit Prozessoren sind im Kommen, aber es dauert noch einige Jahre bis sie sich verbreitet haben und erst das neue Betriebssystem von Windows – Windows Vista – wird davon Gebrauch machen. Dies ist auch der Standardtyp von Variablen, also wenn du es nicht manuell regelst, bekommen alle Variablen, die du NICHT genau deklarierst AUTOMATISCH diesen Typ zugewiesen. Der Typ **.s,** also STRING, speichert Zeichenketten. Man kann in ihnen also Text speichern. Das stimmt zwar so nicht genau, da eigentlich nur ASCII-Zahlen gespeichert werden, aber das vergisst du jetzt erstmal schnell und merkst dir, dass man in ihnen Strings(Zeichenketten) speichern kann. Vergiss, wenn du ihnen einen Wert zuweißt, die Anführungszeichen nicht! Der Typ **.b,** also BYTE, kann Zahlen von -128 bis +127 speichern. Die alten Regeln gelten auch hier wieder. Der Typ **.f**, also FLOAT, speichert FLIEßKOMMAZAHLEN. Diesen Typ braucht ihr nur selten, eigentlich nur für Berechnungen. Soweit, wie du kannst, versuche diesen Typ zu meiden, da er nie wirklich genau arbeitet. Er kann, auch wieder aufgrund der binären Verarbeitung, nur Typen, die man mit dem Teiler 2 errechnen kann, richtig genau speichern. Also nur, wenn du diesen Typ unbedingt brauchst, verwenden! Doch warum jetzt so viele verschiedene Typen für ein paar Zahlen? Die Frage lässt sich leicht beantworten: Die Variablen brauchen unterschiedlich viele Bytes des Speichers. Hier noch mal eine kleine Tabelle aus der PB-Hilfe, die die ganze Sache mal richtig zusammenfasst:

Name	**Erweiterung**	**Speicherverbrauch**	**Bereich**
Byte	.b	1 Byte im Speicher	-128 bis +127
Word	.w	2 Byte im Speicher	-32768 bis +32767
Long	.l	4 Byte im Speicher	-2147483648 bis +2147483647
Float	.f	4 Byte im Speicher	unlimitiert (siehe oben ;-)
String	.s	Länge des Strings + 1	bis zu 65536 Zeichen

Fasst die gleiche Tabelle ist in der Hilfe noch mal nachzulesen. Achja, bevor ich`s vergesse, den Typ **.s** könnt ihr auch mit einem „**$**" (Dollarzeichen) angeben. Dies muss man dann aber IMMER hinter die Variable schreiben. So, dann wäre eigentlich nur noch eine Sache zu klären, die KONSTANTEN. Das sind, wie schon weiter oben erwähnt, eigentlich das gleiche wie Variablen. Allerdings kann man diesen nur EINMAL einen Typ zuweisen. Sie sind wie der Name schon sagt, konstant. Man setzt diesen Variablen immer ein Rautezeichen vor (#). Hier muss man nicht unbedingt einen Typ angeben, er wird automatisch erfasst. Das ganze sieht dann so aus:

```
#Zahl1 = 1213
#Zahl2 = 12323213
```

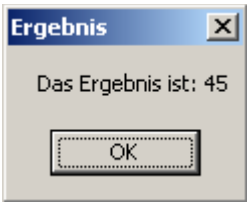
```
#KommaZahl = 2.2424
#Zeichenkette ="Hallo"
```

Diesen Variablen können keine anderen Werte zugewiesen werden. Ihr könnt es gerne mal versuchen, aber ihr werdet eine Fehlermeldung erhalten! Groß und Kleinschreibung wird übrigens bei Variablen ignoriert .Jetzt wäre wohl das allernötigste zu „normalen" Variablen geklärt. Weiter geht´s mit einem bisschen Probieren :-)

Gebt mal folgenden Text in euren Editor ein:

```
Zahl1.l = 20
Zahl2.l = 25
Ergebnis.l = Zahl1 + Zahl2
MessageRequester("Ergebnis","Das Ergebnis ist: "+Str(Ergebnis),0)
End
```

Führt ihn aus. Folgendes müsste bei euch erscheinen:



Versucht es erstmal soweit wie möglich selbst zu überlegen, warum das hier jetzt kommt. Jetzt gehen wir das mal Zeile für Zeile durch. In den ersten 2 Zeilen werden zwei long-Variablen deklariert. Ihnen wird auch gleich ein Wert zugewiesen(20 und 25). In der nächsten Zeile wird auch wieder eine Long-Variable deklariert und ihr wird auch wieder gleich ein Wert zugewiesen. Und zwar das Ergebnis der Rechnung Zahl1 + Zahl2, also der Rechnung 20 + 25. Das ergibt, wie sich hoffentlich jeder denken kann, 45. Die Variable Ergebnis hat nun also einfach den Wert 45. Als nächstes wird der MessageRequester, den du ja schon kennst, aufgerufen. Hier wirst du wahrscheinlich nicht verstehen, aber das klären wir jetzt. Der erste Parameter, dass „Ergebnis" sollte dir klar sein, es ist einfach der Titel dieses MessageRequester (Auch MessageBox genannt!). Beim nächsten wirst du nicht mehr wissen, was das soll, wenn du noch Anfänger bist. Wie du weißt, ist dies der Text, der in dem MessageRequester steht. In diesem Fall steht dort:
„Das Ergebnis ist: " +Str(Ergebnis).
Dieser Parameter erwartet auch wieder einen String. Wir wollen aber unter anderem eine ZAHL ausgeben lassen, nämlich die 45. Ich sagte ja vorhin, dass die man die in Zahlentypen keine Strings speichern kann(gilt natürlich auch umgedreht). Aber auch sagte ich, dass dies alles über Umwege geht. Hier tun wir das, wir machen eine ZAHL zu einem STRING. Dies macht der Befehl **Str()** dieser wandelt die übergebene Zahl in einen **Str**ing um. In unserem Fall wird die Zahl der Variable „Ergebnis", also die 45, in einen String umgewandelt. Der String wird dann einfach zu dem anderen „addiert". Und es entsteht ein String. Zum Schluss steht also kurz gesagt an dieser Stelle: "Das Ergebnis ist: " + "45". Wenn man das Str() weglassen würde, dann käme eine Fehlermeldung, da man versuchen würde, eine Zahl zu einem String zu addieren und dies nicht ohne weiteres möglich ist. Es kommt also eine Fehler-

meldung – So ist das einfach ;-) Der nächste Übergabeparameter ist eine 0. Warum jetzt die 0? Diese Frage habe ich schon weiter oben beantwortet. Ich hätte auch wieder #PB_MessageRequester_Ok schreiben können, da diese Konstante auch den Wert 0 enthält. Diese 0 sagt dem Befehl einfach nur, dass er einen Ok-Knopf zu machen hat. Falls du das mit dem Str() noch nicht kapiert hast, hier noch ein kurzes Beispiel:

```
Zahl.l    = 8
String1.s = "Ich "
String2.s = "bin "
String3.s = "ein "
String4.s = "aus "
String5.s = "Teilen "
String6.s = "zusammengesetzter "
String7.s = "String"
Gesamt.s = String1 + String2 + String3 + String4 + Str(8) + " " + String5 +
String6 + String7
MessageRequester("Info",Gesamt,0)
```

Der folgende Requester müsste auftauchen:

Hier werden einfach nur ein ganzer Haufen Strings aneinander gesetzt um einen String zusammengesetzten zu erhalten, welcher dann in der Variable Gesamt gespeichert wird und dann mal wieder in einem MessageRequester ausgegeben wird. Achtung mit dem Str() kann man nur Byte, Long und Word Variablen umgewandelt werden. Für Floatvariablen musst du **StrF()** verwenden! So, Leute verdaut das jetzt erstmal ganz langsam. Ich hoffe, ihr habt das einigermaßen verstanden, auch, wenn das ein bisschen schwieriger Stoff war, wenn du noch nie Erfahrung mit der Programmierung gemacht hast. Aber tröste dich, hast du das gerafft, dann sollte der Rest auch machbar sein. Also gehen wir jetzt auf zum nächsten Kapitel, der Programmsteuerung, wo wir Themen wie Schleifen, Prozeduren, Sprünge, If-Blöcke und so weiter behandeln, also bereite dich schon mal selig und moralisch darauf vor…

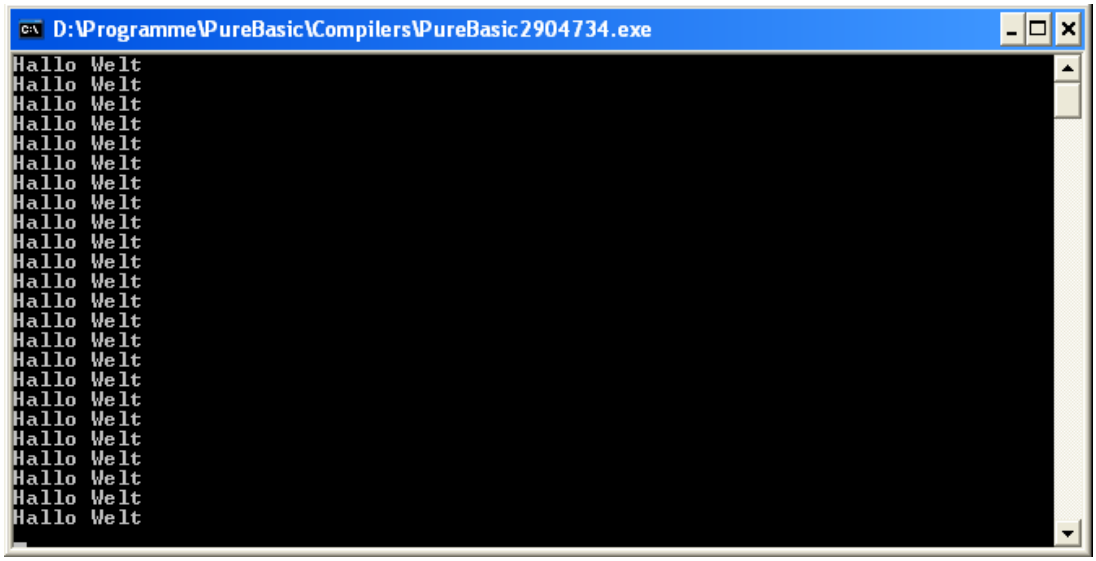
Kapitel II
(Start-Up)

Schleifen

Hi und willkommen im zweiten Kapitel dieses Tutorials! Ich hoffe ihr habt das Erste ohne Nebenwirkungen überstanden, denn wir wollen jetzt wieder voll einsteigen! Was ihr hier so mitbekommt ist in großem und ganzen auch in anderen Sprachen ähnlich, bloß dass der *Syntax* (Die Grammatik) etwas variieren wird. Als Anfänger wirst du dich jetzt sicher fragen: Was ist das – Schleifen? Also: Schleifen sind prinzipiell dazu da, einen bestimmten Abschnitt deines Programmes mehrmals auszuführen. Wozu braucht man das? Ganz einfach .... wir nehmen mal das simple Beispiel, das du 100-mal „Hallo Welt" schreiben willst … und jetzt überleg mal, wie du das machen würdest, du könntest jetzt 100 mal PrintN(”Hallo Welt") eingeben, über den Sinn davon brauchen wir jetzt glaub ich nicht zu streiten, wenn man so als Programmierer arbeiten müsste, würdest du DAS hier sicher nicht lesen können, da es ganz einfach keine Programme gäbe! Ok, das hast du jetzt hoffentlich verstanden, kommen wir nun mal zu der ERSTEN Schleife…ja, genau, davon gibt´s einige :-) Fangen wir mal mit der Schleife

For-Next an und nehmen uns dazu mal das irrsinnige Beispiel von oben und schreiben 100 mal Hallo Welt in eine Console. Der Code dazu sieht dann so hier aus:

```
OpenConsole()           ;Console öffnen

For X = 0 To 99         ;Schleifenbeginn (Kopf)
  PrintN("Hallo Welt")  ;Schreibt Hallo Welt
Next                    ;Schleifenende(Wendepunkt)

Delay(3000)             ;Wartet 3000 Millisekunden->3 Sekunden

CloseConsole()          ;Schließt die Console
End                     ;Beendet das Programm
```



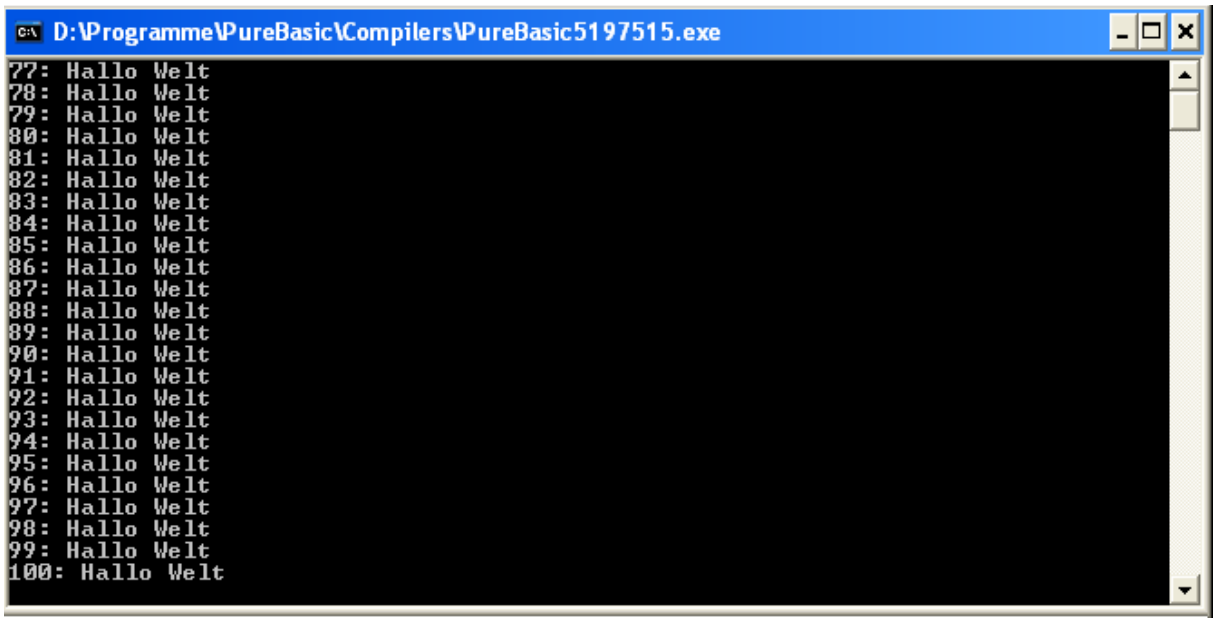
Schön, nicht?! ;-)
Wir werden das jetzt mal wieder Zeile für Zeile durchgehen. Das **OpenConsole** ist klar, aber in der (über)nächsten Zeile wird es schon wieder knifflig… Da steht **For X = 0 to 99** Was bedeutet das? Dies ist der Kopf der For-Next Schleife. Das **For** bezeichnet einfach nur die For-Schleife, es ist das eigentliche Schlüsselwort.
X = 0 bedeutet soviel wie: Deklariere die Variable X (Falls du das nicht schon getan hast) und ordne ihr den Wert 0 zu. Ich hätte auch **X.w = 0** schreiben können, um zu erzwingen, dass X eine Word-Variable wird. Wenn ich nichts angebe, wird entweder der alte Typ genommen(Falls es X schon gäbe), oder, wenn die Variable vorher nicht verwendet wurde, der LONG-Typ(wie in unserem Fall). Das **to 99** bedeutet „bis 99". Sobald die Variable X den Wert 99 überschreitet(!), wird die Schleife verlassen. Solange X also kleiner oder gleich 99 ist, wird die Schleife ausgeführt. Das nächste **PrintN** ist dir ja auch schon bekannt. Allerdings sieht es in der nächsten Zeile mit dem **Next** schon anders aus. Das Next zeigt das Ende der For-Next Schleife an. Alles was ZWISCHEN For und Next steht, wird solange ausgeführt, bis die Bedingung der Schleife erfüllt ist. (X muss größer als 99 sein). Solange X also kleiner oder gleich 99 ist, wird die Schleife ausgeführt. Next springt damit automatisch zum Kopf der Schleife. Außerdem erhöht es die Variable X automatisch um den Wert **1**! Dies macht diese Schleife IMMER, wenn nichts anderes angegeben ist. Wenn du aber willst, dass die Variable X jedes Mal um **2** erhöht wird, dann musst du hinter dem Kopf noch **Step 2** eingeben. Du kannst natürlich jeden anderen Wert (auch negative Werte) einsetzen. Das sieht dann so aus:

```
For X = 0 to 99 Step 2
oder
For X = 0 to 99 Step 10
oder
For Variable = 99 to 0 Step -10
```

Pass hier auf, dass du keine Endlosschleife produzierst. Eine Endlosschleife ist eine Schleife, die durch einen Fehler kein Ende finden. Man könnte so eine Endlosschleife folgendermaßen Schreiben:(Nicht ausprobieren)

```
For Var = 0 To 100 Step 1  ;Normale For-Schleife
  Var = Var - 1            ;Hier wird der Variable Var jedesmal 1 abgezogen
Next                       ;Hier wird wieder 1 dazuaddiert
```

Diese Schleife wird von selbst NIEMALS enden. Hier wird der Variable Var immer in der Schleife 1 abgezogen und am Ende der Schleife immer 1 dazuaddiert. Sie wird deshalb NIE über 100 kommen können. Kommen wir nun zur nächsten Zeile, dem **Delay()** dieser Befehl wartet genau so viele Millisekunden, wie man ihn übergibt, und legt solange den ganzen Programmfluss lahm. Das brauchen wir, da wir ansonsten nix sehen würden, weil Dein Rechner sehr wahrscheinlich so schnell arbeitet, dass du die Console kaum zu Gesicht bekommen würdest. Auf den Rest gehe ich nicht noch mal ein, denn wenn du kein Kopf wie Sieb hast, solltest du noch wissen, was das bedeutet. Und jetzt mal eine kleine Übung zu Auflockerung: Schreib mal wie oben ein Programm, welches 100-mal Hallo Welt in eine Console schreibt, dabei aber vorne angibt, das wievielte mal es schon „Hallo Welt" geschrieben hat. Das Ergebnis sollte folgendermaßen aussehen:



So, jetzt probier mal ein bisschen, denn nur dadurch kannst du die Sache wirklich verstehen. Falls dir überhaupt nichts einfällt, dann denk mal an den Teil, wo du das Zeug mit dem Str() und den Strings verknüpfen über dich ergehen lassen hast. Ließ am besten noch mal nach, und probier, bis du rausbekommst, wie es geht.

NEIN, hier wird jetzt nicht mehr weiter gelesen, denn jetzt kommt der Code, der dir die ganze Arbeit wegnehmen würde ….

So, jetzt hier die Lösung, wie ich das gemacht hab:

```
OpenConsole()
  For X = 0 To 99
    PrintN(Str(x+1)+": Hallo Welt")
  Next
Delay(3000)
CloseConsole()
End
```

Die einzige wirklich interessante Zeile ist wohl das
PrintN(Str(x + 1)+": Hallo Welt")
Das wollen wir noch mal genauer betrachten. Also, du weißt PrintN() möchte einen String haben. Du weißt, das man aus Zahlen mittels *Str()* zu Strings machen kann, und du weißt, dass man Strings verknüpfen (oder auch: aneinanderreihen) kann. Und genau das tun wir hier. Wir machen erst die Variable **X** (ob groß oder klein geschrieben ist egal) Zu einem String, mittels Str(). Dies ist die Variable, die automatisch durch die Schleife immer um 1 erhöht wird. Vorher addieren wir aber immer noch 1 zu der Variablen, bevor wir sie Str() übergeben. Dies ist einfach und logisch dadurch begründet, dass wir bei dieser Schleife bei 0 anfangen zu zählen und bei 99 aufhören. Wir wollen aber bei 1 anfangen und bei 100 aufhören. Deshalb müssen wir immer 1 dazurechen, um ein Ergebnis im Bereich von 1 bis 100 zu bekommen... Wir hätten das auch umgehen können, indem wir **For X = 1 to 100** geschrieben hätten, aber dann wäre ja der Übungseffekt weg ;-) Danach wird dann einfach der String

„: Hallo Welt“ dazu“addiert“. Es steht also einfach zum Beispiel beim ersten Schleifenablauf da:

```
PrintN(”1“+“: Hallo Welt“).
```

Ja, ich weiß: Das ist ziemlich grauer Stoff, aber wir haben noch ein paar solcher Schleifen vor uns. Allerdings werde ich dazu nicht so viele Worte verlieren, da sie an sich gleich arbeiten. Sie haben bloß ein etwas anderes „Layout“. Fachlich besser ausgedrückt, haben sie einen anderen Syntax. Aber egal, das musst du dann erst zu deiner Informatikerausbildung wissen ;-) Als nächste Schleife nehmen wir die Schleife

Repeat-Until

Eine Schleife, die das gleiche macht wie oben, sieht folgendermaßen aus:

```
OpenConsole()
Zaehler.w = 0
Repeat ;Kopf der Schleife
  Zaehler = Zaehler + 1
  PrintN(Str(Zaehler) +": Hallo Welt")
Until Zaehler = 100 ;Fuß der Zeile

Delay(3000)
CloseConsole()
End
```

Du siehst, das dies genau das gleiche ist, wie mit der For-Next Schleife, allerdings wird hier die Bedingung am Fuß der Schleife überprüft wird (in unserem Fall wird die Schleife abgebrochen, wenn die Variable Zaehler den Wert 100 hat.). Hier kann man natürlich auch > 100 (größer als hundert), < 100 (kleiner als hundert), <= 100 (kleinerGleich 100), >= 100 (größerGleichhundert) usw. Dazu kommen wir aber noch mal genauer unter dem Abschnitt „Bedingungen“ weiter unten. Bei dieser Art von Schleife musst du selbst dafür sorgen, dass Sie irgendwo ihr Ende findet, also in unserem Fall muss die Variable Zaehler irgendwann den Wert 100 annehmen, ansonsten wird diese Schleife niemals enden. Dies tun wir oben mit der Zeile Zaehler = Zaehler + 1.
Du kannst statt dem Until auch **ForEver e**insetzen. Dies erzeugt dann eine Schleife, die niemals enden wird: eine Endlosschleife. Dies ist in z.Bsp. in einer WindowEvent-Handling Schleife nützlich. Aber dazu (viel) später! Jetzt gleich die nächste Schleife, damit wir dieses Thema mal endlich abhandeln können und weiter kommen, nämlich die Schleife

While-Wend

```
OpenConsole()
Zaehler.w = 0
While Zaehler < 100;Schleifenkopf
  Zaehler+1 ;Ist das gleiche wie Zaehler = Zaehler +1
  PrintN(Str(Zaehler)+": Hallo Welt")
Wend ;SchleifenEnde

Delay(3000)
CloseConsole()
End
```

Hier wird die *Schleifen-Bedingung* am Kopf der Schleife überprüft. Das While könnte man als „Solange" übersetzen, demnach würde die obige Zeile ins deutsche übersetzt heißen: „Solange Zaehler kleiner als 100 ist". Ist die Bedingung im Kopf also NICHT erfüllt, so wird die Schleife auch nicht weiterhin ausgeführt. Anders ausgedrückt, wird die Schleife solange ausgeführt, wie die Bedingung wahr ist. Es gibt zwar als nächste Schleife noch die ForEach-Next Schleife, allerdings braucht man die nur im Zusammenhang mit LinkedLists, dazu kommen wir aber auch erst später, die Schleife klären wir dann, sobald wir bei diesen LinkedLists sind. Achja, noch eine kleine wichtige Sache dazu: Du kannst innerhalb einer Schleife das Schlüsselwort **Continue schreiben** um automatisch zum Kopf der aktuellen Schleife zu springen, ohne erst bis zum Schluss durchzu-arbeiten. Ausserdem kann man Schleifen ver-schachteln (mehrere Schleifen ineinander schreiben). Es gelten auch hier wieder die gleichen Regeln.

Sprünge

Jetzt kommen wir zu einem weiteren Abschnitt des Tutorials: Den Sprüngen! Wie kannst du das jetzt nun wieder verstehen? Das mit den Sprüngen kannst du (mal wieder) nicht ganz wörtlich nehmen. Also, keine Angst, du kannst dein Fenster ruhig offen lassen, ohne mit der ständigen Angst zu leben, dass du morgen zum MediaMarkt laufen musst und einen neuen PC rausschleppen darfst. (-> Du bist doch nicht blöd ;-) Die Sprünge sind dazu da, um zu einem bestimmten Abschnitt deines Codes zu „springen". Du sagst also dem Programm: Stop, Geh zu dieser Stelle im Code und mach dort weiter. Einen einfachen Sprung führst du mit **GoTo** aus. Den Codeabschnitt zu dem du springen willst, schreibst du einfach hinter das GoTo. Aber grau ist alle Theorie, ein einfacher Sprung sieht folgendermaßen aus:

```
Var1.w = 10

Goto Sprungmarke ;Hier springt das Programm zum Codeabschnitt "Sprungmarke"

;Alles was hier steht wird NICHT ausgeführt.
;Hier könnte ich jetzt jeden Code hinschreiben und
;Er würde nie ausgeführt!
Var1 = Var1 + 100

Sprungmarke: ;Label zu dem gesprungen wird.

;Ab hier wird wieder alles ausgeführt
OpenConsole()
PrintN(Str(Var1))
Delay(3000)
CloseConsole()
End
```

Die Console wird hier natürlich 10 ausgeben, da wir mit dem Sprung das Var1 = Var1 + 100 überspringen. Die Sprungmarken können theoretisch überall im Programm sein. Man könnte mit dem GoTo auch selbst Schleifen basteln, hier ein Beispiel einer sinnlosen Endlosschleife:

```
Kopf:
Goto Kopf
```

Ziemlich aufwendig, nicht ;-) Das Programm besteht quasi nur aus dem einen GoTo! GoTo heißt übrigens übersetzt nichts weiter als „Gehe zu"! Der nächste Punkt, der zu klären währe, ist **GoSub.** GoSub macht prinzipiell das gleiche wie GoTo, allerdings merkt es sich die Stelle, von der es weg-gesprungen ist. Mit dem Schlüsselwort **Return** kann man dann wieder an die Stelle des letzten GoSub zurück-springen. Auch dazu ein kleines Beispiel:

```
Gosub Init ;Geht zum Label Init
;Das hier wird nach dem Return ausgeführt.
  PrintN(Str(Zahl
  Delay(3000)
  CloseConsole()
End

Init:;Hierher springt das Programm zuerst
  Zahl = 100
  OpenConsole()
Return ;Springt zum Gosub zurück
```

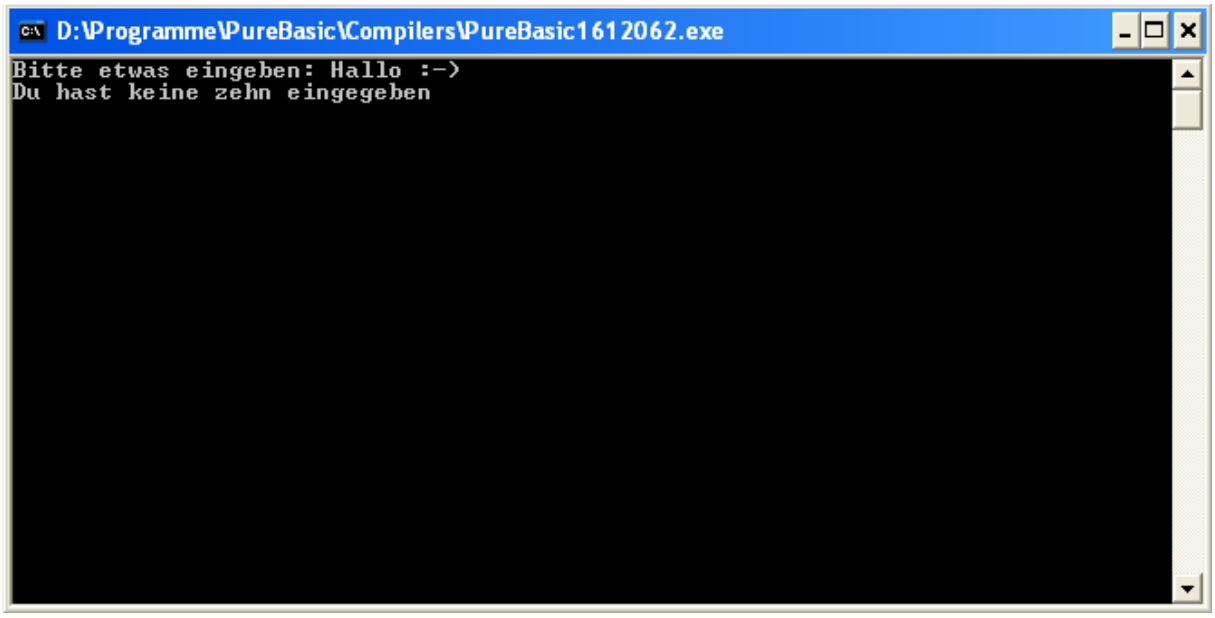
Hier öffnet sich auch wieder eine Console, in der die Zahl 100 steht. Zuerst springt das Programm zu Init und dann kehrt es zum GoSub zurück und führt den Rest des Codes aus. Wenn du aus einem bestimmten Grund irgendwo nach einem GoSub-Sprung noch einen GoTo-Sprung ausführen möchtest, dann musst du vorher das Schlüsselwort **FakeReturn** verwenden. Dies simuliert ein Return OHNE es auszuführen. Ansonsten kann es zu abstürzen führen, also dran denken… Dazu aber jetzt kein Beispiel, denn wenn du wirklich mal soweit bist, dass du solche Befehle brauchst, dann reservier dir schon mal ein Zimmer auf Station 14 … um so was zu verwenden musst du schon kurz vor der Stufe verwirrt stehen ;-) Nein im ernst, allgemein sind Sprünge in der Programmiererszene verrufen. Die meisten meinen, dass es zu unleserlichen Code führt. Und ganz unrecht geben kann ich ihnen nicht. Aber wenn du dir sinnvolle Namen für deine Sprungmarken (Labels) ausdenkst und es mit den Go's nicht gerade übertreibst, können sie sogar zur Leserlichkeit des Codes beitragen. Auch können sie, richtig angewendet, die Geschwindigkeit deines Codes erhöhen. In der Regel kann man sie aber immer umgehen. Diese Sprünge sind genau genommen noch Überbleibsel aus längst vergangenen Zeiten, wo die Prozeduren noch nicht so recht in die Programmier- sprachen Einzug gehalten haben und man deshalb auf Sprünge nicht verzichten konnte. In noch früheren Zeiten musste man in einem Programm am Anfang der Zeile eine Zeilennummer angeben. Meistens wurde dies in 10er-Schritten gemacht. Wenn man ein Goto oder Gosub einsetzen wollte, musste man die Zeilennummer angeben. Was glaubt ihr wie schön der Code aussieht, wenn die Leute, die ein Programm geschrieben haben es irgendwo zwischendrin erweitern mussten?! Sie hatten ja nur einen kleinen Spielraum in den Zehnerschritten. Wenn der Platz zwischen den Zeilennummern nicht reichte, so MUSSTE man mittels einem Gosub irgendwo an das Ende des Programmes springen, um dort zu erweitern. Das hat zu extrem chaotischen und unleserlichen Code geführt, dem so genannten Spagetti- code (neue Rechtschreibung!). Deshalb haben einige sicher noch ihre „Angst" davor, und halten an der prozedualen Schreibweise an und meiden Goto/Gosub wo sie nur können. Also: Setz es ein wenn du's brauchst, aber übertreib's einfach nicht damit! Jetzt wäre eigentlich das wichtigste zu Sprüngen gesagt, also auf zum nächsten Abschnitt…

IF-Blöcke

Kommen wir also zum nächsten Abschnitt dieses Kapitels. Auch hier gilt wieder: Was du hier lernst kannst du auch wieder in anderen Programmiersprachen einsetzen, wie das meist aus diesem Kapitel, dafür ist der „Stoff" ;-) aber zugegebener maßen recht trocken, aber WIRKLICH wichtig. Genug der Vorrede fangen wir an! Diese If-Blöcke (Oder auch If-Statements genannt) sind dazu da, um das Programm auf verschiedene Bedingungen verschieden reagieren zu lassen. Du sagst dem Programm damit: „Wenn DAS hier erfüllt ist, dann tue DAS". Ich zeige euch hier mal einen ganz einfachen IF-Block:

```
OpenConsole()
Print("Bitte etwas eingeben: ") ;Gibt einfach den Text OHNE Return aus
Key.s = Input() ;Lässt den Benutzer etwas eingeben
PrintN("") ;Einfach nur um ein Return(enter) auszugeben

If Key = "10" ;Die Bedingung des IF`s (Kopf)
  PrintN("Du hast eine zehn eingegeben")
Else            ;Wenn Bedingung NICHT erfüllt.
  PrintN("Du hast keine zehn eingegeben")
EndIf           ;Ende des If-Blockes

Delay(3000)
CloseConsole()
End
```

Probiert das Programm mal aus.
Es müsste bei euch in etwa so hier aussehen:



Ich glaube, dass wir dieses Ding hier mal Step-For-Step durchkauen müssen, weil hier jeder Einsteiger nicht mehr dahinter steigen wird… Also, OpenConsole()…Alles klar hoff ich mal so. Print() Auch klar->Wie PrintN, bloß ohne neue Zeile.
Key.s = Input(), ich glaube, das bedarf einer kleinen Erklärung. Also, der Befehl Input() hält das Programm an und wartet, bis der Benutzer etwas in die Console eingegeben hat und Enter gedrückt hat. Dieser Befehl hat(wie die meisten Befehle)

einen *Rückgabewert.* Und dies ist logischerweise der eingegebene Text -> Also immer String. Den speichern wir dann einfach in der String-Variable „Key" und haben ihn :-) Die nächste Zeile lautet: **If** Key = "10" Hier beginnt der eigentliche If-Block. In diesem sehr einfachen Modell wird einfach überprüft, ob die Variable Key den Wert 10 aufweist. Die zehn muss ich in diesem Fall in Anführungszeichen setzen, weil die Zahl in ja in der String-Variable „Key" als Text gespeichert wurde. (es gibt noch die Möglichkeit so eine Zahl aus dem String zu holen mittels dem Befehl **Val**(String), allerdings führt das momentan zu weit, also jetzt noch nur Info!) Wenn die Bedingung wahr (erfüllt; true) ist, so wird der Programmabschnitt darunter ausgeführt. Falls nicht, wird zum **Else** gesprungen(Was soviel wie **sonst** heißt**)** und der Codeabschnitt bis zum **EndIf**, welcher das Ende der Schleife darstellt, ausgeführt. Dieses Else ist natürlich nicht zwingend erforderlich, es wird einfach nur ausgeführt, wenn es vorhanden ist. Wichtig ist allerdings, das du nie vergisst das Endif ans Ende deines Bedingungsblockes zu setzen. Wenn wir aber schon mal beim Else sind, gehe ich auch noch auf die besondere Form des Else´s ein, dem ElseIf. Auch gleich dazu ein Beispiel:

```
OpenConsole()
Print("Bitte etwas eingeben: ") ;Gibt einfach den Text OHNE Return aus
Key.s = Input() ;Lässt den Benutzer etwas eingeben
PrintN("") ;Einfach nur um ein Return(enter) auszugeben
If Key.s = "10" ;Die Bedingung des IF`s (Kopf)
  PrintN("Du hast eine zehn eingegeben")
ElseIf Key = "20"
  PrintN("Du hast eine zwanzig eingegeben.")
ElseIf Key = "30"
  PrintN("Du hast eine dreißig eingegeben.")
Else
  PrintN("Du hast weder eine 10, noch eine 20, noch eine 30 eingegeben)
EndIf           ;Ende des If-Blockes

Delay(3000)
CloseConsole()
End
```

Dieses ElseIf heißt genau übersetzt SonstWenn. So ist es auch zu verstehen. In der Kopfzeile des IF-Blockes wird überprüft, ob die Variable Key den Wert „10" hat. Ist dies der Fall wird der Abschnitt darunter ausgeführt. Wenn NICHT, wird überprüft, ob die Variable den Wert „20" hat. Wenn ja, wird der Codeabschnitt darunter ausgeführt. Wenn nicht, geht`s weiter zur 30. Dies lässt sich theoretisch natürlich endlos erweitern, aber ich glaub das reicht so zum Verständnis. Falls KEINE Bedingung zutrifft, wird der Abschnitt unter ELSE ausgeführt. Die ganze Sache ist natürlich auch wieder optional. Um das bisher mal kurz auf einen Punkt zu bekommen: Mit If-Blöcken kann man den Programmablauf unter bestimmten Bedingungen in verschiedene Wege leiten. Wenn die Bedingung des If-Blocks erfüllt, oder anders ausgedrückt true(dt.:wahr) ist, so wird der entsprechende Abschnitt direkt ausgeführt. Als „wahr" gelten prinzipiell ALLE Werte ungleich 0!! Bei Bedingungskontrollen wird ein wahrer Wert mit **1** ersetzt und ein falscher mit **0.** Also, 1 = true(wahr), 0 = false (falsch). Nehmen wir einmal zur Verdeutlichung das Beispiel von oben. Wir nehmen an, du hast eine 10 eingegeben. Die Variable Key.s beinhaltet nun den String „10". Bei dem If Block wird gleich geprüft, ob die Variable Key den Wert „10" enthält.
If Key = "10"

Wenn man den Inhalt der Variable mal direkt hinschreibt, würde das so aussehen: **If ”10” = ”10”** Jeder Blinde sollte sehen, dass diese Aussage in jeden Fall nur true sein kann. Sie ist also **1!!!** Somit steht dort **If 1**. Das heißt dann also, dass der If-Block wahr ist und er wird somit ausgeführt. Alles klar soweit? Ich hoffe. Wenn nicht ließ bitte noch mal, bis du es gerafft hast. Wenn du Fragen hast -> E-Mail schreiben oder im Forum melden. Probier ruhig mal ein bisschen damit rum. Wenn du soweit bist, klären wir noch den Rest des If-Blockes…

LOGISCHE VERKNÜPFUNGEN

Okay, jetzt wird´s erstmal noch etwas komplizierter. Ich hab das erste Mal eine Zwischenüberschrift gemacht, für unsere Fans der Gliederung ;-) Also, logische Verknüpfungen braucht man in einem IF-Block um mehrere Bedingungen auf einem Mal zu überprüfen. Davon gibt es zumindest prinzipiell erstmal 2 Stück, die PureBasic direkt frisst und das wären **AND** und **OR**. Eigentlich sollte es noch XOR geben, aber das wird nicht so direkt unterstützt oder ich hab irgendwas verpeilt. Aber da ihr das sowieso nur extrem selten brauchen werdet, geh ich erstmal nur auf **AND**(dt.: UND) und **OR**(dt.: ODER) ein. Das kannst du dir wie als eine Verknüpfung zwischen ganz normaler Umgangssprache und Mathematik vorstellen. (Ja, klingt verwirrt, aber irgendwie ist das so ;-). Wir wollen erstmal auf das Umgangssprachliche einzugehen, da die Mathematik erst bei komplexeren logischen Verknüpfungen zum tragen kommt. Umgangssprachlich sagt man doch zum Beispiel: „**WENN** ich heute wieder spät ins Bett gehe **UND** mich wieder vollaufen lasse, **DANN** habe ich morgen garantiert einen Kater“ So in etwa kannst du dir das vorstellen. Dazu mal ein kleines verdeutlichendes Beispiel:

```
OpenConsole()
  PrintN("Bitte nur mit ja oder nein antworten")
  PrintN("") ;Für ein Return
  Print("Hast/Wirst du schon/noch etwas gesoffen/saufen?: ")
  Gesoffen.s = Input() ;Lässt die Antwort eingeben
  PrintN("")
  Print("Gehst du heute zeitig ins Bett?: ")
  Schlafen.s = Input()
  PrintN("")
  If Gesoffen = "ja" And Schlafen = "nein"
    PrintN("Na dannViel Spass morgen! :-)")
  Else
    PrintN("Nochmal Glück gehabt")
  EndIf

  Delay(3000)
CloseConsole()
```

Das einzig interessante an diesem Quelltext sollte die Zeile

```
If Gesoffen = "ja" And Schlafen = "nein"
```

sein. Hier wird erst überprüft, ob die Variable Gesoffen den Inhalt **ja** hat. Falls dies so ist, so ist dieser Teil erfüllt, also **1**. Danach kommt die UND-Verknüpfung. Danach kommt dann Wieder die Überprüfung, ob die Variable Schlafen den Wert „nein“ hat. Nehmen wir an, dass wie im Bild beide Aussagen true sind. Dann steht dort: **If 1 AND 1** Nur wenn der linke UND der rechte Abschnitt von AND true ist, so ist die Bedingung im gesamten Erfüllt. Ist auch nur eine der beiden Aussagen falsch, also 0, so ist die Bedingung NICHT erfüllt. (Es wird dann das else bzw. elseif darunter

ausgeführt). Anders ist das bei **Or.** Hier muss nur eine Seite zwingen erfüllt sein. Es KÖNNEN aber auch beide Seiten erfüllt sein. Genau genommen ist dieses Or ein „UndOder". Also ein **If 1 OR 1** ist genau so wahr, also erfüllt, wie ein **If 1 Or 0** oder einem **If 0 Or 1.** Allerdings ist ein **If 0 Or 0** falsch. Hier mal eine kleine Liste der Möglichkeiten:

If 1 And 1 ->*wahr*
If 1 And 0 ->*falsch*
If 0 And 1 ->*falsch*
If 0 And 0 ->falsch
If 1 Or 1 ->*wahr*
If 1 Or 0 ->*wahr*
If 0 Or 1 ->*wahr*
If 0 Or 0 ->*falsch*

Mehr Kombinationen gibt es logischerweise nicht direkt, wobei ich noch dazusagen muss, dass die 1 als TRUE und die 0 als FALSE anzusehen ist. Wie gesagt, prinzipiell sind alle Werte ungleich 0 gleich true. Damit wäre das Grundlegende gesagt, aber irgendwie währe das doch noch ein wenig zu einfach, findest du nich?! ;-). Da muss es doch noch ne Sache geben, die ich noch nicht erwähnt hab, oder? Natürlich gibt es die :-). Als erstes währe das die Tatsache, dass man theoretisch endlos viele solche AND´s und OR´s hintereinander schreiben kann, zum zweiten ist es die Sache, dass es außer dem Gleichheitszeichen noch andere Rechenzeichen wie größer als, ungleich usw. gibt, zum dritten ist es die Tatsache, dass man die ganzen Bedingungen noch in Klammern setzen kann (auch wieder in endlose Stufen), welche dann wie in der Mathematik stückweise auseinander genommen werden. (Klammern werden zuerst ausgewertet). Na, bist du bedient? Du merkst, dass hier ist ein ziemlich aufwendiger Abschnitt und für Neulinge, garantiert schon jetzt frustrierend und verwirrend zu lesen. Aber keine Sorge, ich bin auch erstmal satt, weil ich jetzt mal auf die schnelle gar nicht weiß, an welcher Ecke ich anfangen sollte…naja, ich schreib morgen weiter, wobei ich euch auch empfehlen würde jetzt ne Pause einzulegen, sonst wird das echt zuviel Stoff mit einem Mal…

Machen wir jetzt also weiter. Also, wie oben erwähnt kann man mehrere solche AND´s und OR´s hintereinander schreiben. Die Auswertung erfolgt dann ganz einfach Stückweise, wie in der Mathematik. Wenn man einen Ausdruck in Klammern schreibt, wird er, auch wie in der Mathematik, VOR den anderen ausge-wertet. Als Rechenbeispiel könnte man zum Beispiel eine einfache Kopfrechenaufgabe nehmen: **4*3*(5+6)**. Hier wird ZUERST die Klammer ausgewertet. Zuerst wird einfach das 5+6 zusammengerechnet. Somit steht dann da 4*3*11. So in etwa kannst du dir die Auswertung bei IF´s auch vorstellen. Jetzt dafür ein einfaches, sinnvolles Beispiel zu finden ist ziemlich schwierig. Ich mache das einfach mal an TRUE und FALSE (wahre oder falsche Aussage) klar. Also dazu mal ein Beispiel:

If (1 *And* 0) *Or* (1 *Or* 1)

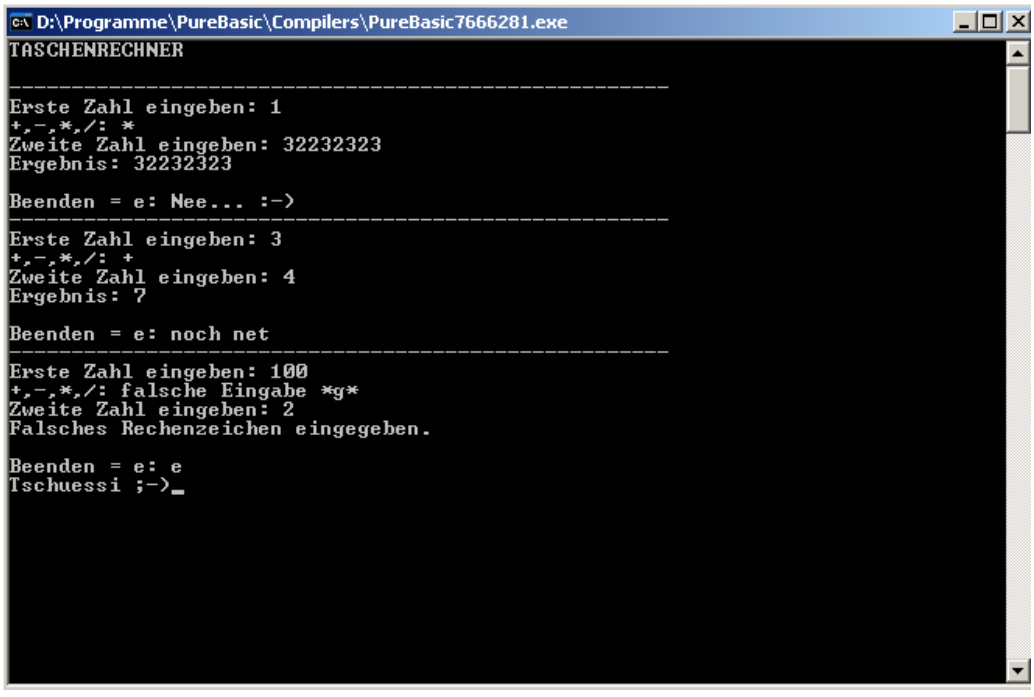
(1 = Wahre Aussage, 0 = falsche Aussage)

Wenn wir jetzt die Schrittfolge durchgehen, und erst die Klammern auswerten, so steht dort:

If 0 Or 1

Da ja 1 AND 0 gleich 0 ist, also FALSCH ist und 1 OR 1 gleich 1 ist, also WAHR ist. Jetzt sieht man, dass die ganze Sache true ist, die Bedingung also erfüllt ist.

Ich gebe euch jetzt mal eine kleine Übung, überlegt mal OHNE es in PB einzutippen, ob diese Aussage wahr oder falsch ist:
If (1 *And* ((0 *Or* 1) *And* (1 *And* 1))) *And* (0 Or (1 *And* 0))
Sieht ganz gut aus, nich? Geh einfach das Ding Stück für Stück und Klammer für Klammer durch. Irgendwann hast du das Ergebnis, hoffe ich. Ich wird jetzt die Sache mal Stück für Stück entschlüsseln, also aufpassen, falls du´s noch nicht raus hast. Wir werten zuerst die obersten Klammerebenen aus, die ich GELB gekennzeichnet habe. Dann sieht die ganze Sache schon so hier aus:
If (1 *And* ((1) *And* (1))) *And* (0 Or (1 *And* 0))
Jetzt werten wir die HELLBLAUEN Klammern aus:
If (1 *And* (1)) *And* (0 Or (0))
Und jetzt kommen wir zur "Krönung", die ROTEN Klammern:
If (1) *And* (0).
Jetzt sieht die Sache doch schon ganz überschaubar aus. Hier steht jetzt einfach noch If 1 And 0, was bedeutet, dass die Gesamtaussage FALSE ist. Ganz easy, nich?! :-P Ich würde sagen, dass wir noch eine kleine Übung im praktischen machen, also Die ganzen einsen und nullen mal mit richtigen Aussagen ersetzen. Am besten wir machen das mal so: Ihr schreibt(NUR(!!) mit den Dingen die wir bisher behandelt haben einen kleinen Taschenrechner. Dieser soll extrem(!!) primtiv ausfallen. Am Anfang soll der Benutzer seine Zahl eingeben. Als zweites sein Rechenzeichen (+,-,*,/). Und als letztes seine zweite Zahl. Zum Schluss soll der Taschenrechner das Ergebnis ausgeben. Danach kommt eine Frage, ob der Benutzer das Programm beenden möchte. Wenn er ein „e" eingibt, soll das Programm beendet werden, andernfalls soll es von vorne starten. Das Programm soll zur ZEIT noch nicht auf Fehleingaben sofort reagieren, schreibt es erstmal so. Bedenkt, dass Input() einen String zurückgibt und ihr mit Val() oder ValF(Für Floats) eine Zahl daraus macht. Das ganze könnte im Bildformat folgendermaßen aussehen:



Kann natürlich Formal bei euch ganz anders (und hübscher ;-) aussehen.
Aber vom Prinzip her sollte es in Etwa so hier aussehen.

Hier mal mein Code für diesen Rechner. Ihr müsst euch nicht 100% an meine Formatierung halten, denn auch hier gilt: Viele Wege führen nach Rom. Was ich habe ist eine Lösung von vielen

```
OpenConsole()
  Quit = 0
  PrintN("TASCHENRECHNER")
  Repeat
    PrintN("")
    PrintN("-----------------------------------------------------")
    ;Eingabeteil
    Print("Erste Zahl eingeben: ")
    Zahl1 = Val(Input()) ;Eingegebenen Text in Zahl umwandeln.
    PrintN("")
    Print("+,-,*,/: ")
    Zeichen.s = Input()
    PrintN("")
    Print("Zweite Zahl eingeben: ")
    Zahl2 = Val(Input())
    PrintN("")
    ;Rechnen
    If Zeichen = "+"
      Ergebnis.f = Zahl1 + Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    ElseIf Zeichen = "-"
      Ergebnis.f = Zahl1 - Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    ElseIf Zeichen = "*"
      Ergebnis.f = Zahl1 * Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    ElseIf Zeichen = "/"
      Ergebnis.f = Zahl1 / Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    Else ;Falsches Rechenzeichen
      PrintN("Falsches Rechenzeichen eingegeben.")
    EndIf
    ;Ende?
    PrintN("")
    Print("Beenden = e: ")
    Zeichen = Input()
    If Zeichen = "e"
      Quit = 1
    EndIf
  Until Quit = 1
PrintN("")
Print("Tschuessi ;-)")
Delay(3000)
CloseConsole()
End
```

Das hier ist der erste mal etwas längere Code. Das erste zumindest bedingt brauchfähige Programm, das du geschrieben hast, cool wa?! 8-) Ich hoffe, hier versteht das so einigermaßen. Wenn nicht, kopiert euch den Text in den PureBasic-Editor und probiert ein wenig, bis ihr es versteht. Dieser Code hier ist aber weit davon entfernt gut zu sein. Die erste Schwachstelle ist, dass er nicht sofort reagiert, wenn der Benutzer eine falsche Zahl eingibt, sondern erst zum Schluss. Es wäre doch viel günstiger, wenn er gleich einen Fehler melden würde. Und das wollen wir zuerst mal ausbügeln. Wir nehmen uns mal diesen Abschnitt hier:

```
PrintN("")
Print("+,-,*,/: ")
Zeichen.s = Input()
PrintN("")
```

Wenn der Benutzer irgendwas außer diesen Rechenzeichen eingibt, soll eine Fehlermeldung erscheinen und der Benutzer soll sofort ein anderes eingeben können. Wir realisieren das am Besten mit einer Repeat-Until Schleife. Also Wiederhole-Bis. Die Schleife soll wiederholt werden, bis entweder ein + ODER ein – ODER ein * ODER ein / eingegeben wurde. Ihr seht, wir haben wieder eine Bedingung mit mehreren OR´s :-). Bloß schreiben wir dies hinter das Until anstatt an ein IF. So sieht der Abschnitt dann ausgeschrieben aus:

```
Repeat
    PrintN("")
    Print("+,-,*,/: ")
    Zeichen.s = Input()
Until Zeichen = "+" Or Zeichen = "-" Or Zeichen = "*" Or Zeichen = "/"
```

Wir fügen ihn der Vollständigkeit halber in den Source des Rechners ein:

```
OpenConsole()
  Quit = 0
  PrintN("TASCHENRECHNER")
  Repeat
    PrintN("")
    PrintN("--------------------------------------------------")
    ;Eingabeteil
    Print("Erste Zahl eingeben: ")
    Zahl1.f = ValF(Input()) ;Eingegebenen Text in Zahl umwandeln.
    Repeat
      PrintN("")
      Print("+,-,*,/: ")
      Zeichen.s = Input()
    Until Zeichen = "+" Or Zeichen = "-" Or Zeichen = "*" Or Zeichen = "/"
    PrintN("")
    Print("Zweite Zahl eingeben: ")
    Zahl2.F = ValF(Input())
    PrintN("")
    ;Rechnen
    If Zeichen = "+"
      Ergebnis.f = Zahl1 + Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    ElseIf Zeichen = "-"
      Ergebnis.f = Zahl1 - Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    ElseIf Zeichen = "*"
      Ergebnis.f = Zahl1 * Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    ElseIf Zeichen = "/"
      Ergebnis.f = Zahl1 / Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    Else ;Falsches Rechenzeichen
      PrintN("Falsches Rechenzeichen eingegeben.")
    EndIf
    ;Ende?
    PrintN("")
    Print("Beenden = e: ")
    Zeichen = Input()
    If Zeichen = "e"
```

```
      Quit = 1
    EndIf
  Until Quit = 1
PrintN("")
Print("Tschuessi ;-)")
Delay(3000)
CloseConsole()
End
```

Jetzt funktioniert er doch eigentlich schon ganz gut, nicht? Aber irgendwie sieht doch der Block mit diesen endlosen ElseIf´s Ziemlich unübersichtlich aus, oder?
Wenn dein Programm unübersichtlich geworden ist, dann kannst du eigentlich gleich einpacken. Um das ein bisschen zu umgehen, lassen wir mal diesen If-Block verschwinden und machen einen Select-Block daraus. Dieses Select arbeitet an sich so ähnlich wie das mit den IF´s allerdings ist es für unseren Fall besser. Wir schreiben dort einfach **Select** *Wert*. Das lässt sich ziemlich schlecht erklären, also mach ich gleich mal ein Beispiel und nehme dazu diesen IF-Block von unserem Taschenrechner:

```
Select Zeichen ;Die Variable Zeichen wird selected(gewählt) (Kopf)
    Case "+" ;Falls Zeichen = "+"
      Ergebnis.F = Zahl1 + Zahl2
      PrintF("Ergebnis: "+StrF(Ergebnis))
    Case "-" ;Falls Zeichen = "-"
      Ergebnis.F = Zahl1 - Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    Case "*" ;Falls Zeichen = "*"
      Ergebnis.F = Zahl1 * Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    Case "/" ;Falls Zeichen = "/"
      ErgebnisF = Zahl1 / Zahl2
      PrintN("Ergebnis: "+StrF(Ergebnis))
    Default ;Wie Else, also wenn KEINE überprüfung zutrifft
      PrintN("Falsches Rechenzeichen eingegeben.")
EndSelect ;Fuß-Ende des Select-EndSelect
```

Select ist also der Kopf der ganzen Sache. Hinter das Select kann man dann die Variable schreiben, die man überprüfen will. Danach kann man mit **Case** *Wert* überprüfen, ob die der Wert dem Wert des Select entspricht. Default ist hier wie das Else im If-Block. Wenn keine der Bedingungen zutrifft, so wird Default ausgeführt, falls vorhanden. EndSelect kennzeichnet dann das Ende des Select-EndSelect Blockes. Auch das implementieren wir gleich in unseren Hauptcode:

```
OpenConsole()
  Quit = 0
  PrintN("TASCHENRECHNER")
  Repeat
    PrintN("")
    PrintN("-----------------------------------------------------")
    ;Eingabeteil
    Print("Erste Zahl eingeben: ")
    Zahl1.f = ValF(Input()) ;Eingegebenen Text in Zahl umwandeln.
    Repeat
      PrintN("")
      Print("+,-,*,/: ")
      Zeichen.s = Input()
    Until Zeichen = "+" Or Zeichen = "-" Or Zeichen = "*" Or Zeichen = "/"
```

```
    PrintN("")
    Print("Zweite Zahl eingeben: ")
    Zahl2.F = ValF(Input())
    PrintN("")
    ;Rechnen
    Select Zeichen ;Die Variable Zeichen wird selected(gewählt) (Kopf)
      Case "+" ;Falls Zeichen = "+"
        Ergebnis.f = Zahl1 + Zahl2
        PrintN("Ergebnis: "+StrF(Ergebnis))
     Case "-" ;Falls Zeichen = "-"
       Ergebnis.f = Zahl1 - Zahl2
       PrintN("Ergebnis: "+StrF(Ergebnis))
      Case "*" ;Falls Zeichen = "*"
        Ergebnis.f = Zahl1 * Zahl2
        PrintN("Ergebnis: "+StrF(Ergebnis))
      Case "/" ;Falls Zeichen = "/"
        Ergebnis.f = Zahl1 / Zahl2
        PrintN("Ergebnis: "+StrF(Ergebnis))
      Default ;Wie Else, also wenn KEINE überprüfung zutrifft
        PrintN("Falsches Rechenzeichen eingegeben.")
    EndSelect ;Fuß-Ende des Slect-EndSelect
    ;Ende?
    PrintN("")
    Print("Beenden = e: ")
    Zeichen = Input()
    If Zeichen = "e"
      Quit = 1
    EndIf
  Until Quit = 1
PrintN("")
Print("Tschuessi ;-)")
Delay(3000)
CloseConsole()
End
```

Ich würde sagen, dass wir jetzt genug an dem Beispiel rumgebastelt haben. Also verdaut alles erstmal schön und wenn ihr alles geschnallt habt, dann geht´s weiter in den nächsten Abschnitt!

Prozeduren

Und nun der letzte Abschnitt dieses Kapitels, die Prozeduren. Prozeduren kannst du dir wie kleine Programme im Programm vorstellen. Man kann damit jederzeit das Programm „Unterbrechen" und erstmal eine Prozedur aufrufen. Das ist so ähnlich wie die Sache mit dem **Gosub**, bloß eben das es wesentlich übersichtlicher ist. PureBasic zählt zu den prozedualen Programmiersprachen, eben hauptsächlich aufgrund dieser Funktionalität. Diese Unterprogramme sind wie ISOLIERT von dem Rest des Programms. D.h.: Ohne Zutun kommt eine Prozedur NICHT an die Variablen vom Hauptprogramm. Man kann in diesen Prozeduren also prinzipiell noch mal Variablen, die es schon im Hauptprogramm gibt noch einmal sogar mit einem anderen Typ deklarieren, ohne die Variablen im Hauptprogramm zu beeinflussen. Doch genug der Vorrede. Hier erstmal ein Beispiel, wie man eine Prozedure kenntlich macht:

```
Procedure Hallo() ;Erstellt eine Prozedur mit dem Namen Hallo
  ;Hier kann das Programm eingegeben werden.
EndProcedure ;Ende der Prozedur
```

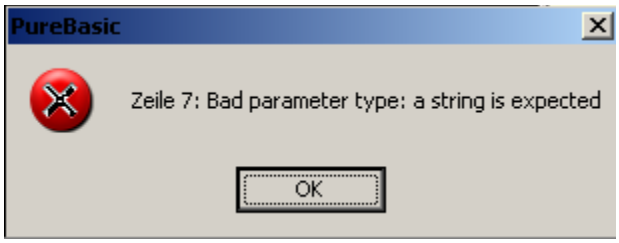
Das Schlüsselwort **Procedure** kennzeichnet den Anfang einer Prozedur. Danach kommt der Name der Prozedur und zwei Klammern. Danach kann man den Code der Prozedur eingeben. Zum Schluss noch ein **EndProcedure** um die Prozedur abzuschließen. Diese Prozeduren führen sich niemals von alleine aus. Man muss sie aufrufen, damit sie ihre Funktionalität erfüllen. Beim Übersetzen des Quelltextes in Maschinencode wird beim Durchgehen einfach nur die Prozedur deklariert. Achtung! Es darf keine Prozeduren gleichen Namens geben. Diese Prozeduren sind an sich nichts weiter als wie die Funktionen die wir bisher behandelt haben, also zum Beispiel Print(*text.s*) oder OpenConsole(), nur das du sie eben selbst schreibst. Ich bring euch jetzt erstmal ein kleines Anwendungsbeispiel:

```
Procedure UnserPrintN(String.s)
  PrintN(String)
EndProcedure

OpenConsole()
UnserPrintN("Ich bin ein String(-tanga ;-)")
Delay(3000)
CloseConsole()
End
```

Hier deklarieren wir zuerst eine Prozedur mit dem Namen: UnserPrintN.
Diese Procedure hat einen *Parameter* (Übergabewert). In unserem Fall ist dieser vom Typ String, abgesehen davon, dass sie zusätzlich noch so heißt. Damit gibt es in dem Unterprogramm schon mal die Variable String.s. Diese bekommt automatisch den Wert zugewiesen, den ich beim Aufrufen der Prozedur übergebe, in unserem Fall also: *Ich bin ein String(-tanga ;-).* Wir rufen in der Prozedur einfach nur den Befehl Print() auf, und übergeben ihn nur noch mal den uns übergebenen Wert. Den Sinn dieser Aktion will ich euch jetzt nicht unbedingt näher bringen ;-). So eine Prozedur kann theoretisch endlos viele Übergabewerte haben, sie müssen bloß alle mit Komma voneinander getrennt sein. Allerdings stellt sich doch noch ein Problem: Wenn diese Prozeduren vom restlichen Programm abgeschnitten sind und sich die Variablen vom Hauptprogramm nicht direkt benutzen lassen, was mache ich, wenn ich sie doch unbedingt brauche? Dafür gibt es 2 Möglichkeiten: Die Erste währe sämtliche Variablen mittels Parameter zu übergeben. Das das manchmal ziemlich nerven kann, könnt ihr euch sicher vorstellen. Und die Zweite Möglichkeit ist, die Variablen im Hauptprogramm als **Global** zu deklarieren. Damit sind sie in sämtlichen Prozeduren verfügbar. Sowas sieht dann so aus:

```
Global IchBinEineGlobaleVariable.s
IchBinEineGlobaleVariable = "Ich bin in jeder Prozedur vorhanden."
IchBinKeineGlobaleVariable.s = "Ich bin nicht in Prozeduren vorhanden."
Procedure UnserPrintN()
  PrintN(IchBinEineGlobaleVariable)
  PrintN(IchBinKeineGlobaleVariable)
EndProcedure
OpenConsole()
  UnserPrintN()
  Delay(3000)
CloseConsole()
End
```

Führt den Code mal aus!



Na Toll! Übersetzt heißt das soviel wie: Falscher Parameter-Typ: Es wurde ein String erwartet. Wenn du jetzt mal darüber nachdenkst wird dir klar, warum das so ist. Überleg mal, PrintN() Erwartet einen String als Übergabewert. Aber bei dem zweiten PrintN() Übergeben wir die Variable mit dem Namen IchBinKeineGlobaleVariable. Jetzt denk mal nach: Diese Variable ist, wie der Name schon sagt, nicht global. In der Prozedur gibt es sie nicht, auch wenn sie im Hauptprogramm sehr wohl vorhanden ist. Variable, die nicht deklariert wurden, aber benutzt werden, sind von Haus aus immer automatisch LONG-Variablen. Somit übergeben wir dem Befehl eine LONG-Variable anstatt eines String, was zu dieser hübschen Fehlermeldung führt. Ganz klar, nich?! ;-) Wenn ihr mal ein Semikolon vor dem zweiten PrintN() setzt, dann kommt keine Error-message und der Globale String wird ausgegeben. Ich hoffe das hat dir die Sache mit dem Global etwas näher gebracht. Doch, jetzt haben wir ein Problem gelöst, aber das nächste folgt zu gleich: Was ist, wenn man in einem Unterprogramm (Prozedur) eine Variable deklariert, aber sie im Hauptprogramm benötigt? Wenn ich ja in einer Prozedur eine Variable deklariere, dann ist sie automatisch lokal und wird nach Beenden der Prozedur automatisch gelöscht, um die verwendeten Ressourcen wieder freizugeben. Dazu gibt es auch wieder 2 Möglichkeiten. Die eine ist so ähnlich wie das mit dem Global. Wenn ich in einer Prozedur eine Variable deklariere und möchte, dass sie auch außerhalb der Prozedur verfügbar ist, so muss ich sie *sharen* (Teilen). Dies geht auch ganz einfach mittels dem Schlüsselwort Share. Auch dazu mal ein kleines Beispiel:

```
Procedure ShareMe()
  Shared Variable1
  Variable1 = 10
  Shared Variable2
  Variable2 = 20
EndProcedure


OpenConsole()
  PrintN(Str(Variable1))
  PrintN(Str(Variable2))
  ShareMe() ;Prozedur aufrufen
  PrintN(Str(Variable1))
  PrintN(Str(Variable2))
  Delay(3000)
CloseConsole()
End
```

Es öffnet sich eine Console, in der zuerst 2 Nullen und dann eine 10 und eine 20 ausgegeben werden. Das lässt sich auch ganz einfach erklären. Bei den ersten beiden PrintN()´s wurde die Prozedur ShareMe() noch nicht ausgeführt. Deshalb wurden die zwei Variablen: Variable1 und Variable2 noch nicht deklariert. Sie sind also noch 0! Danach wird die Prozedur aufgerufen, die nichts weiter macht, als diese 2 Variablen zu Sharen und ihnen einen Wert zuzuweisen. Damit sind sie auch im

Hauptprogramm bekannt und können verwendet werden. Damit hätten wir Möglichkeit nummer 1 geklärt. Doch, wie gesagt, es gibt auch hier wieder 2 Möglichkeiten. Diese Möglichkeit kennst du genau genommen schon. Denk mal zurück an den Befehl Input(). Der Befehl hat den eingegebenen Text zurückgeliefert. Genau so geht das auch mit selbstgeschriebenen Prozeduren. Diese möglichkeit hat eben bloß den Nachteil, dass sie nur eine Variable zurückliefern kann. Das Schlüsselwort dafür heißt **ProcedureReturn** dieses Schlüsselwort beendet die Prozedur. Wenn ein Wert dahintersteht gibt er diesen dann automatisch mit zurück. Allerdings muss man dabei VORHER angeben welchen Typ von Variable eine Prozedur zurückgibt. Dies ist standardmäßig wie bei Variablen immer LONG, soll es aber doch lieber ein anderer sein, so musst du das einfach hinter das Schlüsselwort Procedure schreiben. Das sieht dann so hier aus:

```
Procedure.l Rechnung(Zahl1.l, Zahl2.l)
  Ergebnis = Zahl1+Zahl2
  ProcedureReturn Ergebnis
EndProcedure
OpenConsole()
  PrintN(Str(Rechnung(20,77)))
  Delay(3000)
CloseConsole()
End
```

Hier wird die Prozedur Rechnung deklariert. Diese Prozedur ist vom Typ LONG. Sie hat zwei Parameter eben die zwei Variablen Zahl1 und Zahl2, welche auch vom Typ LONG sind. In der Prozedur werden die Zahlen dann einfach zusammengerechnet und das Ergebnis dieser Rechnung zurückgegeben. Der Rückgabewert wird dann bei dem PrintN() einfach mit dem Befehl Str() in einen String umgewandelt und ausgegeben. Ich hoffe du hast dass jetzt verstanden, denn jetzt währe eigentlich das wichtigste zu Prozeduren gesagt. Ich würde sagen, wir schreiben zum Abschluss unseren Lieblings"taschen"rechner von oben mal so um, dass die Rechnung in einer Prozedur ausgeführt wird, die dann das Ergebnis zurückliefert, was wir dann einfach nur noch ausgeben. Probier ruhig selbst mal, bevor du zur nächsten Seite Blätterst. Wenn du es nämlich selbst bringst, hast die Sache mit den Prozeduren sicher verstanden. Also, hier meine Variante:

```
Declare.f Rechnen(Zahl1.f, Zahl2.f, Rechenzeichen.s)
OpenConsole()
  Quit = 0
  PrintN("TASCHENRECHNER")
  Repeat
    PrintN("")
    PrintN("-------------------------------------------------")
    ;Eingabeteil
    Print("Erste Zahl eingeben: ")
    Zahl1.f = ValF(Input()) ;Eingegebenen Text in Zahl umwandeln.
    Repeat
      PrintN("")
      Print("+,-,*,/: ")
      Zeichen.s = Input()
    Until Zeichen = "+" Or Zeichen = "-" Or Zeichen = "*" Or Zeichen = "/"
    PrintN("")
    Print("Zweite Zahl eingeben: ")
    Zahl2.F = ValF(Input())
```

```
    PrintN("")
    ;Rechnen
    PrintN(StrF(Rechnen(Zahl1, Zahl2, Zeichen)))
    ;Ende?
    PrintN("")
    Print("Beenden = e: ")
    Zeichen = Input()
    If Zeichen = "e"
      Quit = 1
    EndIf
  Until Quit = 1
PrintN("")
Print("Tschuessi ;-)")
Delay(3000)
CloseConsole()
End

Procedure.f Rechnen(Zahl1.f, Zahl2.f, Rechenzeichen.s)
    Select Rechenzeichen ;Die Variable Zeichen wird selected(gewählt)
      Case "+" ;Falls Zeichen = "+"
        Ergebnis.f = Zahl1 + Zahl2
      Case "-" ;Falls Zeichen = "-"
       Ergebnis.f = Zahl1 - Zahl2
      Case "*" ;Falls Zeichen = "*"
        Ergebnis.f = Zahl1 * Zahl2
      Case "/" ;Falls Zeichen = "/"
        Ergebnis.f = Zahl1 / Zahl2
    EndSelect ;Fuß-Ende des Select-EndSelect
  ProcedureReturn Ergebnis
EndProcedure
```

Ich habe in diesem Beispiel gleich das einige neue, was es noch zu sagen gibt zu Prozeduren, verwendet, und zwar das Schlüsselwort Declare. Das brauche ich dazu, dem Programm klar zu machen, dass es diese Prozedur gibt, aber woanders steht. Wenn ich das nicht machen würde, gäbe es in unserem Fall eine Fehlermeldung, dass es die Prozedur eben nicht geben würde. Dies ist nur nötig, wenn man die Prozedur irgendwo an das Ende des Quelltextes verlagert, da das Programm ansonsten nicht weiß, das es sie gibt. Das währe nicht nötig gewesen, wenn ich die Prozedur an den Anfang gesetzt hätte. So, damit wären wir am Ende des zweiten Kapitels. Ich hoffe er hat euch einige neue Erkenntnisse gebracht und euch nicht zu sehr verwirrt und gelangweilt. Es war sehr viel Theorie in diesem Kapitel, aber ohne diese Grundlagen können wir nicht in die Programmierung einsteigen. Aber das krasseste ist jetzt erledigt. Im nächsten Kapitel können wir dann endlich zu interessanteren Themen schreiten, hoffe ich :-)

Kapitel III
(Window-Handling)

Window's

Nach dem letzten, etwas sehr trockenem Kapitel, können wir jetzt zu einem doch schon interessanterem Teil der Programmierung kommen: Den Fenstern, woher auch der Name unser aller Lieblingsbetriebssystems Windows herkommt. Warum diese Dinger ausgerechnet Fenster heißen, weiß der Teufel. So recht was durchsehen kann ich da ehrlich gesagt nicht… Aber was soll's, ich muss ja auch nicht alles verstehen und manche Dinge lassen sich auch nicht irgendwie logisch erklären, das ist bei M$ einfach so. Aber das soll nicht Thema dieses Kapitels sein, wir wollen jetzt mal lieber lernen, wie man diese undurchsichtigen Fenster benutzt. Naja, wenigstens sind die Dinger nicht so schwarz wie unsere Consolen ;-) Also, genug der Vorrede. Kommen wir nun zum Wichtigen: Wie mache ich so ein Fenster auf? Ja, dafür gibt's wie bei der Console einen kleinen Befehl, nämlich **OpenWindow()** sieht irgendwie fast so aus, wie mit der Console…aber ganz so einfach ist das diesmal nicht ;-) Dieser Befehl braucht nämlich so einige Parameter, hier mal der Funktionsaufruf mit Parameterangabe:

```
OpenWindow(#Window, x, y, InnereBreite, InnereHöhe, Flags, Titel$ [,
ParentWindowID])
```

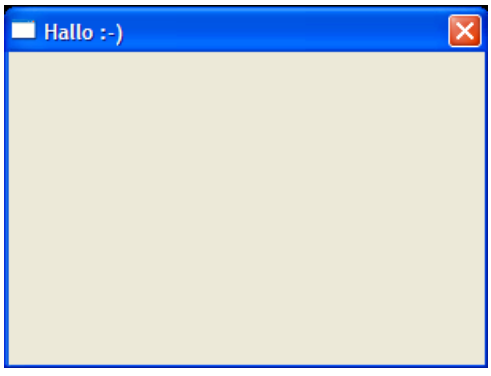
Das Ding gehen wir jetzt gleich mal Parameter für Parameter durch. Der erste Parameter ist eine Zahl, was durch das vorangehende # gekennzeichnet ist. Damit übergibst du dem Fenster eine konstante Nummer. Diese brauchst du, damit du später die Fenster voneinander unterscheiden kannst. Jedes Fenster braucht eine EIGENE Nummer, da du ja mehr als nur 1 Fenster öffnen kannst. x gibt an, an welcher x-Position das Fenster geöffnet werden soll. Dies gibt an, an welchem horizontalen Punkt die linke, obere Ecke des Fensters sein soll. y gibt an, an welcher vertikalen Stelle sich die linke, obere Ecke befinden soll. InnereBreite gibt an, wie groß das Fenster in x-Richtung sein soll. InnereHöhe gibt an, wie groß das Fenster in y-Richtung sein soll. Flags sind spezielle Übergabewerte an das Fenster. Das kann eine oder mehrere der folgenden Konstanten sein:

```
#PB_Window_SystemMenu     : Schaltet das System-Menü in der Fenster-
Titelzeile ein.

#PB_Window_MinimizeGadget ;Fügt das Minimieren-Gadget der Fenster-Titelzeile hinzu.
#PB_Window_System wird automatisch hinzugefügt.

#PB_Window_MaximizeGadget ;Fügt das Maximieren-Gadget der Fenster-
Titelzeile hinzu. #PB_Window_System wird automatisch hinzugefügt.

#PB_Window_SizeGadget     ;Fügt das Größenänderungs-Gadget zum Fenster hinzu.

#PB_Window_Invisible      ;Erstellt ein Fenster, zeigt es aber nicht an. Wird nicht
unter AmigaOS unterstützt.

#PB_Window_TitleBar       ; Erstellt ein Fenster mit einer Titelzeile.

#PB_Window_BorderLess     ;Erstellt ein Fenster ohne jegliche Ränder.
```

```
#PB_Window_ScreenCentered ;Zentriert das Fenster in der Mitte des Bildschirms. Die
Parameter x,y werden ignoriert.

#PB_Window_WindowCentered ;Zentriert das Fenster in der Mitte des
übergeordneten Fenster ("parent window"). Die Parameter x,y werden ignoriert.
```

Wenn du mehrere dieser Konstanten verwendest, musst du sie mit einem „ /" voneinander trennen. Diese Liste ist auch in der Hilfe einsehbar. Titel$ sollte sich von selbst erklären, hier übergibt man den Namen an das Fenster, der in der Titelzeile erscheint. Der nächste Parameter ist allerdings wieder etwas anders, als die anderen. Dieser Parameter ist *optional*. Hier kann ein „Elternfenster" angegeben werden. Aber das lassen wir jetzt erstmal außen vor. So, das ist jetzt zwar schön, dass wir Fenster erstellen können, aber wie wollen wir jetzt merken, dass der Benutzer zum Beispiel auf das Schließkreuz oder zum Beispiel auf einen Button geklickt hat? Nun, das ist ganz einfach (natürlich ;-)! Wir benötigen eine **Event-Loop.** Und schon wieder so ein verwirrender Begriff :-/. Aber wie immer ganz einfach: Was eine Schleife (Loop) ist, wisst ihr ja schon, und ein Event ist ein Ereignis. Kurz gesagt, wir brauchen eine Ereignisschleife. Ein Ereignis entsteht zum Beispiel, wenn der Benutzer auf eine Schaltfläche, wie eben zum Beispiel auf das Schließkreuz, oder einen Knopf(Button) klickt. Auch sind zum Beispiel Ereignisse, wenn das Fenster verschoben wird, oder gezeichnet wird. Das klären wir noch.Wir schreiben jetzt einfach erstmal eine kleine Schleife, die demonstriert, wie so etwas aussieht:

```
OpenWindow(0,0,0,300,200,#PB_WINDOW_SCREENCENTERED |
      #PB_WINDOW_SYSTEMMENU,"Hallo :-)") ;wir Öffnen das Fenster
Repeat ;unsere Schleife
  ;Wir warten auf ein Ereignis des Fensters
  Event = WaitWindowEvent()
  ;Okay, irgendwas ist geschehen auf dem Fenster
  Select Event ;Wir schauen, was passiert ist
    Case #PB_Event_CloseWindow ;Falls das Fenster geschlossen werden soll
      ;Nachfrage, ob das Fenster wirklich geschlossen werden soll
      aw = MessageRequester("Frage","Soll das Fenster geschlossen
      werden?",#PB_MessageRequester_YesNo)
      If aw = 6 ;6 = Ja-Knopf
       End
      EndIf
  EndSelect
ForEver
```



Sieht es nicht Originell aus?! ;-) Wenn du auf das Schließkreuz klickst, dann kommt eine MessageBox(MessageRequester) in der du gefragt wirst, ob du das Fenster schließen möchtest. Klickst du auf „ja" wird das Programm beendet (und damit auch

automatisch alle offenen Fenster geschlossen). Aber wir gehen den Code mal Schrittweise durch, damit ihn alle verstehen. Also, als erstes öffnen wir ein Fenster mittels *OpenWindow()*. Dieses bekommt die Nummer 0 (Dies ist aber jetzt ohne Bedeutung, da wir ohnehin nur ein Fenster geöffnet haben). Dieses WÜRDE an der Position 0,0, also links oben in der Ecke des Bildschirms angezeigt. Es hat eine Ausdehnung von 300*200 Pixel. Die Parameter für dieses Fenster sind *#PB_WINDOW_ScreenCentered* und *#PB_WINDOW_SystemMenu.* #PB_WINDOW_ScreenCentered gibt an, dass das Fenster in der Mitte des Bildschirms angezeigt wird. *#PB_Window_SystemMenu* gibt an, dass das Fenster ein Systemmenü bekommt (Das Ding links oben in der Ecke).Als nächstes geben wir nur noch als Titel „Hallo :-)" an. So, nun, da wir unser Fenster erstmal offen haben, starten wir unsere EventLoop. Wir nehmen einfach mal eine Repeat-ForEver Schleife, da das Programm ja nicht von alleine enden soll. In der Schleife wird zuerst der Befehl *WaitWindowEvent* aufgerufen. Dieser Befehl WARTET(!!) bis ein Ereignis auf irgend einem Fenster diesen Programmes auftritt. Ist dies der Fall gibt es das ausgelöste Ereignis als Rückgabewert an das Programm zurück. Wir speichern das Event einfach in die sinnvoll benannte Variable „Event". Danach verarbeiten wir das Ereignis in einem Select-Block. In diesem Fall ist das natürlich weniger sinnvoll, da wir sowieso nur ein Event abfangen, nämlich das CloseEvent. Dies hat die Bezeichnung *#PB_EVENT_CloseWindow*. Aber in größeren Programmen wollen wir ja meist auf mehrere Ereignisse reagieren können, wobei sich der Select-Block anbietet, zwecks der Übersichtlichkeit. Hier eine Liste aus der Hilfe mit den anderen WindowEvents, die auftreten können. (Ich werde sie aber jetzt hier nicht alle erläutern, alles nacheinander :-)

```
  #PB_Event_Menu        ;ein Menü wurde ausgewählt
  #PB_Event_Gadget      ;ein Gadget wurde gedrückt
  #PB_Event_CloseWindow ;das Schließgadget vom Fenster wurde gedrückt
  #PB_Event_Repaint     ;der Fensterinhalt wurde zerstört und muss neu gezeichnet
werden (nützlich für 2D Grafik-Operationen)
  #PB_Event_MoveWindow  ;das Fenster wurde verschoben
```

Statt dem WaitWindowEvent() kannst du auch einfach *WindowEvent()* verwenden. An sich verrichtet dieser Befehl fast das gleiche, bloß dass er das Programm nicht anhält, sondern einfach weitermacht, auch wenn kein Ereignis vorhanden ist. Du musst dich dann selbst darum kümmern, dass dein Programm nicht zuviel CPU-Leistung beansprucht. Ich denke der Rest von dem Code erklärt sich selbst und ist eigentlich alter Stoff. Noch eine kleine Anmerkung: Wenn wir ein Fenster schließen möchten verwenden wir im Normalfalle *CloseWindow().* Dies ist aber in UNSEREM Fall unnötig, da *End* auch sämtliche Fenster schließt, bevor das Programm dann auch wirklich beendet wird. So, jetzt haben wir es geschafft, ein Fenster zu öffnen, cool nicht?! Naja, gut, es ist vielleicht cool, aber bringen tut uns das auch noch nichts… Also machen wir fröhlich weiter ;-) Kommen wir also zum nächsten Abschnitt, den Gadgets.

Gadgets

Den Begriff hast du vorhin schon mal gelesen, als du die Konstante *#PB_Event_Gadget* gelesen hast. Also, was sind diese Dinger?! Zuerst einmal sei gesagt, dass Gadgets im Normalfalle auch Controls genannt werden. Der Begriff

Gadget ist eigendlich nur in der PB-Szene bekannt. Gadget ist eigentlich ein Sammelbegriff für alle Steuerelemente, die du in PB benutzen kannst. Steuerelemente sind zum Beispiel die Knöpfe, auf die du klicken kannst, oder Optionsfelder, aus denen du auswählen kannst. Falls du so ein Gadget erstellst und darauf eine Aktion durchgeführt wird, so gibt dir *WaitWindowEvent()* als Ergebnis den Wert *#PB_Event_Gadget* zurück. Doch wie bemerke ich dann, wenn ich mehrere Knöpfe (Buttons) auf dem Fenster habe, welcher gedrückt wurde? Dafür gibt es den Befehl *EventGadgetID().* Dieser Befehl gibt zurück, auf welchem Gadget ein Ereignis stattgefunden hat. Jetzt haben wir wieder viel gelabert, es wird Zeit für ein Beispiel:

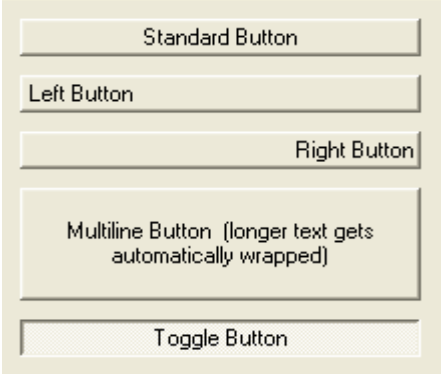
```
;Öffne ein Fenster...
If OpenWindow(0,0,0,100,100,#PB_Window_MinimizeGadget |
#PB_Window_SystemMenu | #PB_Window_ScreenCentered,"Knopf")
  ;Das Fenster konnte erstellt werden
  ;Benutze Fenster 0
  UseWindow(0)
  ;Erstelle eine Liste für das aktuell benutzte Fenster
  If CreateGadgetList(WindowID())
    ;Die GadgetListe konnte erstellt werden :-)
    ;Erstelle ein ButtonGadget(diese Knöpfchen)
    ButtonGadget(0,0,0,100,20,"Knopf 1",#PB_Button_Left)
    ;Und noch einen ^^
    ButtonGadget(1,0,80,100,20,"Knopf 2",#PB_Button_Right)
  EndIf
;Okay alles klar soweit :-)
;Unsere Event-Loop
Repeat
  ;Auf ein Event warten...
  Event = WaitWindowEvent()
  Select Event  ;Das Event herausfinden
    Case #PB_Event_CloseWindow ;Fenster soll geschlossen werden...
      End ;Cya ^^
    Case #PB_EventGadget       ;Es ist ein Ereignis auf einem Knopf
aufgetreten...
      Select EventGadgetID() ;Welches wurde gedrückt?
      Case 0 ;Gadget 0 wurde gedrückt...
        MessageRequester("Hallo","Du hast Knopf 1 gedrückt",0)
      Case 1 ;Gadget 1 wurde gedrückt
        MessageRequester("Hallo","Du hast Knopf 2 gedrückt",0)
      EndSelect
  EndSelect
ForEver
EndIf
```

So.. wie ihr seht, hab ich den Code schon gleich gut durchkommentiert, weil es glaub ich langsam etwas zuviel wird, hier wieder schrittweise durchzugehen. Ich werde nur noch einige Sachen noch mal näher erläutern. Ich überprüfe diesmal mit diesem *If* gleich am Anfang, ob das Fenster überhaupt erstellt werden konnte. Falls dies so ist,

gibt der Befehl *OpenWindow()* einen Wert ungleich 0 zurück, also true. Falls das Fenster nicht erstellt werden konnte, wird der Befehl immer 0 zurückgeben. Dies kann zum Beispiel passieren, wenn nicht mehr genügend Speicher bereit steht. Ich habe es nur vorhin der Einfachheit halber weggelassen. Wenn das Fenster nicht erstellt werden konnte, und ich führe trotzdem weiterhin noch Aktionen darauf aus, kann dies zu Abstürzen führen. Deshalb ist das ziemlich wichtig. Der nächste neue Befehl heißt *UseWindow()* mit ihm setze ich das übergebene Fenster als das aktuell benutzte. Befehle wie *WindowID()* brauchen dies vorher, damit sie wissen, welches Fenster gemeint ist. Daraufhin erstellen wir mithilfe von *CreateGadgetList()* eine Sammlung an Gadgets. In ihr kann ich jetzt beliebig viele Gadgets erstellen. Ich übergebe diesem Befehl den Befehl *WindowID(),* welcher wiederum eine Zahl zurückgibt. Diese Zahl ist die ID des aktuellen Fensters(Was ich mit *UseWindow()* festgelegt habe). Diese ID (Identifikations-Nummer) ist eine intern verwendete Nummer. Du wirst sie als Anfänger an sich selten gebrauchen können. Mit dieser Zahl ist das Fenster dem Betriebssystem bekannt. Du brauchst sie also eigentlich nur, wenn du Befehle des Betriebssystems, die API-Befehle aufrufen möchtest. Das klären wir aber erst viel später, da dies nicht direkt etwas mit PureBasic zu tun hat. Also machen wir mal weiter mit dem nächsten unbekannten Befehl, und zwar *ButtonGadget().* Dieser Befehl erstellt so einen kleinen Knopf, den du da sehen kannst. Der erste Parameter ist wieder eine Zahl, in der du die Nummer des Gadgets übergibst, um es, später, wieder zu erkennen. Die nächsten 4 Parameter sind wieder: X-Position, Y-Position, Breite und Höhe. Der letzte Parameter ist optional. Hier können wieder die Flags, also die Optionen, für dieses Gadget eingestellt werden. Es stehen folgende zur Verfügung:

```
  #PB_Button_Right     ;rechtsbündige Darstellung des Schalter-Textes
  #PB_Button_Left      ;linksbündige Darstellung des Schalter-Textes
  #PB_Button_Default   ;legt das definierte Aussehen des Schalters als Standard-
Schalter für das Fenster fest
  #PB_Button_MultiLine ;Ist der Text zu lang, wird er über mehrere Zeilen
dargestellt
  #PB_Button_Toggle    ;erstellt einen 'Toggle' Schalter: ein Klick und der
Schalter bleibt gedrückt, ein weiterer Klick gibt ihn wieder frei
```

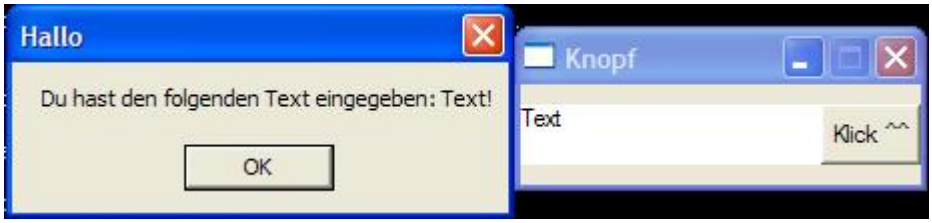
Hier auch noch ein paar Bilder aus der Hilfe, die die Flags bildlich zeigen:



Auch wieder in der Hilfe nach schaubar. Der Großteil vom Rest sollte sich von allein erklären. Der einzige wirklich neue Befehl ist noch *EventGadgetID()*. Aber wie gesagt, gibt dieser Befehl einfach nur die Nummer des Gadgets zurück, auf dem ein Event eingetreten ist. Ja, nun gibt es leider jede Menge solcher Gadgets, nur kann

ich leider nicht auf alle hier eingehen. Das währe wirklich zu umfangreich. Prinzipiell arbeiten sie alle gleich. Ich werde hier nur noch auf ein Gadget eingehen, und das ist das **StringGadget.** Dies mache ich, um euch zu zeigen, dass das wirklich immer das gleiche ist und um noch ein paar neue Sachen zu erläutern, also noch mal schön zuhören. Also, die StringGadget´s sind diese Dinger, in denen man etwas rein schreiben kann. Man kann mit ihnen also richtige Eingaben machen. Dies macht sie noch mal interessant. Die anderen Steuerelemente könnt ihr einfach in der Hilfe nachschauen, wo sie gut erklärt sind. Okay, also zeigen wir mal den Einsatz von so einem StringGadget:

```
;Öffne ein Fenster...
If OpenWindow(0,0,0,200,50,#PB_Window_MinimizeGadget |
#PB_Window_SystemMenu | #PB_Window_ScreenCentered,"Knopf")
  ;Das Fenster konnte erstellt werden
  ;Benutze Fenster 0
  UseWindow(0)
  ;Erstelle eine Liste für das aktuell benutzte Fenster
  If CreateGadgetList(WindowID())
    ;Die GadgetListe konnte erstellt werden :-)
    ;Stringgadget erstellen
    StringGadget(0,0,10,150,30,"Text eingeben",#PB_String_BorderLess)
    ;Ein kleines Hilfefenster hinzufügen...
    GadgetToolTip(0,"Hier irgendwas eingeben")
    ;und noch einen kleinen Button...
    ButtonGadget(1,150,10,50,30,"Klick ^^")
  EndIf
;Okay alles klar soweit :-)
;Unsere Event-Loop
Repeat
  ;Auf ein Event warten...
  Event = WaitWindowEvent()
  Select Event  ;Das Event herausfinden
    Case #PB_Event_CloseWindow ;Fenster soll geschlossen werden...
      End ;Cya ^^
    Case #PB_EventGadget       ;Es ist ein Ereignis auf einem Knopf
aufgetreten...
      Select EventGadgetID() ;Welches wurde gedrückt?
      Case 1 ;Gadget 1 wurde gedrückt
        ;Den Text des Gadgets holen, also den des Stringgadgets
        Text.s = GetGadgetText(0)
        ;Und ausgeben ^^
        MessageRequester("Hallo","Du hast den folgenden Text eingegeben:
"+Text+"!",0)
      EndSelect
  EndSelect
ForEver
EndIf
```

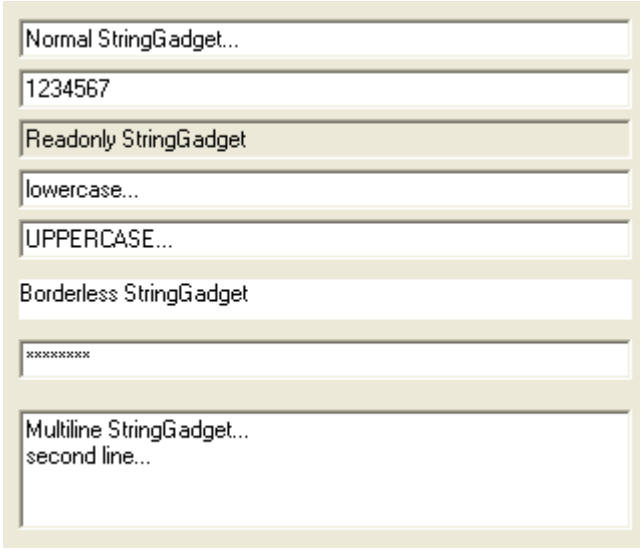


Das Programm ist, wie gesagt, fast gleich geblieben, ich erkläre wieder nur das neue. Der erste neue Befehl ist *StringGadget().* Was es macht, ist ja wohl klar.

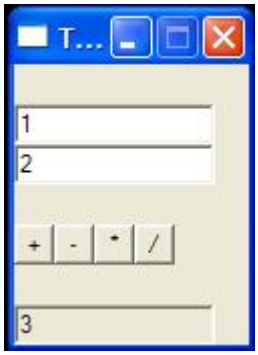
Es erstellt dieses Eingabefeld. Auch hier ist der letzte Parameter wieder optional. Er kann einer der folgenden Werte sein:

```
  #PB_String_Numeric    ;Nur Zahlen werden akzeptiert.
  #PB_String_Multiline  ;Mehrere Zeilen an Text werden akzeptiert.
  #PB_String_Password   ;Passwort-Modus, es werden nur '*' anstelle normaler
Zeichen angezeigt.
  #PB_String_ReadOnly   ;'Read only' Modus. Es kann kein Text eingegeben werden.
  #PB_String_LowerCase  ;Alle Zeichen werden automatisch in Kleinbuchstaben
umgewandelt.
  #PB_String_UpperCase  ;Alle Zeichen werden automatisch in Großbuchstaben
umgewandelt.
  #PB_String_BorderLess ;Es werden keine Ränder rings um das Gadget gezeichnet.
```

hier noch mal das Bild aus der Hilfe für die einzelnen Parameter:



Der nächste neue Befehl ist das *GadgetToolTip()* mit diesem Befehl kann man einem Gadget einen Text zuweisen, der Ausgegeben wird, wenn man mit der Maus drüber stehen bleibt. Der letzte neue Befehl ist *GetGadgetText().* Mit ihm kann man sich die Aufschrift von einem Gadget „holen". Diesen Wert gibt er dann zurück. So… nun hätten wir das geklärt…. Jetzt seit ihr zur Abwechslung auch mal dran. Mit den Informationen, die ihr habt, könnt ihr inzwischen eine neue Version unseres Taschenrechner basteln… wie wär´s?! Das Resultat soll zum Schluss in etwas so hier aussehen:



- Denkt an den Befehl *Val()*
- Mit *GetGadgetText()* könnt ihr den Text eines Gadgets „holen"
- *#PB_String_Numeric*… ; *#PB_String_ReadOnly*…

Ich werde euch hier mal meine Variante posten. Eure kann natürlich mächtig anders aussehen…kein Problem! !!! ABER ERSTMAL SELBER PROBIEREN !!!

```
;Gadgetkonstanten festlegen

;Enumeration
#ButtonPlus = 0
#ButtonMinus = 1
#ButtonMal = 2
#ButtonDurch = 3
#StringZahl1 = 3
#StringZahl2 = 5
#StringErgebnis = 7
;EndEnumeration

#MainWindow = 0

If OpenWindow(#MainWindow,0,0,100,140,#PB_Window_ScreenCentered |
#PB_Window_SystemMenu | #PB_Window_MinimizeGadget,"Taschenrechner")
  UseWindow(0)
  If CreateGadgetList(WindowID())

    ;Gadgets erstellen
    StringGadget(#StringZahl1,0,20,100,20,"1",#PB_String_Numeric)
    GadgetToolTip(#StringZahl1,"Erste Zahl eingeben")
    StringGadget(#StringZahl2,0,40,100,20,"2",#PB_String_Numeric)
    GadgetToolTip(#StringZahl2,"Zweite Zahl eingeben")

    ButtonGadget(#ButtonPlus,0,80,20,20,"+")
    ButtonGadget(#ButtonMinus,20,80,20,20,"-")
    ButtonGadget(#ButtonMal,40,80,20,20,"*")
    ButtonGadget(#ButtonDurch,60,80,20,20,"/")

    StringGadget(#StringErgebnis,0,120,100,20,"3",#PB_String_Numeric |
#PB_String_ReadOnly)
  EndIf
EndIf

Repeat
  Select WaitWindowEvent()

  Case #PB_EVENT_CloseWindow
    End

  Case #PB_Event_Gadget
    Select EventGadgetID()

    Case #ButtonPlus
      SetGadgetText(#StringErgebnis,Str(Val(GetGadgetText(#StringZahl1))+Va
l(GetGadgetText(#StringZahl2))))
    Case #ButtonMinus
      SetGadgetText(#StringErgebnis,Str(Val(GetGadgetText(#StringZahl1))-
Val(GetGadgetText(#StringZahl2))))
    Case #ButtonMal
      SetGadgetText(#StringErgebnis,Str(Val(GetGadgetText(#StringZahl1))*Va
l(GetGadgetText(#StringZahl2))))
    Case #ButtonDurch
      SetGadgetText(#StringErgebnis,Str(Val(GetGadgetText(#StringZahl1))/Va
l(GetGadgetText(#StringZahl2))))
    EndSelect
  EndSelect
ForEver
```

Okay, sagen wir mal, dass meiste hiervon müsstet ihr euch eigentlich erklären können. Das einzige verwunderliche ist aber, dass ich statt Zahlen den Gadgets Konstanten übergeben habe. Doch dies ist eigentlich ganz logisch. Ich weiße doch den Konstanten auch Zahlen zu. Damit werden diese Zahlen dort auch eingesetzt. So haben die Zahlen wenigstens andere Namen. Bedenke, dass jede Zahl immer eine konstante ist! So, jetzt könnt ihr sogar schon eure ersten Fenster erstellen.
Ist doch was :-) Wenn man mal bedenkt, dass ich man innerhalb 36 Seiten soweit kommt, kann man wirklich sagen, dass PB eine leicht erlernbare Programmiersprache ist!

Kapitel IV
(Speicherhandling)

Es wird nun leider wieder Zeit für ein wenig, dafür aber sehr wichtiger, Theorie. Es geht um die Verwaltung von Daten. Als Anfänger kommt dir jetzt auch sicher schon die erste Frage: Was genau sind eigentlich Daten? Wir haben eigentlich schon eine Möglichkeit des Umgangs mit Daten kennen gelernt. Und zwar bei dem Kapitel über Variablen. In so einer Variable konnten wir Zahlen oder Buchstaben speichern. Das sind Daten. An sich sind Daten nichts weiter als Informationen. Eine Zahl ist eine Information, ein Zeichen ist eine Information, ein Zustand (wie false und true) ist eine Information. In großen Programmen muss man sehr häufig mit sehr großen Mengen an Daten zurechtkommen. Man muss sie verwalten können. Das soll Inhalt dieses doch etwas anspruchsvolleren Kapitels werden. In diesem Kapitel wirst du wahrscheinlich sehr oft auf große Fragezeichen stoßen, aber das ist nicht schlimm. Hier hilft es, mit dem neu gelernten Wissen zu experimentieren, da es wirklich teils doch schon etwas schwieriger zu verstehen ist. Aber ich will euch jetzt keine Angst machen, wir fangen mit einem etwas leichteren Abschnitt an: Den **Arrays**.

Arrays

Also, klären wir erst einmal , was diese Dinger überhaupt sind. Arrays sind eigentlich nichts weiter als Variablen. Aber eben keine normalen Variablen. Sie sind eine Sammlung mehrerer Variablen vom gleichen Typ unter dem gleichen Namen. So ein Array sieht im einfachsten Fall zum Beispiel so hier aus:

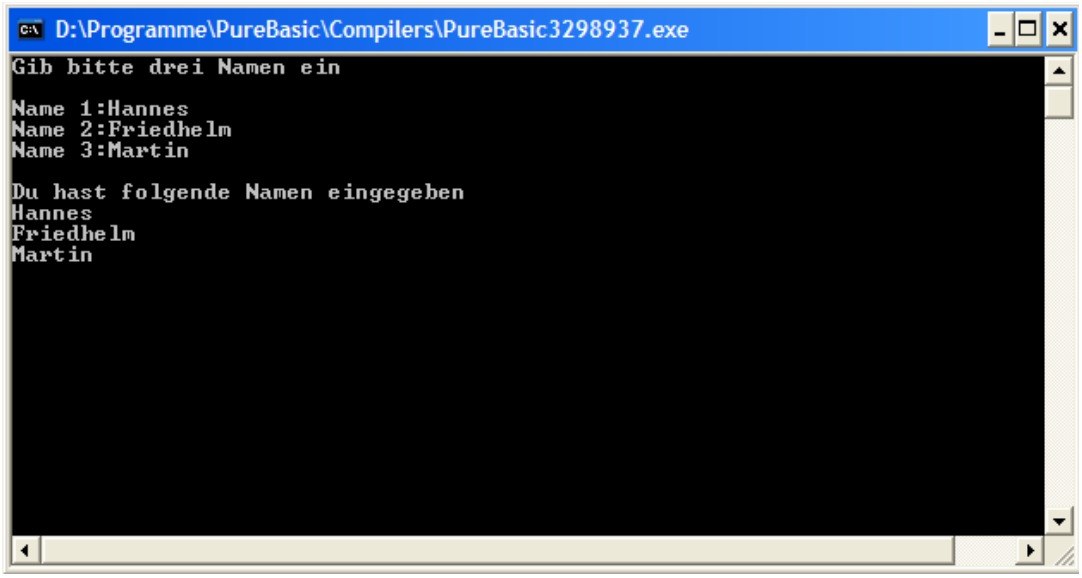
```
Dim Array.l(10)
```

Du siehst, dass das Schlüsselwort für Arrays **Dim** lautet. Dies ist abgeleitet von **Dim**ensionieren. Ihr seht, das der folgende Teil so aussieht, wie eine Normale Variable. Sie trägt den Namen „Array" und ist vom Typ Long. Doch danach kommt ein etwas unverständlicher Teil. Ich schreibe in Klammern die Zahl 10. Was hat das zu bedeuten? Ganze einfach ;-), Ich erstelle eine Sammlung aus 11 Variablen(0 bis 10) vom Typ long unter dem Namen „Array". Ich kann sozusagen unter dem Namen „Array" auf 11 Variablen zurückgreifen. Das sieht jetzt schwieriger aus, als es ist, also wird mal wieder ein Beispiel zur Aufklärung nötig ^^

```
Dim Names.s(2) ;Array mit dem Namen "Names" für 3 Namen

OpenConsole();Wir öffnen eine Console
  PrintN("Gib bitte drei Namen ein")
  PrintN("")
  Print("Name 1:")
  Names(0) = Input();Der Benutzer gibt etwas ein.
  PrintN("")
  ;Das Ergebnis von Input() speichern wir
  ;In die erste Variable des Arrays.
  Print("Name 2:")
  Names(1) = Input();Der Benutzer gibt etwas ein.
  PrintN("")
```

```
  Print("Name 3:")
  Names(2) = Input();Der Benutzer gibt etwas ein.
  PrintN("")
  ;So, jetzt haben wir den Benutzer
  ;Alle 3 Namen (oder sonstwas ;-) eingeben lassen
  ;(Auch wenn das eine Umständliche Lösung war,
  ;aber Aufgrund des Verständnisses so besser)

  ;Jetzt wollen wir die Namen wieder ausgeben lassen
  PrintN("")
  PrintN("Du hast folgende Namen eingegeben")
  For x = 0 To 2
    ;Wir gehen die einzelnen Elemente des Arrays durch
    PrintN(Names(x))
  Next
  Delay(2000)
CloseConsole()
End
```

Die folgende Console sollte beim Ausführen des Codes erscheinen:



Sieht auf dem ersten Blick komplizierter aus, als es ist. Das wichtigste ist allerdings schon Code gesagt. Ich will nur noch einmal kurz auf das **Dim Names.s(2)** kommen. Wie gesagt definiert (oder besser Dimensioniert) ihr hier einen Array mit dem Namen „Names". Er ist vom Typ String und hat 3 Elemente(0, 1, 2). Nach dem Dimensionieren kann man auf die einzelnen Variablen zugreifen, wenn man hinter dem Variablennamen die Nummer des Elements in Klammern schreibt, was man ansprechen möchte. Du solltest aber darauf achten, dass du keine Nummern auserhalb des Bereiches angibst, da dies zu Programmabstürzen führen kann. Ja, nun ist das aber (natürlich ;-) noch nicht alles zum Thema Arrays. Eben haben wir einen Eindimensionalen Array definiert. Ja, richtig erkannt…er kann auch mehrere Dimensionen haben. Das Prinzip ist aber immer noch das gleiche.
Hier mal ein paar Beispiele, wie Mehrdimensionale Arrays aussehen:

```
Dim EinDimensional.l(12)
Dim ZweiDeimensional.l(10,4)
Dim DreiDimensional.l(10,10,5)
Dim VierDimensional.l(10,10,10,342)
```

Die einzelnen Dimensionen werden mit Komma voneinander getrennt. Das ganze sieht fast so aus wie ein Funktionsaufruf, nicht? Das ist jetzt aber etwas schwierig zu verstehen, aber kein Problem. Unser Eindimensionales Array fasst, wie bekannt, 13 Variablen. Das Zweidimensionale Array hingegen fast 11x5 Variablen. Der Dreidimensionale Array fasst 11x11x6 Variablen, usw. Theoretisch ist den Dimensionen keine Grenze gesetzt, praktisch aber ist irgendwann dein Hauptspeicher voll. Doch auch dazu will ich eich mal kein Beispiel vorenthalten, damit ihr die Sache etwas besser verstehen könnt!

```
Dim Persons.s(2,2)
;Erstellt einen 2-Dimensionalen Array.

OpenConsole()
PrintN("Bitte ein paar Personendaten eingeben")
For x = 0 To 2
  PrintN("")
  Print("Vorname: ")
  Persons(x,0) = Input()
  PrintN("")
  Print("Nachname: ")
  Persons(x,1) = Input()
  PrintN("")
  Print("Alter: ")
  Persons(x,2) = Input()

Next
PrintN("")
PrintN("")
PrintN("Du hast folgende Angaben gemacht")
PrintN("")
For x = 0 To 2
  PrintN("Person "+Str(x))
  PrintN("Vorname: "+Persons(x,0))
  PrintN("Nachname: "+Persons(x,1))
  PrintN("Alter: "+Persons(x,2))
Next

Delay(5000)
CloseConsole()
```

Wie ihr seht ist das fast das gleiche Prinzip, wie mit den eindimensionalen Arrays. Ich Speichere auch hier wieder die Namen der Personen, diesmal brauche ich aber zu jeder Person mehr Informationen -> Wir müssen noch eine Dimension anlegen. Die erste Dimension ist in unserem Fall wie eine Nummerierung der einzelnen Personen. Solche Arrays sind sehr nützlich für größere Datensammlungen, da sie relativ unkompliziert arbeiten. Allerdings sind sie nicht sonderlich flexibel, da man nur einen Datentyp in so einem Array speichern kann. Außerdem ist die Größe des Arrays nicht flexibel, man kann mit ihm nur eine bestimmte Menge an Daten zwischenlagern wie eben bei der Erstellung angegeben. Wenn man die Größe des Arrays ändern möchte, ruft man Dim einfach mit dem gleichen Array-Namen auf. Dabei geht aber der Inhalt des vorigen Arrays verloren. Um den Speicher der für das Array verwendet wurde, wieder freizugeben, ruft man einfach Dim mit dem Array-Namen auf und gibt eine 0 als Größe an. Nun haben wir das wichtigste zu Arrays gesagt, kommen wir nun zu einem einer etwas komplexeren Art Daten zu speichern, den Strukturen.

Strukturen

Also, Strukturen sind an sich Speicherbereiche, die nach einem vorgegebenen Muster aufgebaut sind. Dieses Muster kann man bei der Erstellung der Struktur angeben. Man definiert damit einen neuen Datentyp, wie Long oder Byte. Damit kann man also Variablen einen selbst erstellten Typ geben. Dieser Typ fasst aber, im Gegensatz zu anderen Typen in der Regel mehrere verschiedene Variablen(typen) auf einmal. Aber jetzt genug der Rede, ich schreib hier mal eine einfache Struktur als Beispiel.

```
Structure MyStruct
  Element1.s
  Element2.l
  Element3.b
  Element4.w
EndStructure
```

Wie ihr unschwer erkennen solltet, ist das Schlüsselwort für Strukturen **Structure** (wärst du jetzt sicher nicht drauf gekommen ;-) Ich definiere hier eine Struktur mit dem Namen „MyStruct". Sie enthält vier Variablen, nämlich „Element1", „Element2", „Element3" und „Element4" welche alle von verschiedenen Typen sind. Zum Schluss steht dann noch das Schlüsselwort „EndStructure", was einfach besagt, dass die Strukturdefinition hier ein Ende hat. Dieses „MyStruct" können wir jetzt einfach verwenden, als währe es ein Typ wie .w oder .f. Wir können jetzt also einfach eine Variable auf diese Weise deklarieren:

```
Struct.MyStruc
```

Jetzt haben wir die Variable „Struct" mit dem Typ „MyStruct". Soweit, so gut :-)
Um auf die Elemente einer Struktur zuzugreifen, verwendet man einfach ein „\".

```
Structure MyStruct
  Element1.s
  Element2.l
  Element3.b
  Element4.w
EndStructure

Struct.MyStruct

Struct\Element1 = "Hallo"
Struct\Element2 = 123213
Struct\Element3 = 21
Struct\Element4 = 3322

MessageRequester("Strukturen","Das Element 1 der Struktur hat folgenden
Inhalt: "+Struct\Element1+".",0)
```

Es sollte eine MessageBox aufgehen, in der steht, dass das Element1 den Inhalt „Hallo" hat. Solche Strukturen kann man natürlich auch verschachteln. Das geht ganz einfach, indem man einer Variable den Typ einer anderen Struktur zuweist.

```
Structure MyStruct
  Element1.s
  Element2.l
  Element3.b
```

```
  Element4.w
EndStructure

Structure MySecondStruct
  Element1.l
  Element2.b
  Element3.MyStruct
EndStructure

Struktur.MySecondStruct
Struktur\Element3\Element1 = "Hallo"

MessageRequester("Struktur-In-Struktur",Struktur\Element3\Element1,0)
```

Das kann man natürlich noch endlos weiter verschachteln. Allerdings steigert sich dadurch nicht unbedingt die Übersicht. Deshalb gibt es noch eine weitere Methode, Strukturen zu verschachteln. Eine Vorform der Vererbung, die aus C++ bekannt ist, aber das ist eher unbedeutend. Es gibt die Möglichkeit, eine Struktur mit einer anderen zu erweitern. Dieses Schlüsselwort heißt **Extends**. Auch dazu will ich gleich mal ein Beispiel schreiben, da es schwerer zu beschreiben ist, als es eigentlich wirklich ist:

```
Structure Names
  Vorname.s
  Nachname.s
EndStructure

Structure Persons Extends Names
  Alter.b
  Stadt.s
EndStructure

Personen.Persons

Personen\Vorname = "Rudolph"
Personen\Nachname = "Meier"
Personen\Alter = 21
Personen\Stadt = "Timbuktu"
```

In diesem Fall erweitere ich die Struktur „Persons“ um die Struktur „Names“.
Jetzt kann ich die Variablen der erweiterten Struktur verwenden, wie als hätte ich sie direkt hineingeschrieben. Man braucht diese Technik zwar relativ selten, sollte aber im Auge behalten, dass es sie gibt, denn manchmal kann man sich damit viel Arbeit sparen. Dann blieben nur noch wenige grundlegende Dinge zu Strukturen zu sagen. Wichtig ist zu beachten, dass man, wenn man in Strukturen Arrays anlegen will, die Arrayklammern eckige Klammern sein müssen. Allerdings ist das verhalten der Arrays in Strukturen leicht anders. Wenn du in einer Struktur ein Array anlegst, so ist die Größe immer die angegebene Größe minus 1. Auch hier beginnt das zählen immer bei 0. Lege ich in einer Struktur einen Array mit der Größe von 10 an, so hat es auch nur 10 Elemente, nämlich die Elemente von 0 bis 9. Hier ein kleines Beispiel dafür.

```
Structure ArrayStruct
  Array.l[10]
EndStructure

Struktur.ArrayStruct
Struktur\Array[0] = 1
```

```
Struktur\Array[10] = 100 ;Fehler

;im Gegensatz
Dim Array(10)
Array(0) = 1
Array(10) = 100 ;Korrekt
```

Dieses Verhalten ist notwendig, um mit den Windows-Standard konform zu bleiben. Die API des Betriebssystems arbeitet auch auf diese Weise. Dies garantiert die Kompatibilität zum Betriebssystem und ist deshalb durchaus berechtigt. Noch zu erwähnen wäre, dass man natürlich auch Arrays als Strukturen definieren kann. Auch dazu noch ein kleines Beispiel:

```
Structure ArrayStruct
  Array.l[10]
  Element.s
EndStructure

Dim Array.ArrayStruct(10)

Array(0)\Array[0] = 1
Array(0)\Element = "Hallo"
```

Nun bliebe noch eine, etwas unwichtigere Sache zu klären, da ihr sie wirklich nur extrem selten brauchen werdet, und auch nur in Programmen, wo ihr mehrere hunderttausend Strukturen zu verwalten habt und der Speicher des Systems nicht so gut bestückt ist. Dies sind die **Union's**. Mit ihnen kann man mehrere verschieden-artiger Variablen in den gleichen Speicherbereich schreiben. Dies ist nur nützlich, wenn man vorher nicht genau weiß, welchen Typ eine bestimmte Variable in einer Struktur haben soll. Das ganze könnte in etwa so hier aussehen:

```
Structure Strukt
  NichtUnion.l
  StructureUnion
    Var1.f
    Var2.l
  EndStructureUnion
EndStructure

Test.Strukt
Test\Var1 = 1.2
Test\NichtUnion = 100
```

Wie unschwer zu erkennen ist, ist das Schlüsselwort für die Unions **StructureUnion** und zum beenden der Union **EndStructureUnion.** Alle Variablen die zwischen den beiden Schlüsselwörtern stehen teilen sich somit den gleichen Speicherbereich. Damit ist der Speicherbereich immer so groß, wie die größte Variable der StructureUnion. Damit ergibt sich dann, dass man nur eine der Variablen benutzen sollte. Denn ändert man die eine, so ändert man auch gleich alle anderen mit, was eigentlich logisch ist, da die Variablen sich im selben Stückchen Speicher befinden. So, ich würde sagen, das reicht erstmal zu Strukturen, kommen wir zum nächsten Abschnitt, den dynamisch verknüpften Listen.

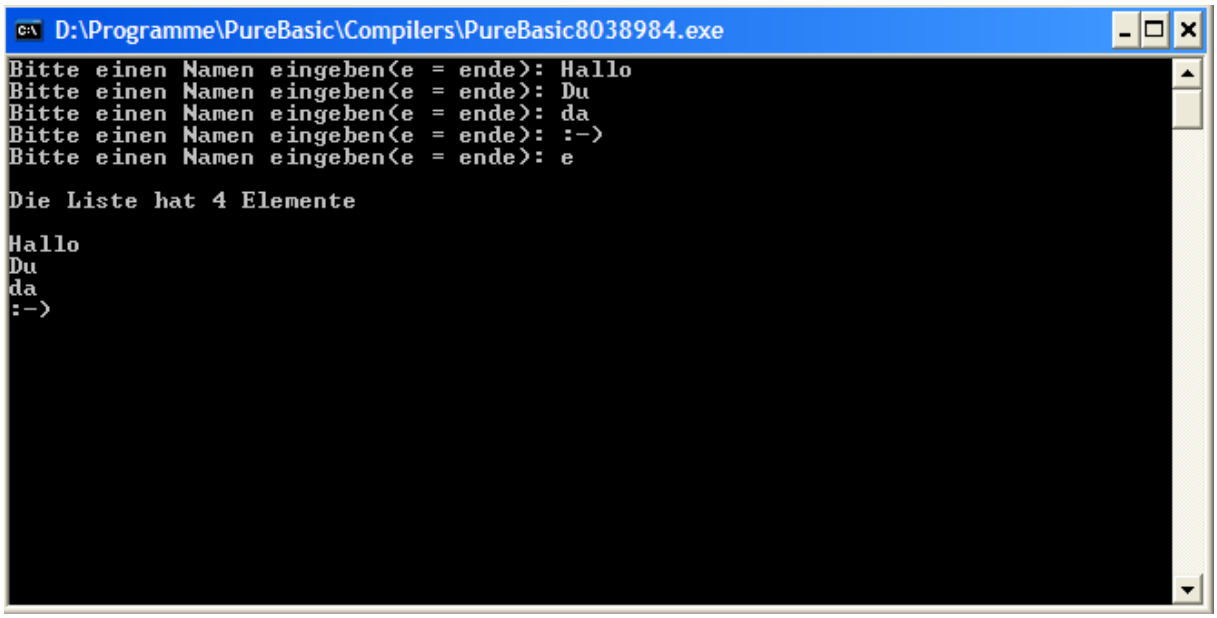
Dynamically Linked Lists

Jetzt spielen wir einen echten Vorteil von PureBasic im Gegensatz zu anderen Sprachen aus: Die verknüpften Listen. Diesen Befehlssatz muss man sich in anderen Sprachen erst mühsam selbst schreiben, aber in PureBasic gibt es ihn schon eingebaut :-) Diese verknüpften Listen werdet ihr häufig brauchen, da sie echt nützlich und wirklich einmal dynamisch sind. Diese Listen sind dazu da, um eine Kette an Daten im Speicher anzulegen, wo vorher nicht bekannt ist, wie viele Elemente diese Kette mal haben soll. In dieser Kette können problemlos jederzeit Elemente entfernt werden und Elemente hinzugefügt werden. Sie sind eine echt komfortable Variante um dynamisch mit dem speicher zu agieren. Wie sieht jetzt so eine Liste aus? Im einfachsten Fall in etwa so hier:

```
NewList Liste.l()
```

Das Schlüsselwort für diese Listen ist unschwer zu erkennen: **NewList.**
Hier erstelle ich eine neue Liste mit dem Name „Liste", kreativ nicht wahr? ;-)
Diese Liste beinhaltet einfach nur LONG-Variablen. Sie ist also eine Kette, die ich unendlich mit Long-Werten versehen kann. Damit ihr das Prinzip versteht, hier mal ein Beispiel:

```
;Erstmal eine Liste mit String-Variablen erstellen
NewList Names.s()


OpenConsole()
  ende = 0
  Repeat
    Print("Bitte einen Namen eingeben(e = ende): ")
    Name$ = Input() ;Etwas eingeben lassen
    PrintN("")
    If Name$ <> "e"
      AddElement(Names())
      Names() = Name$
    Else ;Falls ein "e" eingegeben wurde
      ende = 1
    EndIf
  Until ende = 1
  PrintN("")
  ;Okay, die Namen sind in der Liste gespeichert.
  ;Erstmal die Elemente der Liste anzeigen.
  PrintN("Die Liste hat "+Str(CountList(Names()))+" Elemente")
  ;Jetzt Listen wir die Elemente mal auf
  PrintN("")
  ForEach Names() ;Geht jedes einzelne Element der liste durch
    PrintN(Names())
  Next
  Input()
CloseConsole()
```

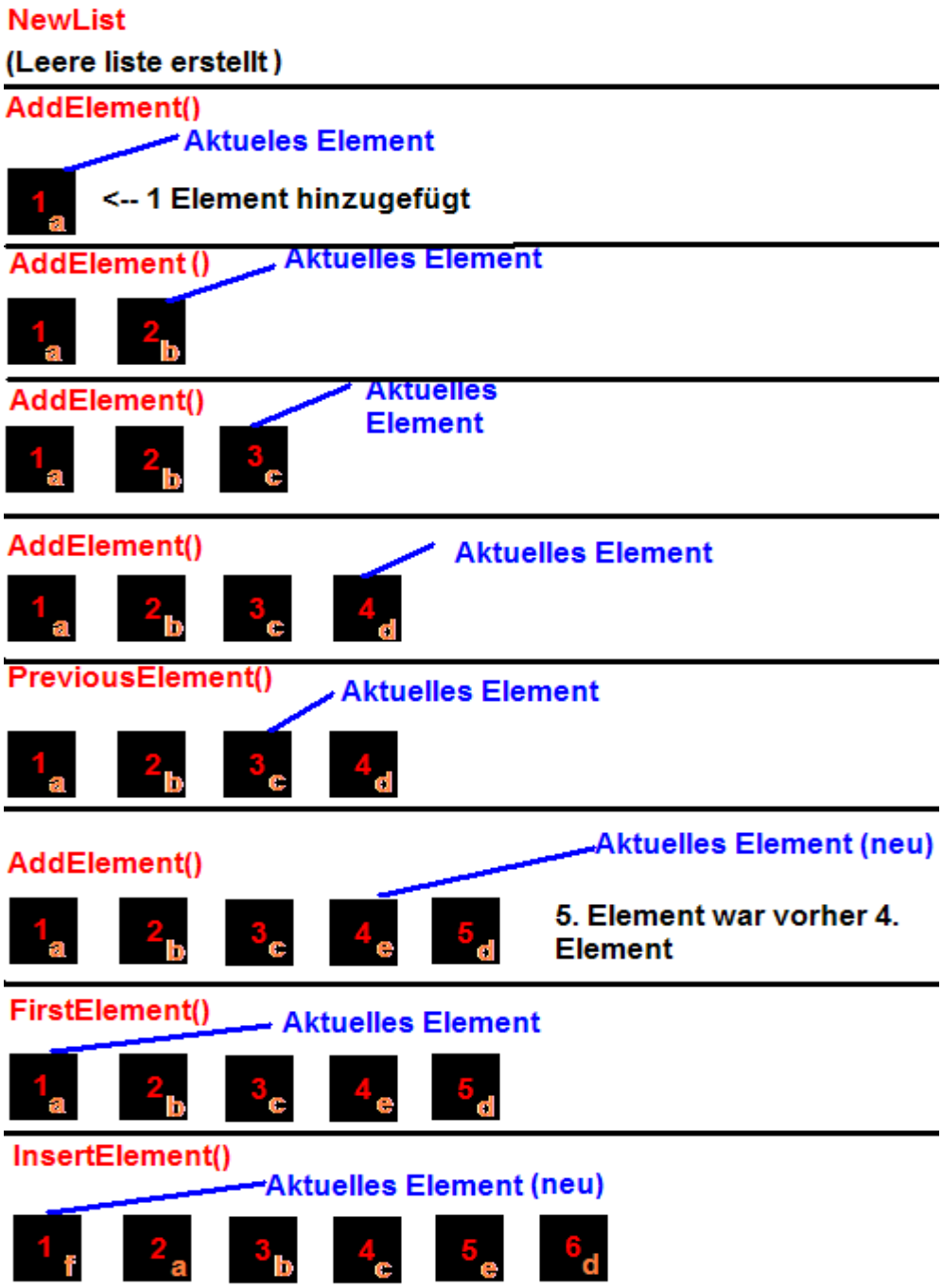


In diesem Code sind gleich mehrere neue Dinge eingebaut. Also gehen wir das mal durch… Zuerst legen wir die Liste „Names" mit String-Variablen an. Dann öffnen wir eine Console und starten eine Repeat-Until Schleife, welche sich beendet, sobald der Benutzer ein „e" eingibt. Dann lassen wir den Benutzer etwas eingeben, sollte seine Eingabe kein „e" sein, so wird mittels **AddElement()** ein neues Element zu der Liste Speicher() hinzugefügt. (Bei LinkedLists müssen immer Klammern hinter dem Namen angegeben werden, also nicht verwirren lassen). Dieses Element wird automatisch immer nach dem aktuellen Element eingefügt. Danach geben wir weisen wir der Liste an der aktuellen Position den eingegeben Text zu. Das wiederholen wir solange, bis der Benutzer ein „e eingibt. Danach geben wir mittels **CountList()** die Gesamtzahl der eingegebenen Elemente aus. Achtung! nicht verwirren lassen, hier fängt das zählen bei 1 an, nicht bei 0. Wenn wir das erledigt haben, gehen wir die Elemente mittels der neuen Schleife, der **ForEach-Next Schleife**, eins nach dem anderen durch. Diese Schleife fängt beim ersten Element der angegebenen Liste an. Es setzt das aktuelle Element der Liste mittels **Next** immer um eins nach „oben". Somit können wir also jedes Element der Liste bearbeiten. Danach geben wir geben dann einfach immer den Inhalt der LinkedList aus. Wem die ForEach-Next Schleife jetzt etwas komisch vorkommt, dem sei noch eine andere Möglichkeit gezeigt, wie man diese Schleife auch selbst schreiben könnte.

```
;Erstmal eine Liste mit String-Variablen erstellen
NewList Names.s()


OpenConsole()
  ende = 0
  Repeat
    Print("Bitte einen Namen eingeben(e = ende): ")
    Name$ = Input() ;Etwas eingeben lassen
    PrintN("")
    If Name$ <> "e"
      AddElement(Names())
      Names() = Name$
    Else ;Falls ein "e" eingegeben wurde
      ende = 1
    EndIf
```

```
  Until ende = 1
  PrintN("")
  ;Okay, die Namen sind in der Liste gespeichert.
  ;Erstmal die Elemente der Liste anzeigen.
  PrintN("Die Liste hat "+Str(CountList(Names()))+" Elemente")
  ;Jetzt Listen wir die Elemente mal auf
  PrintN("")

  ResetList(Names())
  NextElement(Names())
  Repeat
    PrintN(Names())
  Until NextElement(Names()) = 0
  Input()
CloseConsole()
```

Okay, das werde ich jetzt auch mal etwas näher erklären. Der einzige Unterschied ist eigentlich der Teil, der die Namen ausgibt, die ForEach-Next Schleife ist weg. Anstelle dessen steht dort als erstes Neues: **ResetList()**. Mit diesem Befehl setzt man die Liste zum Anfang zurück, vor das 0. Element, was es natürlich nicht gibt. Deshalb setzen wir die Liste mittels **NextElement()** auf das erste legale Element. NextElement wechselt immer zum nächsten Element aus der Liste. Danach kommt die Repeat-Until Schleife zum Einsatz. Die Schleife wird solange wiederholt, bis es keine Elemente mehr in der Liste gibt. Ist dies der Fall, gibt NextElement() automatisch eine 0 zurück. Damit währe dann auch gleich die Abbruchbedingung für die Schleife gegeben. Sie wird also beendet, wenn kein Element mehr in der Liste ist. Ansonsten Printet sie fröhlich die einzelnen Elemente der Liste. Mit **PreviousElement()** könnt ihr das aktuelle Element auf das vorige setzen. Mit **FirstElement()** setzt ihr die Liste auf das erste, mit **LastElement()** auf das letzte und mit **SelectElement()** auf ein beliebiges Element. SelectElement erwartet als ersten Parameter die Liste, die angesprochen werden soll und im zweiten Parameter die Nummer des Elements, auf die sie gesetzt werden soll. Mit **DeleteElement()** wird das aktuelle Element der Liste gelöscht. Das vorige Element wird dadurch automatisch das aktuelle Element. Solltest du das erste Element der Liste löschen, so wird automatisch das nächste Element das aktuelle Element. Mit **InsertElement()** kannst du ein Element VOR dem aktuellen einfügen. Das neue Element wird automatisch das aktuelle Element. **ClearList()** entfernt alle Elemente der aktuellen Liste. Sie hat nach dem Aufruf dieses Kommandos keinen Inhalt mehr. **ListIndex()** gibt die Nummer des aktuellen Elementes zurück(beginnend bei 0). Ganz schön kompliziert, nicht?! Du brauchst dir jetzt nicht gleich die ganzen Befehle auswendig zu merken, denn sie sind jederzeit im der Hilfe nachlesbar. Falls du das noch nicht so recht verstanden hast, werde ich hier mal eine kleine Zeichnung einfügen, die das näher erklären sollte.

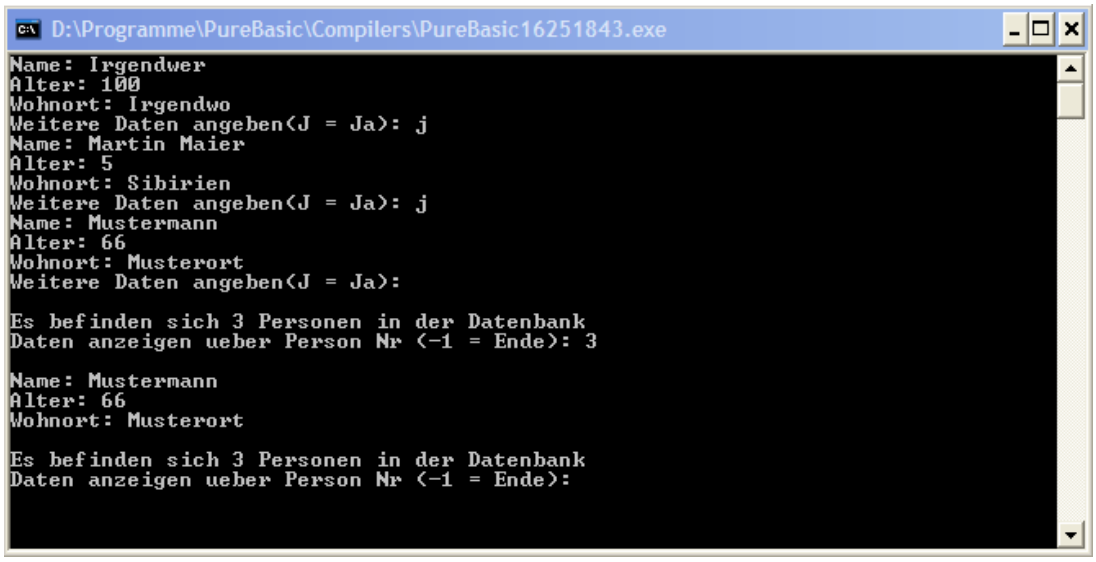


Sieht zwar etwas primitiv aus, aber es sollte das Prinzip beschreiben.
Die Kästchen stellen die Elemente dar, die Zahlen stellen die Nummern der Elemente dar und die Buchstaben die Erstellreihenfolge. In jedem Abschnitt steht der Befehl, der verwendet wird, um diese Aktionen durchzuführen. Das ganze will ich euch aber auch als Programm beweisen. Wir setzen das ganze mal um:

```
;Bitte Debugger anschalten, falls er nicht an ist
;(Compiler->Debugger)

NewList Liste.s()

AddElement(Liste())
Liste() = "a"
AddElement(Liste())
```

```
Liste() = "b"
AddElement(Liste())
Liste() = "c"
AddElement(Liste())
Liste() = "d"
PreviousElement(Liste())
AddElement(Liste())
Liste() = "e"
FirstElement(Liste())
InsertElement(Liste())
Liste() = "f"

;Jetzt nur noch anzeigen

ForEach Liste()
  Debug Liste()
Next
```

Ihr seht das gleiches Resultat. Der Befehl **Debug** ist übrigens nur dazu da, um eine Meldung im Debug-Fenster auszugeben. Das erspart uns in dem Fall nur die Console aufzumachen oder einen anderen umständlichen Weg zu nehmen, die Daten anzuzeigen. Ja, was gibt es noch zu Listen zu sagen? Ja, ganz wichtig ist, dass ihr hier natürlich auch Strukturen-Listen erstellen könnt. Ihr könnt dann wie gewohnt auf die Elemente der Struktur zugreifen, indem ihr, wie gewohnt, nach den Klammern ein „\" setzt und dann den Variablennamen. Jetzt könnt ihr eigentlich auch einmal so zum Spass mal, wie wir das schon mal hatten, eine Eingabeaufforderung erstellen, in der der Benutzer eine dynamisch so viele Namen und Daten über Personen angeben kann, wie er möchte. Das sollte über eine Strukturenliste erfolgen. Wenn der Benutzer die Daten angegeben hat, so sollte er danach zur Eingabe einer Zahl aufgefordert werden. Der Benutzer bekommt dann Daten über diese Person: Name, Alter, Wohnort. Außerdem sollte das Programm auf Fehl-eingaben flexibel reagieren können. Das ganze sollte dann in etwa so aussehen:



Ich gebe zu, das ist diesmal ein wenig komplizierter als die vorherigen Aufgaben, aber durchaus machbar. Versuch das ruhig mal so gut wie möglich hinzubekommen. Hier wieder ein paar Tipps:

- Denk an CountList()
- Denk an Val()
- Denk an die Schleifen
- Denk an SelectElement()
- Achte darauf, dass CountList() bei 1 anfängt zu zählen, nicht bei 0

Das sollte für den Anfang reichen, probier ruhig mal dein Glück ;-)
MEINE Variante kannst du danach immer noch lesen, aber du solltest auch SELBST vorher mal ein wenig probieren. Vielleicht kannst du das ganze ja noch eleganter lösen :-) Also lass dich nicht von meinem Programmierstil beeinträchtigen, das hier ist mal eine Herausforderung. Nimm sie an ;-)

Viel Glück ;-)

```
Structure Person
  Name.s
  Alter.l
  Wohnort.s
EndStructure

NewList Person.Person()

OpenConsole()
  weiter = 0
  Repeat
    Print("Name: ")
    Name$ = Input()
    PrintN("")

    Print("Alter: ")
    Alter.l = Val(Input())
    PrintN("")

    Print("Wohnort: ")
    Wohnort$ = Input()
    PrintN("")

    ;Daten speichern
    AddElement(Person())
    Person()\Name = Name$
    Person()\Alter = Alter
    Person()\Wohnort = Wohnort$
    Print("Weitere Daten angeben(J = Ja): ")
    aw.s = Input()
    If aw <> "J" And aw <> "j"
      ;Keine Angaben mehr machen...
      weiter = 1
      PrintN("")
    EndIf
    PrintN("")
  Until weiter = 1

  Repeat
    PrintN("Es befinden sich "+Str(CountList(Person()))+" Personen in der
Datenbank")
    Print("Daten anzeigen ueber Person Nr (-1 = Ende): ")
    Number = Val(Input())
    PrintN("")
    PrintN("")
    If Number = -1 ;Ende
      End
    EndIf
    If Number > CountList(Person()) Or Number < 1
      ;Nicht in der Liste
      PrintN("Keine Informationen zu dieser Person verfügbar")
    Else ;Daten anzeigen
      SelectElement(Person(),Number-1)
      PrintN("Name: "+Person()\Name)
      PrintN("Alter: " +Str(Person()\Alter))
      PrintN("Wohnort: "+Person()\Wohnort)
      PrintN("")
    EndIf
  ForEver
CloseConsole()
```

Ich möchte jetzt nicht viele Worte zu dem Code verlieren, da alle Befehle eigentlich schon geklärt sein sollten. Das sollte jetzt erstmal reichen zum Thema LinkedLists, da ich dieses Kapitel noch mehr Themen hineingehören ;-)

Zeiger

Jetzt kommen wir langsam in die höhere Kunst der Programmierung, den Zeigern. Viele Leute Schrecken vor diesen Dingern ab, aber wenn man einmal das prinzip verstanden hat, ist das kein Problem. An alle C(++)-Programmierer unter euch: Diesen Abschnitt unbedingt lesen, da es kleine Unterschiede zwischen den Zeigern von C(++) und denen von PureBasic gibt. erstmal ein paar grundlegende Dinge: Was ist ein Zeiger? So ein Zeiger(oder auch Pointer), ist eigentlich nichts weiter als eine Variable, die auf einen Speicher zeigt. Man kann zum Beispiel eine Variable auf eine andere Zeigen lassen, dann kann man den Inhalt der Variable mittels der Zeigervariable verändern. Man braucht diese Zeiger hauptsächlich, wenn man schnell Daten verarbeiten muss, oder die Daten Byte für Byte durchgehen muss.
Auch kannst du mit Zeiger Strukturen oder dynamischen Speicher direkt ansprechen. Jetzt aber erstmal ein bild, wie so ein Zeiger aussieht:

```
Variable.l = 5
*Zeiger.LONG
*Zeiger = @Variable

Debug *Zeiger\l
```

Probier das Beispiel mit eingeschaltetem Debugger aus. Ihr werdet sehen, dass eine 5 ausgegeben wird. Verwirrend, nicht?! Aber ich versuch euch das jetzt mal zu erklären… Ich deklariere erst die Variable „Variable" mit dem Typen long, und gebe ihr den Wert 5. Eigentlich nichts Besonderes. In der nächsten Zeile deklariere ich einen Zeiger mit einer Struktur namens LONG. Diese Struktur ist schon vordeklariert und enthält nichts weiter, als die Variable „l.l".

```
Structure LONG
      l.l
EndStructure
```

Mehr ist das nicht. Wie gesagt, kann ein Zeiger auf einen Speicherbereich zeigen. Wenn man vor eine Variable das @-zeichen setzt, erhält man den Speicherbereich zurück, indem die Variable gespeichert ist. Somit zeigt der Zeiger „*Zeiger" auf den Speicherbereich der Variablen „Variable". Also zeigt er auf die 5. Will ich diesen Wert nun ausgeben oder verändern, muss ich die Variable l aus der Struktur verwenden. Hätte ich einfach nur „Debug *Zeiger" geschrieben, so hätte ich den Speicherbereich, auf dem der Zeiger zeigt, ausgegeben bekommen. Genauso kann ich auch über den Speicher den Wert einer Variablen verändern. Das sieht dan in etwa so hier aus:

```
*ptr.LONG
*ptr = @Var.l
*ptr\l = 5
Debug Var
```

Du wirst sehen, dass wieder eine 5 ausgegeben wird. Ich definiere auch hier wieder

einen Zeiger mit der Struktur LONG. Er hat den Namen „*ptr" Auch ihn lasse ich wieder auf eine Long-Variable zeigen („Var.l"). Als nächstes schreibe ich in den Speicher den Wert 5. Damit hat sich der Inhalt der Variablen „Var" auf 5 geändert. Ziemlich schwierig zu verstehen, nicht? Wie gesagt, mit so einem Pointer kann man direkt in den Speicher schreiben. Doch wozu soll das hier gut sein? Zugegeben, das zu wissen reicht noch nicht unbedingt aus, um etwas Sinnvolles anzustellen, aber selbst schon dieses Wissen kann man verwenden. Zum Beispiel kann eine Prozedur, die die Speicheraddresse einer Variablen übergeben bekommt, die Variable direkt verändern. Das erspart das Rückgabewerte, erspart Zeit und eröffnet neue Möglichkeiten, denn so kann man quasi endlos viele Variablen „zurückgeben". Sowas kann im einfachen Fall so hier aussehen:

```
Procedure TestMe(*Var.WORD)
  *Var\w = 10
EndProcedure

Test.w = 100
TestMe(@Test)
Debug Test
```

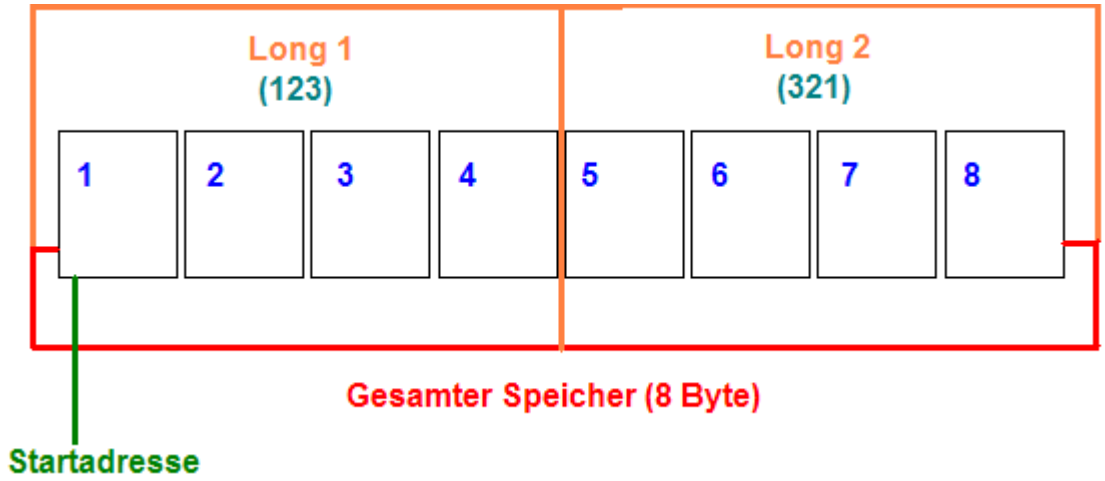
Auch hier wieder das gleiche Prinzip. Ich übergebe der Prozedur die Speicheraddresse von der Variable Test, verändere den Inhalt des Speichers und schon hat die Variable einen anderen Wert. Sehr wichtig sind solche Pointer auch bei dynamischen Speicher. Den wir jetzt auch gleich behandeln werden, da sich dadurch auch die Zeiger besser verstehen lassen.

Dynamischer Speicher

Also, dynamischer Speicher, ist eigentlich nichts weiter als ein Speicherbereich, dessen Größe man zur Laufzeit bestimmen und verändern kann. Es ist einfach nur ein Block, indem man Daten Zwischenspeichern, oder auch Buffern kann. Wenn du so einen Speicher anlegst, so bekommst du automatisch die Startdresse des Speicherbereichs übergeben. Du erhältst also immer einen Pointer zurück. Hier ist das Wissen über Variablen notwendig: Wie groß sind die einzelnen Variablentypen? Also, noch mal zur Erinnerung: Long = 4 Byte; Float = 4 Byte; Byte = 1 Byte; Word = 2 Byte; String = Variabel. Möchte ich also einen Speicherbereich anlegen. Möchte ich also einen Speicherbereich anlegen, der 5 Long Variablen fasst, so muss ich ihn 4*5 Byte groß machen. Doch genug der Vorrede, wie sieht so ein dynamischer Speicher aus? Dazu ein kleiner Code, der unser Wissen in Bezug auf Pointer etwas aufbessern sollte:

```
Structure Test
  long1.l
  long2.l
EndStructure

*Buffer.Test = AllocateMemory(8)
PokeL(*Buffer,123)
PokeL(*Buffer+4,321)

Debug *Buffer\long1
Debug *Buffer\long2
```

Als erstes schreibe ich die Struktur Test, die aus den 2 Variablen „long1" und „long2" besteht. Diese Struktur ist demnach 8 Byte groß. Dann lege ich den Pointer *Buffer an, welcher vom Typ „Test" ist. **AllocateMemory()** ist dazu da, einen Speicher anzulegen. Dieser Speicher ist genau so viele Bytes groß, wie ich dem Befehl übergebe. In diesem Fall 8 Byte. Der Befehl gibt dann automatisch die Startadresse des angelegten Speichers zurück. Somit zeigt der Zeiger *Buffer auf den gerade angelegten Speicherbereich. Der Befehl **PokeL()** ist dazu da, in dem ihn übergebenen Speicher einen LONG-Wert zu schreiben. (Ich hätte dies auch einfach mittels *Buffer\long1 = 123 machen können). Ich schreibe somit an die Startadresse des Zeigers den Long-Wert 123. Damit wären die ersten 4 der 8 Bytes vollgeschrieben. Danach addiere ich einfach den Wert 4 zu der Adresse, um die ersten vier Bytes des Speichers zu überspringen, und schreibe dann gleich ab diesem Punkt den 2. Long Wert, nämlich 321 in den Speicher. Damit habe ich die 8 Bytes des Speichers vollgeschrieben. Da ich dem Speicher eine Struktur zugeordnet habe, kann ich jetzt auch einfach mittel der Struktur die Daten wieder auslesen. Der erste Long-Wert der Struktur symbolisiert die ersten 4 Byte im Speicher, der 2. Long Wert die letzten 4. Du kannst dir die Struktur wie eine Schablone vorstellen, die über den Speicher gelegt wurde, um die einzelnen Variablen voneinander abzugrenzen und ihnen einen Namen zu geben. Ich leg mal diesen Speicher zum Verständnis noch mal zeichnerisch dar.



Ich hoffe das klärt jetzt die meisten Unklarheiten. Allerdings verwendet man dynamischen Speicher selten im diesem Zusammenhang mit Strukturen. Denn in so einem Fall könnte man ja auch die Strukturen ohne umständliche Zeiger verwenden. Im Normalfall liest man die Daten, die man in den Speicher geschrieben hat, mittels Peek() wieder aus. Da es verschiedene Datentypen gibt, gibt es auch verschiedene Formen von Peek und Poke. **PeekB()** und **PokeB()** sind dazu gedacht, Bytes in einen bestimmten Speicherbereich zu lesen und zu schreiben. **PeekL()** und **PokeL()** für Long-Werte, **PeekW()** und **PokeW()** für Word-Werte, **PeekF()** und **PokeF()** für Float-Werte und **PeekS()** und **PokeS()** für String-Werte.
Das will ich euch jetzt mal im Einsatz zeigen.

```
*Buffer = AllocateMemory(Len("Hallo Welt")+1)
PokeS(*Buffer,"Hallo Welt")

For x = 0 To Len("Hallo Welt")
```

```
  Debug PeekS(*Buffer+x,1)
Next
```

Gehen wir das stückweise durch. In der ersten Zeile fordere ich einen Speicher an. Dieser soll viele Bytes groß sein wie der String „Hallo Welt" und noch ein Byte größer. Der Befehl **Len()** gibt mir die Anzahl der Zeichen eines Strings zurück. Da jedes Zeichen genau 1 Byte Speicher verbraucht, können wir es direkt AllocateMemory übergeben. Den Zusatzbyte habe ich noch angehangen, weil jeder String im Speicher mit einer ASCII Null endet, um zu zeigen, dass der String da sein Ende gefunden hat. Der String ist also in Wahrheit immer ein Byte größer, als er sein sollte. In der nächsten Zeile schreiben wir mittels PokeS() den String „Hallo Welt" in den Buffer. Danach gehen wir Den String mittels der For-Next Schleife Zeichenweise durch. Wir lesen bei jedem Schleifendurchgang mittels PeekS() ein Zeichen aus dem Speicher. Der erste Parameter bei PeekS() ist wieder der Startspeicherbereich, von dem gelesen wird. (Dieser Speicherbereich wird auch als Offset bezeichnet) PeekS() liest genau so viele Zeichen ein, wie ich ihm als zweiten Parameter übergebe. Allerdings hätte ich für diese Aktion nicht unbedingt direkt Speicher anfordern müssen, da dies auch einfach mittels Variablen zu machen währe, denn eine Variable ist an sich auch nix weiter als Speicher, ich kann aus einer Variable auch mittels PeekX() Werte auslesen. Das würde dann in etwa so hier aussehen:

```
String.s = "Hallo Welt"

For z = 0 To Len(String)
  Debug PeekS(@String+z,1)
Next
```

Diese Möglichkeit ist sogar wesentlich unkomplexer, aber egal, war ja nur zu Demonstrationszwecken ;-) Ja, was bliebe noch zu sagen zu dem Thema?
Ja, wir haben noch einige wichtige Befehle nicht angesprochen. Der erste währe **ReAllocateMemory().** Dieser Befehl ist dazu da, um einen bestimmten Speicherbereich dynamisch zu vergrößern oder zu verkleinern. Als ersten Parameter erwartet er den zu veränderten Speicherbereich. Als zweiten Parameter erwartet dieser Befehl die Größe des neuen Speichers. Sollte der neue Speicher kleiner sein, so werden die hintersten Bytes einfach abgeschnitten. Sollte er größer sein, werden die neuen Bytes einfach mit Nullen gefüllt. Wenn als erster Parameter eine Null übergeben wird, so wird einfach ein neuer Speicher angelegt. Der Befehl lässt den Originalinhalt des alten Speichers unberührt. Er gibt dann als Rückgabewert den Zeiger der neu erstellten Speicher-Bank zurück. Der nächste wichtige Befehl währe **FreeMemory()**. Hier gibt es nicht viele Worte zu verlieren: Er löscht schlicht und ergreifend den Speicher und gibt ihn wieder frei zur Verwendung. Dann währe da noch **CopyMemory()** wichtig. Er ist dazu da, um Daten von einem Speicher-Buffer in den nächsten zu kopieren. Als ersten Parameter erwartet dieser Befehl den Quelldatenbuffer, als Zweiten den Zieldatenbuffer. Als letzten Parameter erwartet dieser Befehl die Anzahl der zu kopierenden Bytes. Als letztes erwähnendwerte für den Anfang währe vielleicht noch **CompareMemory()**. Dieser Befehl vergleicht 2 Speicherbereiche und gibt 1 zurück, falls die beiden Bereiche übereinstimmen. Der erwartet die gleichen Parameter wie CopyMemory. Die restlichen Befehle sind mehr zur Stringmanipulation zu gebrauchen und daher für den Anfang eher unwichtig. Du kannst sie bei Bedarf auch jederzeit in der Hilfe nachschlagen.

So, ich würde sagen, dass dies den Kapitel zum Speicherhandling abschließt.
Ich weiß manches, vor allem die Zeiger sind schwierig zu verstehen, aber ich hoffe du hast das trotzdem irgendwie geschafft ;-). Falls du etwas noch nicht verstanden hast, oder ich etwas falsch oder unvollständig beschrieben haben sollte-> E-Mail an mich! So, jetzt bleibt mir erstmal nur, dir viel Spaß beim Programmieren zu wünschen :-)