

*KLAUS MEYER-WEGENER
GIA SURGULADZE*

GRUNDLAGEN DER PROGRAMMIERUNG

*(Einleitung in die Programmier-
sprachen C/C++/C#)*

T.2



*Erlangen–Nürnberg–Tbilissi
2007*



PROF. DR. KLAUS MEYER-WEGENER

Lehrstuhlsleiter "Datenbanksysteme"
an der Universität Erlangen-Nürnberg.

Forschungsinteressen:

- Multimedia-Datenbanken
- Dokument- and Prozessverwaltung (Workflow Management)
- Digitale Bibliotheken
- Objektrelationale, objektorientierte verteilte Datenbanksysteme (Oracle)

E-mail:

Klaus.Meyer-Wegener@informatik.uni-erlangen.de



PROF. DR. GIA SURGULADZE

Lehrstuhl MIS an der GTU, Tbilissi.
Sonderforschungsstipendiat DAAD an der
Universität Erlangen-Nürnberg (1991-
2003), Humboldt Universität Berlin (2000-
2004).

Forschungsinteressen:

- UML, OO-Programmierung
- Relationale verteilte Datenbanken
- Petrinetzen, PNML
- .NET Technologie, C#, ADO, ASP

E-mail:

gsurg@gmx.net

INHALTSÜBERSICHT

<i>Einleitung</i>	<i>7</i>
<i>Teil II. Einfache Elemente der Programmiersprache C</i>	<i>9</i>
<i>Lektion 5: EINFÜHRUNG IN DER PROGRAMMIERSPRACHE „C“</i>	<i>9</i>
<i>Lektion 6: OPERATIONEN</i>	<i>20</i>
<i>Lektion 7: OPERATOREN</i>	<i>25</i>
<i>Lektion 8: ZEIGER UND ADRESSE</i>	<i>29</i>
<i>Lektion 9: PROGRAMMSCHLEIFEN</i>	<i>32</i>
<i>Lektion 10: UMSCHALTENANWEISUNG</i>	<i>36</i>
<i>Lektion 11: FENSTERVERWALTUNG</i>	<i>41</i>
<i>Lektion 12: VORÜBERSETZER</i>	<i>47</i>
<i>Lektion 13: SYMBOLPRÜFUNGS- UND TYPEN- UMWANDLUNGSOPERATIONEN</i>	<i>51</i>
<i>Lektion 14: DATENWECHSELOPERATIONEN</i>	<i>53</i>
<i>Lektion 15: FUNKTIONEN</i>	<i>59</i>
<i>KONTROLLFRAGEN UND ÜBUNGEN</i>	<i>63</i>
<i>LITERATURVERZEICHNIS</i>	<i>64</i>

INHALTSVERZEICHNIS

<i>Einleitung</i>	7
<i>Teil II. Einfache Elemente der Programmiersprache C</i>	9
<i>Lektion 5: EINFÜHRUNG IN DER PROGRAMMIERSPRACHE „C“</i>	9
<i>5.1. Programmstruktur und Hauptelementen</i>	
<i>5.2. Formatierte Dateneingabe und Ausgabe.</i>	
<i>5.3. Umbenennung der Datentypen (typedef)</i>	
<i>5.4. Das Unterprogramm</i>	
<i>5.5. Standarte Unterprogramme</i>	
<i>Lektion 6: OPERATIONEN</i>	20
<i>6.1. Zeichenketten, Ein-/Ausgabe</i>	
<i>6.2. Die Funktion: strcat() und strlen()</i>	
<i>6.3. Zeichenweise Ein- und Ausgabe</i>	
<i>6.4. Das Zeichen EOF</i>	
<i>6.5. Konvertierungsoperationen</i>	
<i>6.6. Spaltengerechte ausgabe mit printf()</i>	
<i>6.7. Arithmetische Operationen (+ , - , * , / , %)</i>	
<i>Lektion 7: OPERATOREN</i>	25
<i>7.1. Inkrement- und Dekrementoperatoren</i>	
<i>7.2. Spezielle Operatoren</i>	
<i>7.3. Bedingungen (if - else)</i>	
<i>7.4. Vergleichsoperatoren</i>	
<i>7.5. Logische Operatoren</i>	
<i>Lektion 8: ZEIGER UND ADRESSE</i>	29
<i>8.1. Zeiger (Verweise, Pointer)</i>	
<i>8.2. Definition von Zeigern</i>	
<i>8.3. Der Inhaltsoperator „,*“</i>	
<i>Lektion 9: DIE PROGRAMMSCHLEIFEN</i>	32
<i>9.1. while - Schleife</i>	
<i>9.2. for - Schleife</i>	
<i>9.3. do while - Schleife</i>	
<i>9.4. Rekursive Funktionen</i>	

Lektion 10: UMSCHALTENANWEISUNG	36
10.1. <i>switch</i> – Anweisung	
10.2. Programmbeispiele	
Lektion 11: FENSTERVERWALTUNG	41
11.1. Die Funktion für Fensterverwaltung	
11.2. Programmbeispiele	
11.3. Multifensterprogramm	
11.4. Graphische Windows	
Lektion 12: VORÜBERSETZER	47
12.1. Der Vorübersetzer (#. . .)	
12.2. Präprozessor – Anweisungen	
12.2.1. #define Konstanten	
12.2.2. #define Makros (mit Parametern)	
12.3. Programmbeispiele für den Vorübersetzer	
Lektion 13: SYMBOLPRÜFUNGS- UND TYPEN- UMWANDLUNGSOPERATIONEN	51
13.1. Symbolprüfungsoperationen	
13.2. Typenumwandlungsoperationen	
Lektion 14: DATENWECHSELOPERATIONEN	53
14.1. Wechseloperationen zwischen Funktionen	
14.1.1. Datenwechsel (Datenübertragen) zwischen der zwei Programme	
14.1.2. Datenwechsel mittels Datenadressübertragen	
14.2. Kontrolloperationen für die Unterbrechens- und Hilfstasten	
Lektion 15: FUNKTIONEN	59
15.1. Funktionen	
15.1.1. Typ einer Funktion	
15.1.2. Rückgabewert	
15.2. Zufallsvariablenfunktion	
15.3. Resultatsausgabeprozedur	
KONTROLLFRAGEN UND ÜBUNGEN	63
LITERATURVERZEICHNIS	64

Teil III. Komplexe Elemente der Programmiersprache C

Lektion 16: DATEIEN (Array data)

Lektion 17: ELEMANTARE MATRIZOPERAZIONEN

Lektion 18: DIE CRAMER'SHE REGEL (DETERMINANTENVERFAHREN)

Lektion 19: GAUSSCHES ELIMINATIONSVERFAHREN

Lektion 20: STRUKTUREN

Lektion 21: FILES

21.1: Files mit sequentiellm Zugriff

21.2: Files mit direktem Zugriff

21.3: Beispiele

EINLEITUNG

Dieses Buch hat sich zum Ziel gesetzt, den Student(inn)en eine Einführung in C zu bieten, die noch keine oder eine geringe Programmiererfahrung haben. Es werden lediglich die grundlegenden Kenntnisse im Umgang mit dem Betriebssystem gefordert.

Die Sprache C wurde ursprünglich dazu entwickelt, Betriebssysteme zu schreiben und richtet sich deshalb an fortgeschrittene Programmierer. Wenn Sie wenig oder keine Programmiererfahrung haben, ist es sehr wahrscheinlich, dass Sie nicht alles auf Anhieb verstehen.

Im Buch werden grundlegende Konzepte der Programmierung vorgestellt. Nach einer kurzen Einführung (I-Teil) in die Begriffswelt «Informatik» und Architektur von Rechenanlagen und Programmen werden am Beispiel der Programmiersprache C die Grundkonzepte (II-Teil) von Programmiersprachen eingeführt:

*Datentypen,
Operatoren,
Ausdrücke,
Ablaufkontrolle (Schleifen und Switch),
Rekursion,
Funktionen und
Zeiger und Felder.*

Dann werden komplexe Elemente der Programmierung-C vorgestellt (III-Teil), und zwar:

*Dateien,
Matrizen (einfache, kramerische, gauss),
Strukturen,
Programmstruktur,
Modularisierung,*

Datenorganisations- und Suchverfahren für die sequentiellen (Sequenzzugriff) und direkten Dateien (Direktzugriff).

Im Rahmen der Übungen soll der praktische Einsatz der in der Vorlesung eingeführten Konzepte anhand kleiner Programmierbeispiele geübt werden.

Den Schwerpunkt des C++ Kapitels bilden objektorientierte Analyse und Design. Ihr Ziel ist der Entwurf von Klassen und Objekten mit ihren Merkmalen und Beziehungen.

Hinweise zu graphischen Notationen (in Unified Modelling Language) und Codierungsregeln (mit CASE Werkzeuge) erleichtern die gleichzeitige Erstellung der Dokumentation.

In der traditionellen Software-Entwicklung sind Datenflußanalyse, Strukturierte Analyse und funktionale Dekomposition weitgehend akzeptiert. Beim Datenbankentwurf wird Analyse und Design mittels Entity-Relationship-Diagrammen und Normalisierungsverfahren durchgeführt.

Die Schwerpunkte der visuelle, objektorientierte Programmierung werden mittels Werkzeuge Borland C++ Builder und Visual Studio .Net, und zwar C#.NET für eigene Entwicklungen in den wichtigsten Bereichen und Technologien betrachtet:

- Windows-Anwendungen (Applicationen),*
- Webanwendungen,*
- Objektorientierte Programmierung,*
- Komponentendesign u.s.w.*

Teil I: Die einfache praktische Fragen

Lektion 5

5.1. PROGRAMMSTRUKTUR UND HAUPTELEMENTEN

Ein C-Quellprogramm besteht aus einer Folge von Funktionsdefinitionen. Jede Funktionsdefinition besteht aus:

```
/* ----- Kommentaren ----- */
/* Kopf */
#include <stdio.h>
#include <...>
#define K 100
#define ...

Ergebnistyp Funktionsname ()
{ /* Rumpf */
  Parameterliste
  ...
  Operatoren
  ...
}
```

Abb.2.1

- Ergebnistyp,
- Funktionsnamen,
- Parameterliste,
- Rumpf (s. Abb.2.1).

Eine der Funktionen muß **main** heißen. Sie wird als Hauptprogramm benutzt, das die anderen Funktionen als Unterprogramme aufrufen kann.

Z.B.: das „minimale“ C-Programm.

C-Quellprogrammtext wird mit einem Editor (z.B., tc.exe in dem Verzeichnis C:\TC) vorbereitet.

<pre>/* p l c program myfirst */ #include <stdio.h> main() { printf("hello world\n"); }</pre>	<p>/*.....*/ ist der Kommentar Standard Input/Output Header File</p> <p>begin Anweisung gibt "Text" aus End</p>
---	---

Abb.2.2. „minimale“ C-Programm

Dann bekommen wir z.B. Myfirst.c programmtext. Für den Aufruf des C-Übersetzers (oder Kompailer = compiler) brauchen wir in dem Systemmenü mit F10 und „→“ ein Compile auszuwählen.

Damit wird ein Objektmodul in Maschinensprache id der Datei Myfirst.obj erzeugt.

Das Laufzeitsystem mit den Standard-Funktionen (z.B. printf())

muß noch hinzugebunden werden, damit sich ein lauffähiges Programm (.exe) ergibt. Dazu muß die Bibliothek mit dem Laufzeitsystem angegeben werden (mit F10, Compile → Make Exe File), und anschließend kann der Binder (Linker) aufgerufen werden.

Er erzeugt die Datei Myfirst.exe.

Die Ausführung selbst veranlast mit: F10, Run, oder ctrl+F9.

Ergebnistyp für **main()**:

```
z.B. int main()      /* gibt ergebnis mit Dezimal typ; */  
     float main()   /* gibt ergebnis mit float typ; */  
     . . .  
     void main ()   /* das bedeutet resultat ohne typ. */
```

Abb.5.1. zeigt ein Beispiel ohne typ, deshalb ist er weggelassen. Im „C“ kann man ohne void schreiben, im „C++“ es ist notwendig !
Der Rumpf einer Fnkionsdefinition eingeschloß en in geschweiften „{ }“ Klammern und besteht aus:

- Definition von lokalen Variablen,
- Anweisungsfolge.

Z.B.:

```
#include <stdio.h>  
main()  
{  
int a;  
a=2003;  
printf(„Jahr=%d“, a);  
}
```

Datentypen in C:

int - integer; Definition als short, long oder unsigned (z.B. int money; besitzt im Hauptspeicher 2 Bytes , long int temp; 4 Bytes).

char - character; ein ASCII-Zeichen (z.B. char letter, string; besitzt 1 Byte).

float - gleitkommazahl, besitzt meist 4 bytes (z.B. float koko =27.15;).

double - doppelt genaue Gleitkommazahl, besitzt meist 8 Bytes (z.B. double destik =2.15 E15 ; /* (E15=10¹⁵) */.

Allgemeine Regel : Datentyp, gefolgt von einem oder mehreren Variablenamen, getrennt durch Kommata (,), Abschluß durch Semikolon (;).

```
Z.B. : int score ;
      float average ;
      char grade ;
```

Die Variablen können bei der Definition initialisiert werden. Variablen des gleichen Typs können in der einzigen Anweisung Angegeben werden !

```
Z.B.: int recno=20, key, dit =0;
      float cici=8.221734E-9;      /* 10-9 */
      char *name =“Gwanza“;      /* „*“ - ist Pointer */
      char bell ='\007';          /* symbol-klings */
      double kap, kup, kop;
```

Die Konstanten werden durch die **#define** –Anweisung definiert.

```
Z.B.: #include <stdio.h>
      #define NAME “Giorgi”
      #define PI 3.14
      #define AGE 20
```

NAME, PI und AGE sind Konstante. Sie dürfen nicht im Programm ändern. Variablen können ändern.

Beispielprogramm:

```
/* p2.c sizeof(type) und strlen() */
#include <stdio.h>
#define STR “SOSO”
```

```

main ()
{
  clrscr();
  printf(„      TYPE\n—————\n“);
  printf(„daten mit char besitzen %d bytes\n“, sizeof(char) );
  printf(„daten mit int besitzen %d bytes\n“, sizeof(int) );
  printf(„daten mit long besitzen %d bytes\n“, sizeof(long) );
  printf(„daten mit double besitzen %d bytes\n“, sizeof(double) );
  printf(„\n\n“);
  printf(STR);
  printf(„Die Länge dieser Zeile ist %d oder %d !\n“,
        sizeof STR, strlen(STR));

  getch();
}

```

Abb.2.3. Programmtext mit Daten-Typen

Resultat :

daten mit char besitzen 1 bytes
daten mit int besitzen 2 bytes
daten mit long besitzen 4 bytes
daten mit double besitzen 8 bytes

SOSO

Die Länge dieser Zeile ist 5 oder 4 !

Die Funktion `sizeof` gibt das Ergebnis 5 und `strlen()` – 4 aus, weil in der Hauptspeicher die Zeile „SOSO“ als 5 Symbole gespeichert ist.

S	O	S	O	\0
---	---	---	---	----

`strlen()` ist als spezielle Funktion für Verarbeitung der Zeilen (strings). Sie läßt außer Betracht das letzte Zeichen.

Durch nächste Tabelle werden alle Wertebereiche für die Variablen und Konstanten gegeben.

WERTEBEREICH DER HAUPTDATENTYPEN

N	Datentype	Bytes in Speicher	Diapasone	Unsigned	Spezifikator
1	char	1	-128..127	0..255	%c,%s
2	int	2	- 32768 .. 32767	0..64536	%d,%u
3	long	4	-2147483648 ... 2147483647	0..4294967245	%ld
4	float	4	$-10^{38} \dots 10^{38}$	-	%f
5	double	8	$-10^{307} \dots 10^{307}$	-	%f,%e

Jetzt wird ein Programm mit den *unsigned* Anweisung illustriert.

```

/* p3.c unsigned */
#include <stdio.h>
main()
{
    int i=32767;
    unsigned int j=32767;
    printf(<%d %d %d\n>,i,i+1,i+2);
    printf(<%u %u %u %u %u %u\n>,j,j+1,j+2,j+32768,
j+32769,j+32770);
    getch();
}

```

Abb.2.4. Programmtext mit unsigned Anweisung

5.2. FORMATIERTE DATENEINGABE UND AUSGABE

(scanf (), printf ())

Allgemeine Struktur für die Ausgabe / Eingabe Anweisungen :

```
printf(„Verwaltungszeile %d\n“, Variable_Name);
```

Z.B., z=17;

```
printf(„%d ist %s\n“,z,„mein Lebensalter“);
```

Resultat: 17 ist mein Lebensalter.

```
scanf(„spezifikator des Variables“, [&]Var_Name);
```

/* „&“ nur für Zahlen */

z.B: `scanf(“%s%d%f“, Name, &MonatsNum, &Gehalt); /* Eingabe */`
`printf(“HauptName=%s, Monat=%d, Geld=%.2f Eu\n“,`
`Name, MonatsNum, Gehalt); /*Ausgabe*/`

Resultat: HauptName=Giorgi, Monat=10, Geld=100.55 Eu.

Es wird ein Dialogprogramm mit Eingabe/Ausgabe Anweisungen betrachtet.

```

/* p4.c Dialog*/
#include <stdio.h>
#define VORNAME “Soso”
main()
{ char Name[15];
  int Alter;
  float Gehalt;
  clrscr();
  printf(„Guten Tag %s, wie ist Deine Name ? ,, VORNAME);
  scanf(“%s”, Name);
  printf(“Wie Alt bist Du ? “); scanf(“%d”, &Alter);
  printf(„Geld = ,,); scanf(„%f“, &Gehalt);
  printf(„\n\n Vorname Name Lebensalter Gehalt\n“);
  printf(„%-10s %-15s %-5d %10.2f\n“,
         VORNAME, Name, Alter, Gehalt);
}

```

Abb.2.5. Ein Dialogprogramm

Das nächste Programm zeigt den gleichen Dialog mit kompakten Darstellung der Eingabe-Ausgabe Prozeduren.

```

/* p5.c Dialog-kurz */
#include <stdio.h>
main()
{ char Name[15]; int Age; float Salary;
  printf(“input the date: Name, Age, Salary\n”);
  scanf(“%s%d%f”, Name, &Age, &Salary);
  printf(“\n Results: %s %d %4.2f\n”, Name, Age, Salary);
}

```

Abb.2.6. Kompaktes Dialogprogramm

5.3. UMBENENNUNG DER DATENTYPEN (typedef)

Umbenennung der Datentypen bedeutet Verwendung der alten Typen mit neuen Namen (z.B. für die semantische (inhaltliche) Darstellung).

```
/* p6.c typedef */
#include <stdio.h>
main()
{ typedef float Geld ; /*Geld ist äquivalent als float */
  typedef char *Zeile; /*Zeile ist äquivalent als char mit Pointer */
  Geld Lari=1000, Dollar, Euro;
  Zeile a="Soso", b="Georg";
  Dollar = Lari/2.15;
  Euro = Lari/2.40;
  printf("%s hat %.2f $ \n", a, Dollar);
  printf("%s hat %.2f Eu\n", b, Euro);
  printf(.,\n25/11/2003\n");
}
```

Abb.2.6. Programm für Umbenennung der Datentypen

5.4. DAS UNTERPROGRAMM

Eines der wichtigsten Hilfsmittel zur Rationalisierung des Programmierens ist das Unterprogramm (auch Subroutine genannt). Im Prinzip handelt es sich um eine Befehlsfolge, die, obwohl an verschiedenen Stellen eines Programmes benötigt, doch nur ein einziges Mal programmiert und auch nur einmal gespeichert wird.

Immer dann, wenn diese Befehlsfolge im Verlaufe eines Programmes eingesetzt werden soll, befindet sich an der betreffenden Stelle des Hauptprogrammes ein Sprungbefehl zum ersten Befehl des Unterprogrammes.

Sind die Befehle des Unterprogrammes durchlaufen, so muß der letzte Unterprogrammbefehl wiederum ein Sprungbefehl in das Hauptprogramm sein.

```

Z.B.: /* p7.c  Unterprogramme */
#include <stdio.h>
#define START "Die Stafette beginnt"
main()
{ clrscr();
  printf("%s\n", START);
  printf("Ich bin main und rufe Gwanza an\n");
  Gwanza(); /*UnterprogrammsName für Sprung */
  Hanns();
  printf(„Das Ende der Stafette“);
}

/*———Unterprogramme——*/
Gwanza() /*Erstes*/
{
  printf(„Ich bin Gwanza und rufe Gio an\n“);
  Gio();
}
/*—————*/
Gio() /*Zweites*/
{
  printf(„Ich bin Giorgi, Hello !\n“);
  return(0); /*zurückspringen*/
}
/*—————*/
Hanns() /*Drittes*/
{
  printf(„Ich bin Hanns,Gruss aus Berlin !\n“);
}

```

Abb.2.7. Das Beispiel für die Unterprogrammen

In der Regel wird ein Unterprogramm für die Verarbeitung bestimmter Daten (Operanden) vorgesehen sein und möglicherweise auch noch einige Parameter für diese Verarbeitung benötigen.

Arbeitsdaten und Parameter müssen vom Hauptprogramm jeweils von neuem so zur Verfügung gestellt werden, dass sie für das Unterprogramm erreichbar sind. Erst dann kann der Sprung

in das Unterprogramm selbst erfolgen.

Umgekehrt muss das Hauptprogramm zu den vom Unterprogramm gelieferten Ergebnissen Zugang haben.

Z.B.

```
/* p8.c Unterprogramme mit Parameter */
#include <stdio.h>
#define A 15
int b=25;
main ()
{
    int a, b, c;
    a=5; b=7; c=a+b;
    printf("A=%d a=%d b=%d c=%d\n", A, a, b, c);
    UnterPr1(a,b,c);
    UnterPr2(a,c);
    getch();
}
```

```
UnterPr1(m,n,k)
{
    int p; /*p=24*/
    p=m+n+k;
    printf("p=%d\n", p);
}
```

```
UnterPr2(u,v)
{
    int q;
    q=u+b+v; /*5+25+12=42*/
    printf(„q=%d\n“, q);
}
```

Abb.2.8. Unterprogrammen mit Parameter

Resultats:

A=15 a=5 b=7 c=12

p=24

q=42

5.5. STANDARDER UNTERPROGRAMME

Unterprogramm für allgemeine Probleme, z.B. für Lösung von Aufgabe aus der Mathematik, werden von den Herstellern von DVA zur Verfügung gestellt (z.B. `abs()`, `sin()`, `sqrt()` ...).

Jeder Programmierer kann aber darüber hinaus für seine Bedürfnisse weitere Unterprogramme schreiben.

Die Gesamtheit aller für einen DVA-typ verfügbaren Unterprogramme bezeichnet man als Unterprogramm-bibliothek (oder Standardprogramm-bibliothek).

```
/* p9.c Die Standardprogramme */
#include <stdio.h>
#include <math.h>
main ( )
{
    int a=2, b=34;
    float c, f;
    c=sin((a*b-a*a)/2);
    f=sqrt(a*b*c);
    printf("c=%.2f f=%.2f\n",c,f);
    getch ( );
}
```

Abb.2.9. Beispiele für die Standardprogramme

Oft stellen sich auch Anwender von DVA Bibliotheken mit eigenen Unterprogrammen für anwendungsspezifische Probleme (z.B. ökonomische, medizinische, statistische u.s.w.), die in verschiedenen Hauptprogrammen austauschen können, zusammen.

In wesentlichen lassen sich folgende Gruppen innerhalb einer Programm-bibliothek unterscheiden:

- die Quellprogramm-Bibliothek,
- die Modulbibliothek und
- die Bibliothek der ladbaren Programme.

Programmmodul (Modulbibliothek).

Manche Übersetzungsprogramme liefern das übersetzte Programme noch nicht in der ladbaren und damit ablauffähigen Form, sondern als folge sogenanter Programmmodule.

Vor dem Ablauf müssen diese Module gebunden (durch Binder=Linker) werden. Beim Binden ist es möglich, in unterschiedlichen Übersetzungsläufen gewonnene Programmmodule zu einem ablauffähigen Programm zu verbinden.

Die Übersetzungsprogramme unterschiedlicher Programmier-sprachen liefern Module mit gleichen Aufbau. Deshalb kann man verschiedene Teile eines Programmes in unterschiedlichen Sprachen schreiben, diese Teile in verschiedenen compilers übersetzen und die gewonnenen Programmmodule über den Binder (linker) zu einem Programm zusammenfügen.

Lektion 6

6.1. ZEICHENKETTEN, EIN-/AUSGABE

Ein String (Zeichenkette) ist definiert als Folge von Zeichen (Bytes im Speicher), die mit '\0' (NULL) abgeschlossen wird.

Eine Definitionsmöglichkeit eines Strings:

```
char string[]="Unix und C";
```

Dabei wird exakt der benötigte Speicherplatz belegt:

U	n	i	x			u	n	d			C	\0
---	---	---	---	--	--	---	---	---	--	--	---	----

Die NULL wird bei dieser Initialisierung automatisch erzeugt.

Beispiele:

```
char string[10];  
string="LINUX"; /* verboten !!! */  
string[0]='L';  
string[1]='I';  
string[2]='N';  
string[3]='U';  
string[4]='X';  
string[5]='\0';
```

string ist keine Variable vom Typ Array (sondern eine Konstante).

Nur *string[i]* sind Variable (vom Typ char).

Deshalb:

```
char string1[10], string2[10];  
string1=string2; /* verboten ! */  
string1[0]=string2[0];  
string1[1]=string2[1];
```

u.s.w.

6.2. DIE FUNKTION `strcat()` UND `strlen()`

➤ `strcat()` kopiert einen String an das Ende eines anderen Strings.

➤ `strlen()` gibt die Länge eines Strings zurück. Das NULL-Zeichen am Ende zählt nicht mit.

Beispielprogramm:

```
/* p10.c strcat */
#include <stdio.h>
main ()
{ static char teil1[80]="Drei Chinese";
  static char teil2[]="mit dem Kontrabass";
  strcat(teil1, teil2);
  printf(„%s\n“, teil1);
  printf(„lange=%d\n“, strlen(teil1));
}
```

Abb.2.10. Beispiel für die Funktion

6.3. ZEICHENWEISE EIN- UND AUSGABE

Mit den Funktionen `getchar()` und `putchar()` können einzelne Buchstaben gelesen und geschrieben werden.

<pre>/* p11.c getchar */ #include <stdio.h> main() { char ch; ch=getchar(); putchar(ch); }</pre>	Vereinfachung →	<pre>/* p12.c putchar - getchar */ main() { putchar(getchar()); }</pre>
--	--------------------	---

6.4. DAS ZEICHEN EOF

In der `stdio.h` wird ein spezielles Zeichen, das das Ende einer Datei markiert, als Konstante EOF (End-Of-File) vereinbart.

In UNIX-System erzeugt die Tastenkombination 'ctrl-D' bei Eingabe am Bildschirm das Dateiende. In MS DOS-System 'ctrl-z'.

Beispielprogramme:

```
/*p13.c Kopieren aller Zeichen der Eingabe  
auf die Ausgabe und Zählen der Zeichen*/  
#include <stdio.h>  
main()  
{  
  char ch;  
  int count=0;  
  while((ch=getchar()) !=EOF)  
  {  
    count++;  
    putchar(ch);  
  }  
  printf(,)\n%d Buchstaben wurden gelesen.\n“, count);  
}
```

Abb.2.11. Beispiel für das Zeichen EOF

Ausführung:

```
abcdef^Z  
abcdef  
6 Buchstaben wurden gelesen.
```

6.5. KONVERTIERUNGSOPERATIONEN

%d - int, dezimal mit Vorzeichen,
%o - int, oktall ohne Vorzeichen,
%x, %X - int, hexadezimal ohne Vorzeichen,
%u - int, dezimal ohne Vorzeichen,
%c - char, einzelnes Zeichen,
%s - char, *(string), Ausgabe bis '\0',
%f - float, double als [-] mmm.ddd,
%e, %E - double als [-]m.ddddd[±]xx,
%p - void *, als Zeiger.

6.6. SPALTENGERECHTE AUSGABE MIT printf()

```
/*p14.c Datenausgabe */
#include <stdio.h>
main()
{
    int d=45, k=15360, t=4455;
    clrscr();
    printf(„ %6d %6d %6d\n“, d,k,t);
    printf(„ %6d %6d %6d\n“, k,t,d);
    printf(„ %6d %6d %6d\n“, t,d,k);
    getch();
}
```

Abb.2.12

Ausführung:

```
    45 15360  4455
15360  4455    45
 4455    45 15360
```

6.7. ARITHMETISCHE OPERATIONEN

(+ , - , * , / , %)

```
/*p15.c Arithmetische Operationen-1, +, -, *, /, % */
#include <stdio.h>
main()
{ int n=1, a=100, b=50;
  printf(“%d\n”, 5+a);
  while(n<10)
    { printf(“%10d %d\n”, n, n*n);
      n++; }
  printf(„5/3 = %d\n“, 5/3);
  printf(„5.0/3 = %.2f\n“, 5.0/3);
  printf(„\n5%%3 = %d\n“, 5%3); /* 5 mit Module 3 gibt den Rest
der Division */
}
```

Abb.2.13

```

/* p16.c Arithmetische Operationen-2, Umwandlung
                                     sec =>> hour, min, sec */
#include <stdio.h>
#define SM 60
main()
{ int sec, min, hour, rest;
  printf("Input seconds= ");
  scanf("%d", &sec);
  min=sec/SM; rest=sec%SM; hour=min/SM;
  min=min%SM;
  printf(, "%d seconds = %d hour, %d min, %d sec\n",
                                     sec, hour, min, rest);
}

```

Abb.2.14

```

/* p17.c Arithmetische Operationen-3, abrunden bis ganze Zahl
                                     ceil(), floor(); Bedingte Anweisung if... else... */
#include <stdio.h>
#include <math.h>
main()
{ double a; int b, c;
  a=54.47567;
  b=ceil(a); /*Result 55*/
  c=floor(a); /*Result 54*/
  printf("a=%f b=%d c=%d\n", a, b, c);
  if(b>c)
    printf("2*a=%0.2f\n", 2*a);
  else
    printf(, "Tschuß !!!\n");
  c = (a>b)? 100 : 555;
  printf("c=%d\n", c);
}

```

Abb.2.15

Lektion 7

7.1. INKREMENT- UND DEKREMENTOPERATOREN

Sehr oft ist es notwendig, den Wert einer Variablen um 1 ein zu erhöhen oder erniedrigen.

Hierfür gibt es spezielle Operatoren, jeweils in einer Präfix- und einer Suffix-Notation.

• $++z$ Addiert 1 zu z vor der Nutzung von z (Präfix);
entspricht $z = z + 1$.

• $--z$ Subtrahiert 1 von z vor der Nutzung von z (Präfix);
entspricht $z = z - 1$.

• $z++$ Addiert 1 zu z nach der Nutzung von z (Postfix);
entspricht $(z=z+1) - 1$.

• $z--$ Subtrahiert 1 von z nach der Nutzung von z (Postfix);
entspricht $(z=z-1) + 1$.

Die Inkrement- und Dekrementoperatoren haben Präzedenz vor allem anderen Operatoren, mit Ausnahme der Klammern. Also

$X * Y++$, z.B., $X=3, Y=2$ bedeutet $X * (Y++)=6$ und nicht $(X*Y)++$ /*Syntaxfehler*/.

Operator	z.B.	Bedeutung
$+=$	$a += b$	$a = a+b;$
$-=$	$a -= b$	$a = a-b;$
$*=$	$a *= b$	$a = a*b;$
$/=$	$a /= b$	$a = a/b;$

7.2. SPEZIELLE OPERATOREN

z.B: $sum = sum + a;$ $sum += a;$

```
/* p18.c SpezOperatoren */
#include <stdio.h>
#define N 10
main()
{
    int i=5, Sum=0, Mult=1;
    while(i < N)
    {
        Sum += i;
        Mult *= i;
        i++;
    }
    printf("Summe=%d \nMultiplizieren=%d\n", Sum, Mult);
}
```

Abb.2.16

Ausführung:

Summe = 35, Multiplizieren = 15 120.

7.3. BEDINGUNGEN (*if - else*)

Die Verzweigungs- und Entscheidungsanweisung hat folgende Syntax:

```
if (expression)
    statement
```

expression ist typischerweise ein Vergleich, kann aber auch eine Anweisung sein.

statement ist eine Anweisung oder eine zusammengesetzte Anweisung mit Klammerung durch die geschweiften Klammern { }.

Die Anweisung von **expression** entscheidet über die Ausführung von **Statement**. Ergibt sich 0 (false), wird **Statement** nicht ausgeführt, oder 1 (True), wird **Statement** ausgeführt.

Für bedingte Operation "if - else" gilt es die Syntax:

```
if (expression)
    statement-1
    else
        statement-2
```

Beide Anweisungen **statement-1** und **statement-2** können auch zusammengesetzte Anweisungen sein.

Es gibt in der praktischen Verwendung so genannte „Geschachtelte if’s“ (if...if... else...if...else...if... u.s.w.).

Weil `if(expression) statement` eine Anweisung ist, kann sie überall verwendet werden, wo eine Anweisung erlaubt ist, zum Beispiel in einer if-else Anweisung.

if-else Gruppierungsregel:

Das **else** bezieht auf das unmittelbar vorher stehende **if**, es sei denn, die Klammerung mit `{}` gibt anderes an.

```
if(expr-1)
{
    if(expr-2)
    {
        statement11;
        statement12;
    }
    else
        if(expr-3)
        {
            statement21;
            statement22;
        }
    else
        statement3;
}
else
    statement4;
```

...

Abb.2.17. Das Fragment für geschachtelte if-else Anweisung

7.4. VERGLEICHOPERATOREN

Es gibt 6 Vergleichsoperatoren, und zwar:

- < - kleiner,
- <= - kleiner oder gleich,
- > - größer,
- >= - größer oder gleich,
- == - gleich,
- != - ungleich.

Die Operatoren werden von links nach rechts gruppiert (assoziiert).

7.5. LOGISCHE OPERATOREN

Die logische Operatoren erlauben die Verknüpfung mehrerer Vergleiche (bzw. Ausdrücke) zu sogenannten Booleschen Ausdrücken.

&& - AND, || - OR, ! - NOT

expr1 && expr2 ist **TRUE**, wenn **expr1** und **expr2** **TRUE** sind.

z.B. $5 > 2 \ \&\& \ 4 > 100$ ist **FALSE**.

expr1 || expr2 ist **TRUE**, wenn **expr1** oder **expr2** oder beide **TRUE** sind.

z.B. $5 > 2 \ || \ 4 > 100$ ist **TRUE**.

!expr1 ist **TRUE**, wenn **expr1** **FALSE** ist.

z.B. $!(4 > 100)$ ist **TRUE**.

Beispielprogramm :

```
/* p19.c Logische Operatoren */
#include <stdio.h>
#define YES 1
#define NO 0
main()
{ int ch; int nl=0, nw=0, nc=0; int word=NO;
  while((ch=getchar()) != EOF)
  { if(ch != ' ' && ch != '\n')
    nc++;
```

```

    if(ch == '\n')
        nl++;
    if(ch != ' ' && ch != '\n' && word == NO)
        { word=YES;
          nw++;
        }
    if((ch==' ' || ch == '\n') && word == YES)
        word = NO;
}
printf("char=%d,words=%d,strings=%d\n",nc,nw,nl);
}

```

Ausführung mit Eingabe: **gia gio**
lia maka
marika
^Z

Resultats: **char=19, words=5, strings=3.**

Lektion 8

8.1. ZEIGER (VERWEISE, POINTER)

Zeiger sind die symbolische Repräsentation einer Adresse.

So wie eine int-Variable eine integer-Zahl aufnehmen kann, gibt es Zeiger-Variablen, die Platz für die Adresse bieten.

*Die Adressoperation „&“ liefert die Adresse einer Variable (z.B. **&gewicht**); Es ist die Adresse der Variablen mit Namen **gewicht**.*

*Wenn zeiger eine geeignete Zeiger-Variable ist, dann wird ihr durch: **zeiger = &gewicht**;*

die Adresse von gewicht zugewiesen, so dass nun 'zeiger auf gewicht zeigt'.

*Der Unterschied zwischen **zeiger** und **&gewicht** besteht darin, dass **zeiger** eine Variable und **&gewicht** eine Konstante ist.*

Als Operanden des Adressoperateurs sind nur solche Ausdrücke erlaubt, die auf der linken Seite einer Wertzuweisung stehen dürfen.

Beispielprogramme:

```
/* p20.c Zeiger */
#include <stdio.h>
main ()
{ int sum;
  char *st;  sum=4+5;
  st="Programmiersprache C&C++\n";

  printf("sum=%d &sum=%p\n", sum, &sum);
  printf("st=%c st=%p\n", *st, st);
  printf("Text-st:%s\n", st);
} /* Ende des Programms 20 */

/* p21.c Zyklus mit Zeiger*/
#include <stdio.h>
main()
{
  int ind;
  char *text;
  text="C&C++ für Professional";
  ind=1;
  while(ind <= 5) /* Zyklus 5-mal */
  {
    printf("%2d : %s \n", ind, text);
    ind++;
  }
} /* Ende des Programms 21 */

/* p22.c Dynamische Verteilung der Speicher */
#include <stdio.h>
#include <alloc.h>
#define ZAHL 3
main()
{
  int *u, i;
  u=(int *) calloc(ZAHL, sizeof(int));
  *u = 123;
  *(u+1) = 456;
  *(u+2) = 2003;
  printf(„,Adresse:“);
  for(i=0; i<ZAHL; i++)
    printf(“%5p”, *(u+i));
  printf(„\n Werte:“);
  for(i=0; i<ZAHL; i++)
    printf(“%5d”, *(u+i));
  printf(“\n”);
} /* Ende des Programms 22 */
```

Abb.2.19. Beispielprogramme
für den Inhaltsoperator

Lektion 9

9. DIE PROGRAMMSCHLEIFEN

9.1. WHILE - SCHLEIFE

Syntax:

```
while(expression)  
statement
```

expression - wird vor Ausführung von **statement** ausgewertet.

statement - kann eine einzelne, eine zusammengesetzte Anweisung (mit {...}) oder die leere Anweisung „ ; “ sein.

statement wird nur ausgeführt, wenn **expression** ungleich 0, also **TRUE** ist. Sonst wird mit der folgenden Anweisung fortgefahren.

Nach der Ausführung von **statement** wird **expression** erneut ausgewertet und ggf. **statement** wieder ausgeführt.

In **statement** sollte etwas getan werden, damit **expression** irgendwann **FALSE** wird. Es kann aber auch die **break** – Anweisung benutzt werden, um die **while** – Schleife zu verlassen:

```
...  
while ((ch=getchar()) != EOF)  
{  
  if (ch == '\n')  
    break;  
  else  
    putchar(ch);  
}
```

9.2. FOR - SCHLEIFE

Syntax:

```
for(initialize; exit-test; update)  
statement
```

es ist equivalent:

```
initialize;  
while(exit_test)  
{  
  statement;  
  update;  
}
```

initialize – wird einmal vor Ausführung von **exit-test** ausgeführt.

Wenn **exit-test** ungleich 0, also **TRUE** ist, wird **statement** einmal ausgeführt. Sonst wird die folgende Anweisung ausgeführt

Nach Ausführung von **statement** wird die **update** - Anweisung durchgeführt.

Die vorherigen zwei Schritte werden solange wiederholt, bis **exit-test** zum ersten Mal mit 0, also **FALSE** bewertet wird.

Jeder der Ausdrücke **initialize**, **exit-test** und **update** kann weggelassen werden.

statement kann auch eine zusammengesetzte Anweisung (mit {...}) oder die leere Anweisung „ ; “ sein.

z.B.:

```
Endlosschleifen  
↓  
while(1)  
{  
  ...  
  break;  
}
```

for(;;) ist immer TRUE und bekommen wir einen unendlichen Zyklus.

/* unendlichen Zyklus kann mit ctrl-C enden */

9.3 DO WHILE - SCHLEIFE

Syntax:

```
do
    statement;
while(expression);
```

es entspricht:

```
statement;
while(expression)
    statement;
```

Die Anweisung ist der While-Schleife sehr ähnlich.

Unterschiede:

- *statement* wird solange ausgeführt, wie *expression* TRUE ist.
- Die *do-while* Schleife wird **stets einmal** ausgeführt, weil *expression* erst nach Ausführung von *statement* bewertet wird.

Programmbeispiele:

```
/*p23.c Schleifen */
#include <stdio.h>
#define NUM 5
main()
{
    int count, index, parol;
    /*_____ while _____*/
    count=1;
    while(count <= NUM)
        { printf(“Wir lernen C&C++! \n”);
          count++;
        }
    /*_____ for _____*/
    for(count=1; count <= NUM; count++)
```

```

    printf("Wunderschön ! Gehen wir weiter \n");
for (index=10; index >0; index— )
    printf("%d seconds!\n");
printf("Ba-a-a-ch\n");

for( ; ; ) /* Endlosschleif */
    printf("Bitte tasten Ctrl-C! \n");
/*-----do while ----- */
do
    scanf("%d", &parol);
    while(parol != 94)
        ; /* leere Operator */
}

```

9.4. REKURSIVE FUNKTIONEN

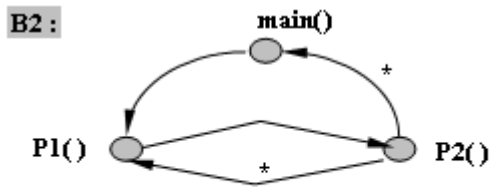
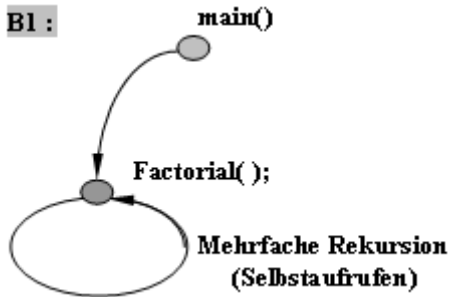
Die Funktion, die direkt oder indirekt selbst sich aufrufen darf, wird als rekursive Funktion verstanden. Durch die Abb.13 werden zwei Beispiele für Rekursion illustriert.

```

/*p24.c   rekursive Funktion */
#include <stdio.h>
double Factorial(int n);
main()
{ int n, double fac;
  printf(Input N : "); scanf("%d", &n);
  fac=Factorial(n);
  printf("%d != %f\n", n, fac);
} /*----- End main() ----- */

double Factorial(int n);
{
  return(n==1)? 1 : n * Factorial(n-1);
}

```



**allgemeines Rekursionsprozeß: P2() – „Child“
kann main() und P1() – „Parents“ aufrufen.**

Abb.2.20. Rekursive Funktionen (sind durch „*“ bezeichnet)

Lektion 10

10.1. SWITCH – UMSCHALTENANWEISUNG

*Vergleichbar der **case** –Anweisung in anderen Sprachen, bietet C mit **switch** eine Mehrfachverzweigung (Abb.2.21).*

Sie könnte auch durch

```

if(...)
    statement1
else
    if(...)
        statement2

```

else
if(..) u.s.w.
erreicht werden.

Syntax:

```
switch(a)      /* a muß schon bekannt sein */  
{  
  case 1:      /* Alternative -1*/  
    statement-1;  
    break;  
  case 2:      /*Alternative-2*/  
    statement-2;  
    break;  
  ....  
  default:    /*Default*/  
    statement-n;  
    break;  
}
```

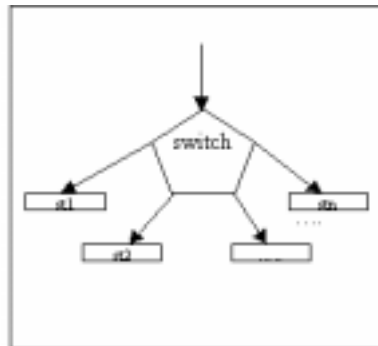
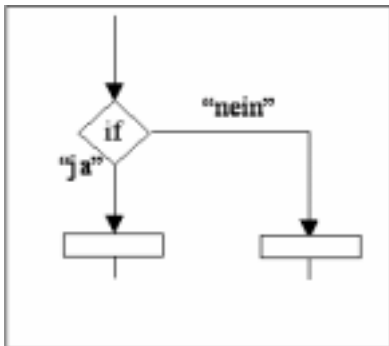


Abb.2.21. Rekursive Funktionen (sind durch „*“ bezeichnet)

Mit switch werden die Alternativen geordnet und auch eine Default-Behandlung vorgesehen.

Mehrere Fälle können auf eine Aktion zurückgeführt werden.

Die Alternativen (case i:) sollte man sich als Marken (Labels) in der Anweisungsfolge des „compound statement“ vorstellen. Wenn der entsprechende Fall eintritt, wird an dieser Stelle mit der Ausführung weitergemacht.

Die Ausführung geht aber nicht nur bis zur nächsten Marke, sondern bis zum nächsten **break** (oder bis zum Ende des „compound statement“).

Mit **break** wird nach Ausführung einer Alternative der switch-Kontrollblock beendet.

Für das Zusammenfassen von Alternativen kann das **break** weggelassen werden:

```
switch(zeichen) /* zi */
{
  case z1:
    .....
  case z2:
    .....
  default: .....
}
```

switch(zeichen) muss als Ergebnis **int** oder **char** liefern. Die case zi:-marken müssen vom typ **int** oder **char** sein.

Z.B. case 27: /* (ASCII für Esc)*/; case 'Y': /* Buchstabe */

10.2. PROGRAMMBEISPIELE:

```
/*p25.c switch() → 1*/
#include <stdio.h>
```

```

main()
{
    char x;
    clrscr();
    while((x=getch()) !=27)        /* Esc */
    switch(x)
    {
        case 'a':
            printf("Ann, Akaki...\n");
            break;
        case 'd':
        case 'e':
            printf("David, Elga....\n");
            break;
        case 'n':
        case 'N':
            printf("Nino, Nikolas...\n");
            break;
        default:
            printf("Bitte, nächsten Buchstabe !\n");
            break;
    }
} /* End */

```

Abb.2.22. Programmbeispiel mit switch()

Jetzt betrachten wir `switch()` Anweisung für die Cursorsteuerung:

```

/*p26.c switch() → 2 */
#include <stdio.h>
#include <conio.h>
#define word "A_U_D_I"
main()
{
    /* 1 */
    int a;

```

```

int i=40,j=10;
clrscr();
gotoxy(i,j);
printf("%s",word);
while((a=getch()) !=27)
{
    /* 2 */
    if(a==0)
        a=getch();
    switch(a)
    {
        /* 3 */
        case 72: /* Up */
            j--; clrscr();
            gotoxy(i,j);
            printf("%s", word);
            break;

        case 75: /* Left */
            i--; clrscr();
            gotoxy(i,j);
            printf("%s",word);
            break;

        case 77: /* Right */
            i++;clrscr();
            gotoxy(i,j);
            printf("%s", word);
            break;

        case 80: /* Dn */
            j++; clrscr();
            gotoxy(i,j);
            printf("%s", word);
            break;
        default: break;
    } /*1*/
} /*2*/
} /*3*/

```

Abb.2.23. switch () Anweisung für Cursorsteuerung

Lektion 11

11.1. DIE FUNKTION FÜR FENSTERVERWALTUNG

Bei der Bearbeitung der Mensch-Rechner Systeme große Bedeutung haben die Funktionen für Fensterverwaltung. Durch Windows-modellierung schafft man ganz originale Benutzerschnittstelle (interface) aufzubauen. Es gibt verschiedene Arte von den solchen Schriftstellen:

- 1. Windows - alls „prompt-menu“ Programme;*
- 2. Graphische Windows und*
- 3. Multiwindows - interface Programme.*

Header-file <conio.h> (consol-input/output) beinhaltet die haupt verwendbaren Funktionen für die Fensterverwaltung, und zwar:

*textbackground(n),
textcolor(n),*

woden n = 0..15;

0 - black	8 - darkgray
1 - blue	9 - lightblue
2 - green	10 - lightgreen
3 - cyan	11 - lightcyan
4 - red	12 - lightred
5 - magenta	13 - lightmagenta
6 - brown	14 - yellow
7 - lightgray	15 - white

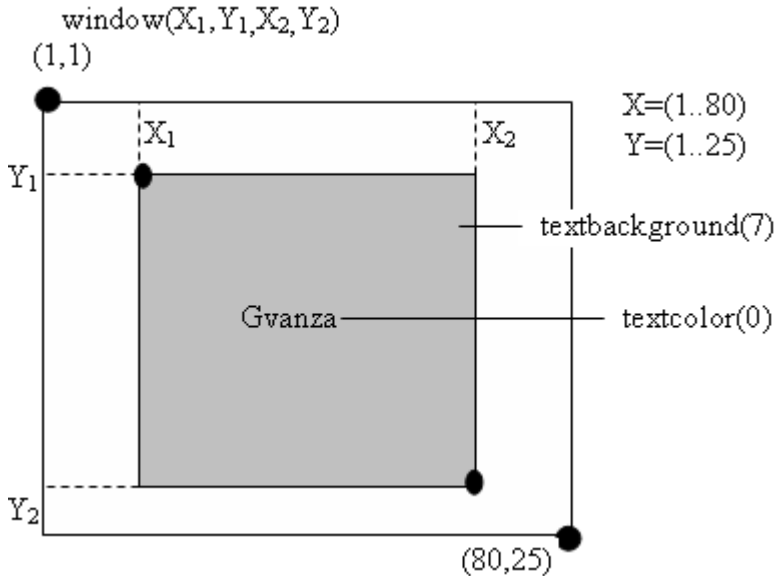


Abb.2.24. Fensterhauptparameter

11.2. PROGRAMMBEISPIELE

```

/* p27.c Menu-prompt */
#include <stdio.h>
#include <conio.h>
#include <io.h>
#define B0 textbackground(0)
#define B1 textbackground(3)
#define B2 textbackground(4)
#define T0 textcolor(15)
#define T1 textcolor(10)
#define T2 textcolor(2)
#define W0 window(1,1,80,25)
#define W1 window(42,20,79,24)
#define C clrscr()

```

```

#define PR_M printf(“%d %s”, i+1,c[i]);
#define MIN 1
#define MAX 3
main()
{
int a, i;
char *b;
char *c[MAX]={“Menu-string_1”,
               “Menu-string_2”,
               “The End    “};

C;
B0;T0;W1;C;
for(i=0; i<3; i++)
{
gotoxy(3,i+1); PR_M;}
B2;T2;
gotoxy(3,1); i=0; PR_M;
while((a=getch())!=13)
{
if(a==0)
a=getch();
switch(a)
{
case 72: /* Up */
B0;T0;gotoxy(3,i+1); PR_M;
i—;
if(i<MIN-1)
i++;
B2; T2; gotoxy(3,i+1); PR_M;
break;
case 80: /* Dn */
B0;T0;gotoxy(3,i+1); PR_M;

```

```

    i++;
    if(i>MAX-1)
        i--;
    B2; T2; gotoxy(3,i+1); PR_M;
    break;
    default: break;
} /*End switch*/
} /*End while*/
gotoxy(1,4); cprintf(“! Enter %d”,i+1);
getch();B0;T0;C;
} /* End main() */

```

Die Aufgabe:

Schreiben Sie, bitte, das Programm „HAUPTMENU“, mit: 5 Menuzeilen; Koordinaten (15,8,65,17); für blinzende Nachrichtzeile „Wellen Sie eine Zeile“ z.B: `textcolor(4+BLINK)`.

Nach der Auswahl der Aufgabe, muss die Nachrichtzeile verschwinden und ausgewähltes Resultat (blinzend) ausgeben.

11.3. MULTIFENSTERPROGRAMM

Bei der Ausführung eines Programms ist es möglich verschiedene Ergebnisse auf die verschiedene Fensters zu verteilen (ausgeben).

Die Benutzeroberfläche (Interface) kann man abhängig von Hauptaufgabe entwerfen.

Z.B. Abb.2.25 zeigt 5 Fensters ($W_i=1..5$).

Jedes Fenster hat sein vordefiniertes Größe, Hintergrund- und Textfarbe.

Alle Fensters haben ihre einzelne Koordinaten, z.B., (1,1). für das Fenster „Gio“, welches befindet sich in der unten-rechten Ecke, existiert auch die (x,y)-Koordinaten mit (1,1) in der linker-oberer Teil des W5-Fensters.

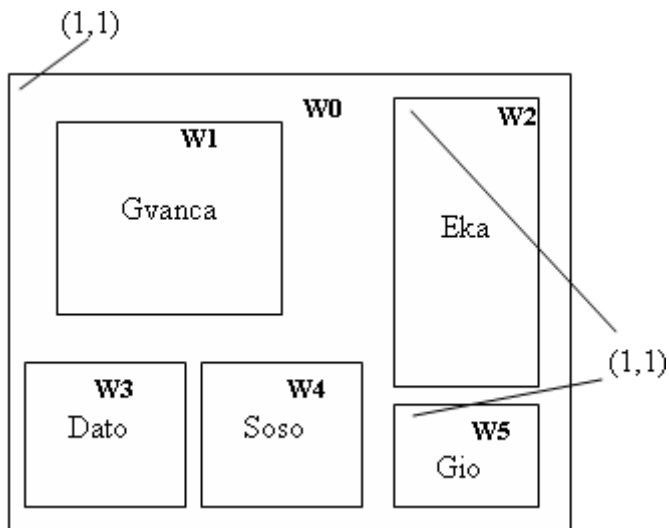


Abb.2.25. Beispiel mit 5 Fensters

Die Aufgabe: schreiben Sie, bitte, das C-Programm für realisierung der Abbildung 19 mit 5 Fensters.

Beispiel des gleichen Programmes:

```

/* p28.c Multifenster */
#include <stdio.h>
#include <conio.h>
#define S1 "Der Zentralkatalog für Datenbank"
#define S2 „die Attribute der Datenbank“
#define S3 „Der Informationsblock“
#define S4 „Das Benutzersarbeitsfeld“
main()
{
    textbackground(0);
    clrscr();
    textcolor(14);
    gotoxy(10,1); printf(“%s\n”,S1);

```

```

gotoxy(50,1); cprintf(“%s\n”,S2);
gotoxy(10,19); cprintf(“%s\n”,S3);
gotoxy(50,19); cprintf(“%s\n”,S4);
textbackground(5); textcolor(15);
window(2,2,39,18); clrscr();
cprintf(“abcdef\n”);
textbackground(2);
textcolor(15);
window(42,2,79,18);
clrscr();
cprintf(“ghijk\n”);
textbackground(4);
textcolor(15);
window(2,20,39,25);
clrscr();
cprintf(“lmnop\n”);
textbackground(1);
textcolor(15);
window(42,20,79,25);
clrscr();
cprintf(“qrstuv\n”);
getch();
}

```

11.4. GRAPHISCHE WINDOWS

Durch die Abbildung 20 ist ein Fensters mit sieben Menü-Block gezeigt. In jedem Block ist eine Aufgabe untergebracht.

Der Benutzer muß eine Aufgabe mit der „?“ und „?“ Tasten auswählen.

Der Hintergrund ausgewähltes Blockes muß „rot“ sein, anderer Blocke –„himmelblau“. Der letzten Block ist für Vollendung („Ende“) der Arbeit mit diesem System .

Programmbeispiel:

```

/* p29.c   graphisches Fenster */
#include <stdio.h>
#include <conio.h>
#include <io.h>
#define B0 textbackground(0)
#define B1 textbackground(3)
#define B2 textbackground(4)
#define T0 textcolor(15)
#define T1 textcolor(7)
#define T2 textcolor(2)
#define W0 window(0,0,80,25)
#define W1 window(20,8,31,16)
#define W2 window(50,8,61,16)
#define C clrscr()
#define C1 "SOSO"
#define C2 "GEORGE"
main()
{
  int a; char *b;
  C;
  B2;T2;W1;C;gotoxy(3,5);
  cprintf("%s\n",C1);
  B1;T1;W2;C;gotoxy(3,5);
  cprintf("%s\n",C2);
  b="GEORGE";
  while((a=getch()) != 27)
  {
    if(a==0)
      a=getch();
    switch(a)
    {
      case 13: /* Enter */
        B0;T0;W0;C;
        printf(,,%s\n",b); break;
      case 75: /* Left */
        B1;T1;W1;C;
        gotoxy(3,5); cprintf("%s\n",C2);
        B2;T2;W2;C;
        gotoxy(3,5); cprintf("%s\n",C1);

```

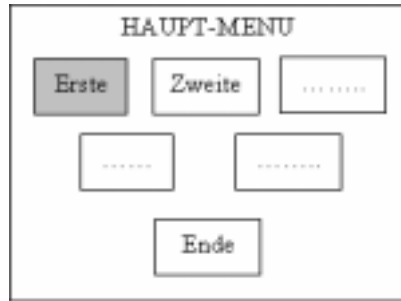


Abb.2.26

```

გაგრძელება
b="GEORGE";
  break;
  case 77: /* Right */
    B1;T 1;W1;C;
    gotoxy(3,5); cprintf("% s\n", C1);
    B2;T 2;W2;C;
    gotoxy(3,5); cprintf("% s\n", C2);
    b="SOSO";
    break;
  default: break
}
}
B0;T0;C;
} // პროგრამის დასასრული

```

Lektion 12

12.1. DER VORÜBERSETZER (#...)

Der Vorübersetzer (engl. *preprocessor*) ist ein Werkzeug zur Generierung und Modifikation von Quellcode. Er interpretiert spezielle, in den normalen C-Code eingestreute Anweisungen (die mit „ # “ beginnen) und modifiziert dabei die Quelldatei nach ganz bestimmten Regeln. Das Ergebnis soll reiner C-Code sein.

Der Vorübersetzer wird vom C-Compiler automatisch aufgerufen, bevor die eigentliche Übersetzung beginnt.

Schema der Übersetzung ist in Abb.2.27 dargestellt.

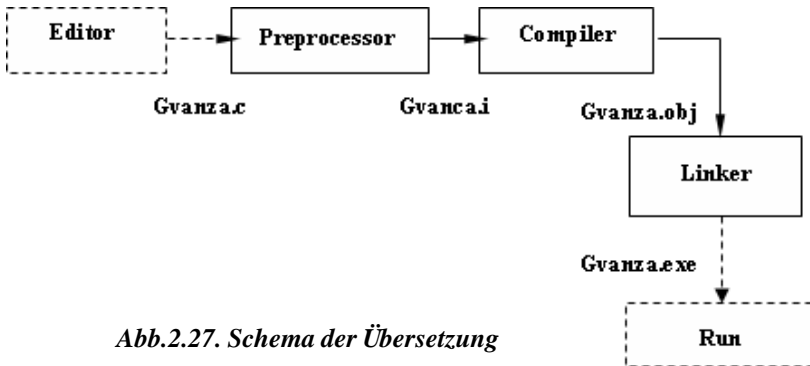


Abb.2.27. Schema der Übersetzung

12.2. PRÄPROZESSOR – ANWEISUNGEN

Präprozessor-Anweisung sind am häufigsten Konstatendefinitionen und das Einkopieren von anderen Quelldateien.

- Eine Präprozessor-Anweisung muß in einer eigenen Zeile stehen, die mit „ # “ beginnt. Unmittelbar dahinter steht der Anweisungsname. „ # “ muß das erste Zeichen der Zeile sein.

- Standardanweisungen für den Präprozessor:
- #define* – Definition einer Konstante oder eines Makros.
- #undef* – Löschen einer Konstante oder eines Makros.
- #include* – Inhalt einer anderen Datei einfügen.
- #if* - Bedingte Einbindung des folgenden Texts bis zum *#endif* oder *#else* aufgrund von Konstantenwerten.
- #ifdef* – Bedingte Einbindung des folgenden Texts bis zum *#endif* oder *#else* aufgrund einer Konstantendefinition.
- #ifndef* – Bedingte Einbindung des folgenden Texts bis zum *#endif* oder *#else* aufgrund einer undefinierten Konstante.
- #else* – Alternative Einbindung des folgenden Texts bis zum *#endif*, falls *#if*, *#ifdef* oder *#ifndef* mit FALSE ausgewertet wurden.
- #endif* – Abschluß von bedingten Text.

12.2.1. *#define* Konstanten

Die Anweisung

***#define* NAME Wert**

kann überall im Quelltext stehen und gilt bis zum Ende der Datei oder bis zu einem *#undef*.

Es gibt zwei Formen der Definition, eine für Konstanten und eine für Makros (mit Parametern).

Beispiele:

#define NAME “Georg”

#define PI 3.14

#define NEUE_ZEILE putchar(‘\n’)

12.2.2. *#define* Makros (mit Parametern)

Syntax:

***#define* name(par-1, . . . , par-n) Folge_Von_Token**

Die Parameterliste kann bei Definition und Aufruf auch leer sein.

Beispiele:

```
#define MAX(a,b) ((a) <= (b)) ? (b) : (a)
```

```
#define STREQ(s1, s2) (!strcmp(s1,s2))
```

12.3. PROGRAMMBEISPIELE FÜR DEN VORÜBERSETZER

```
/* p30.c Vorübersetzer-1 */
```

```
#define TWO 2
```

```
#define MSG „Eile mi Weile !“
```

```
#define FOUR TWO*TWO
```

```
#define PX printf(“X=%d\n”, x)
```

```
#define FMT “X=%d\n”
```

```
main ()
```

```
{ int x=TWO;
```

```
  PX;
```

```
  x=FOUR;
```

```
  printf(FMT,x);
```

```
  printf(„%s\n“,MSG);
```

```
  printf(“TWO:MSG\n”);
```

```
}
```

```
/* p31.c Vorübersetzer-2 */
```

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

```
#include <conio.h>
```

```
#define SQUARE(x) x*x /*manchmal gibt Fehler*/
```

```
#define PR(x) printf(„x %d\n“,x)
```

```
main ()
```

```
{ int x=4; int z;
```

```
  printf(„Richtig !\n“);
```

```
  z=SQUARE(x);
```

```
  PR(z);
```

```
  z=SQUARE(2);
```

```

PR(z);
PR(SQUARE(x));
printf(“—————\n Unrichtig !\n”);
PR(SQUARE(x+2)); PR(100/SQUARE(2)); PR(SQUARE(++x));
/*———— Enderung der Makrodefinition —————*/
#undef SQUARE
#define SQUARE(x) ( (x) *(x) )
x=4;
printf(„\n—————\n Richtig !\n x=%d\n“,x);
PR(SQUARE(x));
PR(SQUARE(x+2));
PR(100/SQUARE(2));
}

```

Beispiel für private h-Datei (file "bool.h")

```

/* p32.c Vorübersetzer-3 */
#include <stdio.h>
#include "c:\tc\soso\bool.h"
main ( )
{ int ch; int count=0;
  BOOL whitescape();
  while((ch=getchar()) != EOF)
    if(whitescape(ch))
      count++;
  printf(“Es gibt %d lehre Symbole\n”, count);
  getch();
}

```

```

BOOL whitescape(c)
char c;
{ if (c == ‘ ‘ || c == ‘\n’ || c == ‘\t’)
  return(TRUE);
  else
  return(FALSE);
}

```

```

/* bool.h file in INCLUDE Directorie*/
#define BOOL int
#define TRUE 1
#define FALSE 0

```

Lektion 13

13.1. SYMBOLPRÜFUNGSOPERATIONEN

Für die Symbolprüfungsoperationen verwendet man <ctype.h> Datei, mit der folgende Funktionen:

isalnum() : für A÷Z, a÷z, . . . , 0÷9

isalpha() : für A÷Z, a÷z

isascii() : wenn kleines Byte hat die Werte in [0 . . 127]

iscntrl() : wenn gegeben ist Löschensymbol oder Steuerungssymbol

isdigit() : für 0 . . 9

isgraph() : für graphischen Symbol (0x21-0x7E)

islower() : für a . . z, . . .

isprint() : für drükenes Symbol (0x20-0x7E)

ispunct() : für trennen Symbol (z.B: „ , „ , „ , „ ; „ u.s.w)

isspace() : für „ , „ , „ \n „ , „ \t „

isupper() : für A . . Z, . . .

isxdigit() : für Hexa-16 Ziffern (0..9, A..F, a..f).

Die Funktion **isalpha(c)** gibt Resultat 1 (True) zurück, falls **c** ein alphabetisches Symbol ist und 0 (False), falls nicht.

```

/*p33.c Programmbeispiel für ctype.h Funktionen*/

```

```

#include <stdio.h>

```

```

#include <ctype.h>

```

```

#define A 'g'

```

```

#define B 25

```

```

#define S ' '

```

```

main()

```

```

{

```

```

    char c;

```

```

    c=isalpha(A)? printf("Char-%c\n",A) : printf("No_Character\n");

```

```

    c=isdigit(B)? printf("No Digital\n") : printf("Digital\n");

```

```

    c=isspace(S)? printf(„,Space\n“) : printf(„,No_Space\n“);

```

```

    if(islower(A))

```

```

    {

```

```

    c=toupper(A);
    printf("Nach der Umwandlung c=%c\n",c);
}
}

```

atof()	character	→	float
atoi()	character	→	integer(int)
atol()	character	→	long int
ecvt(), fcvt(), gcvt()	float	→	char
itoa()	int	→	char
ltoa()	long int	→	char
strtod()	string	→	doppelte Zahl (Genauigkeit)
strtol()	string	→	long int
stroul()	string	→	unsigned long int
toascii()	string	→	ASCII Format
tolower()	string	→	a...z
toupper()	string	→	A...Z
ultoa()	unsigned long int	→	string

13.2. TYPENUMWANDLUNGSOPERATIONEN

Für die Umwandlungsoperationen verwendet man <stdlib.h> Datei, mit der folgende Funktionen:

Programmbeispiel für Symbolsumwandlung:

```
/*p34.c Programmbeispiel für stdlib.h Funktionen*/  
#include <stdio.h>  
#include <stdlib.h>  
main ()  
{  
  char number[9];  
  int value;  
  puts("Input symbole number");  
  gets(number);  
  /*value=number*/ /* mit Fehler*/  
  value=atoi(number); /* ohne Fehler*/  
  printf("value=%d\n", value);  
  getch();  
}
```

Abb.2.28. Beispiel für die Umwandlungsoperation

Lektion 14

14.1. WECHSELOPERATIONEN ZWISCHEN FUNKTIONEN

Datenwechsel zwischen der Funktionen hat bei der Programmierung eine große Bedeutung. Im C-Sprache gibt es verschiedene Mittel dazu. Der Programmist muß selbst diese Mittel (Tools) wählen.

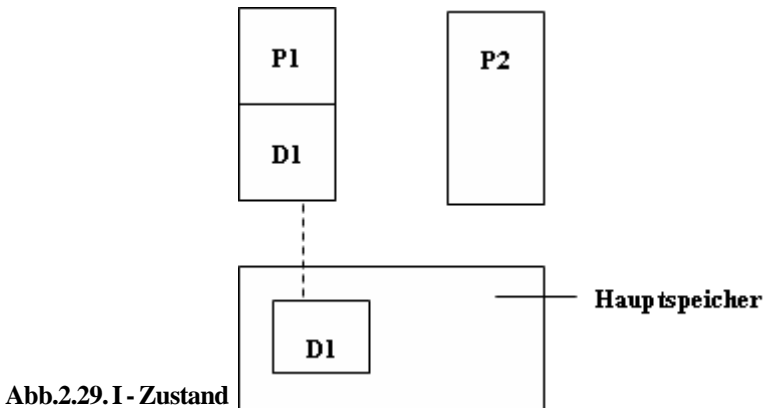


Abb.2.29. I - Zustand

14.1.1. Datenwechsel (Datenübertragen) zwischen der zwei Programme.

I - Zustand: das P1-Programm hat Daten D1 in dem Hauptspeicher. Das P2-Programm hat keine Daten.

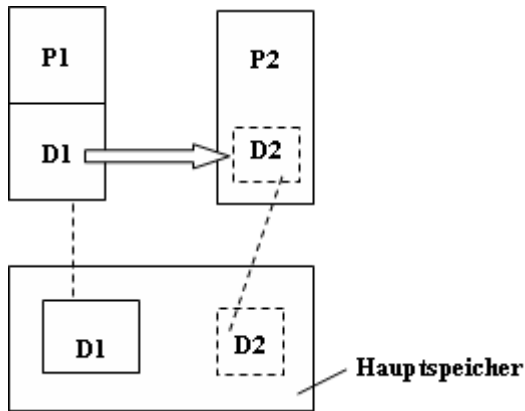


Abb.2.30. II - Zustand

II-Zustand: das P1-Programm schickt (trägt über) die D1 Datenwerte im virtuelle D2 Daten. Das P2-Programm bringt die D2 Daten in den Hauptspeicher. Also, die Daten D1 und D2, die sich im Hauptspeicher befinden, sind verschiedene.

```
/*p35.c Datenwerteübertragung */  
#include <stdio.h>  
main()  
{ int x=5, y=10;  
  printf("1: x=%d y=%d\n", x, y);  
  interchange(x, y);  
  printf("4: x=%d y=%d\n",x,y);  
}
```

```

interchange(int u, int v)
{ int temp;
  printf(„2: u=%d v=%d\n“,u,v);
  temp=u;
  u=v; v=temp;
  printf(„3: u=%d v=%d\n“,u,v);
}

```

Resultat:

1. x=5 y=10
2. u=5 v=10
3. u=10 v=5
4. x=5 y=10 (???)

14.1.2. Datenwechsel mittels Datenadressübertragen

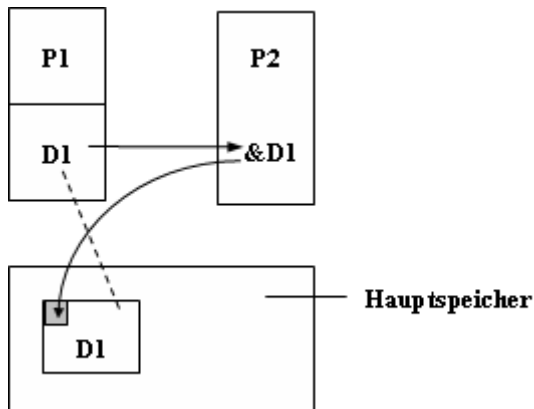


Abb.2.31. Zustand mit &-übertragen

Das P1- Programm schickt für das P2-Programm die Adresse von D1 Daten. &D1-ist die Adresse im Hauptspeicher. Das P2-Programm darf die Daten im Hauptspeicher verarbeiten. Also, P1 und P2 Programme verarbeiten dieselbe Daten.

- Vorteile für Adressübertragung:

1) Die Adressübertragung braucht weniger Zeit als Datenwerteübertragung (Zeitfaktor);

2) Im Hauptspeicher befinden sich keine kopierte Daten (Speicherfaktor).

Programmbeispiel für Datenadressübertragung:

```
/*p36.c Datenadressübertragung */
#include <stdio.h>
main()
{
    int x=5, y=10;
    printf("1: x=%d y=%d\n",x,y);
    interchange(&x,&y);
    printf("4: x=%d y=%d\n",x,y);
}

interchange(int *u, int *v)
{
    int temp;
    printf(",2: u=%d v=%d\n", *u, *v);
    temp=*u;
    *u=*v;
    *v=temp;
    printf(",3: u=%d v=%d\n", *u, *v);
}
```

Resultat:

1. x=5 y=10
2. u=5 v=10
3. u=10 v=5
4. x=10 y=5 (!!!)

14.2. KONTROLLOPERATIONEN FÜR DIE UNTERBRECHENS- UND HILFSTASTEN

Bei der Verwendung *dos.h* und *bios.h* Dateien kann der Programmist die Unterbrechungs- und Hilfsoperationen steuern.

Die Unterbrechung (Interrupt) im Computersystem ist solche Prozedur, wenn eine normale Ausführung des Programmes haltet und ohne normale Beendigung (Umfallsituation) das System (Managerprogramm) soll eine spezielle unterbrechungsverarbeitende Prozedur aufrufen. Dieses Unterbrechungsprogramm kann das Computersystem aus Umfallsituation zu Normalsituation bringen.

Manchmal das Managerprogramm kann keine eindeutige Entscheidung geben. Das heißt eine „Sackgasse“-Situation (Dead-lock). Computerbenutzer muß selbst das System helfen, und zwar mit Hilfstasten (z.B.: *ctrl+alt+Del*).

Solche Hilfstasten werden durch die Systementwerfer und Benutzer vordefiniert, und später Regular verwendet. z.B.: (im *dos.h*):

- ctrbrk()* - aufrufen des Unterbrechungsverarbeitungsprogramm.
- disable()* - verbieten des Unterbrechens.
- enable()* - erlauben des Unterbrechens.
- getdfree()* - informiert über die freie Kapazität des Disksspeichers.
- getfat()* - FAT Information (File Allocation Table).
- setcbrk()* - feststellen der Reaktion für *Ctrl+Break*;
- delay()* - Wartezeit für das Programmunterbrechen;
- sound()* - einschaltet das Mikrofon.
- nosound* - ausschaltet das Mikrofon u.s.w.

Programmbeispiele:

```
/* p37.c Unterbrechensoperationen */  
#include <stdio.h>  
#include <dos.h>  
#define ABORT 0  
int i; int c_break(void)
```

```

{ printf("is pressed botton Ctr-Break\n");
  printf("Ausführung des Programmes ist unterbrochen..\n");
  return(ABORT);
}
main ()
{ ctrlbrk(c_break);
  for(;;)
  { printf(„Arbeit der Zyklus...\n“);
    delay(0); /* Verzögerung in MSsekunde */
    for(i=1; i<8; i++)
      { printf("die Freguenz=%d\n",80*i);
        sound(80*i); delay(800); nosound();
      }
  }
}
}
/* p38.c Wartezeitsteuerungsprogramm */
#include <stdio.h>
#define G 800L
main()
{ long delay=0;
  int i=0, n1=10, n2=5;
  while(delay++ < G)
  while(i++ <n1+n2-1)
  { putchar('\007');
    delay=0;
    while(delay++ < G)
      ; /* leerer Operateur */

    printf(„i=%d delay=%ld \n“, i, delay);
    putchar('\n');
  }
  getch();
  printf(„Wartezeit multipliziert ! \n“);
  for(i=1; i<5; i++)
  {
    delay=0;
    while(delay++ < 10*G)

```

```

        ;
        putchar('\007');
    }
    printf("OK!\n");
}

```

Resultat:

```

    i=1  delay=8001  /* und klingt */
    i=2  delay=8001

```

```

    ...
    i=14 delay=8001

```

Wartezeit multipliziert ! /* klingt länger */
OK!

Lektion 15

15.1. FUNKTIONEN

15.1.1. Typ einer Funktion

Der Typ einer Funktion ist der Typ ihres Rückgabewerts. Er muß bei Definition und Deklaration einer Funktion angegeben werden.

Wird er weggelassen, so wird der Typ **int** angenommen.

Für Funktionen ohne Rückgabewert steht der Typ **void** zur Verfügung, z.B.:

```

double sqrt(float a),
void drucke_bilanz(in jahr).

```

Innerhalb der runden Klammern hinter dem Funktionsnamen werden für jeden Parameter sein Typ und Name paarweise aufgezählt. Die Parameterdeklarationen werden durch Kommata getrennt, z.B.:

```

SendeNachricht(char nachricht[], int leange).

```

Eine leere Parameterliste wird durch (void) deklariert, z.B.:

```

druckeStern(void).

```

Soll die Funktion jedoch das Argument (d.h. die Variable der aufrufenden Funktion) ändern können, so muss **die Adresse** dieser Variable übergeben werden.

15.1.2. Rückgabewert

Die Rückgabewert ist für eine Funktion die einfachste Art, ein Datum an ihren Aufruf zurückgeben.

Wenn eine **return** - Anweisung erreicht wird, wird die aufgerufene Funktion verlassen und die aufrufende Funktion mit der nächsten Anweisung hinter dem Funktionsaufruf fortgesetzt.

Eine Funktion wird auch dann verlassen, wenn ihre letzte Anweisung ausgeführt worden ist.

Durch die **return** - Anweisung kann eine Funktion beim Rücksprung einen Wert an den Aufrufer übergeben. Der Wert des Ausdrucks hinter dem **return** ersetzt für den Aufrufer logisch den Funktionsaufruf, z.B.:

```
return; /* gibt nichts zurück */
return ausdruck; /* gibt Wert von
return (ausdruck); /* gibt Wert von
ausdruck zurück */
```

Programmbeispiele:

```
/* p39.c base_exp x=a^b */
#include <stdio.h>
double base_exp(float base, int exp);
main ( )
{ double x; /* Resultat */
float a; /* base */
int b; /* exponent */
printf("a="); scanf("%f", &a);
printf("b="); scanf("%d", &b);
x=base_exp(a, b);
printf("x=%.2fn",x);
}
double base_exp(float base, int exp)
{ double Answer;
if (exp>0)
{ for(Answer=1.0; exp>0; exp--)
Answer *= base;
return(Answer);
}
else
```

```
if (base!=0)
{ for(Answer=1.0;exp<0;exp++)
Answer/=base;
return(Answer);
}
else /*base=0,exp<=0*/
{ printf("
return(0);
printf("OK ! \n");
}
```

```
Resultat:
a = 3
b = 2
x = 9.000
a = 2
b = -2
x = 0.2500
OK !
```

Abb.28. Programmbeispiel für Funktion $x=a^b$

15.2. ZUFALLSVARIABLENFUNKTION

Zufallsvariablenfunktion ist als Zufallszahlengenerator gannat, z.B.: im C gibt es die Funktion `random ()`:

```
/* p40.c Randomprogramm */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
main()
{
    int i;
    long now;
    printf("V_New\n");
    srand(time(&now)%37);
    for(i=0; i<5; i++)
        printf("%d\n", rand() );

    printf(„OK!\n“);
    getch();
}
```

Im Resultat erhalten wir fünf Variablen innerhalb in dem Intervall [0 .. 32767], z.B.,

Resultat:

V_New	V_New
7272	692
28585	32682
31411	21834
32631	23967
26364	22222
OK !	OK !

15.3. RESULTATSAUSGABEPROZEDUR

Das Programm druckt alle gerade Zahlen aus dem bestimmten Intervall.

```
/* p41.c Datenausgabe */
#include <stdio.h>
main()
{ int a=0, m, n, number;
  printf("Input obere Grenze :"); scanf("%d", &m);
  printf(„Input Spaltenanzahl :“); scanf(„%d“, &n);
  for(number=1; nubmer<=m; number++)
  {
    if(number % 2 == 0)
    {
      printf(“%5d”, number);
      a++;
      if(a % n == 0)
        printf(“\n”);
    }
  }
}
```

<u>Resultat:</u> Input obere Grenze : 30 Input Spaltenanzahl : 5 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30

DIE KONTROLLAUFGABEN

1. Was ist ein Programm, eine Programmierung ?
 - a) Die Phasen des Programmierungsprozesses;
 - b) Zeigen Sie, bitte, durch den Personalrechner das Beispiel des Anwendersprogrammes. Wo befinden sich (C:\Arbeitsplatz\...)
Ihre Programmtexte:
 - Quellprogramm (pname.c)
 - Mashcinenprogramm (pname.obj)
 - Ablaufprogramm (pname.exe)
 - c) Was für eine Verwendung hat:
 - Übersetzungsprogramm (Compiler),
 - Binder (Linker) ?
2. a) C - Quellprogrammstruktur;
b) Wie kann man Konstanten und Variablen in dem C-Programm

definieren und initialisieren ?

- c) Welche Hauptdatentypen kennen Sie ?
- Zeigen Sie, bitte, alle Datentypen mit den Wertebereichen und Spezifikationen
 - Was bedeutet die Anweisung „unsigned“ ?

3.

- Schreiben Sie, bitte, die allgemeine (syntaktische) Struktur für die Ausgabe und Eingabe Anweisungen in der C – Programmierung.
- Konstruieren Sie, bitte, mittels PC das Dialogprogramm P für die folgende Aufgabe (Abb.-A):

Eingabe Variablen: N – die Produktname, P – der Produktpreis, Q – die Produktanzahl;

Ausgabe Variablen: Sum – die Summe, z.B. $Sum = P * Q$;

St – die Steuer, z.B. $St = (Sum * 19)/100$; /*d.h. 19 % von Kosten */

GK - die Gesamtkosten, z.B. $GK = Summ + St$.

4. Schreiben Sie, bitte C-Programm für die 3x3-Dimensionale Schachbretttabelle mit den blaue und rote Zellen. Verwenden Sie die Anweisung - window(). Dann schreiben die Vornamen mit weiß und grüne

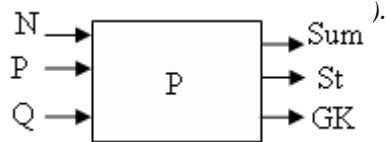


Abb.-A

DATO	NINO	VANO
LIA	GIO	MARI
BACHO	KATE	TEMO

Abb.-B

5. Rechnen Sie, bitte folgende quadratische Gleichung :

$$Ax^2 + Bx + C = 0, \text{ wo } A=5, B=-7, C=-15$$

6. Es sind zwei Wurzel der quadratische Gleichung gegeben: $x_1=8$ und $x_2=-7$. Berechnen Sie, bitte die Werte von Koeffizienten: A, B und C.

7. Schreiben Sie, bitte C-Programm für Berechnung der Summe von gerade und ungerade Zahlen im Intervall $N=[0..100]$. Die konkrete Bedeutung N eingeben Sie, bitte durch Anweisung `scanf()`.

Dabei verwenden Sie die Schleife „for“ oder „while“ und Konstruktion „if...else“. (Anmerkung: die Zahl (z) ist gerade, wenn durch die Division ($z/2$) bekommen im Rest „0“ !!!).

8. Schreiben Sie, bitte C-Programm für die Berechnung der Multiplikation ($M=Mult(S_i, i=1,5)$) von Summen ($S_i = x1_i + x2_i, i=1,5$) der Wurzeln ($x1$ und $x2$) der Quadratische Gleichung. Die Koeffizienten A, B und C , geben Sie, bitte mit Anweisung `scanf()` ein. Für die 5 Varianten Verwenden Sie, bitte Anweisung „for“ oder „while“.

Eingabedaten:

N	A	B	C
1	5	-3	-2
2	1	10	-20
3	13	-31	-43
4	7	0	-10
5	55	14	-5

Ausgabedaten:

N	x1		x2	S_i
1	?	+	?	$S_1=?$
2	?	+	?	$S_2=?$
3	?	+	?	$S_3=?$
4	?	+	?	$S_4=?$
5	?	+	?	$S_5=?$
	$M=S_1*S_2...*S_5$			$M=?$

Literatur

1. Kernighan B. W., Ritchie D. M.: Programmieren in C, 2. Ausgabe, Carl Hanser, München, Wien; Prentice Hall, London, 1990.

2. Meyer-Wegener K., Surguladze G. Grundlagen der Programmierung (Einleitung in die Programmiersprachen C/C++/C#). Erlangen-Tbilissi. 2005.

3. Gogitschaischvili G., Surguladze G., Schonja O. Programmier -verfahren mit C&C++. GTU, Tbilisi, 1997.