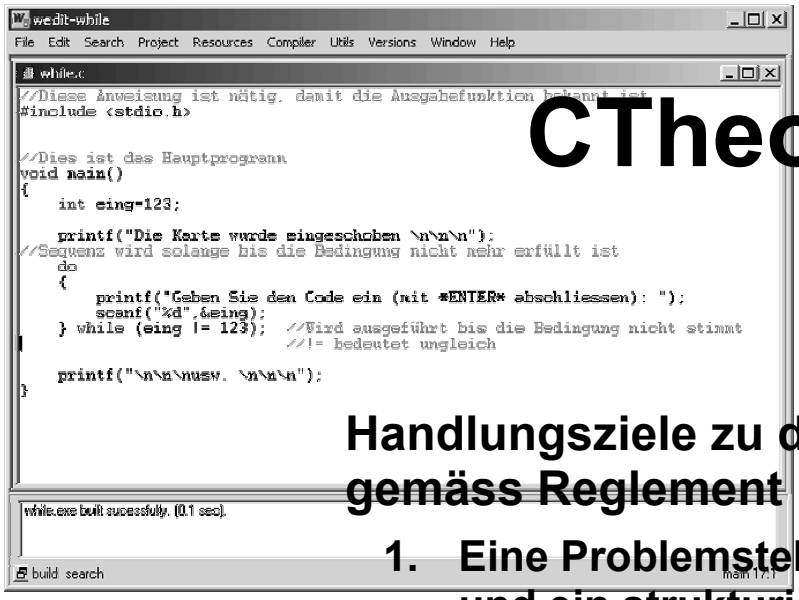
Modul 118



CTheorie

Handlungsziele zu diesem Modul gemäss Reglement

1. **Eine Problemstellung analysieren und ein strukturiertes Design erstellen**
2. **Ein Design interpretieren und erklären**
3. **Ein Design prozedural umsetzen**
4. **Ein Programm in Teilmodule und Funktionen gliedern**
5. **Eine Testmethode auswählen und den Test durchführen**
6. **Eine vollständige Dokumentation erstellen (Design, Struktur, Funktionstest usw.)**

Inhaltsverzeichnis

1 Grundrechenarten (Willms S.43)

1.1 Testen der Grundrechenarten

Aufgabe: Testen sie die Grundrechenarten (+, -, *, /, %)!
Untersuchen sie auch ++, -- usw.
Den Vorschlag für ein Testprogramm entnehmen sie den nächsten Zeilen (rechnen.c). Wenn sie das Programm testen wird es bei der Übersetzung einen Fehler geben. Der Compiler kennt in main den Aufruf res = berechnung(zahl, zahl1) noch nicht. Sie müssen ihm dies vor main sagen mit einem Prototyp (Willms S.29).
Achten sie bei den Tests auch auf den Einsatz von Datentypen. Überlegen sie sich wo der Datentyp float geeignet wäre! Ändern sie das Programm entsprechend ab.

```
#include <stdio.h>

void main()
{
   int zahl, zahl1, res;

   printf(„1. Zahl :“);
   scanf(„%i“,&zahl);
   fflush(stdin);
   printf(„2. Zahl :“);
   scanf(„%i“,&zahl1);
   fflush(stdin);

   res = berechnung(zahl, zahl1);   //Aufruf Funktion mit Parameter
   printf(„Resultat: %i\n“,res);
}

int berechnung(int op1, int op2)    //op1 und op2 Parameter, berechnung Rückgabewert mit Datentyp
{
   int resultat;

   resultat = op1 % op2;
   return(resultat);                //Hier wird das Resultat ans Hauptprogramm zurückgegeben
}
```

1.2 Unwandlung von Datentypen (Willms S.121)

Sie möchten 4 durch 3 teilen. Es handelt sich hier zwar um eine Division von zwei ganzen Zahlen. Das Resultat weist aber Nachkommastellen auf. D.h. das Resultat ist vom Datentyp float (Zahl mit Nachkommastellen). Das folgende Programm gibt Aufschluss über die Problematik (type_cast.c).

```
#include <stdio.h>

void main()
{
   int ires;
   float fres;

   ires = 4/3;
   printf(„Resultat: %i\n“,ires);
   fres = (float)(4/3);
   printf(„Resultat float: %f\n“fres);
   fres = (float)(4)/(float)(3);
   printf(„Resultat float: %f\n“fres);
   fres = 4/3.0;
   printf(„Resultat float: %f\n“fres);
}
```

Das Resultat beinhaltet immer den genausten Datentyp der Berechnung!

2 Sichtbarkeitsbereiche (Willms S.73)

Wie sie vielleicht aus dem Modul 103 noch wissen, können aus dem Hauptprogramm verschiedene Funktionen aufgerufen werden. Es gibt nun verschiedene Möglichkeiten Variablen zu deklarieren. Je nach dem, wo der Ort der Deklaration ist, kann diese eine unterschiedliche Bedeutung haben.

2.1 Lokale Variable

Aufgabe: Studieren sie den untenstehenden Code. Was erwarten sie für eine Ausgabe. Testen sie dazu das Programm lo.c.
Bearbeiten sie das Programm auch mit dem Debugger um die Arbeitsweise mit Funktionen genau zu studieren. Sie können zusätzlich im Hauptprogramm auch zahl1 ausgeben.

```
#include <stdio.h>

int lo()
{
   int zahl = 100;      //lokale Variable in lo

   printf(„Dies ist die lokale Zahl von lo : %i\n“,zahl);
   if (zahl == 100)
   {
      return(0);
   }
   else
   {
      return(1);
   }
}

void main()
{
   int zahl = 1000;     //lokale Variable in main
   int zahl1;

   printf(„Dies ist die lokale Zahl von main : %i\n“,zahl);
   zahl1 = lo();
}
```

2.2 Globale Variable

Aufgabe: Studieren sie den untenstehenden Code. Was erwarten sie für eine Ausgabe. Testen sie dazu das Programm glo.c.
Bearbeiten sie das Programm auch mit dem Debugger um die Arbeitsweise mit Funktionen genau zu studieren.
Betrachten sie noch den Fall, wenn sie in glo auch noch eine Variable namens zahl deklarieren. Welche Variable wird bevorzugt?

```
#include <stdio.h>

int zahl;         //globale Variable

void glo()
{
   zahl = 333;
}

void main()
{
   zahl = 555;

   printf(„Die Zahl im main : %i\n“,zahl);
   glo();
   printf(„Die Zahl nach dem Aufruf von glo : %i\n“,zahl);
}
```

Versuchen sie wenn möglich die globalen Variablen zu verhindern. Dies ist vor allem dann wichtig, wenn sehr viele Leute an der Entwicklung eines Porgrammes beteiligt sind. Die Gefahr ist gross, dass eine globale Variable dann fälschlierweise manipuliert wird.

2.3 Statische Variable (Willms S. 79)

Der Wert einer statischen Variable wird beim verlassen der Funktion nicht gelöscht.
Eine statische Variable muss bei der Deklaration initialisiert werden!!

Aufgabe: Testen Sie den Zusammenhang mit dem Beispiel im Willms.

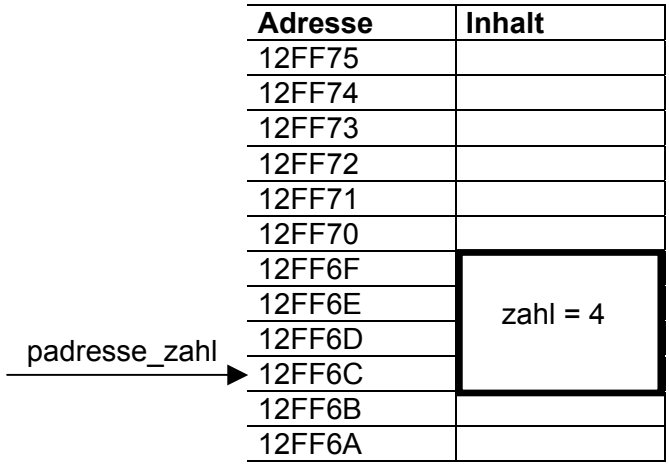
3 Strukturen

Wenn wir Daten haben, welche zusammengehören, dann können wir diese in Strukturen zusammenfassen. Als Beispiel könnte man sich eine Adressliste vorstellen. Zu jeder Adresse gehören Vorname, Nachname, Strasse, Strassennummer, PLZ, Ort. Das würde dann wie folgt aussehen (strukturen.c).
Achten sie neben den Strukturen auch auf das Kopieren von Worten mit strcpy (Willms S. 180) .

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

struct Adressen           //Definition der Struktur
{
   char nachname[40];
   char vorname[40];
   char strasse[40];
   unsigned short strnummer;
   unsigned short plz;
   char ort[40];
};

void ausgabe (struct Adressen aadresse)
{
   printf(„Name: %s %s\n“,aadresse.nachname,aadresse.vorname);
   printf(„Strasse: %s %hu\n“,aadresse.strasse, aadresse.strnummer);
   printf(„Ort: %hu %s\n“,aadresse.plz, aadresse.ort);
}



void main()
{
   //Hier wird der Speicherplatz für drei Strukturen reserviert
   struct Adressen adresse = {„Meier“,“Fritz“,“Cool“,1,2222,“Seen“};   //Mit Initialisierung
   struct Adressen adressen[2];     //Array mit zwei Elementen

   strcpy(adressen[0].nachname, „Saegesser“);
   //adressen[0].nachname = „Saegesser“ geht nicht!! strcpy befindet sich in string.h
   strcpy(adressen[0].vorname, „Andreas“);
   strcpy(adressen[0].strasse, „Grossackerstr.“);
   adressen[0].strnummer = 65;
   adressen[0].plz = 8041;
   strcpy(adressen[0].ort, „Zuerich“);

   strcpy(adressen[1].nachname, „Mosimann“);
   strcpy(adressen[1].vorname, „Daniela“);
   strcpy(adressen[1].strasse, „Lengacker“);
   adressen[1].strnummer = 6;
   adressen[1].plz = 2000;
   strcpy(adressen[1].ort, „Fribourg“);

   ausgabe(adresse);
   ausgabe(adressen[0]);
   ausgabe(adressen[1]);
}
```

4 Zeiger oder Pointer oder Vektoren (Willms S. 155)

Wie gesagt, jede Variable wird an einem Speicherplatz mit einer gewissen Nummer abgelegt. Diese Nummer nennt man Zeiger, Pointer oder Vektor. Testen sie das Programm (pointer.c)!




```
#include <stdio.h>

void main()
{
   int zahl;               //Reservation für Zahl mit 4 Bytes
   int *padresse_zahl;     //Reservation für Adresse auf int

   padresse_zahl = &zahl;  //Hier wird nun die Adresse inititalisiert!
                           //& liefert die Adresse der zahl
   printf("Dies ist die Zahl : %i\n",zahl);                          //Ausgabe der zahl
   printf(„Dies ist die Zahl via Pointer : %i\n“,*padresse_zahl);   //Ausgabe der zahl
   printf("Dies ist die Adresse der zahl : %p\n",padresse_zahl);      //Ausgabe der Adresse
}
```

4.1 Bedeutung von & und * im Zusammenhang mit Zeigern

Die Bedeutung von *: Es ist etwas verwirrend, dass * in C zwei unterschiedliche Bedeutungen hat.

1. Bei der Deklaration bedeutet *, dass es sich um einen Zeiger handelt.
   Bsp: short *pzahl;
   Dies ist ein Zeiger auf eine Variable vom Typ short.
2. Bei einer Zuweisung bedeutet *, dass man über den Zeiger auf den entsprechenden Speicherplatz zugreift.
   Bsp: *pzahl = *pzahl++;
   Hier wird der Wert, auf welchen pzahl zeigt um 1 erhöht.

Die Bedeutung von &: Mit & kann die Adresse einer Variable bestimmt werden.

Bsp: pzahl = &zahl;
Hier wird der Zeiger pzahl mit der Adresse der Variable zahl geladen.

Wenn sie Zeiger einsetzen müssen sie immer genau überlegen, welche Nebeneffekte auftreten könnten!

Es ist sinnvoll, die Variablennamen von Zeigern so zu wählen, dass man dem Code sofort ansieht, dass es sich um einen Zeiger handelt. Mein Vorschlag setzen sie ein p an den Anfang (pzeichen, pbuchstabe usw.)

4.1.1 Problem Array

Der Name eines Arrays ist verbunden mit einem Pointer auf die Startadresse des Speicherblocks. D.h. ein Array kann direkt ohne & einem Pointer zugewiesen werden. Den Zusammenhang sehen sie am Besten am Beispiel arr_poin.c.

```
#include <stdio.h>

void main()
{
   int zahlen[10] = {1,2,3,4,5,6,7,8,9,10};
   int *pzahlen = zahlen;     //ist schon Pointer darum kein &
   int *pzahl1 = &zahlen[0];  //beim einzelnen Element braucht es dann &
   int *pzahl2 = &zahlen[1];

   printf("Dies ist der Pointer pzahlen %p\n",pzahlen);

   printf("Dies ist der Pointer pzahl1 %p\n",pzahl1);

   printf("Dies ist der Pointer pzahl2 %p\n",pzahl2);
}
```

5 Verschiedene Arten von Parameterübergabe

5.1 Parameterübergabe by value

Diese Art Parameterübergabe haben wir unbewusst immer verwendet. Wenn der Wert einer Variable im Unterprogramm verändert wird, so hat dies keinen Einfluss auf den Wert der Variable im Hauptprogramm.

```
#include <stdio.h>

void sub(int sub_zahl)     //Aufruf Unterprogramm mit Parameter (mehrere Parameter möglich)
{
   sub_zahl = sub_zahl + 2;

   printf(„Wert im Unterprogramm: %i\n“,sub_zahl);
}

void main()
{
   int main_zahl = 1000;

   printf(„Wert im Hauptprogramm : %i\n“,main_zahl);
   sub(main_zahl);
   printf(„Wert nach dem Aufruf von sub : %i\n“,main_zahl);
}
```

Aufgabe: Ändern sie das Programm by_val.c so ab, dass mehrere Parameter übergeben werden.

5.2 Parameterübergabe by reference

Wenn nun der Wert einer Variable des Hauptprogrammes im Unterprogramm verändert werden soll, so muss als Parameter ein Pointer übergeben werden. Über diesen Pointer geschehen nun die entsprechenden Veränderungen. Das untenstehende Programm by_ref.c zeigt den Zusammenhang eindrücklich.

```
#include <stdio.h>

void sub(int *sub_zahl)
{
   *sub_zahl = *sub_zahl + 2;

   printf(„Wert im Unterprogramm: %i\n“,*sub_zahl);
}

void main()
{
   int main_zahl = 1000;
   int *pmain_zahl = &main_zahl     //Deklaration Pointer plus Initialisierung

   printf(„Wert im Hauptprogramm : %i\n“,main_zahl);
   sub(pmain_zahl);
   printf(„Wert nach dem Aufruf von sub : %i\n“,main_zahl);
}
```

5.2.1 Spezialfall Arrays

```
#include <stdio.h>

void sub(int sub_array[2])
{
   sub_array[0]++;
   sub_array[1]++;

   printf(„Wert im Unterprogramm: %i, %i\n“,sub_array[0],sub_array[1]);
}

void main()
{
   int array[2] = {100, 1000};

   printf(„Werte im Hauptprogramm : %i, %i\n“,array[0],array[1]);
   sub(array);
   printf(„Werte nach dem Aufruf von sub : %i, %i\n“,array[0],array[1]);
}
```

Aufgabe: Testen sie das obige Programm array.c. Welches sind ihre Folgerungen?

5.2.2 Call by reference mit Strukturen

Das folgende Programm sollte Aufschluss darüber geben, wie die Strukturen als Pointer übergeben werden. Achten sie auf den speziellen Zugriffsmechanismus! Das Programm finden sie unter cal_str.c

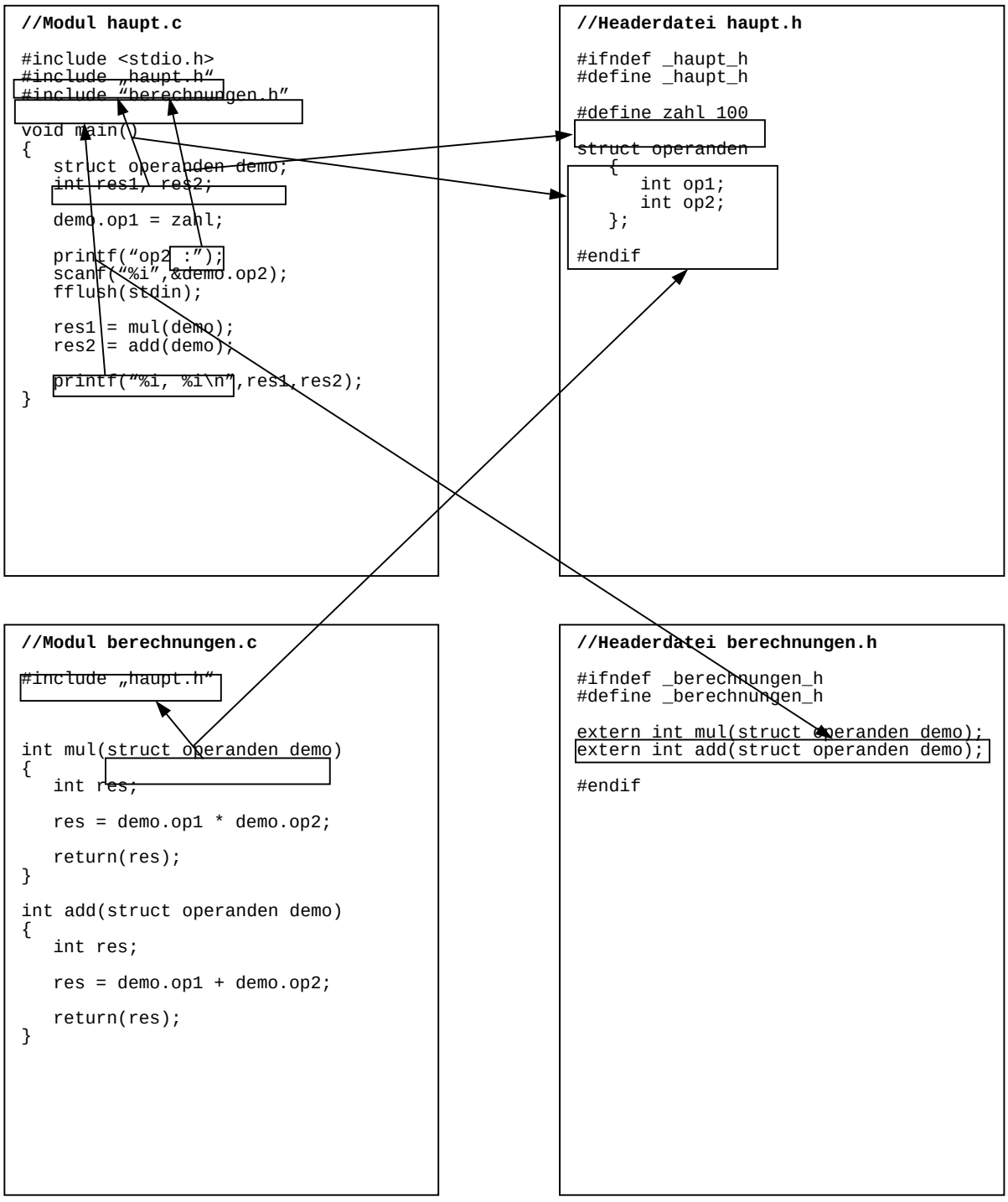
```
#include <stdio.h>

struct test             //Struktur wird festgelegt
{
   short wert1;
   short wert2;
};

void sub(struct test *pwerte)
{
   pwerte -> wert1 = 100;  //So wird über ein Pointer auf die Elemente der Struktur zugegriffen
   pwerte -> wert2 = 1000;
}

void main()
{
   struct test werte;
   struct test *pwerte;

   pwerte = &werte;
   sub (pwerte);     //Übergabe des Pointers auf die Struktur
   printf(„Die werte sind: %hu %hu\n“,werte.wert1, werte.wert2);
}
```

6 Modulares Programmieren (Willms S. 349)

Bei grösseren Projekten ist es sinnvoll verschiedene Funktionen mit ähnlichen Bedeutungen in Modulen zusammenzufassen. Die Module werden unabhängig voneinander compiliert und in einem zweiten Schritt zu einem lauffähigen Programm gebunden.

Ziel ist es auch, dass die Module bereits getestet werden können. Manchmal müssen dazu spezielle Testroutinen geschrieben werden.

6.1 Und so wird's gemacht



Es ist darauf zu achten, dass alle Funktionen eines Modules welche auch in anderen Modulen zur Verfügung stehen sollten, in dessen Headerdatei als extern deklariert werden. Wenn wir nun in einem anderen Modul die Funktion verwenden möchten so ist nur die entsprechende Headerdatei zu importieren. Damit steht dann ein Prototyp dieser Funktion zur Verfügung.

Definieren sie möglichst die Konstanten in den Headerdateien. So können spätere Anpassungen einfach dort vorgenommen werden!

Mit den Präprozessorbefehlen #ifndef #endif verhindern wir, dass zum Beispiel die Headerdatei haupt.h mehrmals übersetzt wird.

Statt eine Headerdatei zu schaffen könnte man auch einfach den Text in der Datei an die Stelle schreiben wo der #include Befehl steht. Die Headerdateien sollten helfen bei grossen Projekten die Übersicht zu wahren.

7 Dynamische Speicherverwaltung (Willms S. 279)

Wenn wir Variablen deklarieren, d.h. wenn wir für gewisse Variablen Speicherplatz reservieren, dann wissen wir immer schon während der Entwicklung, wie viele Elemente das Programm verwalten muss. Dies ist aber häufig in der Praxis nicht der Fall. Wenn man zum Beispiel eine Datenbank mit Adressen verwalten möchte, so kann man während der Entwicklung noch nicht sagen wie viele Adressen dann tatsächlich verwaltet werden müssen.

Damit ist es möglich, während der Laufzeit des Programms Speicherbereiche anzufordern. Auch können diese Speicherbereiche wieder freigegeben werden.

7.1 Und so wird's gemacht

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


void main(void)
{
   void *nullptr;    //Pointer auf undefinierten Typ
   int *ptr1, *ptr2; //zwei Pointer auf int

   nullptr=malloc(sizeof(int));  //Speicherblock der Grösse int wird reserviert
                                 //malloc(Grösse) gibt den Pointer auf die Startadresse des Be-
reichs
   ptr1=(int*)(nullptr);         //Typecast von Pointer auf void zu Pointer auf int

   if(ptr1)                      //ptr ungleich 0 wenn Reservierung erfolgreich
      {
         *ptr1=30;
         printf(„Der Pointer auf int zeigt auf die Adresse %p\n“,ptr1);
         printf("Der dynamische Speicherblock hat den Wert %i gespeichert\n",*ptr1);
         free(ptr1);             //Der reservierte Speicherblock wird wieder freigegeben!
      }

   nullptr=malloc(sizeof(int));
   ptr2=(int*)(nullptr);

   if(ptr1)
      {
         *ptr2=100;
         printf(„Der Pointer auf int zeigt auf die Adresse %p\n“,ptr2);
         printf("Der dynamische Speicherblock hat den Wert %i gespeichert\n",*ptr2);
         free(ptr2);
      }

}
```

Aufgabe: Testen sie das obige Programm dyn.c.
Nehmen sie die Zeile free(ptr1) aus dem Code. Was beobachten sie und welches ist ihre Interpretation dazu?
Versuchen sie eine Situation zu provozieren, in welcher die Reservierung des Speichers nicht mehr funktioniert.

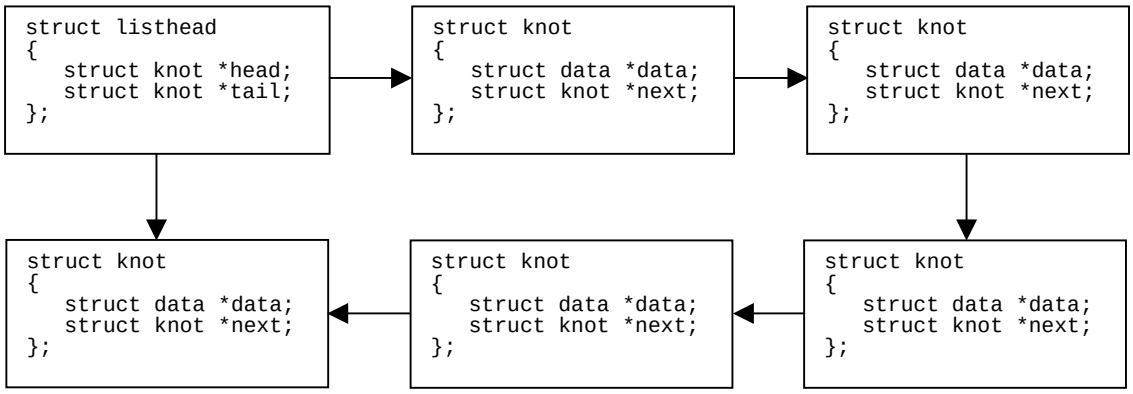
8 Listen

8.1 Ausgangslage

Auch wenn wir den Speicher zur Laufzeit des Programms reservieren, wurde der Zeiger schon zur Zeit der Übersetzung angelegt. Der entscheidende Schritt zur wirklichen dynamischen Verwaltung des Speichers geschieht, indem wir Strukturen anlegen, die ihrerseits Zeiger auf weitere Strukturen beinhalten. Das ist das Grundprinzip aller dynamischen Datenstrukturen in C.

8.2 Einfach verkettete Liste

Eine einfach verkettete Liste besteht aus einem Listenkopf, welcher die beiden Zeiger auf das erste und auf das letzte Element beinhaltet. Die einzelnen Knoten sind aus den Nutzdaten und dem Zeiger auf das nächste Element aufgaut.



Es werden drei verschiedene Strukturen eingesetzt (Listenkopf, Knoten und Nutzdaten).

8.2.1 Listenkopf erzeugen

Diese Funktion gibt den Pointer auf den Listenkopf zurück. Zudem werden head und tail mit NULL initialisiert.

```
struct ListHead *ListCreate()                          //Diese Funktion gibt einen Pointer zurück
{
   struct ListHead *head=malloc(sizeof(struct ListHead));  //Pointer auf Listenkopf

   if(head)
      {  //Wenn Speicher verfügbar
         head->head=0;                                    //Liste leer
         head->tail=0;
         return(head);                                    //Zeiger zurückgeben
      }
   return(0);                                             //Aktion nicht erfolgreich
}
```

8.2.2 Neues Element anhängen

Dieser Funktion werden die Nutzdaten und der Zeiger auf den Listenkopf übergeben. Es ist zu erwähnen, dass der Zeiger auf den neuen Knoten nicht zurückgegeben wird. Der letzte Knoten der Liste zeigt auf diesen neu erzeugten Knoten. Dies ist genau der Sinn einer Liste. Die Information, wo sich das nächste Element befindet ist im Knoten enthalten.

```
int ListAppend(struct ListHead *hd, struct Telefon *tel)
{
   struct Knot *ap=malloc(sizeof(struct Knot));//neuer Zeiger auf Knoten

   if(ap)
      {                                        //wenn erfolgreich
         ap->telefon=tel;                      //Nutzdaten laden
         ap->next=0;                           //letztes Element

         if(!(hd->head))
            {                                  //Liste war noch leer
               hd->head=ap
               hd->tail=ap;
               return(1);
            }
         else
            {                                  //Liste hatte mind. ein Element
               hd->tail->next=ap;
               hd->tail=ap;
               return(1);
            }
      }
   return(0);
}
```

8.2.3 Liste ausgeben

Um die Liste auszugeben wird der Listenkopf benötigt. Nun muss die Liste immer wieder über den Zeiger auf den nächsten Knoten abgearbeitet werden.

```
void ListShow(struct ListHead *hd)
{
   struct Knot *cur=hd->head;              //Zeiger auf aktuellen Knoten

   if(!cur)
      {                                    //kein Zeiger vorhanden
         printf("Die Liste ist leer!\n");
      }

   while(cur)
      {                                    //solange nicht letztes Element
         printf("Name: %s, %s. Tel.:%s\n",  //Nutzdaten anzeigen je nach Nutzdaten unterschied-
lich!
                 cur->telefon->nachname,cur->telefon->vorname,cur->telefon->telefon);
         cur=cur->next;                    //akt. Zeiger auf das nächste Element
      }
}
```

8.2.4 Liste löschen

Es ist wichtig, dass reservierter Speicher nach seiner Verwendung wieder freigegeben wird. Je grösser die Anwednungen und die Strukturen desto wichtiger wird dieser Vorgang. Es ist nicht beliebig viel Speicher zur Verfügung.

```
void ListDelAll(struct ListHead *hd)
{
   struct Knot *cur=hd->head, *nex;       //Zeiger auf akt. und nächsten Knoten

   while(cur)
      {
         nex=cur->next;
         free(cur->telefon);              //Nutzdaten freigeben
         free(cur);                       //Knoten freigeben
         cur=nex;
      }
   free(hd);                              //Listenkopf freigeben
}
```

8.2.5 Hauptprogramm

Nun braucht es noch ein Hauptprogramm, aus welchem die verschiedenen Funktionen aufgerufen werden.

- Zuerst Liste erzeugen
- Dann Auswahl
  - Neues Element einfügen
  - Liste anschauen
  - Programm beenden
- Wenn Programm beendet Liste löschen

Es würde vielleicht Sinn machen, die verschiedenen Operationen im Zusammenhang mit Listen in einem Modul op_list zusammenzufassen. Wenn dann in einem Programm eine Liste verwendet wird, kann einfach die Headerdatei op_list.h importiert werden.

Auftrag: Ändern sie das Programm so ab, dass sie die CD Datenbank aus der Prüfung „verwalten“ können.
Fassen sie dazu die Listenoperationen in einem Modul zusammen. Achten sie darauf, dass sie nur die Listenoperationen ins Modul integrieren, welche keinen direkten Zugriff auf die Nutzdaten machen. Diese Operationen sind für verschiedenste Nutzdaten immer gleich!

9 Presistente Daten (Willms S. 219)

Unsere Daten in den Variablen oder in Arrays sind weg, sobald die flüchtigen Speicher die Daten verlieren. Nun sollen unsere Daten also auch auf der Festplatte gespeichert werden sollen.

9.1 Unterschied Textdateien Binärdateien

Textdateien: Textdateien interpretieren den Zeilenvorschub automatisch richtig. Ausserdem werden Textdateien mit EOF (End Of File) beendet.

Binärdateien: Alles ausser Text wird in sogenannten Binärdateien abgelegt.

9.2 Öffnen einer Datei

Das untenstehende Beispiel erstellt eine Textdatei. Die verschiedenen Optionen beim Öffnen sind dem Willms S.220/226 zu entnehmen.

```
#include <stdio.h>

void main()
{
   FILE *handle;           //Pointer auf FILE

   handle = fopen("i:\\c_proj\\file\\files.txt","w");//Angabe von Pfad und Option

   fputs("Dies ist ein Test!\n",handle);             //Wird in die Datei geschrieben

   fclose(handle);                                   //Datei schliessen
}
```

9.2.1 Die wichtigsten Funktionen zusammengefasst

	Befehl	**Beschreibung**
Textdateien	wert = feof(handle)	0 wenn Dateiende nicht erreicht ungleich 0 wenn Dateiende erreicht
	fputs(zeichenkette,handle)	Schreibt Zeichenkette in Datei
	fgets(zeichenkette,länge,handle)	Liest Zeichenkette mit max. Länge aus Datei
	fputc(zeichen,handle)	Schreibt Zeichen in Datei
	zeichen = fgets(handle)	Liest Zeichen aus Datei
	fprintf	Gleiche Funktion wie printf, ausser Handle Bsp: fprintf(handle,„Test %s“,text);
	fscanf	Gleiche Funktion wie scanf, ausser Handle
Binärdateien	anz=fwrite(adr,gr,anzahl,handle)	adr: Hier steht die zu speichernde Variable gr: Grösse (Anz. zu speichernde Bytes). Kann mit sizeof(x) ermittelt werden anzahl: Anzahl Elemente
	anz=fread(adr,gr,anzahl,handle)	
Allgemein	rewind(handle)	Setzt den Positionszeiger an den Anfang
	ok=remove(namen)	Löscht die Datei mit dem namen
	ok=rename(altername,neuname)	Datei unbenennen

Auftrag: Es wäre nun toll, wenn sie ihre Listenoperationen noch um die Optionen SaveList und LoadList erweitern könnten!

Einige Tips zu SaveList:
- beim File handelt es sich um eine Binärdatei
- es werden nur die Nutzdaten abgelegt

möglicher Prototyp:
void ListSave(struct ListHead *hd, char dateiname[20])

Einige Tips zu LoadList:
- es muss eine neue Liste angelegt werden
- Daten aus File lesen und in Liste einfügen

möglicher Prototyp:
struct ListHead *ListLoad(char dateiname[20])