Eine Einführung

für die dritte Klasse Elektronik an der HTL Anichstrasse

Version 0.5

Zusammengestellt von Markus Signitzer
Oktober 2021
HTL Anichstraße

Inhaltsverzeichnis

1 Vorwort und Voraussetzungen

Hallo Leute!
Anbei ein Versuch von mir die grundlegenden Aspekte der Programmiersprache C zu erklären - mit Berücksichtigung, dass ihr schon zwei Jahre lang Python programmiert. Absolut kein Anspruch auf Vollständigkeit!

Voraussetzungen: Grundlegende Programmierkenntnisse in Python sowie ein funktionierendes Linux, entweder als dual boot auf euren Laptops oder als Virtuelle Maschine. Alternativ kann auch am Raspi programmiert werden. Wer unbedingt under MS Windows entwickeln will muss sich selber einen C Compiler installieren, z.B. von http://www.mingw.org/

Vieles wird euch von Python schon bekannt vorkommen. Hauptunterschied in der Syntax sind vermehrt geschwungene Klammern für Code-Blöcke und ein ; hinter Befehlszeilen. Der Rest ist easy ;-)

Viel Spaß!
Markus Signitzer

2 Hintergrundinformationen

2.1 Geschichte

C ist eine imperative und prozedurale Programmiersprache, die der Informatiker Dennis Ritchie (1941-2011) in den frühen 1970er Jahren an den Bell Laboratories entwickelte. Seitdem ist sie eine der am weitesten verbreiteten Programmiersprachen.

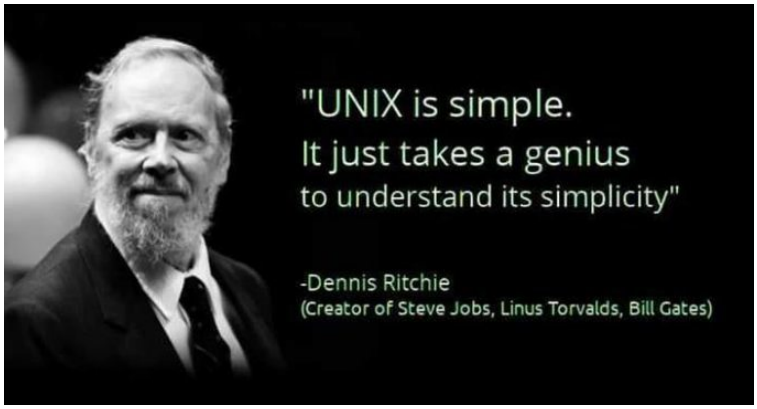
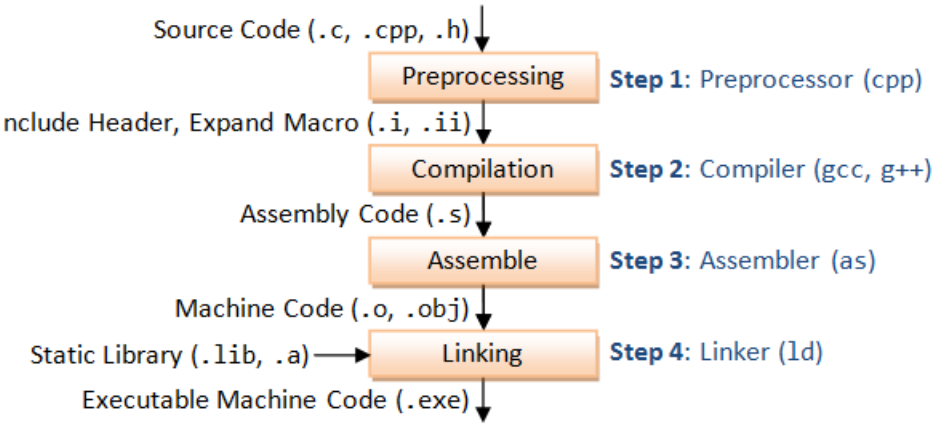


Abbildung 1: Dennis Ritchie, ©www.computationalthinkers.com

Die wichtigsten Anwendungsbereiche von C sind die **System- und Anwendungsprogrammierung, sowie Hardware-Nahe-Programmierung**. Die grundlegenden Programme aller Unix-Systeme und die Systemkernel vieler Betriebssysteme sind in C programmiert. Zahlreiche Sprachen, wie C++, Objective-C, C#, D, Java, PHP, uvm., orientieren sich an der Syntax und anderen Eigenschaften von C.
C wurde mehrfach standardisiert **ANSI** (C89/C90, C99, C11), wobei ANSI für das American National Standards Institute steht und die Zahlen für die Jahre in denen die Standardisierung stattgefunden hat, also ANSI C11 wurde im Jahr 2011 definiert. Abgesehen vom Mikrocontrollerbereich, wo eigene Dialekte existieren, sind die meisten aktuellen PC-/Server-Implementierungen eng an den Standard angelehnt.

2.1.1 Compiler

Im Gegensatz zu einem Interpreter (Python) wird der Quelltext vom Compiler auf einmal in einen Maschinencode übersetzt und kann dann schnell abgearbeitet werden. Fehler werden nicht erst beim Laufenlassen des Programms, sondern bereits beim Übersetzten aufgespürt.
Was genau ein Compiler macht und welchen Schritten er folgt ist wieder von Sprache zu Sprache unterschiedlich. Im Folgenden wird der C/C++ Compiler (GCC-Gnu Compiler Collection) genauer betrachten:
Die Compiler-Schritte werden in verschiedene Phasen gegliedert, die jeweils verschiedene Teilaufgaben des Compilers übernehmen. Einige dieser Phasen können als eigenständige Programme realisiert werden (s. Precompiler, Präprozessor).



2.1.1.1 Präprozessor Der Präprozessor stellt die erste Instanz vor dem eigentlichen Kompilieren in C dar. Musste früher ein alleinstehendes Programm für die Aufgaben des Präprozessors vor dem Kompilieren genutzt werden, so ist der Präprozessor heute ein fester Bestandteil der meisten Compiler.
Einfach ausgedrückt durchsucht der Präprozessor den Quelltext und ersetzt Teile des Codes nach bestimmten Vorgaben zur Aufbereitung für den Kompilierungsvorgang.
Aufgaben des Präprozessors:

- Kommentare des Programmierers entfernen
- Headerfiles z.B. *#include<stdio.h>* einbinden, d.h. die Codezeile wird durch den gesamten Inhalt des Header Files ersetzt.
- Makros (z.B. symbolische Konstanten) können definiert werden z.B: *#define PI 3.1415926f* und der Präprozesser erstetzt im Quelltext das Wort *PI* durch den Wert 3.1415926f.

Manueller Aufruf: *cpp helloworld.c > helloworld.i*

2.1.1.2 Compiler Ein C-Compiler erhält den vom Präprozessor bereits vollständig vorbereiteten Quellcode und übersetzt diesen aus der Hochsprache in einen Maschinencode.
Dabei werden die zwei vom Compiler ausgeführten Phasen in jeweils drei unterschiedliche Arbeitsschritte unterteilt:

- **Analyse-Phase**
  - Lexikalische Analyse: Erkennt "Rechtschreibfehler"wie *3,14 statt 3.14* oder *iff statt if*
  - Syntaktische Analyse: Prüfung auf Syntax-Fehler wie *flaeche = pi * r * ;* - da fehlt was hinter dem zweiten *.
  - Semantische Analyse: Funktioniert das Programm? Dabei kann keine Prüfung auf pragmatische Korrektheit erfolgen (”Leistet der Quellcode, wofür er geschrieben wurde?”), sondern lediglich ob der Quelltext über die lexikalischen und syntaktischen Spezifikationen hinaus sinnvoll ist. Dazu werden beispielsweise Datentyp-Überprüfung gemacht. Ein Beispiel:
  *int x = 7 * meineMethode(5);* was wäre wenn meineMethode() einen String zurück gibt?
- **Synthese-Phase**
  - Zwischencode-Erzeugung: Grobe Übersetzung in einen Zwischencode welcher eine fundamentale Operation pro Zeile sowie eine unbegrenzte Speicheradressierung und Registeranzahl aufweist - d.h. langer und unoptimierter code!
  - Zwischencode-Optimierung Läuft je nach Compiler anders und kann eventuell in verschiedenen Graden aufgerufen werden. Mögliche Arbeitsschritte sind: Schleifen optimieren, Konstanten auflösen, toten code entfernen, usw.
  - Assemblercode-Erzeugung: Im letzten Übersetzungsschritt wird der erzeugte und optimierte Zwischencode in einen Assemblercode übersetzt.

Manueller Aufruf: *gcc -S helloworld.i*

2.1.1.3 Assembler Die Hauptaufgabe vom Assembler ist es die Maschinensprache (noch Quelltext) in binärcodes zu transformieren. Es gibt teilweise auch pseudo-assembly instructions (das sind Befehle die es auf der HW gar nicht gibt, die aber sehr einfach emuliert werden können) die der Assembler noch auflöst (z.B.: es gibt keinen Befehle für bedingter Sprung wenn Register den Wert 0 hat. Der Assembler könnte das mit einer TEST Instruktion und einem Spring bei ZERO-Flag übersetzten). Das fertige Object-File (.o) ist nun bestmöglich optimiert und nutzt nur so viel Speicher, wie es wirklich benötigt.
Manueller Aufruf: *as helloworld.s -o helloworld.o*

2.1.1.4 Linker Der Linker bindet nach erfolgreichem Kompilieren alle entstandenen Object-Files zu einem zentral ausführbaren Programm zusammen und bindet eventuell noch benötigte statische libraries (.lib) oder dynamically linked libraries (.dll) ein.
Manueller Aufruf ist beim Linker schwer möglich weil man alle benötigten Libraries übergeben muss. Prinzipiell ginge der Aufruf mit:
ld -o runMe.sh helloWorld.o ...libraries...
Es kann auch überprüft werden welche dll-files eine fertige .exe verwendet, bzw. welche zu ihr gelinkt sind:
ldd helloWorld.exe

2.2 Vergleich mit Python

Python wird oft als Interpreter-Sprache, oder Sprache ohne Compiler bezeichnet, wobei dies eigentlich nicht ganz richtig ist.
Python hat ebenfalls einen Compiler, der allerdings nicht so viele Code-Optimierungen wie der C-Compiler macht, sondern den Code in einen Byte-Code übersetzt (ähnlich wie Java), und dieser Byte-Code wird dann im eigentlichen Interpreter (wie eine Virtuelle-Maschine, wieder wie in Java) ausgeführt.
$Programmdatei : programm.py \rightarrow COMPILER \rightarrow Byte-Code : programm.pyc \rightarrow INTERPRETER \rightarrow Programm-laeuft$

Vorteile: Plattformunabhängig (mit der Einschränkung, dass der Interpreter (VM) vorhanden sein muss), schnelle Übersetzung;
Nachteil: Der Code ist nicht so gut optimiert und damit langsamer bzw. größer in der Ausführung, als in C.

Noch ein Wort zum Interpreter. Der Standard-Python-Interpreter ist *CPython*. Viele vereinfachte Python-Befehle greifen im Hintergrund auf C-Funktionen zu. Dies erlaubt es auch C Programme einzubinden.
Es gibt aber noch andere Python-Interpreter z.B. *Jython* oder *IronPython* die in Java oder C# implementiert sind und damit eine Verbindung zur Java Runtime Enviroment JRE bzw. dem .NET Framework herstellen.

3 Erste Schritte und Hello World

In einem Texteditor eurer Wahl bitte folgendes Programm schreiben und unter dem Filenamen *HelloWorld.c* speichern:
Zum Vergleich (und zur Erinnerung an die 1. + 2. Klasse :) daneben das gleiche Programm in Python.

```c
/* Mein erstes C Programm.
 * Hello World
 * (c) by me
 */

#include <stdio.h>

int main()
{
    //dies ist eine Kommentar
    printf("Hello C-World!\n");
    return 0;
}
```

```python
""" Mein erstes Python Programm.
    Hello World
   (c) by me """

def main():

    #dies ist eine Kommentar
    print("Hello Python-World!")

if __name__ == "__main__":
    main()
```

In einem Terminal kann man das Programm nun Compilieren. Annahme: Man befindet sich im gleichen Unterverzeichnis wie das .c file.

```
gcc HelloWorld.c -o runHelloWorld
./runHelloWorld
```

3.1 Hello World - erklärt

Welche Unterschiede zu Python fallen auf?

- **Kommentare**: Es gibt mehrzeilige Kommentare

  ```c
  /* Hier steht
   * der Kommentar über
   * mehrere Zeilen
   */
  ```

  oder einzeilige Kommentare

  ```c
  // nur ein einzeiliger Kommentar
  ```

- **include-statements**: haben eine Raute # vorausgestellt und der Name des Header-Files (deshalb file-Endung .h) steht in spitzen Klammern.

  ```c
  #include <stdio.h>  //standard IO-Library
  ```

- **Main** oder die main-methode ist in Python nicht unbedingt notwendig. Man verwendet sie nur dann wenn der Python code als *stand alone* code und als Modul, eingebettet in ein anderes Programm, verwendet werden kann. Wenn der Code dirket ausgeführt wird und nicht als Modul, wird vom Interpreter der String `"__main__"` in die Variable `__main__` geschrieben. Dies kann über ein if-statement abgefragt werden und dann die main() Methode ausgeführt werden.
  C braucht die main() Methode als Startpunkt des Programms. Das *int* bedeutet, das main einen Return-Wert vom Type Integer (ganze Zahl) hat. return 0 bedeutet dabei ein erfolgreiches Beenden (das Programm hat getan was es tun sollte und ist ohne Fehler durchgelaufen) bzw. return 1 bedeutet, das ein Fehler aufgetreten ist. Diese Werte werden dem ausführenden Betriebssystem weitergegeben. (*Linux*: Den return code eines gerade in der Bash ausgeführten Befehls kann man mit `echo $?` anzeigen)

- **printf** erwartet einen String (durch " gekennzeichnet) der an den std-output ausgegeben wird. Es gibt verschiedene escape character wie un unserem Beispiel `\n`, welcher für *new-line* steht. Für mehr Info zu escape-characters einfach nach `"escape sequences in C"` googlen ;-)

- **;** ist in C hinter jeder Code-Zeile notwendig, dafür sind die unter Python verpflichtenden Einrückungen nun optional.

4 Datentypen

Im Unterschied zu Python, müssen in C die Datentypen einer Variable definiert werden. Folgendes Beispiel deklariert eine int Variable mit Namen x und weist ihr den Wert 5 zu:

```
int x = 5;
```

4.1 Auflistung der Datentypen

4.1.1 char

char steht für **Character** und wird verwendet um einen einzelnen Character zu speichern. Achtung, tatsächlich wird aber nicht der Buchstabe oder Character gespeichert, sondern eine 8-bit lange Binärzahl (ASCII Code):

```
char zeichen = 'A';    /* intern gespeichert wird nicht der Buchstabe "A" sondern
                        * die anhand der ASCII-Tabelle errechnete Binärzahl "01000001" */
printf("%d", zeichen); /* gibt "01000001" als Dezimalzahl aus, also: "65" */
printf("%c", zeichen); /* gibt "01000001" als ASCII-Zeichen aus, also: "A" */
```

4.1.2 int

int steht für **Integer**, also eine Ganzzahl. Je nachdem wie viel Speicherplatz man für die Zahl braucht, gibt es verschieden Deklarations-Schlüsselwörter, wobei der gnu compiler dann eine mindestgröße an Speicherplatz zur Verfügung stellt. Je nach Betriebssystem kann auch mehr zugewiesen werden.

```
char ganzzahl = 1;      /* mindestens 8 Bit, also 256 mögliche Werte */
short ganzzahl = 2;     /* mindestens 16 Bit, also 65536 mögliche Werte */
int ganzzahl = 3;       /* mindestens 16 Bit, also 65536 mögliche Werte */
long ganzzahl = 4;      /* mindestens 32 Bit, also 4294967296 mögliche Werte */
long long ganzzahl = 5; /* mindestens 64 Bit, also 18446744073709551616 mögliche Werte */
```

Wenn man keine negativen Zahlen braucht kann das key-word *unsigned* vorausstellen. Also z.B. `unsigned int x;` Um herauszufindne wie groß die Speicherbereiche auf der eigenen Maschine sind, kann man die Funktion *sizeof()* verwenden, welche die tatsächlich Speichergröße in Byte wiedergibt z.B.

```
#include <stdio.h>

int main(void)
{
   printf("char     : %d Byte\n" , sizeof(char));
   printf("int      : %d Bytes\n", sizeof(int));
   printf("long     : %d Bytes\n", sizeof(long int));
   printf("float    : %d Bytes\n", sizeof(float));
   printf("double   : %d Bytes\n", sizeof(double));

   return 0;
}

ergibt (auf meiner Maschine) den folgenen Output:
char     : 1 Byte
int      : 4 Bytes
long     : 8 Bytes
float    : 4 Bytes
double   : 8 Bytes
```

4.1.3 float

float steht für floating point number, also auf Deutsch eine Fließkommazahl. Im Wort steckt auch schon der größte Unterschied: Punkt statt Komma!

```
float kommazahl = 0.000001;           /* mindestens 32 bit */
double kommazahl = 0.000000000000002; /* mindestens 64 bit */
long double kommazahl = 0.3;          /* Genauigkeit ist implementierungsabhängig */
```

4.1.4 bool

bool für boolean und gibt es als eigenen Datentyp erst seit dem C99 Standard. Variablen können nun als `_Bool` deklariert werden und einen der beiden Werte 0 (falsch) oder 1 (wahr) aufnehmen. Inkludiert man den Header stdbool.h kann auch der Alias `bool` statt `_Bool` verwendet werden, sowie `false` und `true` statt 0 und 1.

```
#include <stdbool.h>
bool b = false;
```

4.1.5 Überläufe

Achtung auf Überläufe! Wenn man eine int Variable, die z.B. ihren maximalen positiven Wert hat, um eins erhöht, springt sie zu ihrem kleinst möglichen Wert!

4.2 Unveränderliche Variablen

Mit Hilfe des Schlüsselwortes *const* können Variablen unveränderlich, also zu Konstanten, gemacht werden.

```
const int pi = 3.1415;
```

Wenn man versucht eine mit *const* deklarierte Variable nachträglich zu verändern, bringt der Compiler eine Fehlermeldung.

4.3 Typumwandlungen

Cast oder **casting** nennt man eine explizite Typumwandlung. Hierbei wird der Ziel-Datentyp in Klammern vor den zu umwandelnden Variablennamen geschrieben.

```
int i;
char c = 'A';

i = (int) c;
printf("char c nach int i: %d\n", i);

i = 67;
c = (char) i;
printf("int i nach char c: %c\n", i);

float f = 2.345;
i = (int) f;
printf("float f nach int i: %d\n", i);

//output
char c nach int i: 65
int i nach char c: C
float f nach int i: 2
```

5 Simple User Input/Output

Vorausstellend: C (wie Linux) behandelt alle Devices wie z.B. Display und Tastatur wie ein File (Vorteil: Schreiben auf das Display oder in ein File - ähnliche Syntax!)
Wenn ein C Programm läuft, werden automatisch drei Files geöffnet welche folgendes default mapping haben:

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard errot	stderr	Screen

Es gibt in C ein Vielzahl von sehr spezifischen Ein- und Ausgabe-Funktionen z.B. `getchar()` und `putchar()` um einzelne character einzulesen bzw. auszugeben, oder `fgets()` um die Eingabe solange in einen Buffer zu schreiben bis ein newline oder EOF (End Of File) kommt.
Aus Zeitgründen werden wir uns nur zwei Funktionen genauer ansehen:

5.1 printf() und scanf()

5.1.1 printf()

Die (für uns) standard Ausgabefunktion. Die Funktion besteht aus einem Formatierungsteil der von Anführungszeichen `" "` eingeschlossen ist. In diesem Formatierungsteil wird der auszugebende Text und eventuelle Platzhalter für Variablen-Werte, durch das Prozentzeichen `%` dargestellt, sowie weitere Formatierungszeichen mittels escape-character (backslash) \ z.B. `\n` für eine neue Zeile.

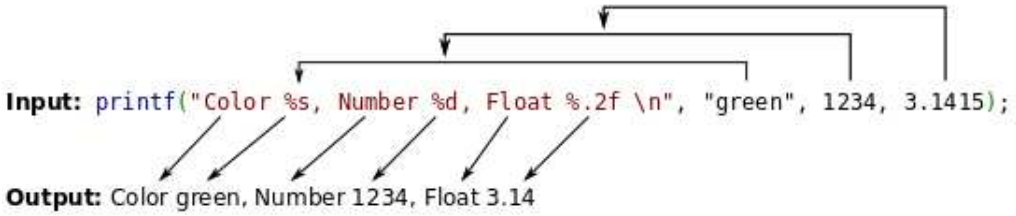


Abbildung 2: printf() Beispiel

Im oberen Beispiel wurde noch eine zusätzlich Formatierungsinfo für die Gleitkommazahl (floating point number) 3.1415 angegeben und zwar steht beim Platzhalter: `%.2f`
Dies bedeutet, das nur zwei Stellen nach dem Komma (point) ausgegeben werden.

Ein Auszug aus den gängisten Platzhaltern:

Typ	Platzhalter
int (dezimal)	%d
hexadezimal	%x
float/double	%f
char	%c
string (char*)	%s
pointer (void*)	%p

Es gibt noch viele weitere Formatierungsoptionen für printf(). Auf diese weiteren Optionen einzugehen, würde aber den Rahmen dieses Dokumentes sprengen. Aus diesem Grund wird hier an die Internet-Suchengine ihrer Wahl verwiesen :)

5.1.2 scanf()

`scanf()` liest beliebig lange Eingaben, welche erst durch drücken der ENTER Taste beendet werden. ACHTUNG: Alle Eingaben die wir machen landen zuerst im Tastaturpuffer, bevor die Zeichen von unserem Programm verarbeitet werden. Da wir unsere Eingaben mit der ENTER-Taste bestätigen müssen, landet auch dieses Zeichen (ENTER = line feed) im Puffer, was eventuell Probleme verursachen kann.
Es können aus dieser einen Eingabezeile natürlich auch mehrere Werte eingelesen werden (standard mäßig durch Leerzeichen getrennt, kann aber angegeben werden). Die verwendete Syntax ist ähnlich zu `printf()` mit dem Unterschied, das nicht auf Variablen sondern auf die Adressen der Variablen verwiesen wird. Die Adresse einer Variable erhält man durch den Addressoperator `&` - also wenn `x` die Variable ist, ist `&x` die Speicheradresse von `x`. Mehr zu Adressen und Pointer gibt es im Kapitel 6 auf Seite 13.
Ein Beispiel:

C-Code:

```
#include<stdio.h>

int main(void)
{
  int x;
  float y;
  char z[100];

  printf("Bitte int float string eingeben\n");
  printf("Trennzeichen = Leerzeichen\n");
  scanf("%d %f %s",&x, &y, z);
  printf("Die Eingabe war: %d %f %s\n",x,y,z);

   return 0;
}
```

Zugehöriger Input und Output:

```
Bitte int float string eingeben
Trennzeichen = Leerzeichen
42 1.23456789 HelloC
Die Eingabe war: 42 1.234568 HelloC
```

Was fällt auf? Warum muss auf das char Array z nicht mit Adressoperator & verwiesen werden? Mehr zu Arrays gibt es im Kapitel 11 auf Seite 22.

Noch ein Beispiel zum Tastatur-Buffer-Problem:

C-Code:

```
#include <stdio.h>

int main(void)
{
   char a,b,c;
   printf("1. Buchstabe : \n");
   scanf("%c",&a);
   printf("2. Buchstabe : \n");
   scanf("%c",&b);
   printf("3. Buchstabe : \n");
   scanf("%c",&c);
   printf("Sie gaben ein : %c %c %c ",a,b,c);

   return 0;
}
```

Zugehöriger Input und Output:

```
1. Buchstabe :
a
2. Buchstabe :
3. Buchstabe :
b
Sie gaben ein : a
 b
```

Was fällt auf? Man kann den zweiten Buchstaben gar nicht mehr eingeben, weil die ENTER-Taste zum quittieren der Eingabe des Buchstabes a, schon als Eingabe für den zweiten Buchstaben gewertet wird.
Mögliche Lösung: Wenn man nur einen Character einlesen will, kann man die ENTER-Taste abfangen, indem man explizit zwei chars einliest. Das LineFeed der Entertaste steckt dann im zweiten eingelesenen char und kann verworfen werden:

C-Code:

```
#include <stdio.h>

int main(void)
{
   char a,b,c,temp;
   printf("1. Buchstabe : \n");
   scanf("%c%c",&a,&temp);
   printf("2. Buchstabe : \n");
   scanf("%c%c",&b,&temp);
   printf("3. Buchstabe : \n");
   scanf("%c%c",&c,&temp);
   printf("Sie gaben ein : %c %c %c ",a,b,c);

   return 0;
}
```

Zugehöriger Input und Output:

```
1. Buchstabe :
a
2. Buchstabe :
b
3. Buchstabe :
c
Sie gaben ein : a b c
```

Noch eine Sicherheits-Anmerkung:

Die Funktion scanf() ist nicht gegen einen Pufferüberlauf (Buffer-Overflow) geschützt und somit unsicher, d. h., sie könnte für einen Hack des Programms durch eine andere Person missbraucht werden. Damit ist gemeint, dass die Funktion nicht die Anzahl der eingegebenen Zeichen überprüft und es damit zu Fehlern kommen kann bzw. ein Fehlverhalten von außen provoziert werden kann. Abgesehen davon ist scanf() (und auch printf()) ein guter Kanidat für Format String Exploits. Viele Compiler monieren scanf() auch als unsichere Funktion. Der Compiler von Microsoft VC++ z. B. rät, stattdessen die Funktion scanf_f()zu verwenden. Beachten Sie hierbei allerdings, das scanf_f() keine Standard-C-Funktion und somit auch nicht portabel ist.

6 Zeiger/Pointer

Pointer oder Zeiger sind ein key-concept von C und als solches sehr wichtig, leider aber auch (zumindest zu Begin) etwas verwirrend.
Variablen der verschiedenen Datentypen werden ja im Arbeitsspeicher des Rechners (genauer: im ausführenden Prozess-Speicher des laufenden C-Programms und dort entweder im Heap oder im Stack) gespeichert.
Dies ist ein Aspekt den wir bisher nicht berücksichtigt haben. Denken wir nochmals kurz darüber nach, welche Aspekte zu einer C Variable gehören. Am besten an einem einfachen Beispiel:

```
int x = 42;
```

Was wissen wir über diese Variable?

- Sie hat den *Bezeichner/Namen* x. Er erlaubt es die Variable innerhalb ihres Gültigkeitsbereichs zu verwenden.
- Sie hat den aktuellen *Wert* 42. Der kann ausgelesen oder verändert werden.
- Ihr *Datentyp* ist int (Integer) und definiert welche Werte repräsentiert werden können. In diesem Fall Ganzzahlen.
- **Und zusätzlich und für uns neu:** Die *Speicheradresse* gibt den Ort im Speicher an, wo der Wert der Variabel gespeichert ist. Und wo steht die???

Wir haben Variablen bisher immer über den Bezeichener bzw. Namen angesprochen. Man kann aber auch direkt über die Speicheradresse auf den Variablenwert zugreifen! Warum das sinnvoll ist und welche Möglichkeiten dadurch für den Programmierer entstehen werden wir später noch genaue sehen.

6.1 Deklaration von Zeigern/Pointern

Wie bei normalen Variablen muss auch bei Zeigern angeben, welcher Wertetyp an der Speicheradresse zu finden ist.
Also in unserem Beispiel mit der Variable `int x = 42;` müssen wir einen integer Zeiger anlegen.
Eine Zeigervariable wird mit dem Sonderzeichen Asterisk * vor dem Namen des Zeigers deklariert!

```
int *p;
```

Momentan ist unsere Zeigervariable noch nicht initialisiert, und kann irgendwo hinzeigen. Zum Vergleich gibt es auch einen sogenannter *Null-Zeiger* oder *Null-Pointer*: `int *p1 = NULL;` der nirgends hinzeigt, bzw. im C99 auf die Adresse 0 zeigt.

6.2 Zuweisung von Zeigern/Pointern

Der Adressoperator & liefert die Adresse (einen Zeiger) einer bestehenden Variable.

```
int *p = &x;
```

Kleine Visualisierung wie das am Stack aussieht:

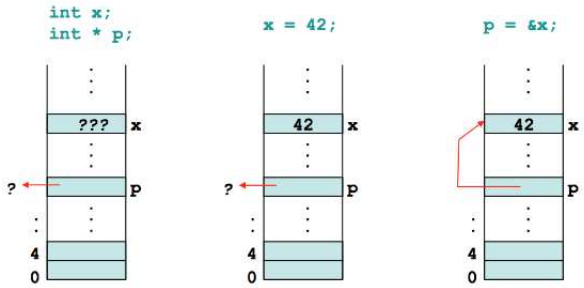


Abbildung 3: Variablen und Zeiger am Stack ©softech.cs.uni-kl.de

6.3 Zeiger Beispiel-Code

Der folgende Code:

```
#include <stdio.h>

int main(void)
{
    int x = 42;
    printf("Wir erstellen eine Variable: int x = 42;\n");
    printf("Die Adresse von x ist &x und hat den Wert %p \n", &x);
    printf("Der Wert von x ist %d \n",x);
    printf("\n");
    int *p = &x;
    printf("Nun erstellen wir eine Zeiger-Variable: int *p = &x;\n");
    printf("Die Adresse von p ist &p und hat den Wert %p \n", &p);
    printf("Der Wert von p ist %p \n",p);
    printf("Der Wert auf den p zeigt ist *p und ist %d \n",*p);

    return 0;
}
```

liefert folgenden output:

```
Wir erstellen eine Variable: int x = 42;
Die Adresse von x ist &x und hat den Wert 0x7ffca9e4844c
Der Wert von x ist 42

Nun erstellen wir eine Zeiger-Variable: int *p = &x;
Die Adresse von p ist &p und hat den Wert 0x7ffca9e48450
Der Wert von p ist 0x7ffca9e4844c
Der Wert auf den p zeigt ist *p und ist 42
```

Na, alles klar ?!?

7 Verzweigungen

Einfache Verzweigungen sollten von Python (if, else, elif) hinreichend bekannt sein, so dass an dieser Stelle nur sehr kurz auf die C-Syntax eingegangen wird.

7.1 if - else - else if

Wie in Python, nur das man statt `elif` ein vollständiges `else if` schreiben muss. Einrückungen und Klammer setzten wir laut der HTL-C-Coding-Guideline von Prof. Walch.

```
if (test_expression)
{
    if (nested_test_expression)
    {
        //do something
    }
    else
    {
        //do something else
    }
}
else if (test_expression)
{
    //do something else entirely
}
else
{
    //or maybe do nothing at all
}
```

7.2 Vergleichs- und logische Operatoren

Sind gleich wie in Python!

Bedeutung	Symbol
gleich	==
ungleich	!=
kleiner	<
kleiner gleich	<=
größer	>
größer gleich	>=
logisches UND	&&
logisches ODER	||

7.3 switch - case

Neu für uns sind Switch-Case Statements die es als solches in Python nicht gibt. Sie sind sehr nützlich wenn man viele Fälle unterscheiden möchte.
In die Klammern nach dem Schlüsselwort `switch` schreiben wir den Ausdruck, welchen wir auswerten möchten. Danach folgen mit dem Schlüsselwort `case` die verschiedenen Fälle, nach dem Doppelpunkt die auszuführenden Befehle. Der case Block wird mit `break` abgeschlossen. Dies ist unbedingt notwendig. Mit break wird bei erfolgreichem Ausführen eines Falles die switch Anweisung verlassen. Wird kein Fall erreicht, wird der `default` Block ausgeführt. Ein Beispiel:

```
int a=2;

switch(a)
{
    case 1: printf("a ist eins\n"); break;
    case 2:
    printf("a ist zwei\n");
    printf("und nicht drei!\n");
    break;
    case 3: printf("a ist drei\n"); break;
    default: printf("a ist irgendwas\n"); break;
}
```

Bei kurzen Statements kann man einen case block auch in einer Zeile schreiben oder auf mehrere Zeilen aufteilen, wie im Beispiel oben.

8 Schleifen

Sehr ähnlich zu Python, deshalb der Vollständigkeit halber nur ein kurzer Überblick:

8.1 break; continue; goto;

Es gibt in C drei Control-Statments mit denen man den Ablauf einer Schleife beeinflussen kann.

Statement	Syntax	Erklärung
break statement	break;	Beendet die laufende Schleife oder Verzweigung.
continue statement	continue;	Die momentane Schleifen-Iteration (-Durchlauf) wird gestoppt, und sofort zum nächsten Durchlauf gesprungen.
goto statement	goto lableName; lableName: statement;	**Achtung: gefährlich und umstritten:** die Programmausführung wird einfach an einem anderen Teil (durch das label gekennzeichnet) vorgeführt.

8.2 while

Kopfgesteuerte Schleife! Die Bedingung nach dem Schlüsselwort `while` muss erfüllt sein, sonst wird die Schleife gar nicht erst gestartet.
An einem Beispiel erklärt:

```
int i = 10;
while (i > 0)
{
    printf("i = %d\n",i);
    i--;
}
```

8.3 do while

Fußgesteuerte Schleife! Der `do`-Block wird mindestens einmal ausgeführt/durchlaufen und erst dann wird die Bedingung evaluiert.
An einem Beispiel erklärt:

```
int i = 10;
do
{
    printf("i = %d\n",i);
    i--;
}
while (i > 0);
```

8.4 for

Kopfgesteuert und zusätzlich noch zählergesteuerte Schleife. Wird meist verwendet wenn die Anzahl der gewünschten Durchläufe bekannt ist.
Beispiel-Syntax:

```
for(int i = 10; i > 0; i--)
{
    printf("i = %d\n",i);
}
```

Achtung: Deklaration der Zählvariable in der Schleife geht erst seit dem C99-Standard!

9 Funktionen

Funktionen sind sinnvoll um große Programme in logische Teilprobleme zu gliedern. Weiters verwenden wir Funktionen um auf vorgegebene Funktionalitäten (Libraries) zu zugreifen.
Grundsätzliche Syntax:

9.1 Funktions-Deklaration

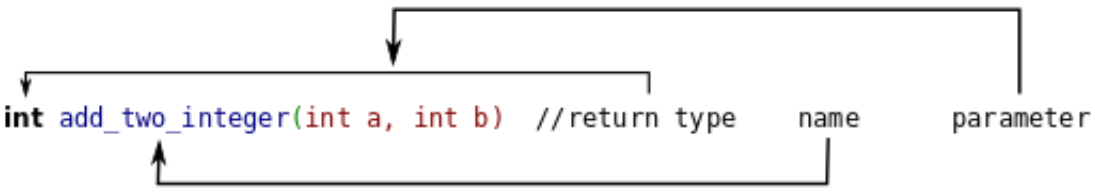


Abbildung 4: Funktions Syntax

- Eine Funktionsdeklaration hat immer einen return-Typ, selbst wenn sie eigentlich nichts zurück gibt, dann ist er `void`.
- Funktionsnamen sollten kurz und informativ sein. Mehrere Wörter werden durch Unterstrich _ getrennt.
- Bei Parameterübergaben muss der Typ und der Name, welcher innerhalb der Funktion gilt, angegeben werden.
  Achtung: C verwendet **Pass-By-Value** für die Parameterübergabe! Mehr dazu siehe unten.
- Um eine Funktion in main() verwenden zu können, muss sie dem Compiler bekannt sein. Dies kann dadurch erreicht werden, dass die Funktion vor der main() geschrieben wird, was bei mehreren Funktionen aber schnell unübersichtlich wird, weil die main() immer weiter nach unten wandert.
  Besser ist es Funktionen nur vor der main() zu **deklarieren** und sie nach der main() aus-zu-codieren.
  Bei der Deklaration legt man return typ, name und Anzahl, Reihenfolge und Typ der Übergabe-Parameter fest.
  Codebeispiel:

```
#include<stdio.h>

//declaration the function
int add_two_integer(int,int);

int main()
{
    int a = 3;
    int b = 4;
    //calling the function inside the printf statement
    printf("3+4=%d\n",add_two_integer(a,b));
    return 0;
}

//definition of the function
int add_two_integer(int a, int b)
{
    int c = a + b;
    return c;
}
```

9.2 Parameterübergabe

Die meisten Programmiersprachen verwenden default mäßig das *pass-by-value* System um Parameter an eine Funktion zu übergeben. Das heist es, wird eine Kopie des Wertes der Variable übergeben und die ursprüngliche Variable wird nicht verändert.
Pass-by-reference bedeutet, das eine Referenz (ein Zeiger) auf den Variablen-Wert übergeben wird, und dass alle Modifikationen innerhalb der Funktion, sich auch auf den Variablen-Wert ausserhalb der Funktion auswirken.
C verwendet pass-by-value, man kann aber pass-by-reference simulieren, wenn man Zeiger auf die Variablen, statt Variablen-Namen übergibt.
Am Besten erklären sich die Unterschiede zwischen den zwei Arten an einem Beispiel:

9.2.1 pass-by-value

Beispiel: Der Wert einer int Variable `a` soll in einer `update()` Funktion um eins erhöht werden.
Code:

```
#include<stdio.h>
void update(int);

int main()
{
    int a = 42;
    printf("in main(): a=%d\n",a);
    printf("in main(): update(a) aufrufen\n");
    update(a);
    printf("in main(): a=%d\n",a);
    return 0;
}

void update(int a)
{
    printf("    in update(): a=%d\n",a);
    printf("    in update(): a++ ausführen\n");
    a++;
    printf("    in update(): a=%d\n",a);
}
```

Zugehöriger Output:

```
in main(): a=42
in main(): update(a) aufrufen
    in update(): a=42
    in update(): a++ ausführen
    in update(): a=43
in main(): a=42
```

Man sieht, der Wert der Variable `a` wird in der `main()` **nicht** verändert.

9.2.2 pass-by-reference

Nun überlisten wir das default *pass-by-value* von C, indem wir einen Pointer auf die Variable übergeben. C macht zwar brav *pass-by-value* mit unserem Pointer, aber wir bekommen trotzdem die Möglichkeit den Wert der ursprünglichen Variable in der `update()` Funktion zu verändern.

Code:

```
#include<stdio.h>
void update(int*);

int main()
{
    int a = 42;
    printf("in main(): a=%d\n",a);
    printf("in main(): a=%p\n",&a);
    printf("in main(): update(&a) aufrufen\n");
    update(&a);
    printf("in main(): a=%d\n",a);
    return 0;
}

void update(int *a)
{
    printf("    in update(): *a=%d\n",*a);
    printf("    in update(): a=%p\n",a);
    printf("    in update(): (*a)++ ausführen\n");
    (*a)++;
    printf("    in update(): *a=%d\n",*a);
}
```

Zugehöriger Output:

```
in main(): a=42
in main(): &a=0x7ffc6aa1a404
in main(): update(&a) aufrufen
    in update(): a=0x7ffc6aa1a404
    in update(): *a=42
    in update(): (*a)++ ausführen
    in update(): *a=43
in main(): a=43
```

Man sieht, der Wert der Variable `a` wird, im Vergleich zu vorher, in der `main()` **verändert**!
Nochmals eine kurze Erklärung dazu: Wir übergeben der `update()` Funktion diesmal einen Pointer (in meinem Beispiel: 0x7ffc6aa1a404) welcher auf den Wert (42) der Variable `a` zeigt. C macht das übliche *pass-by-value* und erstellt eine Kopie dieses Zeigers. In der `update()` Funktion gibt es nun eine Variable `a` die eine Zeiger-Variable ist und den Wert 0x7ffc6aa1a404 hat. Achtung: In der `main()` haben wir eine integer Variable mit Namen `a`. Wenn wir nun ein update mit der Zeigervariable ausführen, ändern wir den Wert an der Speicheradresse 0x7ffc6aa1a404, und damit auch den Wert der `int a` Variable in `main()`, weil sie ja auf den gleichen Speicherbereich verweist!
Wer mit den Zeiger-Notationen `*` und `&` noch Schwierigkeiten hat, soll nochmals im Kapitel 6 auf Seite 13 nachlesen!
Wer von den ganzen `a` Variablen in `main()` und `update()` verwirrt ist, wird im Kapitel 10 auf Seite 20 hoffentlich Klarheit erfahren.

9.3 Rekursive Funktionen

Eine *rekursive* Funktion ruft sich selbst immer wieder auf, bis eine Bedingung erfüllt ist. Rekursive Funktionen eignen sich zum Beispiel für Durchläufe von Baumstrukturen wie z.B. Verzeichnissen oder für die Abbildung von mathematischen Folgen.
Ein nettes Beispiel ist die Fibbonacci-Folge:
$f_n = f_{n-1} + f_{n-2}$ mit den Anfangswerten $f_1 = 1$ und $f_0 = 0$.

Code:

```
#include<stdio.h>

int fibbonacci(int);

int main()
{
   int n = 14;
   printf("Fibbonacci Folge:\n");
   for(int i = 0;i<n;i++) {
      printf("fib(%d)=%d\n",i,fibbonacci(i));
   }
}

int fibbonacci(int n)
{
   if(n == 0){
      return 0;
   } else if(n == 1) {
      return 1;
   } else {
      return (fibbonacci(n-1) + fibbonacci(n-2));
   }
}
```

Zugehöriger Output:

```
fib(0)=0
fib(1)=1
fib(2)=1
fib(3)=2
fib(4)=3
fib(5)=5
fib(6)=8
fib(7)=13
fib(8)=21
fib(9)=34
fib(10)=55
fib(11)=89
fib(12)=144
fib(13)=233
```

Achtung. Bei zu vielen rekursiven Aufrufen kann es zu einem Stack-Overflow kommen. Klar worum?

10 Gültigkeitsbereich von Variablen - variable scope

Variablen besitzen eine Gültigkeitsbereich (*scope*). Bis jetzt haben wir nur ***lokale Variablen*** benutzt: Sie sind **nur in innerhalb der Funktion bekannt und gültig, in der sie deklariert sind!**
Daher ist es auch kein Problem, Variablen mit demselben Namen in verschiedenen Funktionen zu deklarieren - es sind trotzdem unterschiedliche Variablen!

10.1 Globale Variablen

Man kann auch ***globale Variablen*** deklarieren - ausserhalb von allen Funktionen - auch der main(). Diese sind dann in allen Funktionen gültig und bekannt - mit einer Einschränkung: Sollte es innerhalb einer Funktion eine *lokale* Variable mit dem gleichen Namen wie der *globalen* Variable geben, ist nur die *lokale* Variable gültig!

So, warum dann nicht mit *globalen* Variablen arbeiten? Scheint doch viel feiner zu sein! Man könnte sich die ganze Parameter-Übergabe und -Rückgabe sparen!

Aber: Der Einsatz von *globalen* Variablen wiederspricht dem Ansatz der *modularen* Programmierung, bei dem man sich Unabhängigkeit und klare Schnittstellen zwischen den einzelnen Funktionen eines Programms wünscht.
Wenn man sich eine Funktion anschaut, sollte klar erkennbar sein welche Daten von außen hereinkommen, wie sie bearbeitet werden und was am Ende der Funktion wieder nach außen gesendet wird.
Der Einsatz von *globalen* Variablen macht diese Nachvollziehbarkeit viel schwieriger und Programme schwer lesbar und fehleranfälliger.

Code:

```
#include<stdio.h>
//declare global variable
int x = 3;
//declare functions
void func_1(void);
void func_2(void);

int main()
{
    printf("in main(): x=%d\n",x);
    func_1();
    func_2();
    printf("in main(): x=%d\n",x);
    return 0;
}

void func_1(void)
{
    //use global variable x
    printf("in func_1(): x=%d\n",x);
    x = 42;
}

void func_2(void)
{
    //use local variable x
    int x = 1;
    printf("in func_2(): x=%d\n",x);
}
```

Zugehöriger Output:

```
in main(): x=3
in func_1(): x=3
in func_2(): x=1
in main(): x=42
```

10.2 Statische Variablen

Eine *locale* Variable existiert nach dem Ende einer Funktion nicht mehr. Falls man bestimmt Informationen über mehrere Funktionsaufrufe hinweg behalten möchte, kann man ein *statische* Variable definieren.
Man deklariert eine *statische* Variable mit dem Schlüsselwort `static` und gibt ihr bei der Deklaration gleich einen definierten Startwert (wird bei späteren Aufrufen ignoriert).
Code:

```
#include<stdio.h>

//declare functions
void func(void);

int main()
{
    func();
    func();
    func();
    return 0;
}

void func(void)
{
    static int x = 1;
    printf("x=%d\n",x);
    x++;
}
```

Zugehöriger Output:

```
x=1;
x=2,
x=3;
```

11 Arrays

Was Arrays (oder Felder, oder Listen) sind und für was sie gut sind wird an dieser Stelle nicht mehr wiederholt, das sollte nach zwei Jahren Python klar sein. Hier wird nur auf die C-Syntax zu Arrays eingegangen. Der Hauptunterschied zu Pythons Listen, besteht darin, dass Arrays eine fixe Größe haben und nicht dynamisch wachsen können. Für dynamische Strukturen siehe Kaptiel: 15 auf Seite: 32

11.1 Deklaration von Arrays

`float messwerte[3];` legt ein Array vom Typ *float* an mit der Größe 3. Der Index des Arrays läuft von 0 - 2.

Deklaration und Zuweisung von Werten in einem Schritt:

```
int myArray[3] = {1,2,42};
```

11.2 Mehrdimensionale Arrays

Sind *nur* Arrays in Arrays ;-)
Für zweidimensionale Arrays lässt sich dies leicht wie eine Spielfläche eines Brettspiels vorstellen z.B. Schach mit 8x8 Felder kann als `int schach_brett[8][8]` realisiert werden.
Auch dreidimensionale Arrays `int wuerfel[z][y][x]` kann man sich mit etwas Fantasie noch ganz gut vorstellen, aber dann wird es für die menschliche Vorstellungsgabe schon etwas schwierig. Für C ist es aber kein Problem ein n-dimensionales Array zu erzeugen: Einfach eine eckige Klammer für jede Dimension
`int crazy_array[d1][d2]...[dn]`.
Für die weiteren Zeilen beschränke ich mich auf zweidimensionale Arrays, weil sie am häufigsten gebraucht werden.

11.2.1 Initialisierung

Es gibt zwei Arten zweidimensionale Arrays zu initialisieren:
Eine leicht lesbare:

```
int array[2][4] = {
    {10, 11, 12, 13},
    {14, 15, 16, 42}
};
```

oder

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 42};
```

was zum gleichen Ergebnis führt, aber auf den ersten Blick weniger leicht verständlich ist.

11.2.2 Verschachtelte Schleifen

Oft nützt man verschachtelte Schleifen um zweidimensionale Arrays mit Werten zu füllen bzw. auszugeben. Im folgenden wird die Verwendung von zwei verschachtelten for-loops zur Ausgabe eines tic-tac-toe Spielfeldes demonstriert:

```
#include <stdio.h>
int main()
{
    //Initialisierung zu einem zufälligen
    //tic tac toe Spielstand
    char ttt[3][3] = {
        {'X','O',' '},
        {' ','O',' '},
        {' ','X',' '},
    };
    //Ausgabe mit vertikalen Trennstrichen
    for(int y = 0; y < 3; y++){
        printf("|");
        for(int x = 0; x < 3; x++){
            printf("%c|",ttt[y][x]);
        }
        printf("\n");
    }
    return 0;
}
```

Zugehöriger Output:

```
|X|O| |
| |O| |
| |X| |
```

12 Strings

Strings sind in C char arrays. Da man sehr oft mit Strings arbeiten muss, gibt es in der `string.h` nützliche Funktionen für die String-Manipulation.

String (`char[]`) deklarieren und zuweisen:
`char text[30] = {"Ich habe Hunger!"};`

Ausgabe einzelner Zeichen bzw. des ganzen Textes:
`printf("Text=%s\n",text);`
`printf("Zeichen=%c\n",text[10]);`

12.1 Formatierte Ausgabe von Strings

Man kann eine links oder rechtsbündige Ausgabe und eine Limitierung der maximalen Zeichen einstellen:

```
#include <stdio.h>

int main()
{
    //normale Ausgabe
    printf(":%s:\n","Hello World");
    //rechtsbuendig auf 20 Zeichen aufgefuellt
    printf(":%20s:\n","Hello World");
    //linksbuendig auf 20 Zeichen aufgefuellt
    printf(":%-20s:\n","Hello World");
    //auf 8 Zeichen beschnitten
    printf(":%.8s:\n","Hello World");
    return 0;
}
```

Zugehöriger Output:

```
:Hello World:
:         Hello World:
:Hello World         :
:Hello Wo:
```

12.2 Funktionen aus string.h

Die String.h Library hat viele spezielle Funktionen um mit Strings zu arbeiten. Sie Link:
https://www.tutorialspoint.com/c_standard_library/string_h.htm

- `strcpy` (String Copy) Man kann eine Zeichenkette in eine andere kopieren.
  Bsp:

```
char string_one[30];
char string_two[30];
//Zeichenkette in String Variable kopieren
strcpy(string_one, "Hi Lord Helmi");
//String Variable in andere String Variable kopieren
strcpy(string_two, string_one);
```

13 Strukturen - selbst definierte Datentypen

Structs oder auf Deutsch Strukturen sind selbstdefinierte Datentypen, die aus mehrern primitiven Datentypen bestehen, aber für uns einen thematischen Zusammenhang haben. Am besten erklärt es sich mit einem Beispiel: Angenommen wir arbeiten an einer Schülerdatenbank und sind damit konfrontiert die Adressen der Schüler abzubilden. Dann brauchen wir dazu mindestens vier Variablen: Name, Straße, PLZ und Land, welche aber zusammen die Adresse bilden.
(P.S. wer von Euch schon Erfahrung mit Objekt-Orientierter-Programmierung hat, dem werden einige der folgenden Punkte bekannt vorkommen ;-)

13.1 Strukturen definieren

Funktioniert mit dem Schlüsselwort `struct` und wirst meist noch vor der `main()` deklariert, damit sie im gesamten Programm *global* bekannt ist.

```
//Definition der Struktur
struct adresse
{
    //Komponenten der Struktur
    char name[30];
    char strasse[30]
    int PLZ;
    char land[30];
};
```

13.2 Struct-Daten erzeugen bzw. zuweisen

Es gibt zwei Möglichkeiten Variablen einer selbst definierten Struktur mit Werten zu füllen:

- Direkt bei der Deklaration der Variable, ähnlich wie bei Arrays.
- Oder über die Punkt-Notation (wie in der Objektorientierung). Diese wird auch verwendet wenn auf die einzelnen Werte der Strukturvariable zugegriffen wird. z.B. bei der Ausgabe.

Beispiel:

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

//Definition der Struktur
struct adresse
{
    //Komponenten der Struktur
    char name[30];
    char strasse[30];
    int plz;
    char land[30];
};
int main()
{
    //Deklaration und Zuweisung in einem Schritt
    struct adresse adress1 = {"John Snow", "Winterfell 42a", 7777, "Westeros"};

    //Deklaration von Variablen der Struktur
    struct adresse adres2, adres3;

    //Zuweisung über Punkt-Notation
    strcpy(adres2.name, "Sepp Hinterkircher");
    strcpy(adres2.strasse, "Hinter der Kirch 12");
    adres2.plz = 6161;
    strcpy(adres2.land, "Almland");

    //Zuweisung mittels Kopie
    adres3 = adres2;

    return 0;
}
```

13.2.1 Arrays von Strukturen

Man kann aus den eigenen Strukturen natürlich ebenso wie aus primitiven Datentypen Felder bzw. Arrays erzeugen.
`struct adresse adressFeld[10];`
Der Zugriff erfolgt nun über den Array-Index und die Punkt-Notation:
`adressFeld[7].plz = 1234;`

13.2.2 Zeiger auf Strukturen

Beim Übergeben von Strukturen an Funktionen hat man (wie bei primitiven Variablen) die Möglichkeit *pass-by-reference* über Zeiger zu realisieren. Dies läuft wie normales *pass-by-reference* (Siehe Kapitel 9.2.2 auf Seite 18) mit der Ausnahme, das es einen Pfeiloperator `->` gibt.
Er ist die vereinfachte Schreibweise für die Verwendung des Inhaltsoperators `*`. Da der Punktoperator `.` Vorrang vor dem Inhaltsoperator `*` hat, muss er in Klammer gesetzt werden und dadurch wird die Schreibweise etwas lang und die Verwendung des Pfeiloperator `->` ist kürzer. Beispiel:
Annahme: Es gibt einen Zeiger auf eine Struktur-Variable:

```
struct adresse *p;
//Zuweisung mit Inhalts- und Punktoperator
(*p).plz = 6161;
//Zuweisung mit Pfeiloperator
p->plz = 6161;
```

14 File I/O

File I/O ist wichtig, vorallem auch in Hinblick auf hardwarenahes Programmieren unter Linux (...where everything is a file).
Unter Linux bzw. Unix hat man zwei Möglichkeiten für I/O-Programmierung. Zum einen die standard ANSI-C-Bibliothek, die Funktionen der höheren Ebene wie `fprintf(), fopen(), fgets()` etc. bereithält und mit *Dateizeiger* (Filepointern) arbeitet. Und zum anderen die elementaren I/O-Funktionen des Kernels wie `open(), read() oder wri`
Sie arbeiten mit Filedeskriptoren, sind natürlich ebenso genormt aber nicht so bequem zu bedienen.
Wir konzentrieren uns an dieser Stelle auf die höheren Funktionen (ANSI-C-Standard).

Zur Bearbeitung von Dateien wird ein **Dateizeiger** benötigt - `FILE*`. Dabei handelt es sich um eine Struktur, die in etwa wie folgt aussieht:

```
typedef struct _iobuf {
  char*  _ptr;
  int    _cnt;
  char*  _base;
  int    _flag;
  int    _file;
  int    _charbuf;
  int    _bufsiz;
  char*  _tmpfname;
} FILE;
```

Darin sind Informationen enthalten wie:

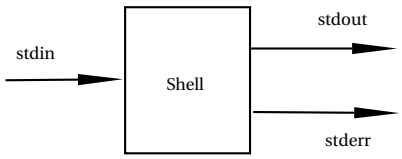
- Anfangsadresse des Puffers
- Puffergröße
- aktueller Pufferzeiger
- Filedeskriptor
- Position des Schreib-/Lesezeigers
- Fehlerflag
- EOF-Flag

Vereinfachte Vorstellung für uns C-Einsteiger: Er markiert die aktuelle Stelle, an der man sich innerhalb der Datei befindet.

14.1 FILE-Zeiger = Stream

Bei dem FILE-Zeiger spricht man auch von einem Datenstrom (englisch Stream). In Linux stellt jeder Prozess drei vordefinierte Standardstreams zur Verfügung:

- FILE* stdin – Standard-engabestrom ist voreingestellt auf die Tastatur
- FILE* stdout – Standardausgabestrom ist voreingestellt auf den Monitor
- FILE* stderr – Standardausgabestrom für Fehlermeldungen ist voreingestellt auf den Monitor



Folgendes Beispiel erlaubt das Einlesen und Ausgeben von der Shell: Soweit nichts Neues, nur mit Exception-Handling. Warum? Wird klar wenn wir nicht selbst die Daten eingeben sonder zum Beispiel den stdin für unser C-Programm so umleiten, dass der Input aus einem File kommt.

```
#include <stdio.h>

int main()
{
    int number;
    for(;;){
        printf("Enter a number or Ctrl-D to exit\n");
        int result = scanf("%d",&number);
        if (result == EOF){
            printf("\n ciao \n");
            break;
        }else{
            if (number == 0){
                fprintf(stderr,"0 geht nicht!\n");
            }else{
                printf("Zahl %d hat den Kehrwert %f\n",number,1.0/number);
            }
        }
    }
    return 0;
}
```

Das Programm einfach einmal normal, dann mit umgeleiteten Input und auch mit umgeleiteten stdout ausprobieren:

```
$ ./runme < numbers.txt               #eingabe der zahlen aus einem txt file
$ ./runme < numbers.txt > output.txt  #nun auch stdout umgeleitet
```

14.2 File öffnen

Zur Bearbeitung von Dateien wird ein *Dateizeiger* (engl. filepointer) benötigt. Er markiert die aktuelle Stelle, an der man sich innerhalb der Datei befindet.
Das eigentliche Öffnen der Datei erfolgt mit dem Befehl *fopen*.

Syntax: `fopen("myFile.txt","r")`

fopen braucht zwei Parameter: Einen String welcher den Filenamen (eventuell mit Pfad) enthält und einen Sting welcher den *mode* (Modus) der Öffnung beschreibt. Zur Auswahl stehen:

- **"r"**: (read) Öffnen einer Datei zum Lesen. Wenn die Datei nicht existiert oder nicht geöffnet werden konnte, gibt fopen() NULL zurück.
- **"w"**: (write) Anlegen einer Datei zum Ändern. Wenn die Datei nicht geändert werden kann bzw. wenn keine Schreibberechtigung besteht, liefert hier fopen() NULL zurück. Wenn unter Windows/MS-DOS die Datei ein Read-only-Attribut hat, kann sie nicht geöffnet werden. Wenn sie schon existiert wird sie überschrieben.
- **"a"**: (append) Öffnet die Datei zum Schreiben oder zum Anhängen an das Ende der Datei. Wenn die Datei nicht vorhanden ist (oder keine Zugriffsrechte bestehen), liefert fopen() wieder NULL zurück.
- **"r+"**: Öffnet die Datei zum Lesen und Schreiben, also zum Verändern. Bei Fehlern oder mangelnden Rechten liefert fopen() auch hier NULL zurück.
- **"w+"**: Anlegen einer Datei zum Ändern. Existiert eine Datei mit gleichem Namen, wird diese zuvor gelöscht. Bei Fehlern oder mangelnden Rechten liefert fopen() hier NULL zurück.
- **"a+"**: Öffnen einer Datei zum Lesen oder Schreiben am Ende der Datei. Falls noch keine Datei vorhanden ist, wird eine angelegt. Bei Fehlern oder mangelnden Rechten liefert fopen() NULL zurück.

Beispielcode:

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

int main(){
   //filepointer anlegen
   FILE *meineDatei;

   //Datei mittels fopen öffnen
   //Bitte Pfad und Dateinamen anpassen
   meineDatei = fopen("test.txt", "r");

   //oder mit langem Pfad unter Linux
   meineDatei = fopen("/home/markus/test.txt", "r");

    //oder mit langem Pfad unter Windows und doppel \\
   meineDatei = fopen("c:\\Windows\\test.txt", "r");

   //hat es funktioniert?
   if(meineDatei == NULL) {
      printf("Konnte Datei \"test.txt\" nicht öffnen!\n");
      return 1;
   }
   return 0;
}
```

14.3 File schliessen

Die Funktion *fclose()* schließt eine Datei (Stream), die zuvor mit *fopen()* geöffnet wurde. Warum sollte man das machen? Wenn sich ein Programm beendet, schließen sich automatisch alle noch offenen Streams. Es gibt zwei gute Gründe, dies dennoch selbst zu tun:
1.) Die Anzahl der geöffneten Dateien ist begrenzt. Sie ist in der Konstante FOPEN_MAX in der Headerdatei <stdio.h> deklariert. Wird versucht, FOPEN_MAX+1 Dateien zu öffnen, dann schlägt dies fehl. Mit fclose() kann wieder ein FILE-Zeiger freigegeben werden.
2.) Wenn eine Datei im Schreibmodus geöffnet wurde, wird diese erst beschrieben, wenn der Puffer (im Hintergrund) voll ist. Ist der Puffer nur teilweise voll und das Programm beendet sich mit einem Fehler, dann sind die Daten im Puffer verloren.
Die Syntax von fclose() ist:

```
#include <stdio.h>

int fclose(FILE *f);
```

14.4 Zeichenweise lesen und schreiben mit fgetc() und fputc()

Beispiel: Zeichenweise lesen mit fgetc():

```
//Zeichenweise aus Datei lesen und ausgeben
#include <stdio.h>
#include <stdlib.h>

int main(){
   int c;
   FILE *datei;

   datei=fopen("test.txt", "r");
   if(datei != NULL) {
      while( (c=fgetc(datei)) != EOF)
         printf("%c\n",c);
   }
   else {
      printf("Konnte Datei nicht finden bzw. oeffnen!\n");
      return 1;
   }

   //file pointer schließen
   fclose(datei);
   return 0;
}
```

Beispiel: Zeichenweise schreiben mit fputc():

```
//Zeichenweise das ABC in eine Datei schreiben mit fputc()
#include <stdio.h>
#include <stdlib.h>

int main(){
   int c;
   FILE *datei;

   datei=fopen("test.txt", "w");
   if(datei != NULL) {
      //ASCII code für A = 65
      for (char c = 65; c <= 90; c++){
         fputc(c, datei);
      }
      fclose(datei);
   }
   else {
      printf("Konnte Datei nicht finden bzw. oeffnen!\n");
      return 1;
   }

   //file pointer schließen
   fclose(datei);
   return 0;
}
```

14.5 Zeilenweise lesen und schreiben mit fgets() und fputs()

Beispiel: Zeilenweise aus Datei einlesen und an STDOUT schreiben.

```
//Zeilenweise einlesen
#include<stdio.h>

int main(){

   //filepointer
   FILE *fp;

   //file oeffnen im lese modus
   fp = fopen("hello.txt", "r");

   //hat es funktioniert?
   if (fp==NULL){
      fprintf(stderr, "Can't open file!\n");
      return 1;
   }

   //zeilen einlesen - muessen in buffer gespeichert werden
   char buffer[100];
   while(fgets(buffer, 100, fp)){
      //gleich wieder rausschreiben
      fputs(buffer, stdout);
   }

   //file pointer schließen
   fclose(fp);

   return 0;
}
```

14.6 Formatiertes lesen und schreiben mit fscanf() und fprintf()

fprintf() und fscanf() funktionieren eigentlich genau gleich wie das schon bekannte scanf() und printf(). Zusätzlich muss nur die Quelle, bzw. das Ziel angegeben werden.
Beispiel: Einlesen einer CSV-Datei die den Aufbau Zahl1, Zahl2, Zahl3: Text haben soll:

```
//reading from a file CSV file, changeing the order and writing
//to another file
#include<stdio.h>

int main(){

   //we need a file pointer to handle files
   FILE *fpCSVin;
   FILE *fpCSVout;

   int a,b,c;
   char buffer2[100];

   fpCSVin = fopen("input.csv","r");
   fpCSVout = fopen("output.csv","w");
   if (fpCSVin == NULL || fpCSVout == NULL){
      fprintf(stderr, "Can't open file!\n");
      return 1;
   }
   while((fscanf(fpCSVin,"%d,%d,%d:%s\n",&a,&b,&c,buffer2)) != EOF ){
      fprintf(fpCSVout,"%d,%d,%d:%s\n",c,b,a,buffer2);
   }

   //closing the file streams
   fclose(fpCSVin);
   fclose(fpCSVout);

   return 0;
}
```

15 Dynamische Speicherverwaltung

Dieses Kapitel ist für Python-Programmierer Neuland und wird deshalb etwas ausführlicher behandelt!
Da C im Vergleich zu Python keine dynamischen Listen bzw. Felder hat, muss ich der Programmierer selbst um dynamische Speicherbereiche kümmern!

15.1 Speicherbereiche eines Prozesses

Wenn ein Programm (oder Prozess) startet, wird im vom Betriebssystem ein Speicherbereich (RAM) zugewiesen. Er darf nur auf diesen Speicherbereich zugreifen! Der Speicherbereich wird in vier Bereiche unterteilt:

- **Code (oder Text)**: Der Maschinencode des Programms. Der Code-Speicher wird in den Arbeitsspeicher geladen, und von dort aus werden die Maschinenbefehle der Reihe nach in den Prozessor (genauer gesagt in die Prozessor-Register) geschoben und ausgeführt.
- **Daten**: m Daten-Speicher befinden sich alle statischen Daten, die bis zum Programmende verfügbar sind (globale und statische Variablen).
- **Stack**: Im Stack-Speicher werden die Funktionsaufrufe mit ihren lokalen Variablen verwaltet. Kann dynamisch wachsen und schrumpfen. Bei rekursiven Funktionsaufrufen kann es zu einem Stack-Overflow kommen.
- **Heap**: Dynamisch reservierbarer Speicher. Der Heap funktioniert ähnlich wie der Stack. Bei einer Speicheranforderung erhöht sich der Heap-Speicher, und bei einer Freigabe wird er wieder verringert. Wenn ein Speicher angefordert wurde, sieht das Betriebssystem nach, ob sich im Heap noch genügend zusammenhängender freier Speicher dieser Größe befindet. Bei Erfolg wird die Anfangsadresse des passenden Speicherblocks zurückgegeben.

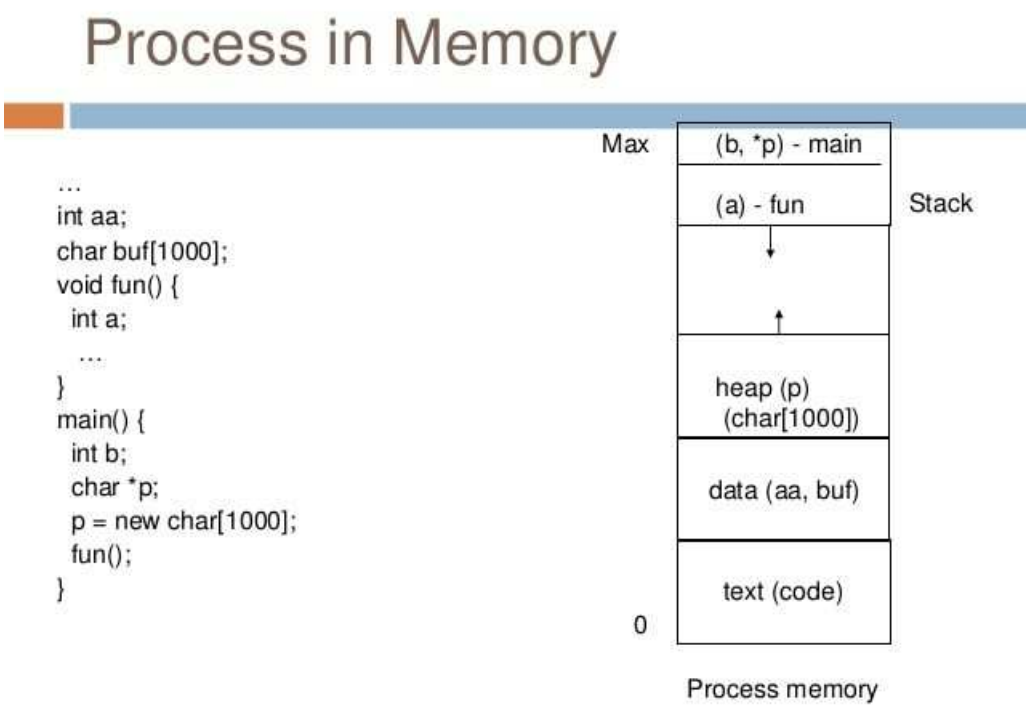


Abbildung 5: Heap und Stack Beispiel ©Birju Tank, IEEE Seminar

15.2 Speicher anfordern mit malloc()

Mit **Memory Allocation** vordern wir zur Laufzeit des Programms auf sichere Art Speicherplatz vom **Heap** an, z.B. für Felder. Mit dem Parameter **size** wird die Größe des Speicherbedarfs in Byte übergeben. Der Rückgabewert ist ein **void-Zeiger** auf den Anfang des Speicherbereichs oder ein **NULL-Zeiger**, wenn kein freier Speicher mehr zur Verfügung steht. Der void-Zeiger soll aussagen, dass der Datentyp des belegten Speicherbereichs unbekannt ist.

```
#include <stdlib.h>

void *malloc(size_t size);
```

Zwei Dinge erscheinen hier eventuell als verwirrend:

- Der Datentyp **size_t**. Dieser ist vom Typ **long int** und wird für die Angabe einer Datengröße in Byte verwendet. Die Funktion sizeof liefert bei Übergabe eines Datentyps wie int dessen benötigte Speichergröße in Byte. Der Datentyp des Rückgabewertes ist size_t.
- Der Rückgabewert ist ein **void-Zeiger** auf den Anfang des Speicherbereichs oder ein NULL-Zeiger, wenn kein freier Speicher mehr zur Verfügung steht. Der void-Zeiger soll aussagen, dass der Datentyp des belegten Speicherbereichs unbekannt ist. Das Casten des **void-Zeiger** auf den gewünschten Datentyp ist nicht unbedingt notwendig, da der void-Zeiger automatisch in den richtigen Typ transformiert wird. Es ist jedoch sauberer und macht den Code leichter lesbar!

15.2.1 Beispiel: Dynamische Speicheranforderung

Angenommen der User möchte eine unbekannte Anzahl an Integern speichern, also soll das Programm dynamisch den dafür benötigten Speicher anlegen:

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

int main() {
        int size=0;
        int* array;

        printf("Bitte Array-Groesse eingeben: ");
        scanf("%d", &size);

        // Speicher reservieren
        array = (int *) malloc(size * sizeof(int));

        if(array != NULL) {
                printf("\nSpeicher ist reserviert\n");
        }else {
                printf("\nKein freier Speicher vorhanden.\n");
                return 1;
        }

    // Zugriff auf das Array
    for(int i = 0; i < size; i++)
    {
        array[i]=i*i;
    }

    printf("A two numbers from the array are:\n");
    printf("array[0]=%d and array[7]=%d\n",array[0],array[7]);

    return 0;
}
```

15.3 Speicher freigeben mit free()

Wenn wir Speicher vom Heap angefordert haben, sollten wir diesen auch wieder zurückgeben. Der allozierte Speicher wird mit folgender Funktion freigegeben:

```
#include <stdlib.h>

void free (void *p)
```

Der Speicher wird übrigens auch ohne einen Aufruf von `free()` freigegeben, wenn sich das Programm beendet, aber das hängt von der Speicherverwaltung des Betriebssystems ab. Es ist auf alle Fälle die sauberere Lösung es im Programm zum machen!

15.3.1 Beispiel: Speicher mit free() freigeben

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

int main(void) {
   int* p = (int*) malloc(sizeof(int));

   if(p != NULL) {
      *p=99;
      printf("Allokation erfolgreich ... \n");
      }
   else {
      printf("Kein virtueller RAM mehr verfügbar ... \n");
      return 1;
   }
   if(p != NULL)
      free(p);
   return 0;
}
```

Da der **Heap** üblicherweise aus Performance-Gründen nicht wieder reduziert wird, kann man eventuell auf den freigegebenen Speicherplatz und dessen Inhalt immer noch zugreifen. Aber dieses Verhalten ist Betriebsystem abhängig und sollte auf keinen Fall willentlich verwendet werden!
Am besten überschreibt man nach der Freigabe den Zeiger mit einem NULL-Zeiger, dann kann man definitiv nicht mehr auf den Speicherbereich zugreifen.

```
free(p);
p = NULL;
```

Dies könnte man auch wie folgt in ein Makro verpacken:

```
#define my_free(x)  free(x); x = NULL
```