2. Die Programmiersprache C - Auffrischung

编程语言C - 进修

Michael Schöttner
Betriebssysteme und Systemprogrammierung

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

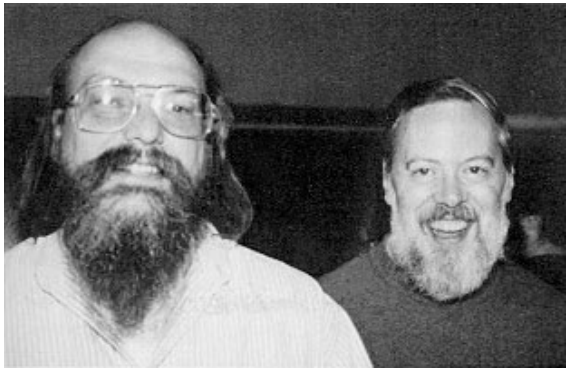
Warum C?

- Die kommerziellen Betriebssysteme (UNIX, MacOS, Linux, Microsoft Windows) sind alle in C geschrieben
- In der Forschung gibt es viele Betriebssysteme in unterschiedlichen Programmiersprachen, auch eigens erfundene
- C ist sehr weit verbreitet, knapp hinter Python, Java und C++
  - Für fast alle Plattformen verfügbar
  - C-Programme erlauben große Effizienz
  - C ist sehr flexibel, hat dadurch aber einige Nachteile, die manchmal zu oft sehr schwer auffindbaren Fehlern führen
    - Kein strenges Typsystem
    - Array-Grenzen werden nicht geprüft
    - Manuelle Speicherverwaltung

商业操作系统（UNIX、MacOS、Linux、Microsoft Windows）都是用C语言编写的。
- 在研究中，有许多不同编程语言的操作系统，包括那些内部发明的操作系统
- C语言的应用非常广泛，仅次于Python、Java和C++。
- 可用于几乎所有平台
- C语言程序允许极大的效率 - C是非常灵活的，但这也有一些缺点
这有时会导致通常非常难以发现的错误
- 没有严格的类型系统 - 不检查阵列的边界 - 手动内存管理

Erfinder von C

- UNIX wurde von Dennis Ritchie und Ken Thompson entwickelt.
- C wurde 1969–1973 von Dennis Ritchie in den Bell Laboratories für die Programmierung von UNIX entwickelt. Er schrieb den ersten C Compiler.
  - Er stützte sich dabei auf die Programmiersprache B, die Ken Thompson und Dennis Ritchie in den Jahren 1969/70 geschrieben hatten.
  - Der Name C entstand als Weiterentwicklung von B.



https://de.wikipedia.org/wiki/C_(Programmiersprache)#/media/File:Ken_n_dennis.jpg

- UNIX是由Dennis Ritchie和Ken Thompson开发的。
- C语言是由Dennis Ritchie于1969-1973年在贝尔实验室开发的，用于UNIX编程。他写了第一个C语言编译器。
- 他以Ken Thompson和Dennis Ritchie在1969/70年编写的编程语言B为基础。
- C的名字出现了作为B的进一步发展。

2.1 Vorschau

- Sprachstandards
- Datentypen
- Funktionen
- Präprozessor
- Module und Makefiles
- Zeiger
- Arrays
- Dynamischer Speicher
- Ein- und Ausgabefunktionen
- man pages

- 语言标准
- 数据类型
- 职能
- 预处理程序
- 模块和Makefile
- 指针
- 数组
- 动态内存
- 输入和输出功能
- 人工页

2.2 Hello-World

用C语言编译器进行翻译

- Übersetzen mit dem C-Compiler: `cc -o hello hello.c`

通过调用./hello来执行

- Ausführen durch Aufruf von `./hello`

预处理程序指令

Präprozessoranweisung 包括库中的标准IO（用于printf）。

→ Bibliothek Standard-IO einbinden (für printf)

Hauptfunktion, als Einstieg ins Programm

Programm hat normal terminiert.

```
#include <stdio.h>

int main (void) {
    printf("Hallo Welt\n");
    return 0;
}
```

Zeilenvorschub

- Bemerkungen: Syntax ist ähnlich zu Java
- Achtung `cc -o hello.c hello` überschreibt die Quelltextdatei

注意 `cc -o hello.c hello` 覆盖了源代码文件

2.3 Sprachstandards

有几种语言规格，有细微的差别
- 可以用一个编译器开关来指定，例如-std=c90
- C89、C90和ANSI C表示同一标准

- Es gibt verschiedene Sprachspezifikationen mit kleinen Unterschieden
- Kann mit einem Compiler-Schalter festgelegt werden, z.B. `-std=c90`
  - C89, C90 und ANSI C bezeichnen den gleichen Standard

Compiler-Schalter	Standard
-std=c89	Erster offizieller Standard; ANSI 1989
-std=c90	ISO Version von C89 im Jahr 1990
-std=c95	Erweiterungen, u.a. Unicode-Unterstützung in Bibliotheken
-std=c99	Mit C99 flossen einige aus C++ bekannte Erweiterungen in die Sprache C ein, zum Beispiel Inline-Funktionen und die Möglichkeit, Variablen innerhalb der for-Anweisung zu deklarieren.
-std=c11	Unter anderem Unterstützung von Multithreading

GNU-Sprachstandards

GNU语言标准

- Es gibt weitere Sprachvarianten für den GNU-Compiler, welche Erweiterungen zu den ISO Standards definieren

还有一些GNU编译器的语言变体，它们定义了对ISO标准的扩展。

Compiler-Schalter	Standard
-std=gnu89	ISO C90 mit einigen GNU-Erweiterungen
-std=gnu90	Siehe gnu89
-std=gnu99	Erweiterungen zu C99
-std=gnu11	Erweiterungen zu C11
...	...

- Details hier: http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html

2.4 Elementare Datentypen

符号：有符号（带符号；默认）；无符号（无） 值的范围部分取决于语言标准和处理器的字宽 → sizeof(变量)返回以字节为单位的大小

- Vorzeichen: **`signed`** (mit Vorzeichen; default); **`unsigned`** (ohne)
- Wertebereich teilweise abhängig vom Sprachstandard und Wortbreite des Prozessors → **`sizeof(variable)`** liefert Größe in Bytes

Datentyp	Bezeichner	Größe
Zeichen	`char`	1 Byte
Ganze Zahl	`short`	2 Byte
“	`int`	2 od. 4 Byte
“	`long`	4 od. 8 Byte
Gleitkommazahl	`float`	4 Byte
“	`double`	8 Byte
“	`long double`	10 Byte

Weitere elementare Datentypen

从C99开始，在stdint.h中增加了以下类型
- 设置每个数据类型的字节数

- Ab C99 wurden folgende Typen ergänzt, in **`stdint.h`**
- Legen Anzahl der Bytes für jeden Datentyp fest

Datentyp	Bezeichner	Größe
Zeichen	`int8_t` `uint8_t`	1 Byte
Ganze Zahl	`int16_t` `uint16_t`	2 Byte
“	`int32_t` `uint32_t`	4 Byte
“	`int64_t` `uint64_t`	8 Byte

Eigene Datentypen

- Eigene Datentypen mit **typedef**

```
#include <stdio.h>

typedef int Integer;

int main (void) {
    Integer i = 0;
}
```

Unser Name als Datentyp int

typedef 关键字，您可以使用它来为类型取一个新的名字。下面的实例为单字节数字定义了一个术语 BYTE：

typedef unsigned char BYTE;
在这个类型定义之后，标识符 BYTE 可作为类型 unsigned char 的缩写，例如：

BYTE b1, b2;
按照惯例，定义时会大写字母，以便提醒用户类型名称是一个象征性的缩写，但您也可以使用小写字母，如下：

typedef unsigned char byte;

Strukturen als Datentypen

- Strukturen und **typedef**

```c
#include <stdio.h>

struct datum {
     int tag;
     int monat;
     int jahr;
};

int main (void) {
     struct datum d;

     d.tag = 1;
     d.monat = 1;
     d.jahr = 2019;
}
```

```c
#include <stdio.h>

struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
} book = {"C 语言", "RUNOOB",
"编程语言", 123456};
```

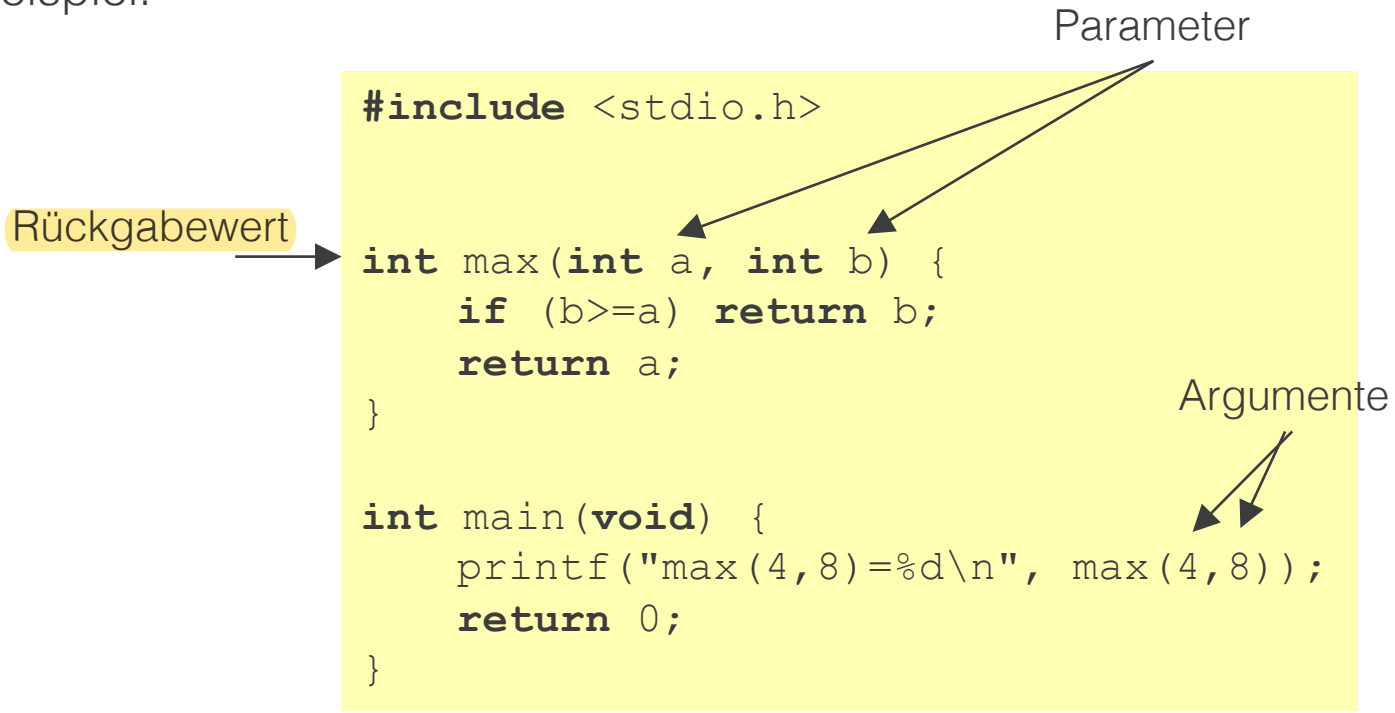
```c
#include <stdio.h>

struct datum {
     int tag;
     int monat;
     int jahr;
};

typedef struct datum datum_t;

int main (void) {
     datum_t d;

     d.tag = 1;
     d.monat = 1;
     d.jahr = 2019;
}
```

Neuer Datentyp-name

```c
/* Book1 详述 */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "Nuha Ali");
  strcpy( Book1.subject, "C Programming
Tutorial");
  Book1.book_id = 6495407;
```

2.5 Funktionen

- Beispiel:




```
#include <stdio.h>


int max(int a, int b) {
    if (b>=a) return b;
    return a;
}

int main(void) {
    printf("max(4,8)=%d\n", max(4,8));
    return 0;
}
```

Funktionen (2)

- Beispiel:

```
#include <stdio.h>

void inkrement(void) {
    static int i = 1;
    printf("Wert von i: %d\n",i);
    i++;
}

int main(void) {
    inkrement();
    inkrement();
    inkrement();
    return 0;
}
```

Statische Variable
- Wert bleibt zwischen Funktionsaufrufen erhalten
- Im Prinzip eine globale Variable

- 在函数调用之间保留值
- 原则上，一个全局变量

Ausgabe:

```
Wert von i: 1
Wert von i: 2
Wert von i: 3
```

Funktionsdeklaration

- Soll eine Funktion vor ihrer Definition verwendet werden, kann sie durch eine Deklaration bekannt gemacht werden (Prototyp) →

- Oft ist die Deklaration in einer separaten Header-Datei (siehe später)

- 如果一个函数在被定义之前就要被使用，可以通过声明的方式让它知道
(原型) →通常情况下，声明是在一个单独的头文件中。
- 通常情况下，声明是在一个单独的头文件中
(见后)

```c
#include <stdio.h>

void inkrement(void);

int main(void) {
    inkrement();
    inkrement();
    inkrement();
    return 0;
}

void inkrement(void) {
    static int i = 1;
    printf("Wert von i: %d\n",i);
    i++;
}
```

2.6 Präprozessor

2.6 预处理程序

- 在C源码被传递给C编译器之前，它要被一个宏预处理器处理。
- 对预处理程序的指令由 #字符在行的开头
- 预处理程序语句的语法与语言无关。
- 预处理程序语句不以;!

- Bevor eine C-Quelle dem C-Compiler übergeben wird, wird sie durch einen Makro-Präprozessor bearbeitet
- Anweisungen an den Präprozessor werden durch ein #-Zeichen am Anfang der Zeile gekennzeichnet
- Die Syntax von Präprozessoranweisungen ist unabhängig von der Sprache
- Präprozessoranweisungen werden nicht durch ; abgeschlossen!

Einfügen von Dateien

插入文件

#include将另一个文件的内容插入到C源文件中；通常是几个源文件都需要的头文件。

- **`#include`** fügt den Inhalt einer anderen Datei in eine C-Quelldatei ein; i.d.R. Header-Dateien, die für mehrere Quelldateien benötigt werden:
  - Deklaration von Funktionen, Strukturen, externen Variablen
  - Definition von Makros

- 函数、结构、外部变量的声明
- 宏的定义

如果文件名用< >括起来，就会把一个标准的头文件复制到
- 根据安装情况搜索路径

- Wird **Dateiname** durch **< >** geklammert, wird eine **Standard-Header-Datei** einkopiert
  - Suchpfade gemäß Installation

```
#include < Dateiname >
oder
#include " Dateiname "
```

如果文件名用""括起来，用户的头文件就会被复制进来（路径细节与C文件的路径相对）

- Wird **Dateiname** durch **" "** geklammert, wird eine Header-Datei des Benutzers einkopiert (Pfadangaben relativ zum Pfad der C-Datei)

Standard-Headerdateien - Beispiele

Include-Datei	Beschreibung des Inhalts
`math.h`	Mathematische Funktionen
`stdarg.h`	Funktionen für variable Anzahl an Parametern 参数数量可变的函数
`stdio.h`	Standard-I/O
`stdlib.h`	Verschiedene Hilfsfunktionen
`string.h`	Zeichenketten-Funktionen
`time.h`	Datum und Uhrzeit
`...`	...

Makrodefinitionen

预处理器。用于在编译器处理程序之前预扫描源代码
C语言标准规定，预处理是指前4个编译阶段（phases of translation）。
三字符组与双字符组的替换
行拼接（Line splicing）：把物理源码行（Physical source line）中的换行符转义字符处理为普通的换行符，从而把源程序处理为逻辑行的顺序集合。
单词化（Tokenization）：处理每行的空白、注释等，使每行成为token的顺序集。
扩展宏与预处理指令（directive）处理.

有两种宏：
类似对象的宏（无参数的宏）
类似函数的宏（带参数的宏），在第一个标识符与左括号之间，绝不能有空格

- 宏允许进行简单的文本替换
- 一个宏是由#define语句定义的。通常用大写字母

- Makros ermöglichen einfache textuelle Ersetzungen
- Ein Makro wird durch die **`#define`**–Anweisung definiert, i.d.R. mit Großbuchstaben
- Syntax: **`#define`** `MAKRONAME Ersatztext`

语法 `#define MAKRONAME 替换文本`

- Eine Makrodefinition bewirkt, dass der Präprozessor im nachfolgenden Text der C-Quelle alle Vorkommen von Makroname durch Ersatztext ersetzt

- 一个宏定义会使预处理程序在C源的以下文本中用替换文本替换所有出现的宏名称

- Beispiel: **`#define PI 3.1415926f`**

Makrodefinitionen (2)

- Es sind auch parameterisierte Makros erlaubt

```
#define MAX(x,y)  ( (x)<=(y) ? (y) :(x) )
```

- 有条件的编纂
- 使用#ifdef symbol和#endif，只有当符号被定义时，程序文本的中间部分才被编译。
- 例如，对于调试版本

- Bedingtes Compilieren
  - Mit **#ifdef symbol** und **#endif** wird der dazwischen liegende Bereiche des Programmtextes nur compiliert, wenn das **symbol** definiert ist.
  - Zum Beispiel für Debug-Versionen

如果宏已经定义，则返回真

#if　如果给定条件为真，则编译下面代码

```
void inkrement(void) {
    #ifdef DEBUG
        printf("Hallo Welt\n");
    #endif
    ...
```

结束一个 #if……#else 条件编译块

2.7 Module

一个C程序的部分内容可以分布在几个.c文件（C源文件）中。

- Teile eines C-Programms können auf mehrere `.c`-Dateien (C-Quelldateien) verteilt werden
- Logisch zusammengehörende Daten und die darauf operierenden Funktionen sollten in einem **Modul** (eine C Datei) zusammengefasst werden
- Jede C-Quelldatei kann separat übersetzt werden (Option `-c`)
  - Zwischenergebnis der Übersetzung wird in je einer `.o`-Datei abgelegt

```
% cc -c prog.c      (erzeugt Datei prog.o )
% cc -c f1.c        (erzeugt Datei f1.o )
% cc -c f2.c f3.c   (erzeugt f2.o und f3.o )
```

- 逻辑上相关的数据和对其进行操作的函数应结合在一个模块中（一个c文件）。
- 每个C源文件都可以单独翻译（选项-c）。
- 翻译的中间结果分别存储在一个.o文件中。

Module (2)

cc （®V7）：C编译器

- Das Kommando **cc** kann auch mehrere **.o**–Dateien (Module) zusammen in ein Ergebnis binden:

```
% cc -o prog prog.o f1.o f2.o f3.o f4.c f5.c
```

- 在一个函数可以从另一个模块调用之前，它必须被声明→包含头文件。
- 注意：千万不要在.c源文件的帮助下使用

- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muss sie deklariert werden → Header-Datei einbinden

- Achtung: Auf keinen Fall **.c**–Quelldateien mit Hilfe der **#include**–Anweisung in andere Quelldateien einkopieren

Module (3)

```c
/* helper.h */

void inkrement(void);
```

```c
/* helper.c */

void inkrement(void) {
    static int i = 1;
    printf("Wert von i: %d\n", i);
    i++;
}
```

```c
/* main.c */

#include <stdio.h>
#include "helper.h"

int main(void) {
    inkrement();
    inkrement();
    inkrement();
    return 0;
}
```

```
% cc -o prog helper.c main.c
```

Globale Variablen

- Falls diese in anderen Modulen genutzt werden müssen, so ist eine Deklaration in der Header-Datei notwendig:

```
extern int global;
```

如果这些必须在其他模块中使用，则有必要在头文件中进行声明。

关键字extern，可以在一个文件中引用另一个文件中定义的变量或者函数

- Möchte man das verhindern, so muss in der .c-Datei das Schlüsselwort **`static`** vor dem Typnamen gesetzt werden

如果你想防止这种情况，你必须在.c文件中的类型名称前面设置关键字static。

- Achtung: globale Variablen möglichst vermeiden.
  Diese sind problematisch bei Multi-Threading (siehe später)

注意：如果可能的话，避免使用全局变量。
这些都是多线程的问题（见下文）

Makefiles

带有命令的文本文件，作为程序make的输入。
- 用于编译由几个或许多文件组成的大型程序。
- 通常也有安装命令
- 主要成分。
- 规则：针对一个目标；命令和依赖性
- 变量：为了提高可读性
- 评论：用于解释
- 指令：额外的控制选项（类似于C语言）。

- Textdatei mit Befehlen als Eingabe für das Programm **`make`**
  - Zum Übersetzen von größeren Programmen die aus mehreren oder vielen Dateien bestehen
  - Oft auch mit Installationsbefehlen

https://www.ruanyifeng.com/blog/2015/02/make.html

- Hauptbestandteile:
  - **Regeln**: für ein Ziel; Befehle und Abhängigkeiten
  - **Variablen**: zur Verbesserung der Lesbarkeit
  - **Kommentare**: zur Erklärung
  - **Direktive**: zusätzliche Steuerungsmöglichkeiten (ähnlich in C)

Makefiles (2)

Kommentar

Variablen

Tab vor jedem Befehl!

Eine Regel

```
# Ein Beispiel
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary

$(OUTPUT): helper.o main.o
    $(CC) $(CFLAGS) helper.o main.o -o $(OUTPUT)

helper.o: helper.c
    $(CC) $(CFLAGS) -c helper.c -o helper.o

main.o: main.c
    $(CC) $(CFLAGS) -c main.c -o main.o

clean:
    rm *.o $(OUTPUT)
```

调试-版本，所有警告
Debug-Version, All Warnings

Abhängigkeiten

2.8 Formatierte Eingabe 格式化的输入

- Mithilfe von **`scanf`** aus **`stdio.h`**
- Parameter ähnlich zu **`printf`**
- Liest ein oder mehrere Argumente von der Standardeingabe (i.d.R. Tastatur)
- Beispiel:

- 在scanfaus stdio.h的帮助下
- 与printf相似的参数
- 从标准输入（通常是键盘）读取一个或多个参数。

```c
#include <stdio.h>

int main() {
    int zahl;

    printf("Zahl eingeben: ");
    scanf("%d",&zahl);
    printf("Eingabe = %d\n", zahl);
    return 0;
}
```

要读入的变量是用&操作符指定的（后面会有更多介绍）。

- Einzulesende Variablen werden mit dem &-Operator angegeben. (mehr dazu später)

2.8 Formatierte Eingabe

- Bemerkungen
  - `scanf` prüft nicht die Eingabe
  - Probleme mit Pufferung:
    - Betriebssystem puffert Eingaben
    - Unter Umständen liefern mehrfache `scanf`-Aufrufe nicht das gewünschte Ergebnisse
    - Abhilfe schafft hier zeichenweises Einlesen mithilfe von beispielsweise `getchar()`

- 备注
- scanf不检查输入
- 缓冲的问题。
- 操作系统缓冲区的输入
- 多次调用scanf可能不会得到期望的结果。
- 例如，这可以通过使用getchar()一次读入字符来补救。

2.9 Einschub: Quizfrage 1

- Was gibt dieses Programm aus?

```c
#include <stdio.h>

int main(void) {
    int a=5, b=10, c=15;

    printf("%d %d %d", a, b, c);

    return 0;
}
```

2.9 Einschub: Quizfrage 1

- Was gibt dieses Programm aus?

  → 5 10 15

```c
#include <stdio.h>

int main(void) {
    int a=5, b=10, c=15;

    printf("%d %d %d", a, b, c);

    return 0;
}
```

2.9 Einschub: Quizfrage 2

- Was steht in den Variablen a, b und c nach dem **`scanf`**-Aufruf?

```c
#include <stdio.h>

int main(void) {
    int a, b, c;

    scanf("%d %d %d", a, b, c);

    return 0;
}
```

2.9 Einschub: Quizfrage 2

- Was steht in den Variablen a, b und c nach dem **`scanf`**-Aufruf?
  → Das Programm stürzt ab mit einer Segmentation fault
  → Der Adressoperator **`&`** fehlt bei den Parametern in **`scanf`**!

→ 程序崩溃，出现分段故障
→ scanf的参数中缺少地址操作符&!

```c
#include <stdio.h>

int main(void) {
    int a, b, c;

    scanf("%d %d %d", a, b, c);

    return 0;
}
```

scanf("%d", &i);

2.10 Zeiger

Einordnung

- **Konstante**
  Bezeichnet einen Wert

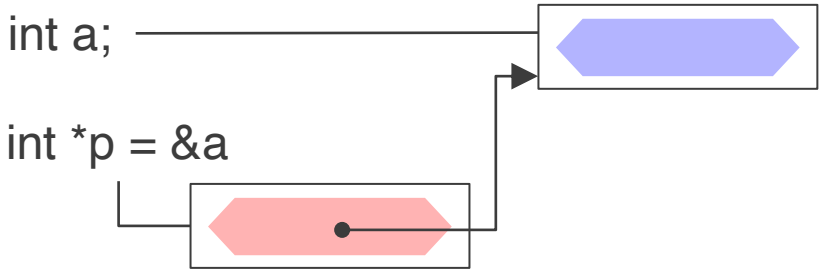
- 恒定
表示一个值

- **Variable**
  Bezeichnet ein Datenobjekt

- 变化的
指定一个数据对象

- **Zeiger-Variable (engl. pointer)**
  Bezeichnet eine Referenz auf ein Datenobjekt

- 指针变量 指定对一个数据对象的引用。



2.10 Zeiger

你可以通过一个指针间接地访问另一个变量。
- 指针是"正常"的变量。
- 它们将一个内存地址存储为一个值。
- 它们通常有一个类型（例外：void*）。
- 使用的例子。
- 指向一个数组/字符串的开头的指针。
- 动态数据结构
- 调用参数应可改变的函数（见后）

Einordnung

- Über einen Zeiger kann man indirekt auf eine andere Variable zugreifen

- Zeiger sind ”normale“ Variablen:
  - Speichern als Wert eine Speicheradresse
  - i.d.R. haben sie einen Typ (Ausnahme: `void *`).

- Anwendungsbeispiele:
  - Zeiger auf den Beginn eines Arrays / Zeichenkette
  - Dynamische Datenstrukturen
  - Funktionen deren Aufrufparameter veränderbar sein sollen (siehe später)

Zeiger-Operatoren

- Adressoperator &
  - liefert die Referenz/Adresse einer Variablen im Speicher

- 地址操作员 &
- 返回内存中一个变量的引用/地址

- 撤消运算符 *
- 返回一个被引用的变量的内容
- * 属于该类型，但被写成了名字

- Dereferenzierungsoperator *
  - Liefert den Inhalt einer referenzierten Variablen
  - * gehört zum Typ, wird aber beim Namen geschrieben

- Zeigerdefinition mithilfe des Symbols *
  - Bei einer Variablendefinition
  - Zeigt an, dass ein Zeiger definiert wird

- 使用符号*的指针定义
- 对于一个变量的定义
- 表明正在定义一个指针

Adressoperator

```
void beispiel(void) {
    int a = 0xCAFE;
    int *p = &a;

    printf("Zeiger: %x\n", p);
    printf("Inhalt: %x\n", *p);
}
```

Dereferenzierung

Zeiger als Argumente beim Funktionsaufruf

- Parameterübergabe erfolgt in C **call-by-value**
- Funktion erhält eine Kopie des Variableninhalts

- 参数在C语言中是逐值传递的
- 函数接收变量内容的副本
- 这也适用于指针
- 被调用的函数不能改变指针。
- 但它可以访问被引用的内存内容并改变内容。
- 这种参数转移对应的是逐次调用的语义。

- Dies gilt auch für Zeiger
  - Die aufgerufene Funktion kann den Zeiger nicht verändern
  - Aber sie kann darüber auf den referenzierten Speicherinhalt zugreifen und den Inhalt ändern

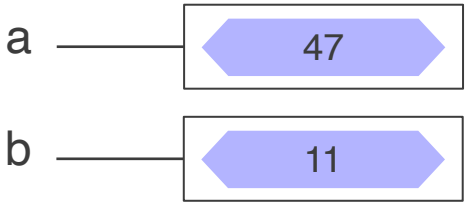
- Diese Parameterübergabe entspricht der **call-by-reference** Semantik

Zeiger als Argumente beim Funktionsaufruf (2)

- Beispiel

```
void swap (int *px, int *py) {
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;
}

int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
    ...
}
```

STOP

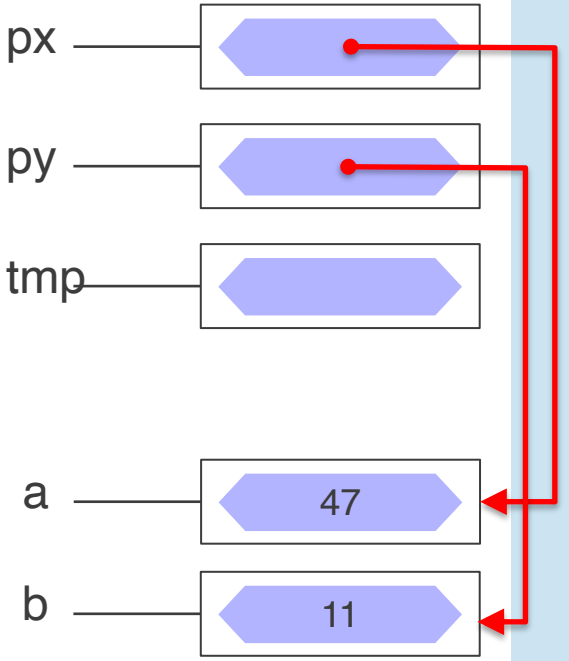


Zeiger als Argumente beim Funktionsaufruf (2)

- Beispiel

STOP

```
void swap (int *px, int *py) {
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;
}

int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
    ...
}
```

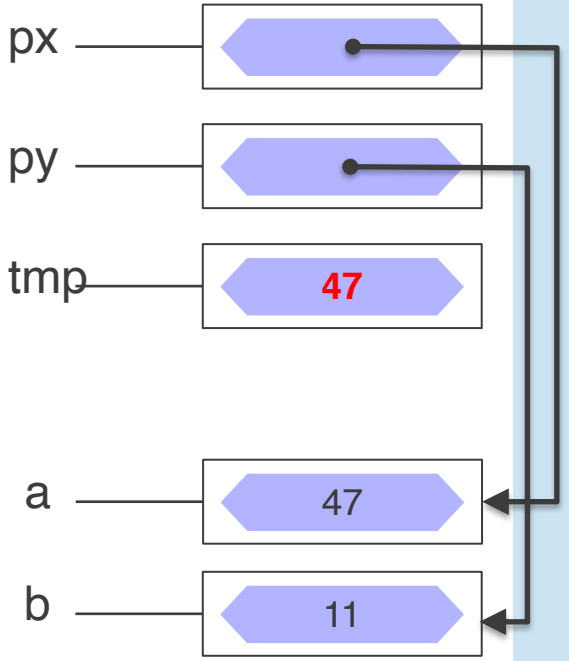


Zeiger als Argumente beim Funktionsaufruf (2)

- Beispiel

```
void swap (int *px, int *py) {
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;
}

int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
    ...
}
```

STOP

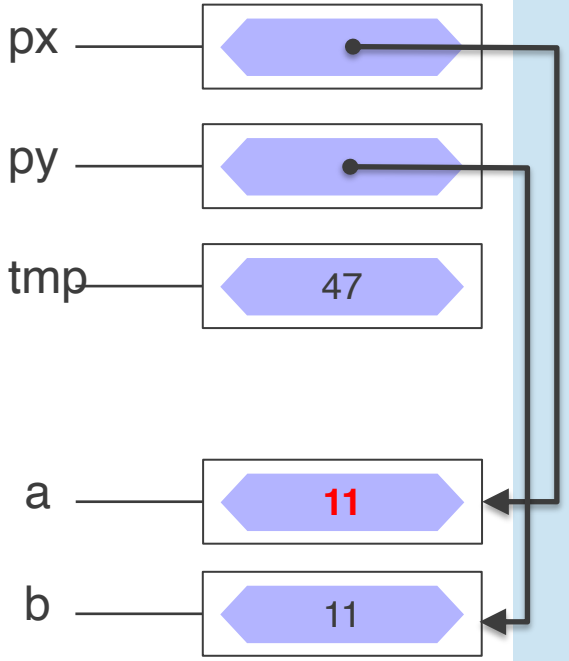


Zeiger als Argumente beim Funktionsaufruf (2)

- Beispiel

```
void swap (int *px, int *py) {
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;
}

int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
    ...
}
```

STOP

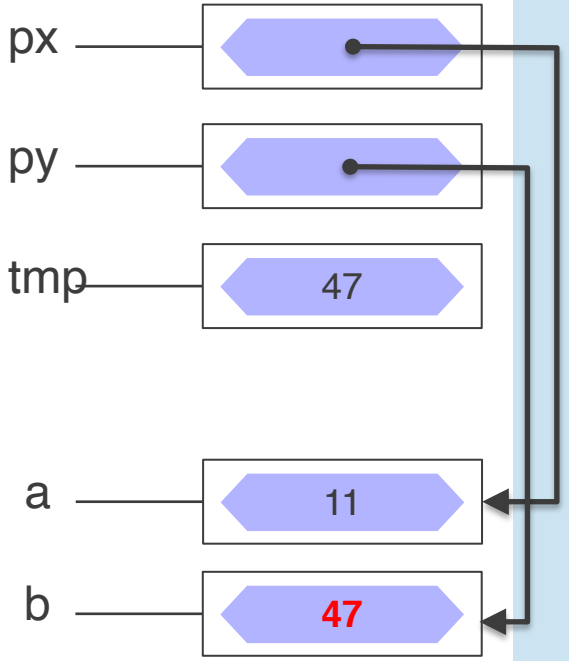


Zeiger als Argumente beim Funktionsaufruf (2)

- Beispiel

```
void swap (int *px, int *py) {
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;
}

int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
    ...
}
```

STOP



Zeiger auf Strukturen

- 以前，一个指针引用了一个变量的内存位置
- 指针也可以引用内存

- Bisher hat ein Zeiger einen Speicherplatz einer Variable referenziert
- Ein Zeiger kann auch den Speicher-platz einer **`struct`** referenzieren →
- Dereferenzieren eines Zeigers auf eine **`struct`** mit **`(*dat).jahr`** oder eleganter mit **`dat->jahr`**

解除一个指向structmit(*dat).year的指针。
或更优雅地使用dat->year

```c
#include <stdio.h>

struct datum {
     int tag;
     int monat;
     int jahr;
};

typedef struct datum datum_t;

void change(datum_t *dat) {
     dat->jahr = 1900;
}

void no_change(datum_t dat) {
     dat.jahr = 1900;
}
```

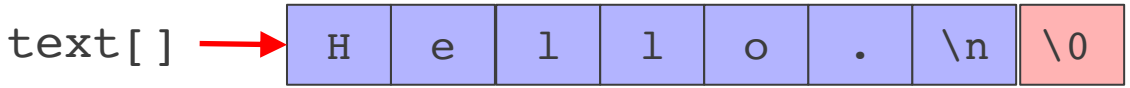
2.11 Arrays - Grundlagen

- Anlegen eines Arrays mit: `int arr[10];`
- Oder mit Initialisierung:
```
int arr[] = {1,2,3};
int arr[3] = {1,2,3};
```
- Zugriff ab Index 0

- Achtung: Unterschiede zu Java
  - Es gibt kein `.length` → Länge muss separat verwaltet werden!
  - Beim Zugriff findet keine Indexprüfung statt → gefährlich

Zeichenketten

- Zeichenketten werden in einem Array gespeichert
- Beispiel: `char text[] = {"Hello.\n"};`
- Alternative: `char *text = "Hello.\n";`
- Speicherformat



- Wichtig ist der Terminator `\0` am Ende.
  Damit kann `strlen(text)` die Länge einer Zeichenkette ermitteln

- String-Funktionen in `string.h`

2.12 Main mit Parameter

- Übergabe von Argumenten vom Terminal / Konsole
- Der Kommandoname ist immer in **`argv[0]`**
- Beispiel: **`$ test 1 2 3`**

- 从终端/控制台传递参数
- 命令名称总是在argv[0]中。
- 例如：$ test 1 2 3

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;

    for (i=0; i<argc; i++) {
        printf("%d: %s\n", i, argv[i]);
    }
    return 0;
}
```

2.13 Dynamischer Speicher

- Notwendig für dynamische Datenstrukturen, Puffer etc.
- Speicher anfordern mit **`void* malloc(size_t size)`** (size: Anzahl Bytes)
- Freigabe von Speicher mit **`void free(void *ptr);`**

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


int main(int argc, char *argv[]) {

    int *ptr = (int*) malloc( 1024*sizeof(int) );
    if (ptr != NULL)
        free(ptr);
}
```

Speicher für 1024 Integer

Typecast

Hat malloc geklappt?

2.13 Dynamischer Speicher

- NULL ist ein vordefinierter Zeiger
  - wird zum Testen von Funktionsergebnissen verwendet
  - `malloc` liefert NULL zurück, falls Aufruf fehlschlägt

- Achtung:
  - C hat keine Garbage Collection wie Java
  - Nicht mehr benötigter Speicher muss manuell freigegeben werden
  - Aber nicht zu früh, sonst gibt es undefinierte Zugriffe

- NULL是一个预定义的指针
- 是用来测试函数结果的
- 如果调用失败，malloc返回NULL
- 请注意。
- C语言没有像Java那样的垃圾收集
- 不再需要的内存必须被手动释放
- 但不能太早，否则会出现未定义的访问

Beispiel: einfach verkettete Liste (Auszug)

```c
#include <stdio.h>

struct node {
    int element;
    struct node *next;
};

typedef struct node node_t;
typedef struct node *node_p;

int main (void) {
    node_p ptr = (node_p) malloc( sizeof(node_t) );

    ptr->element = 1;
    ptr->next = NULL;

    free(ptr);
}
```

2.14 Ein- und Ausgabefunktionen

- Stream-Funktionen in **`stdio.h`** deklariert
  - **`fopen`**: Datenstrom ¨offnen
  - **`fclose`**: Datenstrom schließen
  - **`feof`**: testet auf Dateiende im Stream
  - **`ferror`**: testet auf Fehler im Stream
  - **`fgetc`**: zeichenweise lesen vom Stream
  - **`fgets`**: zeilenweise lesen vom Stream
  - **`fgetpos`**: Position im Stream ermitteln
  - **`fprintf`**: formatierte Ausgabe an Stream
  - **`fputc`**: zeichenweise schreiben in den Stream
  - **`fputs`**: schreibt einen String in den Stream
  - **`fscanf`**: formatierte Eingabe vom Stream
  - **`fseek`**: Dateizeiger neu positionieren

- 在stdio.h中声明的流函数
- fopen: 打开数据流
- fclose: 关闭流
- feof: 测试流中文件的结束情况
- ferror：测试流中的错误。
- fgetc：从流中逐个读取字符
- fgets: 逐行读取流
- fgetpos: 确定流中的位置
- fprintf: 格式化的输出到流
- fputc：逐个字符写入流中。
- fputs：将一个字符串写到流中。
- fscanf：来自流的格式化输入
- fseek。重新定位文件指针

2.14 Ein- und Ausgabefunktionen

- Weitere Funktionen aus `stdio.h`
  - `getchar`: Zeichenweise lesen von `stdin`
  - `gets`: liest String von `stdin` (unsichere Funktion)
  - `printf`: Formatierte Ausgabe an `stdout`
  - `putchar`: zeichenweise schreiben an `stdout`
  - `puts`: Zeichenkette an Stream `stdout`
  - `scanf`: formatierte Eingabe von `stdin`
  - `sprintf`: formatierte Ausgabe in einem String
  - `sscanf`: formatierte Eingabe aus einem String

- 来自stdio.h的其他函数
- getchar：从stdin中逐个读取字符。
- gets: 从stdin读取字符串(不安全函数)
- printf: 格式化的输出到stdout
- putchar：逐个字符写到stdout。
- 把。字符串到stdout流
- scanf：来自stdin的格式化输入
- sprintf：在一个字符串中进行格式化输出
- sscanf：从一个字符串格式化输入

2.15 man pages

- Bietet ausführliche Information zu vielen Programmen und Funktionen
- Ist aufgeteilt in verschiedene Sections:
  - (1) Kommandos
  - (2) Systemaufrufe
  - (3) C-Bibliotheksfunktionen
  - (5) Dateiformate (spezielle Datenstrukturen etc.)
  - (7) Verschiedenes (z.B. IP, GPL, Zeichensätze, ...)

- 提供关于许多方案和功能的详细信息。
- 被分为不同的部分。
- (1) 指挥
- (2) 系统调用
- (3) C库函数
- (5) 文件格式（特殊数据结构等）。
- (7) 杂项（如IP、GPL、字符集...）。
- 调用：man 3 printf或直接调用man printf

- Aufruf: `man 3 printf` oder einfach `man printf`