

Einführung in die Programmiersprache C

Thomas Stibor

Fakultät für Informatik
Technische Universität München

Danke an Christian Schneider für die ursprünglichen Folien

25.10.2010

Themen

Warum heute noch C?

Grundlegende Syntax

Vom Quellcode zum Programm

Der Präprozessor

Arrays und Pointer

Komplexe Datentypen

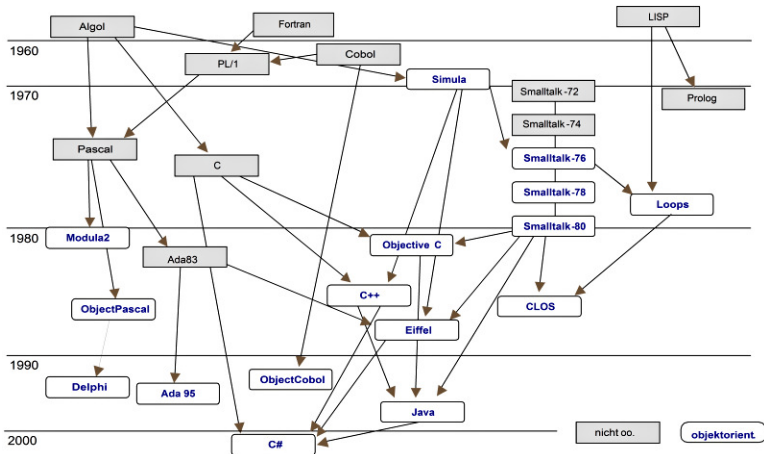
Funktionsparameter

Bibliotheksfunktionen

Programmierumgebungen

Literatur

Historie der Programmiersprachen



- ▶ Erste Standardisierung 1989: **C89**
- ▶ Überarbeiteter Standard 1999: **C99**

Erscheinungsjahr von C: 1972

Stand der damaligen **Computertechnik**:

- ▶ Seit **kurzem** Verwendung von **Magnetspeichern**
 - ▶ Davor: Lochstreifen
- ▶ **Kein** virtueller Speicher
- ▶ **Kein** Multi-Tasking
- ▶ **Keine** graphische Benutzeroberfläche

Einfache **Mikroprozessoren**, z. B. Intel 8008:

- ▶ **500kHz** Taktfrequenz
- ▶ **16kB** Arbeitsspeicher
- ▶ **100kB** Festspeicher

Oberstes Ziel: Effiziente Programmierung

Heutige Relevanz

Ist C nicht längst obsolet?

Heutige Relevanz

Ist C nicht längst obsolet? **Nein!**

Heutige Relevanz

Ist C nicht längst obsolet? **Nein!**

- ▶ **Maschinennahes** Programmieren bei gleichzeitig hohem **Abstraktionsniveau**
 - ▶ Direkter Speicherzugriff über Pointer
 - ▶ Einfache Bit-Arithmetik
 - ▶ Einbinden von Inline-Assemblercode
 - ▶ Komplexe Datenstrukturen und Kapselung
- ▶ Grundlage für **speicher-** und **recheneffiziente** Programmierung
 - ▶ Betriebssysteme, eingebettete Systeme, Grafikanwendungen, ...
- ▶ **Vorbild** für viele moderne Programmiersprachen
 - ▶ C++, Java, C#, Perl, PHP, ...
 - ▶ Wer C beherrscht, kommt schnell mit diesen Sprachen klar

Kritik an C

Aus **heutiger Sicht** ergeben sich einige **Probleme**:

- ▶ Freier **Speicherzugriff** ist **fehlerträchtig**
 - ▶ 80% der Programmierfehler in C sind solche Fehler
- ▶ Explizite **Speicherverwaltung** führt zu **Speicherlecks**
- ▶ **Trennung von Deklaration und Definition** in Header und Quelle ist umständlich
 - ▶ Später mehr dazu
- ▶ Hohe Ausdrucksstärke kann zu **schwer verständlichem** Code führen

Themen

Warum heute noch C?

Grundlegende Syntax

Vom Quellcode zum Programm

Der Präprozessor

Arrays und Pointer

Komplexe Datentypen

Funktionsparameter

Bibliotheksfunktionen

Programmierumgebungen

Literatur

Kontrollstrukturen

Syntax der **Kontrollstrukturen** von Java sehr ähnlich zu C:

- ▶ `for (;;;) {...}`
- ▶ `while () {...}`
- ▶ `do {...} while ();`
- ▶ `if () {...} else {...}`
- ▶ `switch () { case (): ...; break; default: ...; }`
- ▶ `() ? ... : ...;` (Kurzform für if-else)
- ▶ `goto ...;`

Verwendung von Variablen

Syntax im **Umgang mit Variablen** von Java sehr ähnlich zu C:

- ▶ Deklaration von Variablen: `int i;`
- ▶ Zuweisung: `a = b;`
- ▶ Arithmetik: `+` `-` `*` `/` sowie `++` `--` `+=` `-=` `*=` `/=`
- ▶ Vergleiche: `==` `!=` `>=` `<=` `>` `<`
- ▶ Logische Operatoren: `&&` (AND), `||` (OR), `!` (NOT)
- ▶ Bitweise Arithmetik: `<<` und `>>` shiften, `~` invertieren, `&` (AND), `|` (OR), `^` (XOR-Verknüpfung)

Unterschiede zu Java in der Behandlung von Variablen:

- ▶ Müssen **am Anfang** von Funktionen deklariert werden
 - ▶ Seit C99 nicht mehr
- ▶ Müssen **explizit initialisiert** werden, sonst Wert **undefiniert**

Verwendung von Funktionen

- ▶ Definition und Verwendung von **Funktionen** analog zu Java
- ▶ Bei Verwendung der Funktion **vor der Definition** muss diese zuerst **deklariert** werden:

```
1  /* declare max(...), but don't define it yet */
2  int max(int a, int b);
3
4  /* use later defined max(...) */
5  int lower_limit_zero(int x) {
6      return max(x, 0);
7  }
8
9  /* definition of max(...) */
10 int max(int a, int b) {
11     return (a > b) ? a : b;
12 }
```

Eintrittsfunktion: `main`

- ▶ `main`-Methode bildet **Eintrittspunkt** beim Start eines Programms: `int main(int argc, char **argv);`
- ▶ Parameter `argc` liefert Anzahl der Kommandozeilen-Parameter
- ▶ Parameter `argv` ist ein String-Array mit den Parametern
- ▶ **Rückgabewert** ist **Terminalstatus** des Programms
 - ▶ Wert null: normal beendet
 - ▶ Wert ungleich null: Fehlercode

Beispiel: „Hello world“-Programm in C:

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4     printf("Hello world.\n");
5     return 0;
6 }
```

Datentypen: Ganzzahlen

- ▶ **Größe** und **Wertebereich** der Ganzzahlen kann auf unterschiedlichen Architekturen **variieren**
- ▶ C-Standard garantiert nur **Mindestgröße** für Datentypen
 - ▶ Einzige Ausnahme: **char** ist immer **genau 1 Byte** groß
- ▶ Typische Werte für Windows/Linux auf x86 und MacOS:

<i>Typ</i>	<i>Bytes</i>	<i>Wertebereich</i>
char	1	-128 ... +127
unsigned char	1	0 ... +255
short	2	-32 768 ... +32 767
unsigned short	2	0 ... +65 535
int	4	-2 147 483 648 ... +2 147 483 647
unsigned int	4	0 ... +4 294 967 295
long	4	-2 147 483 648 ... +2 147 483 647
unsigned long	4	0 ... +4 294 967 295
(long long)	8	-2^{63} ... $+2^{63} - 1$
(unsigned long long)	8	0 ... $+2^{64} - 1$

Achtung: ANSI-C garantiert nur 2 Byte große **ints**!

Datentypen: Gleitkommazahlen

- ▶ Verwendung der **IEEE-Formate** in 32 und 64 Bit
- ▶ **Größe** der Datentypen ist **standardisiert** wie folgt:

<i>Typ</i>	<i>Bytes</i>	<i>Wertebereich</i>	<i>Genauigkeit</i>
float	4	$\pm 1,2 \cdot 10^{-38} \dots \pm 3,4 \cdot 10^{+38}$	6 Stellen
double	8	$\pm 2,3 \cdot 10^{-308} \dots \pm 1,7 \cdot 10^{+308}$	15 Stellen

Konvertierung von Datentypen

- ▶ Operation mit Ganzzahlen erzeugt wieder Ganzzahl
- ▶ Bei **gemischten** Operanden Verwendung von **Gleitkomma-Arithmetik**
- ▶ Explizite **Konvertierung** von verschiedenen Datentypen mittels (`<Typ>`) möglich, z. B.: `int i = (int) 1.9;`

Konvertierung von Datentypen

- ▶ Operation mit Ganzzahlen erzeugt wieder Ganzzahl
- ▶ Bei **gemischten** Operanden Verwendung von **Gleitkomma-Arithmetik**
- ▶ Explizite **Konvertierung** von verschiedenen Datentypen mittels (`<Typ>`) möglich, z. B.: `int i = (int) 1.9;`

Problem: Ergebnistyp wird für Operation nicht berücksichtigt

```
1 int i = 2, n = 5;  
2 float f = i/n;
```

Konvertierung von Datentypen

- ▶ Operation mit Ganzzahlen erzeugt wieder Ganzzahl
- ▶ Bei **gemischten** Operanden Verwendung von **Gleitkomma-Arithmetik**
- ▶ Explizite **Konvertierung** von verschiedenen Datentypen mittels (`<Typ>`) möglich, z. B.: `int i = (int) 1.9;`

Problem: Ergebnistyp wird für Operation nicht berücksichtigt

```
1 int i = 2, n = 5;  
2 float f = i/n;           ← f = 0
```

⇒ **Unerwartetes Ergebnis**

Konvertierung von Datentypen

- ▶ Operation mit Ganzzahlen erzeugt wieder Ganzzahl
- ▶ Bei **gemischten** Operanden Verwendung von **Gleitkomma-Arithmetik**
- ▶ Explizite **Konvertierung** von verschiedenen Datentypen mittels (`<Typ>`) möglich, z. B.: `int i = (int) 1.9;`

Problem: Ergebnistyp wird für Operation nicht berücksichtigt

```
1 int i = 2, n = 5;  
2 float f = i/n;           ← f = 0
```

Lösung: Konvertierung eines Operanden in Gleitkommazahl

```
1 int i = 2, n = 5;  
2 float f = (float)i/n;    ← f = 0,4
```

⇒ Unerwartetes Ergebnis

Definition eigener Datentypen

- ▶ Schlüsselwort `typedef` erlaubt Definition eigener Datentypen
- ▶ Verwendung wie normale Typen

```
1  /* define "byte" */  
2  typedef  
3     unsigned char byte;  
4  byte b = 255;  
5  byte buf[1024];
```

Definition eigener Datentypen

- ▶ Schlüsselwort `typedef` erlaubt Definition eigener Datentypen
- ▶ Verwendung wie normale Typen

```
1  /* define "byte" */
2  typedef
3     unsigned char byte;
4  byte b = 255;
5  byte buf[1024];
```

Beispiel: Eigener Typ `bool`

- ▶ C kannte ursprünglich **keinen** eigenen Datentyp für logische Werte
- ▶ Ganzzahlen stellen **implizit boole'sche Werte** dar:
 - ▶ Wert von **null** entspricht `false`
 - ▶ Wert **ungleich null** entspricht `true`
- ▶ Boolean-Typ lässt sich leicht simulieren

```
1  /* poor man's bool */
2  #define true 1
3  #define false 0
4  typedef int bool;
```

Themen

Warum heute noch C?

Grundlegende Syntax

Vom Quellcode zum Programm

Der Präprozessor

Arrays und Pointer

Komplexe Datentypen

Funktionsparameter

Bibliotheksfunktionen

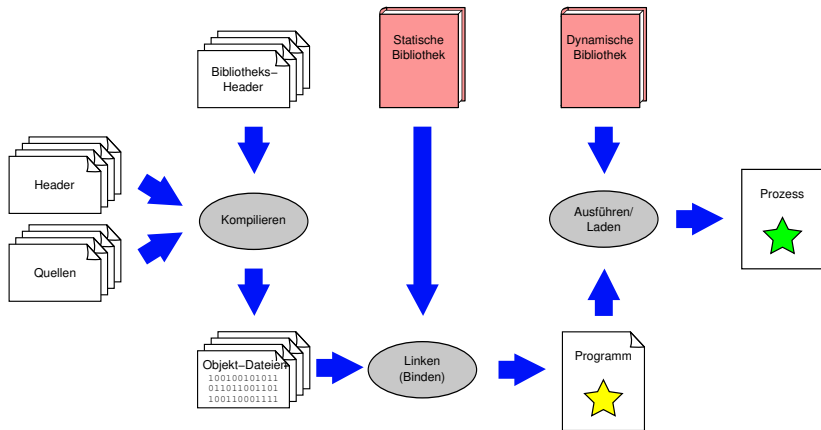
Programmierumgebungen

Literatur

Dateitypen

- ▶ **Headerdatei**, Endung `.h`
 - ▶ Nach außen sichtbare Schnittstelle (Funktionen, Datentypen, globale Variablen)
 - ▶ Wird in Quelldateien eingebunden
 - ▶ Keine Implementierung von Funktionalität
- ▶ **Quelldatei**, Endung `.c`
 - ▶ Implementierung der Funktionalität
 - ▶ Nutzung der Schnittstellen anderer eingebundener Header
- ▶ **Objektdatei**, Endung `.o`
 - ▶ Kompilierter Maschinencode (binär)
 - ▶ Enthält noch Referenzen auf Symbole aus anderen Objekten und Bibliotheken
- ▶ **Programmdatei**, Endung `.exe` bzw. Execute-Bit gesetzt
 - ▶ Ausführbares Programm
 - ▶ Enthält evtl. noch Referenzen zu dynamischen Bibliotheken

Kompilieren, Binden, Laden



Organisation des Quellcodes

- ▶ Aufteilung von Deklaration und Implementierung in zwei Dateien: **Header** und **Quelle**
 - ▶ Lokale Funktionen müssen nicht zuvor deklariert werden

Header-Datei: foo.h

```
1 void foobar(int x);
```

Quelldatei: foo.c

```
1 #include "foo.h"
2
3 /* local function */
4 int max(int a, int b) {
5     return (a > b) ? a : b;
6 }
7
8 /* exported function */
9 void foobar(int x) {
10     int i = max(x, 0);
11     ...
12 }
```

Kompilieren des Quellcodes

Annahme:

- ▶ Zwei Quelldateien `foo.c` und `bar.c`
- ▶ Programm soll `foobar` heißen
- ▶ Verwendung des *GNU C Compilers* (andere Compiler analog)

Kompilieren:

1. `gcc -Wall -c foo.c` (erzeugt `foo.o`)
2. `gcc -Wall -c bar.c` (erzeugt `bar.o`)

Binden (Linken):

3. `gcc -o foobar foo.o bar.o` (erzeugt `foobar`)

Kompilieren des Quellcodes

Annahme:

- ▶ Zwei Quelldateien `foo.c` und `bar.c`
- ▶ Programm soll `foobar` heissen
- ▶ Verwendung des *GNU C Compilers* (andere Compiler analog)

Kompilieren:

1. `gcc -Wall -c foo.c` (erzeugt `foo.o`)
2. `gcc -Wall -c bar.c` (erzeugt `bar.o`)

Binden (Linken):

3. `gcc -o foobar foo.o bar.o` (erzeugt `foobar`)

Hinweis:

- ▶ Schalter `-Wall` nicht notwendig, aber **dringend** empfohlen
- ▶ Aktiviert sämtliche Compilerwarnungen
- ▶ Warnungen sind **meist Programmierfehler**, daher ernst nehmen

Themen

Warum heute noch C?

Grundlegende Syntax

Vom Quellcode zum Programm

Der Präprozessor

Arrays und Pointer

Komplexe Datentypen

Funktionsparameter

Bibliotheksfunktionen

Programmierumgebungen

Literatur

Funktionsweise des Präprozessors

- ▶ Wird automatisch **vor dem Kompilieren** einer Quelldatei ausgeführt
- ▶ Ersetzt **symbolische Namen** im Quelltext durch definierte Werte
- ▶ Ermöglicht **bedingte Kompilierung** auf Basis von definierten Konstanten und der Systemumgebung
- ▶ Erlaubt die Definition von **Makros**, die syntaktisch wie normale C-Funktionen verwendet werden können
- ▶ Wird durch Befehle mit einleitendem „#“ angesprochen
 - ▶ Beispiel: `#include "foo.h"`

Definition von Konstanten

Syntax: #define <Symbol> <Wert>

Definition von Konstanten

Syntax: #define <Symbol> <Wert>

Problem: Array-Größe durch Konstante festlegen

```
1  const int bufsize = 1024;
2  char buf[bufsize];
3
4  /* copy at most bufsize chars from src to buf */
5  strncpy(buf, src, bufsize);
```

Definition von Konstanten

Syntax: #define <Symbol> <Wert>

Problem: Array-Größe durch Konstante festlegen

```
1  const int bufsize = 1024;
2  char buf[bufsize];    ← Compilerfehler: variably modified »buf« at file scope
3
4  /* copy at most bufsize chars from src to buf */
5  strncpy(buf, src, bufsize);
```

Definition von Konstanten

Syntax: #define <Symbol> <Wert>

Problem: Array-Größe durch Konstante festlegen

```
1  const int bufsize = 1024;
2  char buf[bufsize];    ← Compilerfehler: variably modified »buf« at file scope
3
4  /* copy at most bufsize chars from src to buf */
5  strncpy(buf, src, bufsize);
```

Lösung: Definition einer Präprozessor-Konstanten

```
1  #define BUFSIZE 1024
2  char buf[BUFSIZE];
3
4  /* copy at most BUFSIZE chars from src to buf */
5  strncpy(buf, src, BUFSIZE);
```

Definition von Konstanten

Syntax: #define <Symbol> <Wert>

Problem: Array-Größe durch Konstante festlegen

```
1  const int bufsize = 1024;
2  char buf[bufsize];    ← Compilerfehler: variably modified »buf« at file scope
3
4  /* copy at most bufsize chars from src to buf */
5  strncpy(buf, src, bufsize);
```

Lösung: Definition einer Präprozessor-Konstanten

```
1  #define BUFSIZE 1024    ← Definition des Symbols BUFSIZE als 1024
2  char buf[BUFSIZE];     ← Ersetzen von BUFSIZE durch 1024
3
4  /* copy at most BUFSIZE chars from src to buf */
5  strncpy(buf, src, BUFSIZE); ← Ersetzen von BUFSIZE durch 1024
```

Bedingte Kompilierung (1)

Syntax: `#ifdef <Symbol>, #ifndef <Symbol>, #endif`

Bedingte Kompilierung (1)

Syntax: #ifdef <Symbol>, #ifndef <Symbol>, #endif

Beispiel 1: Debug-Ausgabe

```
1 void foo(int n)
2 {
3     #   ifdef DEBUG
4         /* debug output */
5         printf("%s:%u): n=%d\n", __FILE__, __LINE__, n);
6     #   endif
7         ...
8 }
```

Bedingte Kompilierung (1)

Syntax: #ifdef <Symbol>, #ifndef <Symbol>, #endif

Beispiel 1: Debug-Ausgabe

```
1 void foo(int n)
2 {
3 #   ifdef DEBUG      ← Folgenden Code nur übersetzen, falls DEBUG definiert ist
4     /* debug output */
5     printf("(s:%u): n=%d\n", __FILE__, __LINE__, n);
6 #   endif           ← Ab hier wieder normal fortfahren
7     ...
8 }
```

Bedingte Kompilierung (1)

Syntax: #ifdef <Symbol>, #ifndef <Symbol>, #endif

Beispiel 1: Debug-Ausgabe

```
1 void foo(int n)
2 {
3 #   ifdef DEBUG      ← Folgenden Code nur übersetzen, falls DEBUG definiert ist
4     /* debug output */
5     printf("(%s:%u): n=%d\n", __FILE__, __LINE__, n);
6 #   endif          ← Ab hier wieder normal fortfahren
7     ...
8 }
```

Aktivierung der Debug-Ausgaben auf zwei Arten möglich:

1. In Header-Datei global definieren: `#define DEBUG`
2. Durch Compilerschalter, z. B. beim GCC durch `-D<Symbol>`:
`gcc -DDEBUG -o foo foo.c`

Bedingte Kompilierung (2), Fehlermeldungen

Syntax:

```
#if <Bedingung>, #elif <Bedingung>, #else, #error <Nachricht>
```

Bedingte Kompilierung (2), Fehlermeldungen

Syntax:

`#if` <Bedingung>, `#elif` <Bedingung>, `#else`, `#error` <Nachricht>

Beispiel 2: Einbinden von systemabhängigen Headern

```
1 #if (__WIN32__ || _MSC_VER)
2 #   include <windows.h>
3 #elif (__unix__ || __linux__)
4 #   include <unistd.h>
5 #else
6 #   error "Unknown operating system"
7 #endif
```

Bedingte Kompilierung (2), Fehlermeldungen

Syntax:

`#if` <Bedingung>, `#elif` <Bedingung>, `#else`, `#error` <Nachricht>

Beispiel 2: Einbinden von systemabhängigen Headern

```
1 #if (__WIN32__ || _MSC_VER)           ← Vom Compiler definierte Symbole
2 #   include <windows.h>
3 #elif (__unix__ || __linux__)        ← Vom Compiler definierte Symbole
4 #   include <unistd.h>
5 #else
6 #   error "Unknown operating system"  ← Kompilierung abbrechen
7 #endif
```

Wechselseitige Includes

Problem: Wechselseitige Includes führen zu endloser Rekursion:

Header-Datei: foo.h

```
1 #include "bar.h"  
2  
3 void foo();
```

Header-Datei: bar.h

```
1 #include "foo.h"  
2  
3 void bar();
```

⇒ Compilerfehler

Wechselseitige Includes

Problem: Wechselseitige Includes führen zu endloser Rekursion:

Header-Datei: foo.h

```
1 #include "bar.h"
2
3 void foo();
```

Header-Datei: bar.h

```
1 #include "foo.h"
2
3 void bar();
```

⇒ Compilerfehler

Lösung: Einschachtelung in Präprozessor-Bedingungen:

Header-Datei: foo.h

```
1 #ifndef FOO_H
2 #define FOO_H
3 #include "bar.h"
4
5 void foo();
6 #endif
```

Header-Datei: bar.h

```
1 #ifndef BAR_H
2 #define BAR_H
3 #include "foo.h"
4
5 void bar();
6 #endif
```

Makros als Funktionen

Syntax:

```
#define <Symbol>(<Param1>, <Param2>, ...) <C-Code>
```

Makros als Funktionen

Syntax:

```
#define <Symbol>(<Param1>, <Param2>, ...) <C-Code>
```

Beispiel: Funktionen für Minimum und Maximum

```
1 #define min(a, b) ((a < b) ? a : b)
2 #define max(a, b) ((a > b) ? a : b)
3
4 int x = 5, y = 10;
5 int i = min(x, y), j = max(x, y);
```

Makros als Funktionen

Syntax:

```
#define <Symbol>(<Param1>, <Param2>, ...) <C-Code>
```

Beispiel: Funktionen für Minimum und Maximum

```
1 #define min(a, b) ((a < b) ? a : b)
2 #define max(a, b) ((a > b) ? a : b)
3
4 int x = 5, y = 10;
5 int i = min(x, y), j = max(x, y);
```

Warnung:

- ▶ Verwendung solcher Makros als Funktionen kann **unerwartete Nebeneffekte** haben und wird **nicht** empfohlen
- ▶ Beispiel: `int i = max(x++, y++);`
- ▶ Deshalb: „Echte“ Funktionen verwenden, wo möglich

Themen

Warum heute noch C?

Grundlegende Syntax

Vom Quellcode zum Programm

Der Präprozessor

Arrays und Pointer

Komplexe Datentypen

Funktionsparameter

Bibliotheksfunktionen

Programmierumgebungen

Literatur

Statische Arrays

- ▶ Deklaration von Arrays mit **statischer** Größe: `int a[5];`
 - ▶ Größe muss bei der Übersetzung bekannt sein
 - ▶ Inhalt des Arrays wird **nicht initialisiert**
- ▶ Deklaration von Arrays **mit Initialisierung**:
`int a[] = {23, 99, -8, 42, 12};`
- ▶ **Kein** Operator zur Ermittlung der Länge
 - ▶ Länge eines Arrays muss separat gespeichert werden:
`const int a_size = 5;` oder `#define A_SIZE 5`
- ▶ Größe kann **nicht** zur Laufzeit geändert werden

	a[0]	a[1]	a[2]	a[3]	a[4]
a:	23	99	-8	42	12

Zeichenketten

- ▶ Strings in C sind **null-terminierte Arrays** von Zeichen
- ▶ Ende der Zeichenkette ist durch dezimalen Wert null gekennzeichnet
 - ▶ Wird bei statischer Initialisierung automatisch eingefügt
- ▶ `char s[] = "hello";` ergibt:

s:

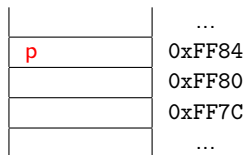
h	e	l	l	o	\0
---	---	---	---	---	----

Konzept der Pointer

- ▶ Zeiger auf **beliebige Adresse** im Speicherbereich
 - ▶ **Inhalt** des Speicherbereichs spielt **keine Rolle**
 - ▶ Inhalt des Speichers **kann** entsprechend des **Pointer-Typs** interpretiert werden

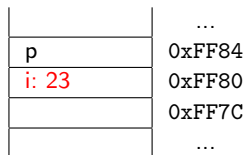
Konzept der Pointer

- ▶ Zeiger auf **beliebige Adresse** im Speicherbereich
 - ▶ **Inhalt** des Speicherbereichs spielt **keine Rolle**
 - ▶ Inhalt des Speichers **kann** entsprechend des **Pointer-Typs** interpretiert werden
- ▶ Pointer eines Typs mit * deklarieren:
`int *p;`



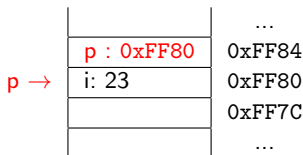
Konzept der Pointer

- ▶ Zeiger auf **beliebige Adresse** im Speicherbereich
 - ▶ **Inhalt** des Speicherbereichs spielt **keine Rolle**
 - ▶ Inhalt des Speichers **kann** entsprechend des **Pointer-Typs** interpretiert werden
- ▶ Pointer eines Typs mit * deklarieren:
`int *p;`
- ▶ Adresse einer beliebigen Variable mit & ermitteln (**referenzieren**):
`int i = 23;`



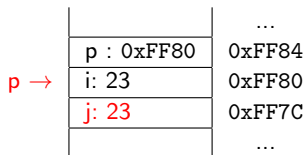
Konzept der Pointer

- ▶ Zeiger auf **beliebige Adresse** im Speicherbereich
 - ▶ **Inhalt** des Speicherbereichs spielt **keine Rolle**
 - ▶ Inhalt des Speichers **kann** entsprechend des **Pointer-Typs** interpretiert werden
- ▶ Pointer eines Typs mit * deklarieren:
`int *p;`
- ▶ Adresse einer beliebigen Variable mit & ermitteln (**referenzieren**):
`int i = 23; p = &i;`



Konzept der Pointer

- ▶ Zeiger auf **beliebige Adresse** im Speicherbereich
 - ▶ **Inhalt** des Speicherbereichs spielt **keine Rolle**
 - ▶ Inhalt des Speichers **kann** entsprechend des **Pointer-Typs** interpretiert werden
- ▶ Pointer eines Typs mit * deklarieren:
`int *p;`
- ▶ Adresse einer beliebigen Variable mit & ermitteln (**referenzieren**):
`int i = 23; p = &i;`
- ▶ Auf Inhalt des referenzierten Speichers mit * zugreifen (**dereferenzieren**):
`int j = *p;`



Pointer versus Arrays

Arrays in C sind nichts anderes als **Pointer**: `int a[5];`

	a[0]	a[1]	a[2]	a[3]	a[4]
a:	23	99	-8	42	12

- ▶ a ist Pointer auf Speicherbereich der Größe $5 \cdot \text{sizeof}(\text{int})$
- ▶ $a[n]$ dereferenziert Speicher an Adresse $a + n \cdot \text{sizeof}(\text{int})$

Pointer versus Arrays

Arrays in C sind nichts anderes als **Pointer**: `int a[5];`

	a[0]	a[1]	a[2]	a[3]	a[4]
a:	23	99	-8	42	12

- ▶ a ist Pointer auf Speicherbereich der Größe $5 \cdot \text{sizeof}(\text{int})$
- ▶ a[n] dereferenziert Speicher an Adresse $a + n \cdot \text{sizeof}(\text{int})$
- ▶ Folgende Beispiele sind semantisch äquivalent:

```
1 int sum = 0;
2 int i;
3 for (i = 0; i < 5; i++)
4     sum += a[i];
```

```
1 int sum = 0;
2 int *p;
3 for (p = a; p < a + 5; p++)
4     sum += *p;
```

Achtung: Operatoren auf Pointern rechnen **implizit in Schritten** von `sizeof(<Typ>)`

Dynamische Arrays

- ▶ Für dynamische Arrays muss Speicher **zur Laufzeit reserviert** werden: `void *malloc(int size)`
 - ▶ Reserviert `size` Bytes Speicher auf dem Heap
 - ▶ Gibt Pointer auf reservierten Speicherbereich zurück
- ▶ Nicht mehr benötigter Speicher muss wieder **explizit freigegeben** werden: `void free(void *ptr)`
 - ▶ Bereich darf anschließend **nicht mehr verwendet** werden

```
1 int i, *a, a_len = 5;
2 /* Allocate memory for a */
3 if (! (a = malloc(a_len * sizeof(int))) ) {
4     printf("Out of memory\n");
5     exit(1);
6 }
7 for (i = 0; i < a_len; i++)
8     a[i] = i * i; /* Use array */
9
10 free(a); /* Free unused memory */
```

Themen

Warum heute noch C?

Grundlegende Syntax

Vom Quellcode zum Programm

Der Präprozessor

Arrays und Pointer

Komplexe Datentypen

Funktionsparameter

Bibliotheksfunktionen

Programmierumgebungen

Literatur

Structs – Urahnen der Klassen

- ▶ Structs **gruppieren** Daten zu einer **Einheit**
- ▶ Zugriff auf einzelne Elemente mittels Qualifizierer „.“
- ▶ Deklaration mittels Schlüsselwort `struct`
 - ▶ Entweder direkte Deklaration und einzelne Verwendung (wie statische Klasse in Java)
 - ▶ Oder Definition als Typ und mehrfache Verwendung (wie Objekte einer Klasse in Java)

Structs – Urahnen der Klassen

- ▶ Structs **gruppieren** Daten zu einer **Einheit**
- ▶ Zugriff auf einzelne Elemente mittels Qualifizierer „.“
- ▶ Deklaration mittels Schlüsselwort **struct**
 - ▶ Entweder direkte Deklaration und einzelne Verwendung (wie statische Klasse in Java)
 - ▶ Oder Definition als Typ und mehrfache Verwendung (wie Objekte einer Klasse in Java)

```
1 /* declare struct-vars p1, p2 */
2 struct {
3     int x, y;
4 } p1, p2;
5 /* use structs directly */
6 p1.x = p2.y = 5;
7 p1.y = p2.x = 12;
```

```
1 /* define struct-type "point" */
2 typedef struct {
3     int x, y;
4 } point;
5 /* declare and use two points */
6 point p1, p2;
7 p1.x = p2.y = 5;
8 p1.y = p2.x = 12;
```

Pointer auf Structs

- ▶ Pointer auf Structs funktionieren wie auf elementaren Typen
- ▶ Qualifizierer „->“ dereferenziert Struct-Pointer und greift auf Elemente zu: `p->x` äquivalent zu `(*p).x`

Pointer auf Structs

- ▶ Pointer auf Structs funktionieren wie auf elementaren Typen
- ▶ Qualifizierer „->“ dereferenziert Struct-Pointer und greift auf Elemente zu: `p->x` äquivalent zu `(*p).x`
- ▶ **Problem:**
 - ▶ Verwendung von Typen erst **nach** ihrer Definition möglich
 - ▶ Structs benötigen oft Pointer auf **eigenen Typ** (z. B. in Listen)
- ▶ **Lösung:** Verwendung eines Namens für das Struct

```
1  /* define list element */
2  typedef struct _elem {
3      int x;
4      struct _elem *next;
5  } elem;
6
7  /* initialize list */
8  elem *head = malloc(sizeof(elem));
9  head->next = 0;
10 head->x = 23;
```

Pointer auf Structs

- ▶ Pointer auf Structs funktionieren wie auf elementaren Typen
- ▶ Qualifizierer „->“ dereferenziert Struct-Pointer und greift auf Elemente zu: `p->x` äquivalent zu `(*p).x`
- ▶ **Problem:**
 - ▶ Verwendung von Typen erst **nach** ihrer Definition möglich
 - ▶ Structs benötigen oft Pointer auf **eigenen Typ** (z. B. in Listen)
- ▶ **Lösung:** Verwendung eines Namens für das Struct

```
1  /* define list element */
2  typedef struct _elem {
3      int x;
4      struct _elem *next;
5  } elem;
6
7  /* initialize list */
8  elem *head = malloc(sizeof(elem));
9  head->next = 0;
10 head->x = 23;
```

← Bezeichne Struct als `_elem`

← Pointer auf Struct `_elem`

Themen

Warum heute noch C?

Grundlegende Syntax

Vom Quellcode zum Programm

Der Präprozessor

Arrays und Pointer

Komplexe Datentypen

Funktionsparameter

Bibliotheksfunktionen

Programmierumgebungen

Literatur

Funktionsparameter

- ▶ Übergabe von Parametern erfolgt als Kopie (*call-by-value*)
- ▶ Übergabe von Pointern ermöglicht Veränderung der Parameter (*call-by-reference*)
- ▶ Übergabe von Array-Parametern erfolgt ebenfalls als Pointer

Beispiel 1: Referenzübergabe

```
1 void lower_limit_zero(int *x) {
2     *x = (*x > 0) ? *x : 0;
3 }
4
5 int i = -5;
6 lower_limit_zero(&i);
7 /* i is now 0 */
```

Beispiel 2: Array-Parameter

```
1 int asum(int *array, int len) {
2     int i, ret = 0;
3     for (i = 0; i < len; i++)
4         ret += array[i];
5     return ret;
6 }
7
8 int a[] = {23, 18, -9, 34, 3};
9 int sum = asum(a, 5);
```

Rückgabe von dynamischen Arrays (1)

- ▶ Gesucht: Funktion `strcat`, die zwei Strings konkateniert und zurück gibt
- ▶ Erster Versuch:

```
1  /* concatenate two strings, limited to total length of 1024 */
2  char *strcat(char *a, char *b) {
3      char ret[1024];
4      int i = 0, j = 0;
5      while (a[i] && j < 1023)
6          ret[j++] = a[i++];
7      i = 0;
8      while (b[i] && j < 1023)
9          ret[j++] = b[i++];
10     ret[j] = 0;
11     return ret;
12 }
```

Rückgabe von dynamischen Arrays (1)

- ▶ Gesucht: Funktion `strcat`, die zwei Strings konkateniert und zurück gibt
- ▶ Erster Versuch:

```
1  /* concatenate two strings, limited to total length of 1024 */
2  char *strcat(char *a, char *b) {
3      char ret[1024];
4      int i = 0, j = 0;
5      while (a[i] && j < 1023)
6          ret[j++] = a[i++];
7      i = 0;
8      while (b[i] && j < 1023)
9          ret[j++] = b[i++];
10     ret[j] = 0;
11     return ret;
12 }
```

- ▶ **Problem:** Nach **Verlassen** der Funktion `strcat` wird `ret []` **freigegeben**, Pointer zeigt ins „Nirvana“

Rückgabe von dynamischen Arrays (2)

- ▶ Zweiter Versuch: reserviere Heap-Speicher

```
1  /* concatenate two strings */
2  char *strcat(char *a, char *b) {
3      int i = 0, j = 0;
4      char *ret = malloc(strlen(a) + strlen(b) + 1);
5      while (a[i])
6          ret[j++] = a[i++];
7      i = 0;
8      while (b[i])
9          ret[j++] = b[i++];
10     ret[j] = 0;
11     return ret;
12 }
```

Rückgabe von dynamischen Arrays (2)

- ▶ Zweiter Versuch: reserviere Heap-Speicher

```
1  /* concatenate two strings */
2  char *strcat(char *a, char *b) {
3      int i = 0, j = 0;
4      char *ret = malloc(strlen(a) + strlen(b) + 1);
5      while (a[i])
6          ret[j++] = a[i++];
7      i = 0;
8      while (b[i])
9          ret[j++] = b[i++];
10     ret[j] = 0;
11     return ret;
12 }
```

- ▶ Problem gelöst?

Rückgabe von dynamischen Arrays (2)

- ▶ Zweiter Versuch: reserviere Heap-Speicher

```
1  /* concatenate two strings */
2  char *strcat(char *a, char *b) {
3      int i = 0, j = 0;
4      char *ret = malloc(strlen(a) + strlen(b) + 1);
5      while (a[i])
6          ret[j++] = a[i++];
7      i = 0;
8      while (b[i])
9          ret[j++] = b[i++];
10     ret[j] = 0;
11     return ret;
12 }
```

- ▶ Problem gelöst? Ja, aber...
 - ▶ Neuer Speicher wird in strcat **reserviert**, aber **nicht freigeben**
 - ▶ Aufrufer ist für Freigabe verantwortlich ⇒ **schlechter Stil**

Rückgabe von dynamischen Arrays (3)

- ▶ Dritter Versuch: Aufrufer muss Speicher reservieren

```
1  /* concatenate two strings, dst must provide enough space */
2  void strcat(char *dst, char *a, char *b) {
3      int i = 0, j = 0;
4      while (a[i])
5          dst[j++] = a[i++];
6      i = 0;
7      while (b[i])
8          dst[j++] = b[i++];
9      dst[j] = 0;
10 }
```

Rückgabe von dynamischen Arrays (3)

- ▶ Dritter Versuch: Aufrufer muss Speicher reservieren

```
1  /* concatenate two strings, dst must provide enough space */
2  void strcat(char *dst, char *a, char *b) {
3      int i = 0, j = 0;
4      while (a[i])
5          dst[j++] = a[i++];
6      i = 0;
7      while (b[i])
8          dst[j++] = b[i++];
9      dst[j] = 0;
10 }
```

- ▶ Oder eleganter (aber schwieriger verständlich):

```
1  void strcat(char *dst, char *a, char *b) {
2      while ( (*dst++ = *a++) );
3      dst--;
4      while ( (*dst++ = *b++) );
5  }
```

Rückgabe von dynamischen Arrays (3)

- ▶ Dritter Versuch: Aufrufer muss Speicher reservieren

```
1  /* concatenate two strings, dst must provide enough space */
2  void strcat(char *dst, char *a, char *b) {
3      int i = 0, j = 0;
4      while (a[i])
5          dst[j++] = a[i++];
6      i = 0;
7      while (b[i])
8          dst[j++] = b[i++];
9      dst[j] = 0;
10 }
```

- ▶ Oder eleganter (aber schwieriger verständlich):

```
1  void strcat(char *dst, char *a, char *b) {
2      while ( (*dst++ = *a++) );
3      dst--;
4      while ( (*dst++ = *b++) );
5  }
```

- ▶ **Achtung:** Gefahr des **Buffer-Overflows**

Parameter der main-Methode

Wiederholung:

- ▶ main-Methode: `int main(int argc, char **argv);`
- ▶ Parameter `argc` liefert Anzahl der Kommandozeilen-Parameter
- ▶ Parameter `argv` ist ein String-Array mit den Parametern
 - ▶ `argv[0]` ist Dateiname des Programms selbst

Beispiel: `foo.c`

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4     int i;
5     for (i = 0; i < argc; i++)
6         printf("%d: %s\n", i, argv[i]);
7     return 0;
8 }
```

Aufruf:

```
gcc -Wall -o foo foo.c
./foo hello world
```

Ausgabe:

```
0: ./foo
1: hello
2: world
```

Themen

Warum heute noch C?

Grundlegende Syntax

Vom Quellcode zum Programm

Der Präprozessor

Arrays und Pointer

Komplexe Datentypen

Funktionsparameter

Bibliotheksfunktionen

Programmierumgebungen

Literatur

Wichtige Bibliotheksfunktionen

- ▶ Dokumentation über C-Bibliotheken füllt ganze Bücher
- ▶ Hier nur Nennung der wichtigsten Header und Funktionen, die Sie für die Praktika benötigen werden
 - ▶ Kein Anspruch auf Vollständigkeit
- ▶ Unter **Linux** (oder in **Cygwin** unter Windows):
Bibliotheksreferenzen in **Manpages**
 - ▶ Für einzelne Funktion z. B. `man 3 printf`
 - ▶ Für Übersicht der Bibliothek z. B. `man stdio.h`
- ▶ *Hinweis*: Manpages lassen sich auch leicht mittels Internet-Suchmaschinen finden

Ein-/Ausgabe: `stdio.h`

Arbeiten mit Zeichenketten

`printf` Formatierte Ausgabe von Text

`sprintf` Formatierten Text in Zeichenkette speichern

`sscanf` Zeichenkette anhand von Formaten zerlegen

Arbeiten mit Dateien

`fopen` Öffnen von Dateien zum Lesen und Schreiben

`fclose` Schließen von offenen Dateien

`feof` Auf Dateiende prüfen

`fread` Aus Dateien lesen

`fgets` Zeilenweise aus Dateien lesen

`fprintf` Formatierten Text in Dateien schreiben

`fscanf` Formatierte Daten aus Datei lesen

Stringverarbeitung: `string.h`

Operationen auf Zeichenketten

- `strlen` Länge einer Zeichenkette ermitteln
- `strncpy` Kopieren von Zeichenketten
- `strncat` Anhängen von Zeichenketten
- `strstr` Teilstring in Zeichenkette finden
- `bzero` Zeichenkette mit Nullen füllen

Grundlegende Funktionen: `stdlib.h`

Speicherverwaltung

`malloc` Speicher reservieren

`calloc` Speicher für Arrays reservieren

`free` Speicher freigeben

Hilfsfunktionen

`abs` Absolutwert einer Ganzzahl

`atoi` Ganzzahl aus String parsen

`atof` Gleitkommazahl aus String parsen

Sonstiges

`rand` Zufallszahl ermitteln

`system` Befehl des Betriebssystems ausführen

`exit` Programm verlassen

Operationen auf Gleitkommazahlen

- ▶ Trigonometrie
- ▶ Logarithmen, Potenzen
- ▶ Auf-/Abrunden

Themen

Warum heute noch C?

Grundlegende Syntax

Vom Quellcode zum Programm

Der Präprozessor

Arrays und Pointer

Komplexe Datentypen

Funktionsparameter

Bibliotheksfunktionen

Programmierumgebungen

Literatur

IDE (Integrated development environment)

- ▶ Anwendungsprogramm zur Entwicklung von Software
- ▶ IDE hat folgende Komponenten
 - ▶ Texteditor
 - ▶ Compiler bzw. Interpreter
 - ▶ Linker
 - ▶ Debugger
 - ▶ Quelltextformatierungsfunktion
- ▶ Empfohlene IDEs
 - ▶ Visual Studio (Express): Windows
 - ▶ Eclipse CDT: Windows, Mac OS X, Linux
 - ▶ QtCreator: Windows, Mac OS X, Linux
- ▶ Liste C/C++ Compiler/IDE
(<http://www.thefreecountry.com/compilers/cpp.shtml>)

Themen

Warum heute noch C?

Grundlegende Syntax

Vom Quellcode zum Programm

Der Präprozessor

Arrays und Pointer

Komplexe Datentypen

Funktionsparameter

Bibliotheksfunktionen

Programmierumgebungen

Literatur

Literaturempfehlungen

- ▶ Helmut Herold: *C-Kompaktreferenz*, Addison-Wesley
- ▶ B. W. Kernighan, D. M. Ritchie: *The C Programming Language*, Prentice Hall
 - ▶ Detailliertes Expertenwissen der Erfinder von C
- ▶ Jürgen Wolf: *C von A bis Z*, Galileo Press
 - ▶ Als „Open Book“ kostenlos elektronisch verfügbar
 - ▶ http://www.galileo-press.de/openbook/c_von_a_bis_z/