

Einführung in C / C++

Thomas Kiesel

Eingebettete Systeme und Betriebssysteme (EOS)
Otto-von-Guericke-Universität Magdeburg

Wintersemester 2009/2010

Motivation und Ziele

Ziele der Übung

- Vertiefung der vorgestellten Konzepte
- *praktische Anwendung* aus Sicht des (System-)Programmierers

Motivation und Ziele

Ziele der Übung

- Vertiefung der vorgestellten Konzepte
- *praktische Anwendung* aus Sicht des (System-)Programmierers

Verwendung von ...

- Betriebssystem UNIX
- Programmiersprache C / C++

Warum C?

- Eine der wichtigsten Sprachen obwohl relativ alt (*erste Version ab 1971*)
- Fast alle gängigen Betriebssysteme in C/C++ geschrieben
- Sehr maschinennahe Sprache, deshalb gut für Systemprogrammierung geeignet
- C für die Betriebssystemprogrammierung erdacht (*UNIX*)

Warum C?

- Eine der wichtigsten Sprachen obwohl relativ alt (*erste Version ab 1971*)
- Fast alle gängigen Betriebssysteme in C/C++ geschrieben
- Sehr maschinennahe Sprache, deshalb gut für Systemprogrammierung geeignet
- C für die Betriebssystemprogrammierung erdacht (*UNIX*)

Fazit:

Wer sich mit Systemprogrammierung beschäftigt,
kommt um C/C++ nicht herum!

Beispiel

Beispiel OO-StuBS

```
#include "user/appl.h"
#include "device/cgastr.h"

CGA_Stream kout;

int main() {

    Application application;

    int x, y;

    // Bildschirm loeschen
    for (y=0; y<25; y++)
        for (x=0; x<80; x++)
            kout.show (x, y, ' ', CGA_Screen::STD_ATTR);

    // Startmeldung ausgeben
    kout.setpos (36,10);
    kout << "OO-Stubs" << endl << endl;

    // Anwendungscode aufrufen
    application.action();

    return 0;
}
```

Inhalte

- Datentypen
- Zeiger (*Pointer*)
- Funktionen
- Klassen
- Vererbung
- Toolchain

Datentypen und Deklaration

- Eindeutiger Typ der mögliche Anweisungen/Befehle bestimmt
- Compiler/Laufzeitsystem überwacht korrekte Typisierung
- Kombination verschiedenen Datentypen benötigt Konvertierung
 - *implizite/explizite casts*

Datentypen und Deklaration

- Eindeutiger Typ der mögliche Anweisungen/Befehle bestimmt
- Compiler/Laufzeitsystem überwacht korrekte Typisierung
- Kombination verschiedenen Datentypen benötigt Konvertierung
 - *implizite/explicite casts*
- Syntax: [Spezifizierer] Basistyp Deklarator [Initialisierer]
 - Spezifizierer** definierte Speicherart (const, extern, static, volatile)
 - Deklarator** Name [Operator] (Zeiger *, Referenz &, Feld [], Funktion ())
 - Initialisierer** Vorgabewert für die Variable (*implizite Typkonvertierung*)

Beispiel – Deklaration

```
char string[20] = {'H', 'a', 'l', 'l', 'o', '\\0'};
```

Datentypen in C

C kennt folgende Datentypen

```
char c;           /* 8 Bit */
short s;         /* >= 16 Bit */
int i;           /* >= 16 Bit */
long l;          /* >= 32 Bit */
float f;         /* i.d.R. 4 Byte */
double d;        /* i.d.R. 4-8 Byte */
long double ld; /* i.d.R. 4-8 Byte */
```

- Unsigned Variante für jedem ganzzahligen Typ
- Größe der Datentypen ist systemabhängig, wobei gilt:
 - `int` ist die *natürliche* Größe der Plattform
 - `short` \leq `int` \leq `long`
- Tatsächliche Größe für **alle** Datentypen mit dem Operator `sizeof` (`Datentyp|Variable`)

bool

- Kein expliziter Datentyp `bool` in ANSI-C
- Konvention (*TRUE* \neq 0 und *FALSE* = 0)

Beispiel – bool

```
int x=0;
int y=1;

if (x-y)
    printf("a");
else
    printf("b");
```

enum

- Verwendung sprechender Namen
- Aufzählung ist eine Menge von Integer-Konstanten¹
- Verwendung kann equivalent zu `int` erfolgen

Beispiel – enum

```
enum ASCII
{
    ASCII_NULL,      // 0
    ASCII_SOH,       // 1
    ASCII_STX,       // 2
    ASCII_A = 65,    // 65
    ASCII_B,         // 66
    ...
};

for (ASCII zeichen = ASCII_A; zeichen <= ASCII_Z; zeichen++)
{
    \\ Durchlaufen aller Zeichen von A-Z
}
```

¹Enumeratoren

struct

- Entspricht Java Klassen ohne Methoden und Zugriffsrechten
- Zusammenfassung von Elementen verschiedener Typen
 - Elemente einer Struktur müssen im Speicher nicht direkt hintereinander liegen
 - Initialisierung wie bei Arrays möglich

Beispiel – struct

```
struct person {  
    int alter;  
    char name[25];  
} element = {20, "Hans_Mueller"};  
  
element.alter = 30;           /* Zugriff mit Instanze von struct */  
element->name = "Hans_Werner"; /* Zugriff mit Pointer auf struct */
```

union

- Eine `union` vereint mehrere Datentypen in einem Typ
- Syntaktisch zu Strukturen aber eine Union kann zu jedem Zeitpunkt nur eines der Felder beinhalten
 - Felder einer Struktur geben unterschiedliche Speicherbereiche innerhalb eines größeren Typs an
 - Felder einer Union geben unterschiedliche Datentypen für denselben Speicherbereich an!
- Größe einer Union entspricht der des größten enthaltenen Datentyps
- Kann auch in Arrays und Strukturen verwendet werden

typedef

- Verwendung: typedef <Definition> <Typenname>
 - Umbenennen von Datentype
 - Definition eigener Datentypen
 - **Vorsicht** bei rekursiven Strukturen

Beispiel – typedef

```
typedef unsigned char byte;
typedef struct {
    int alter;
    char name[25];
} person;

typedef struct _plist { /* Fehlerhafte Zuweisung aufgrund von Verschachtelung */
    person element;
    struct _plist *next;
} plist; /* plist erst hier definiert */
```

Casts

- Automatische Typumwandlung
- Manuelle Typumwandlung
 - C-Cast-Operatoren: (typ)ausdruck
`float f = (int)3.6*(int)4.76; // ergibt 12`
 - C++-Cast-Operatoren
 - 1 Typüberprüfung bei der Übersetzung (`static_cast`)
 - 2 Typüberprüfung zur Laufzeit² (`dynamic_cast`)
 - 3 Typumwandlung ohne Überprüfung (`reinterpret_cast`)

²RTTI = RunTime Type Identification

Achtung:

Eine der wichtigsten Fehlerquellen in C/C++ sind Zeiger!

Achtung:

Eine der wichtigsten Fehlerquellen in C/C++ sind Zeiger!

- Unkontrollierter Zugriff auf beliebige Speicherstellen
 - Programmabbruch durch das Betriebssystem (*gut!*)
 - Überschreiben prozesslokaler Daten (*schlecht!*)
- Fehler durch falsche Zeiger sind oft *sehr schwer* zu finden

Zeiger alg.

- Enthält Adresse einer Variable im Speicher
 - `&` liefert Adresse einer Variablen (Adressoperator)
 - `*` liefert Variableninhalt zu einer Adresse³
 - `NULL` spezielle Adresse (*Zeiger ins Nichts*)
- Addition/Subtraktion auf Zeigern ändert Zeiger um Größe des referenzierten Datentyps

Zeigerverwendung

```
int a = 0;
int *b = &a;
*b=1; /* a==1 */
b=0;
*b=2; /* Fehler */
int (*func)(int*);
```

³Dereferenzierungsoperator

Beispiele



Referenzen

- sind alternative Namen für ein Objekt
- nach Initialisierung immer gültig
- Referenzen werden nicht explizit dereferenziert
- **Hauptanwendung:** Rückgabewerte von Funktionen

Beispiel

```
void swap(int &a, int &b) { int c = a; a = b; b = c; }
```

Kontrollstrukturen und Kontrollfluß

Weitgehend so wie in Java, aber ...

- Kommentare werden laut ANSI-C in `/* Kommentar */` eingeschlossen
- Statt boolean werden beliebige Werte akzeptiert
`int i = 42; while(i) {i--;}`

Vorsicht:

- Zuweisungen haben zugewiesenen Wert als Rückgabewert
z.B. `a = b = 6*7;`
- Zuweisungen in if-Anweisungen sind legal:
`if (a=b) { ... } anstatt if (a==b) { ... }`

Funktionen in C

- Definition syntaktisch wie Methodendefinition in Java
 - **Syntax:**[Returntyp] Name ([Parameter(Basistyp Name)])
- Parameterübergabe **immer** *call-by-value*
- Rückgabe **immer** *by-value*
- Es können nur bereits deklarierte Funktionen verwendet werden!

Funktionsdeklaration ohne Definition

```
char foo(int, double, float);
```

Funktion main

- Signatur: `int main(int argc, char **argv);`
- `argv` Array der Länge `argc` mit Zeigern auf `char*`
- `argv[0]` Name des aufgerufenen Programms
- `argv[i]` `i`-ter Parameter als NULL-terminierter String

Beispiel Program echo

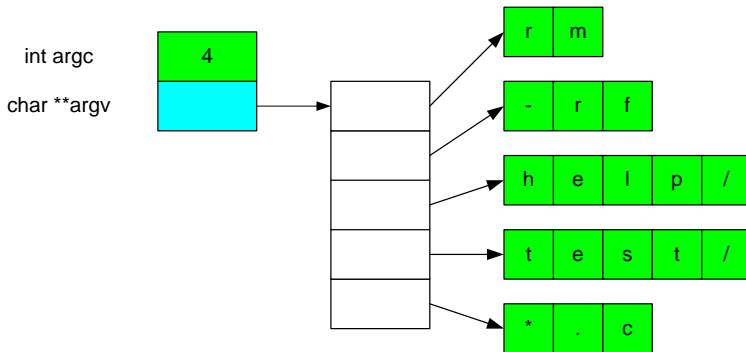
```
int main(int argc, char **argv)
{
    int i = 0;

    for (i=1; i<argc; i++)
        printf("%s ", argv[i]); printf("\n");
    return 0;
}
```


main Beispiel

Beispiel Program echo

```
rm -rf help/ test/ *.c
```



Gültigkeitsbereich von Variablen

- Variablen außerhalb von Funktionen sind global gültig
- Variablen gelten im Block, in dem sie definiert wurden sowie allen darin enthaltenen Blöcken
- Variablen aus äußeren Blöcken können in inneren Blöcken überlagert (*neu deklariert*) werden
- `static`
 - Global `static` Variablen nur innerhalb einer Datei gültig
 - Lokale `static` Variablen behalten Wert über Funktionsaufruf hinweg
- `extern` (z.B. `extern int x;`)
 - Keine Reservierung von Speicher
 - Linker-Fehler wenn Variable nach Compilierung nicht vorhanden

Klassen vs. Objekte

Klasse Datenstruktur mit Daten (*Felder*) und darauf arbeiten Funktionen (*Methoden*)

Objekt Exemplar (*Instanz*) einer Klasse

- Zugriff nur über Interfaces
- Kapselung von Daten und Funktionalitäten
- Umsetzung von Vererbung, Polymorphie, Operandenüberladung, Zugriffskontrolle

Wichtig:

Unterscheidung zwischen Deklaration und Definition einer Klasse.

Klassendeklaration

Beispiel – Klassendeklaration

```
class Staff : public Person {  
public:           // oeffentlich  
    Staff();  
    ~Staff();  
    void setID(const int id);  
    int  getID() const;  
private:       // klassenintern  
    int id_;  
};
```

- Klassendeklaration ist eine Typdeklaration (*Wichtig Semikolon am Ende*)
- class ist Erweiterung von struct in C
- Zugriffskontrolle (public, protected, private)

Methodendefinition

- Methoden sollten in der Klassendeklaration nur deklariert, aber erst anschließend definiert werden
- **Wichtig:** Ausserhalb der Klassendeklaration muss Bezug (*scope*) verwendet werden

Wichtig:

```
void Staff::setID(const int id) { id_ = id; }
```

Zugriff auf Felder und Methoden

- Objekte verwenden den Operator `.`
- Objektzeiger verwenden den Operator `->`
- Aktuelles Objekt spricht Felder/Methoden direkt an
- Alternativer Zugriff mit `this`-Pointer

Methodenzugriff

```
Klasse A;  
Klasse *B;  
  
A.methode(); /* Methodenzugriff ueber Klasseninstance */  
B->methode(); /* Methodenzugriff ueber Klassenpointer*/  
(*B).methode(); /* Methodenzugriff ueber Pointer mit Dereferenzierung*/
```

Konstruktor

- Konstruktor für Speicherzuweisung und Feld-Initialisierung
- Gleicher Name wie Klasse und kein Rückgabewert
- Mehrere Konstruktoren möglich (*verschiedene Signaturen*)
- Feld-Initialisierung über Initialisierungsliste

Beispiel – Klassendefinition

```
Person::Person(char *name, int year) : name_(name), year_(year)
{ ... }
```

Destruktor

- Aufruf beim Löschen/Entfernen eines Objektes
- Destruktor muss nicht explizitt definiert werden, aber
 - Default-Konstruktor löscht Felder nur flach
 - Referenzen werden gelöscht, nicht aber deren Inhalte
- Virtuelle Klassen benötigen virtuellen Destruktor

Überladen von Operatoren

- Operatoren sind in C++ Methodenaufrufe
 - Überladen = mit neuer Funktionalität versehen
 - Nur bestehende Operatoren überladbar
 - Priorität und Assoziativität von Operanden nicht veränderbar

Ausgabe von Objekten^a

^aManipulatoren

```
ostream& operator<<(ostream os, const Person &rhs) {
    return rhs.print(os);
}

ostream& Person::print(ostream &os) const {
    os << name_ << ", " << year_;
    return os;
}
```

Vererbung

- Drückt eine IST EINE-Beziehung aus
- Erweiterung od. Spezialisierung einer vorhandenen Klasse
- Interface-Definition über `virtual class` möglich

MehrfachVererbung

Java Jede Klasse erbt von genau einer Klasse

- Klassenstruktur ist Baum mit Object an der Wurzel

C++ Klasse kann von beliebig vielen Klassen erben

- Klassenstruktur ist zyklenfreier Graph
- Konflikt von gleichnamigen Feldern od. Methoden möglich

Virtuelle Methoden

- Drückt aus, dass eine Methode in abgeleiteten Klassen
 - überschrieben werden kann ...
 - oder überhaupt erst implementiert werden muss (*pure virtual*)
- Polymorphie, late binding

C++ Eine Methode, die in einer abgeleiteten Klasse überladen werden kann, muss ausdrücklich als `virtual` deklariert werden

Java Alle Methoden sind virtuell; *pure virtual* Methoden heißen `abstract`

Codeaufteilung – Interface

- Klassendeklaration (*Interface*) immer in Header-Datei
- Header-Datei wird von Klassendefinition und Anwender eingebunden

Beispiel – Klassendeklaration

```
#ifndef __Person_h__
#define __Person_h__
class Person {
public:    // oeffentlich
    Person(char *name="", int year=0);
    void setName(char *name);
private: // klassenintern
    char name_[25];
};
#endif
```

- **Faustregel:** Keine Reservierung von Speicherplatz

Codeaufteilung – Implementierung

- Klassendefinition benötigt Header-Datei

Beispiel – Klassendefinition

```
#include "Person.h"

void Person::setName(char *name) {
    name_ = name;
}
```

- Anwendungsprogramm benötigt Header-Datei

- Ausführung in mehreren Schritten
 - 1 Präprozessor
 - 2 Compiler
 - 3 Linker: Erzeugung eines lauffähigen Programms aus Objekt- und Bibliotheksdateien
- Abarbeitung aller Schritte auf einmal und ohne Benutzereingriff

Präprozessor

- Textuelle Ersetzung von Präprozessoranweisungen
- Vorverarbeitung des Quelltextes
- Textersetzung definierter Präprozessoranweisungen
 - Bedingte Übersetzung (z.B. *systemspezifische Teile*)
 - Auflösen von Makros (*für C++ kaum von Bedeutung*)

Präprozessor Direktiven

- `#include <Datei>` Einbinden von *Header-Dateien*
- `#define ...` Definition von Konstanten und Makros
- `#if / #ifdef / #ifndef` Bedingte Übersetzung
- `#else #endif` Unerfüllte Bedingung

GNU C Compiler (gcc)

- Übersetzung in Maschinencode (*Objekt-Dateien*)
- Aufruf: `gcc <datei.c>`
 - `-Wall` Ausgabe von Warnungen
 - `-o <datei>` Name der zu erzeugenden Datei
- Aufruf von `gcc` bewirkt Übersetzen und Linken
- Weitere Optionen
 - `-c` Kein Linken
 - `-E` Ausführung des Präprozessors
 - `-g` Einbinden von Debugg-Informationen
 - `-O`, `-O1`, `-O2`, `-O3` Optimierungsstufen

Make

- Automatische Erzeugung von Programmen aus Quelltexten
- Berücksichtigt Abhängigkeiten zwischen Dateien
- Steuerung der Übersetzung durch Regeln

Makefile Regeln

```
GENSYS          = Linux
ASMOBJFORMAT   = elf

ifeq ($(GENSYS), Linux)
LDHEAD = $(shell g++ --print-file-name=crti.o)
else
...
endif

(OBJDIR)/%.o : %.c
    $(CC) -c $(CFLAGS) -o $@ $<

clean :
    @rm -f $(OBJDIR)/*.[oO] $(OBJDIR)/*.img
    @rm -rf ./build
```

GNU Debugger: gdb, ddd

- Aufruf: `gdb <programm>`
- Vorgehensweise für Fehler zwischendurch
 - `run <parameter>` Programm starten
 - `kill` Programm abbrechen
 - `break <funktion>` Programm gezielt unterbrechen
 - `print <ausdruck>` Programmvariablen ausgeben
 - `cont` Programm fortsetzen
 - `backtrace` Aufrufhierarchie verfolgen
- Graphisches Display für gdb ist der ddd (*Data Display Debugger*)

Literatur I



Kyle Loudan and Matthias Kalle Dalheimer.

C++ kurz und gut.

O'Reilly, 2003.



Scott Meyers.

Effective C++.

Addison-Wesley, 2007.



Scott Meyers.

More Effective C++.

Addison-Wesley, 2007.



Bjarne Stroustrup.

Die C++ Programmiersprache.

Addison-Wesley, 2004.



Jürgen Wolf.

C von A bis Z.

Galileo Press, 2 edition, 2006.