Einführung in die GNU-Toolchain

Thomas Kiebel

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

Wintersemester 2010/2011

Motivation und Ziele

Ziele der Übung

- Vertiefung der vorgestellten Konzepte
- *praktische Anwendung* aus Sicht des (System-)Programmierers
- Low–Level *(Treiber)* Programmierung

Motivation und Ziele

Ziele der Übung

- Vertiefung der vorgestellten Konzepte
- *praktische Anwendung* aus Sicht des (System-)Programmierers
- Low–Level *(Treiber)* Programmierung

Verwendung von ...

- Betriebssystem UNIX
- GNU–Toolchain
- Programmiersprache C / C++

GNU-Toolchain I

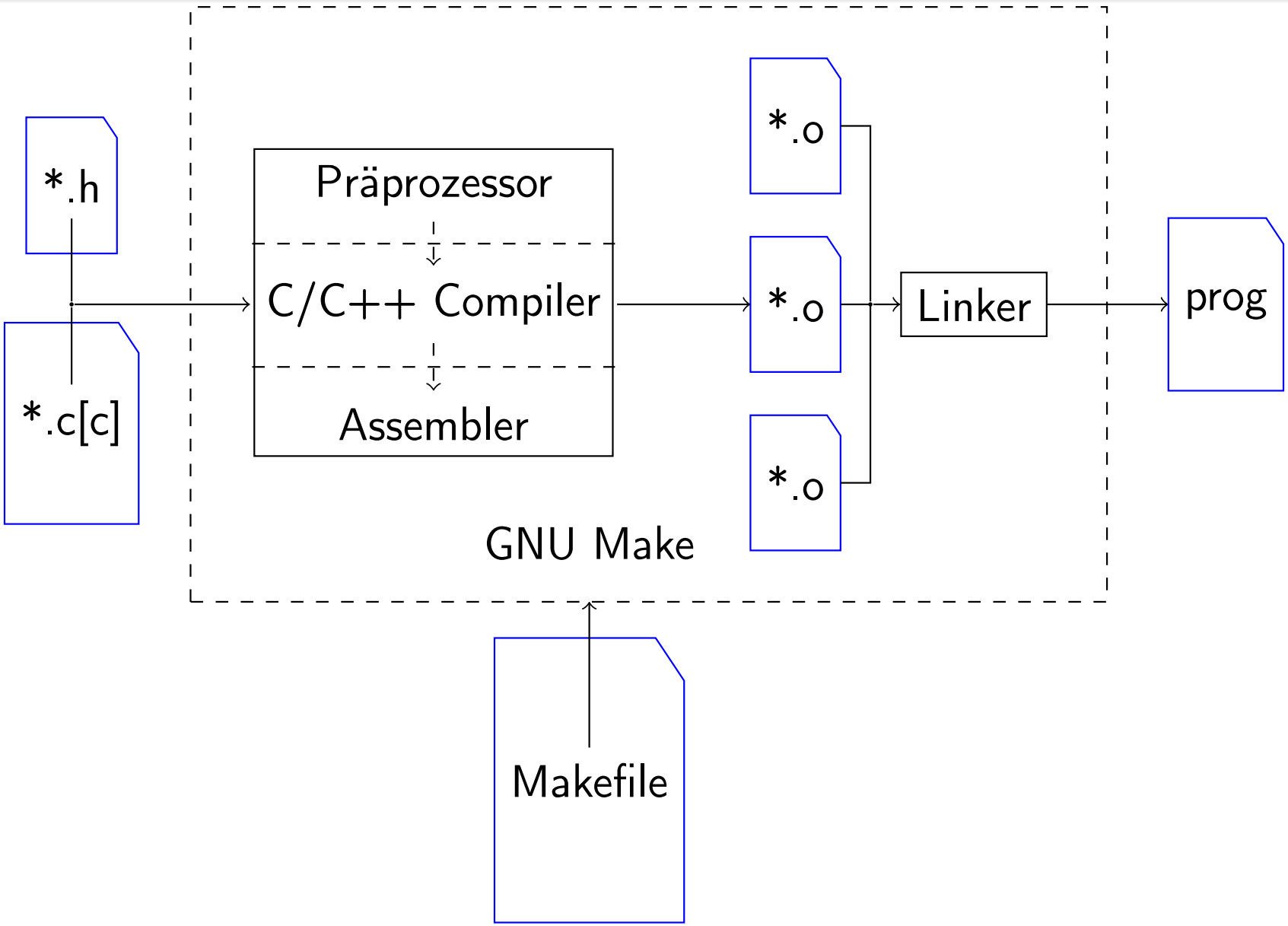
Ziele

- Erzeugung lauffähiger Programme/Bibliotheken aus Quellcode
- Architekturunabhänigkeit
- Abarbeitung aller Schritte autonom ohne Benutzereingriff

Tools

- GNU Make (make)
- gcc Präprozessor (gcc)
- gcc Compiler (gcc/g++)
- GNU Assembler (gas)
- GNU Linker (ld)

GNU-Toolchain II



GNU Make (make)

- allgemeiner Aufruf: `make <Ziel>`
- Steuerung durch Datei "Makefile"
- regelbasierte Syntax

Ablauf einer Regel

1. Auflösen der Abhänigkeiten
2. Überprüfung auf Aktualität
3. Ausführen der Befehle

Makefile Regeln

```
TARGET=example
SOURCES=example.cc assembler.S

${TARGET}: ${SOURCES}
	${CXX} ${CFLAGS} $^ -o $@
```

GNU Compiler Collection (gcc/g++)

- Übersetzung in ausführbaren Maschinencode (*Programme/Bibliotheken*)
- Aufruf für C: `gcc <datei.c>`
- Aufruf für C++: `g++ <datei.cc>`
  - `-Wall` Ausgabe von allen Warnungen
  - `-o <datei>` Name der zu erzeugenden Datei, sonst a.out
  - `-g` Einbinden von Debug-Informationen für GDB
  - `-O[0|1|2|3|s]` Optimierungsstufen
- Wenn nur Teilschritte ausgeführt werden sollen:
  - `-c` nur Object-Code erzeugen
  - `-S` nur Assembler-Code erzeugen
  - `-E` nur Präprozessor ausführen

Präprozessor (gcc/g++)

- Textuelle Ersetzung zur Vorverarbeitung
- Einsatzzwecke
  - Bedingte Übersetzung (*z.B. systemspezifische Teile*)
  - Auflösen von Makros (*für C++ kaum von Bedeutung*)
  - Definition von Konstanten (*unter C++ unschön*)

Häufige Präprozessor Direktiven

- `#include <Datei>` Einbinden von *Header*-Dateien
- `#define <Name> <Wert>` Definition von Konstanten und Makros
- Bedingte Übersetzung des folgenden Codeblocks:
  - `#if <Bedingung>` Test auf Bedingung
  - `#ifdef <Name>` Test ob Name bereits definiert
  - `#ifndef <Name>` Gegenteil zu `#ifdef <Name>`
  - `#else` Alternative
  - `#endif` Ende eines bedingt Codeblocks

GNU Assembler (gas)

- Aufruf über gcc: `gcc <Datei.S>`
- direkter Aufruf: `as <Datei.S>`

Besipiel

```
.data                           /* Begin der Datendefinition */
base:                           /* Label */
zero:   .long 0                 /* reservieren von vorbelegtem Speicher */
one:    .long 1

.text                           /* Begin des Codes */

.global return1                 /* Extern verfuegbare Symbole */

return1:                        /* Begin einer Funktion */
        mov $base, %ecx
        movl 4(%ecx), %eax
        ret                     /* Ruecksprung */
```

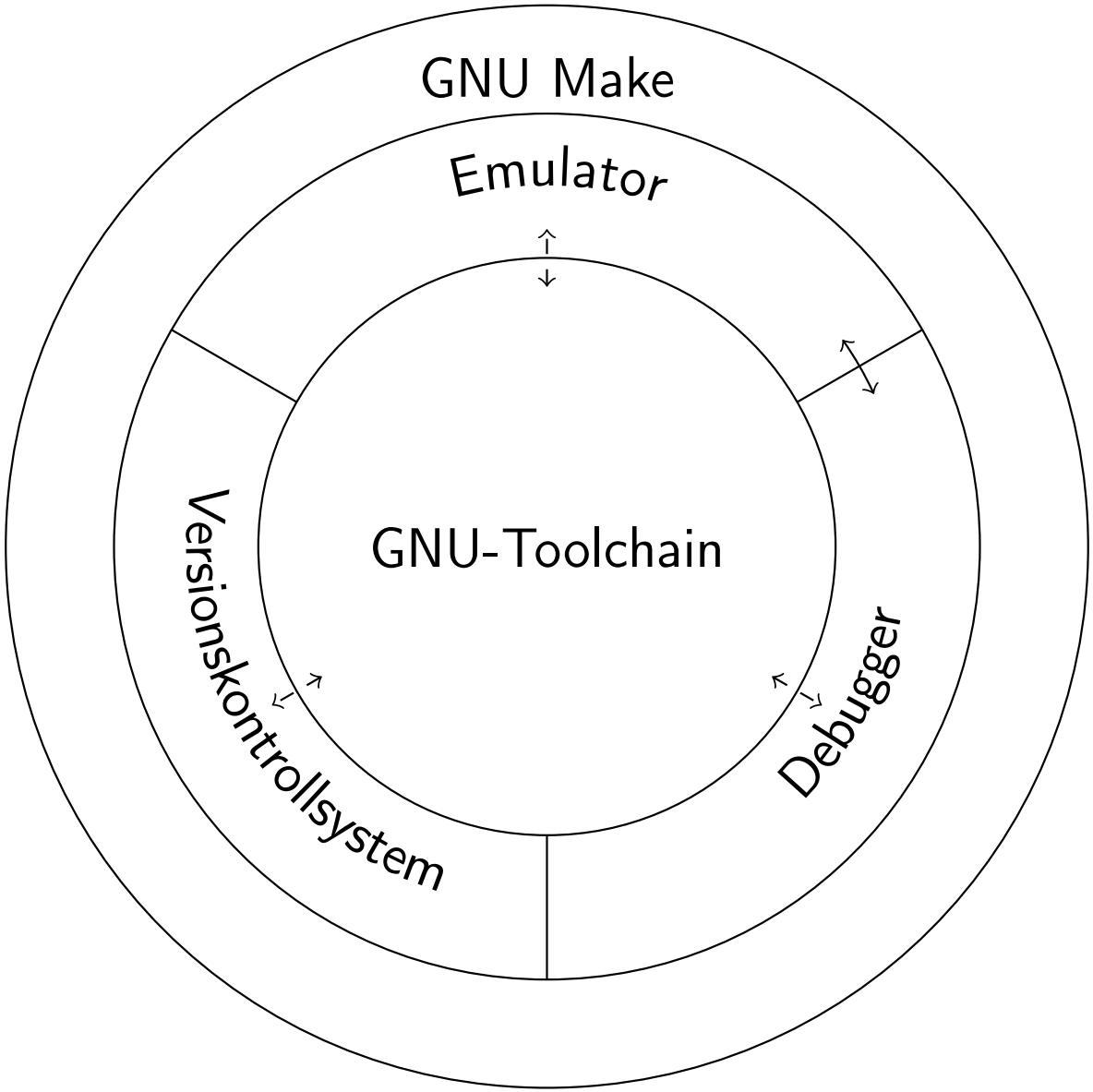
GNU Linker (ld)

Syntax

- Aufruf über gcc: `gcc < Datei1.o Datei2.o .. > -o <output>`
- direkter Aufruf: `ld < Datei1.o Datei2.o .. > -o <output>`

- Aufgabe: Auflösen von unbekannten Symbolen zwischen Objektdateien
- kann über Scriptdatei gesteuert werden, für Betriebssystem nötig
- Bei Aufruf über gcc zusätzliche interne Schritte zur Erzeugung von Programmen

Zusätzliche Entwicklertools



GNU Debugger: gdb

- Aufruf: `gdb <programm>`
- Mächtiges Werkzeug zur Analyse vom Programmverhalten
- Graphische Oberflächen für `gdb`:
  - `ddd` (*Data Display Debugger*)
  - `insight`

Wichtige Befehle

- `run <parameter>` Programm starten
- `kill` Programm abbrechen
- `break <funktion>` Programm gezielt unterbrechen
- `print <ausdruck>` Programmvariablen ausgeben
- `cont` Programm fortsetzen
- `backtrace` Aufrufhierachie verfolgen

Emulator QEMU

- Multi-Architektur Emulator
- Emulation von x86-CPU sowie Peripherie:
  - Chipsatz (Intel i440FX)
  - Grafikkarte (Standard VGA oder Cirrus CLGD 5446)
  - Eingabegeräte (PS/2 Tastatur und Maus)
  - Netzwerkkarten (e1000)
  - Soundkarten (AC97, ES1370 oder Creative Soundblaster 16)
  - verschiedene Ports (USB, Seriell, Parallel)
- Besondere Features:
  - direktes Laden von Betriebsystemkernen (Qemu $>$ 0.10)
  - Debugging mithilfe von GDB
  - Skriptfähig, da Konsolenapplikation

Programmiersprache C / C++

- C ist meist verwendeten Sprachen obwohl relativ alt (*erste Version ab 1971*)
- Fast alle gängigen Betriebssysteme in C/C++ geschrieben
- Sehr maschinennahe Sprache, deshalb gut für Systemprogrammierung geeigent
- C++ syntaktisch zu C kompatibel aber mächtiger

Programmiersprache C / C++

- C ist meist verwendeten Sprachen obwohl relativ alt (*erste Version ab 1971*)
- Fast alle gängigen Betriebssysteme in C/C++ geschrieben
- Sehr maschinennahe Sprache, deshalb gut für Systemprogrammierung geeigent
- C++ syntaktisch zu C kompatibel aber mächtiger

Fazit:

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

Typisierung und Deklaration

- Jede Variable hat eindeutigen Typ der mögliche Anweisungen/Befehle bestimmt
- Trennung von Deklaration und Definition
- Compiler überwacht korrekte Typisierung
- Bei inkompatiblen Datentypen Umwandlung nötig:
  - *implizite/explizite casts*

Typisierung und Deklaration

- Jede Variable hat eindeutigen Typ der mögliche Anweisungen/Befehle bestimmt
- Trennung von Deklaration und Definition
- Compiler überwacht korrekte Typisierung
- Bei inkompatiblen Datentypen Umwandlung nötig:
  - *implizite/explizite casts*
- Syntax: [Spezifizierer] Basistyp[Modifikator] Objektname[Modifikator];

  Spezifizierer Veränderung der internen Repräsentation (`const`, `extern`, `static`, `volatile`)

  Modifikator Veränderung des Typs (`Zeiger *`, `Referenz &`, `Array []`, `Funktion ()`)

Beispiel – Deklaration und Definition

```
string output;
output="Test1";
```

Basisdatentypen in C

C kennt folgende Datentypen, Größen für x86-Architektur

```
char        c;      /* 8   Bit, Achtung: standardmaessig signed */
short       s;      /* 16  Bit */
int         i;      /* 32  Bit */
long        l;      /* 32  Bit */
long long   ll;     /* 64  Bit */
float       f;      /* 32  Bit, Achtung: in OOStuBs nicht verwenden*/
double      d;      /* 64  Bit, Achtung: in OOStuBs nicht verwenden*/
long double ld;     /* 128 Bit, Achtung: in OOStuBs nicht verwenden*/
```

- Unsigned und Signed Variante für jeden ganzzahligen Typ
- Größe der Datentypen systemabhängig, wobei gilt:
  - `int` ist *natürliche* Größe der Plattform
  - `short` $\leq$ `int` $\leq$ `long`
- Tatsächliche Größe eines Datentyps mit Operator `sizeof (Datentyp|Variable)` in Bytes

Datentyp bool

- Kein explizieter Datentyp `bool` in ANSI-C
- Als Erweiterung in C definiert, in C++ standard
- implizite Konvertierung ($0 = \textit{false}$ und $0 \neq \textit{true}$)

Beispiel – bool

```
int t=1;
int f=0;

int main(){
    if (t)
        printf("true\n");
    if (!f)
        printf("false\n");
}
```

Enumerationen

- Deklaration von Aufzählungen
- Alternativ auch zur Konstantendefinition mit sprechenden Namen
- ist Menge von Integer-Konstanten
- Verwendung equivalent zu `int`
- Definierte Enumerationen sind jeweils eigener Typ

Beispiel

```
enum TextIndices{
        Object,
        Interface,
        Test1,
        Test2
};

const void* getText(enum TextIndices index);
```

Funktionen in C

- Definition syntaktisch wie Methodendefinition in Java
- Auch hier Trennung von Deklaration und Definition möglich
- Es können nur bereits deklarierte Funktionen verwendet werden!
- Trennung von Interface und Implementierung:
  - Definitionen in Header-Dateien, beschreibt Interface
  - Definition in C-Dateien, beschreibt Implementierung

Funktionsdeklaration

```
const void* getText(enum TextIndices index);
```

Funktionsdefinition

```
const void* getText(enum TextIndices index){
        return Text[index];
}
```

Gültigkeitsbereich und Allokation I

Definition

Ein **Codeblock** ist ein von { und } eingeschlossener Bereich des C/C++-Codes

Gültigkeitsbereiche

- lokal, innerhalb eines Codeblocks
- global, außerhalb von **allen** Codeblöcken

- Zugriff auf Variablen nur auf gleicher oder höherer Verschachtelungsebene
- Umdeklaration von Variablen in verschachtelten Blöcken ist möglich, jedoch schlechter Stil

Gültigkeitsbereich und Allokation II

Allokation

- lokale Variablen werden auf dem **Stack** alloziert
- globale Variablen liegen im **Datensegment** der ausführbaren Datei

Modifikation des Standardverhaltens

- `extern` deklarierte Variablen, werden nicht erzeugt, ihre Existenz wird vorausgesetzt
- `static` deklarierte **lokale** Variablen behalten ihren Wert auch bei Verlassen der Gültigkeit
- `static` deklarierte **globale** Variablen gelten nur in der Datei in der sie definiert wurden

Klassen vs. Objekte

Unterschied

- **Klasse** Datenstruktur mit Daten (*Felder*) und darauf arbeiten Funktionen (*Methoden*)
- **Objekt** (*Instanz*) einer Klasse

- Beschreibung der Klasse ist Deklaration
- Erzeugung des Objekts ist Definition
- Deklarationen gehören in Header-Files
- Definitionen gehören in CC-Files

Klassendeklaration

- Klassendeklaration ist eine C++ Typdeklaration (*Wichtig:* Semikolon am Ende)
- Nahezu identisch zu Java, jedoch `:` statt `extends` oder `implements`
- Zugriffskontrolle durch `public`, `protected`, `private` (Standard: `private`)

Beispiel – Klassendeklaration

```
class DataPrinter : public DataPrinterInterface{
        private:
                ostream& out;
        public:
                void printOne();
};
```

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

Zugriff

```
ostream& DataPrinter::operator<<(Data &d){
        out << d.s << endl;
}
void DataPrinter::printOne(){
        this->out << "one: " << return1() << endl;
}
```

Methodendefinition

- Methoden sollten wie Funktionen getrennt Definiert und Deklariert werden
- Deklaration innerhalb der Klasse, Definition in der Implementierung (CC-Datei)
- **Wichtig:** Ausserhalb der Klassendeklaration muss Bezug (*scope*) verwendet werden

Wichtig:

```
ostream& DataPrinter::operator<<(const void* address){
        out <<  hex << address << endl;
```

Konstruktoren und Destruktoren

Konstruktor

- Konstruktor für Speicherzuweisung und Member, sowie Basis-Initialisierung
- Gleicher Name wie Klasse und kein Rückgabewert
- überladbar (*verschiedene Signaturen*)
- Member bzw. Basis-Initialisierung über Initialisierungsliste

Destruktor

- Aufruf, wenn Objekt seinen Gültigkeitsbereich verlässt
- keine Parameter, kein Rückgabewert
- muss nicht explizit definiert werden, aber:
  - Default-Destruktor löscht Felder nur flach
  - Referenzen werden gelöscht, nicht aber deren Inhalte

Konstruktoren und Destruktoren

Achtung bei Vererbung

Soll der Desktruktor der Basisklasse mit aufgerufen werden, müssen **alle** Destruktoren der Vererbungskette `virtual` deklariert werden.

Beispiel

```
class DataPrinter : public DataPrinterInterface{
        private:
                ostream& out;
        public:
                DataPrinter();
                virtual ~DataPrinter();
};

DataPrinter::DataPrinter() : DataPrinterInterface(), out(cout){
}

DataPrinter::~DataPrinter(){
}
```

Strukturen und Unions

struct

- Strukturdeklaration durch `struct`, abwärtskompatibel zu C
- In C++ entspricht entspricht Klasse mit Standardzugriff `public`
- Reihenfolge der Elemente wird vom Compiler nicht verändert
- Compiler kann freie Speicherstellen zwischen den Elementen einfügen

union

- Eine `union` vereint mehrere Interpretationen eines Stück Speichers in einem Typ
- Syntaktisch äquivalent zu Strukturen
- die Gesamtgröße entspricht der Größe des größten Members
- Jedes Element entspricht einer Interpretation des darunter liegenden Speichers

Beispiel: Strukturen und Unions

Beispiel: struct

```
struct Data{
        string s;
};
```

Beispiel: union

```
union DataUnion{
        Data data;
        const void* ptr;
};
```

Überladen von Operatoren

- Operatoren sind in C++ Methodenaufrufe
  - Überladen wie in Java
  - Nur bestehende Operatoren überladbar
  - Priorität und Assoziativität von Operanden nicht veränderbar

Beispiel

```
class DataPrinter : public DataPrinterInterface{
        public:
                virtual ostream& operator<<(const void* address);
};
```

Virtuelle Methoden

- virtuelle Methode ermöglichen polymorphes verhalten
- abstrakte Methoden `<Methodendefinition> = 0;` müssen virtuell sein, da sie von der abgeleiteten Klasse implementiert werden müssen.
- Klassen die abstrakte Klassen enthalten sind selbst abstrakt und können nicht instanziert werden (vgl. Interfaces in Java)

Unterschiede zwischen Java und C++

- In Java jede Methode virtuell
- In C++ virtuelle Methoden speziell gekennzeichnet `virtual`

Beispiel

```
class DataPrinterInterface{
        public:
                virtual ostream& operator<<(const void* address) = 0;
};
```

Mehrfach-Vererbung

Wichtiger Unterschied zu Java

C++-Klassen können von mehr als einer Basisklasse erben

- Interfaces sind C++-Klassen mit abstrakten Methoden
- durch Mehrfachvererbung implementieren von Interfaces möglich
- Insgesamt aber mächtiger und komplexer (Deadly Dimond)
- Klassenstruktur ist zyklenfreier Graph
- Konflikt von gleichnamigen Member od. Methoden möglich

typedef

- Verwendung: `typedef <Definition> <Typenname>`
  - Umbenennen von Datentype
  - Definition eigener Datentypen

Beispiel

```
typedef const char* string;
```

Casts

- implizite: führt der Compiler automatisch durch
- explizite: werden vom Programmierer angefordert:
  - C-Cast: (<neuer Typ>)<variable>
    unsicher → nicht verwenden
  - C++-Casts: spezielle Checks, daher error bei invaliden Casts
    (<> gehören zur Syntax) :
    - const_cast<neuer Typ>(Variable) Cast von const Objekt zu nicht const Objekt
    - static_cast<neuer Typ>(Variable) Cast die Vererbungskette hinab, Check zur Compile-Zeit
    - dynamic_cast<neuer Typ>(Variable) Cast die Vererbungskette hinab, Check zur Laufzeit
    - reinterpret_cast<neuer Typ>(Variable) für alles andere

Beispiel

```
string output;
const void* ptr=reinterpret_cast<const void*>(output);
```

Zeiger und Referenzen

Achtung

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

Zeiger und Referenzen

Achtung

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

Grund:

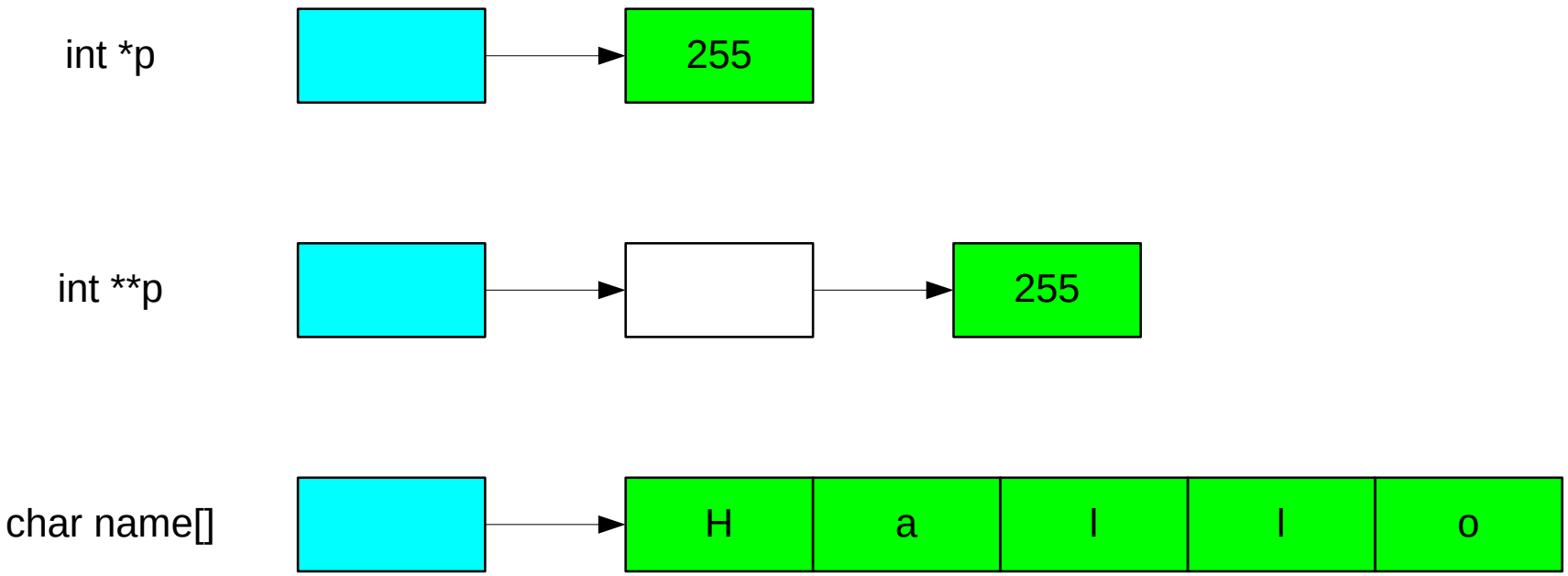
- Unkontrollierter Zugriff auf beliebige Speicherstellen
  - Programmabbruch durch das Betriebssystem (*gut!*)
  - Überschreiben beliebiger Daten (*schlecht!*)
- Fehler durch falsche Zeiger sind oft *sehr* schwerzu finden

Zeiger und Referenzen II

- Enthält Adresse einer Variable im Speicher
- Addition/Subtraktion auf Zeigern ändert Zeiger um Größe des Datentyps, auf den der Zeiger zeigt
- `NULL` spezieller Zeiger (*Zeiger auf Addresse 0*) $\rightarrow$ stets invalide Adresse
- Operationen:
  - `&` liefert Adresse einer Variablen (Adressoperator)
  - `*` liefert Variableninhalt zu einer Adresse[1]
- Referenz ähnlich zu Zeiger, jedoch:
  - keine besonderen Operationen für Objektzugriff
  - keine NULL-Referenz, daher stets gültig
  - Bei Definition an Objekt gebunden

[1] Dereferenzierungsoperator

Beispiele



Literatur I

Beispiel Projekt: Makefile und Assembler-Code

Makefile

```
TARGET=example
SOURCES=example.cc assembler.S

${TARGET}: ${SOURCES}
	${CXX} ${CFLAGS} $^ -o $@
```

Assembler Code (assembler.S)

```
.data                           /* Begin der Datendefinition */
base:                           /* Label */
zero:   .long 0                 /* reservieren von vorbelegtem Speicher */
one:    .long 1

.text                           /* Begin des Codes */

.global return1                 /* Extern verfuegbare Symbole */

return1:                        /* Begin einer Funktion */
        mov $base, %ecx
        movl 4(%ecx), %eax
        ret                     /* Ruecksprung */
```

Beispiel Projekt: example.h I

```
#ifndef __example_header__
#define __example_header__

#include <iostream>

using std::cout;
using std::hex;
using std::endl;
using std::ostream;

typedef const char* string;

enum TextIndices{
        Object,
        Interface,
        Test1,
        Test2
};

const void* getText(enum TextIndices index);

struct Data{
        string s;
};
```

Beispiel Projekt: example.h II

```
union DataUnion{
        Data data;
        const void* ptr;
};

class DataPrinterInterface{
        public:
                virtual ~DataPrinterInterface();
                virtual ostream& operator<<(Data &d) = 0;
                virtual ostream& operator<<(const void* address) = 0;
};

class DataPrinter : public DataPrinterInterface{
        private:
                ostream& out;
        public:
                DataPrinter();
                virtual ~DataPrinter();
                void printOne();
                virtual ostream& operator<<(Data &d);
                virtual ostream& operator<<(const void* address);
};

extern "C" unsigned int return1();
#endif
```

Beispiel Projekt: example.cc I

```
#include "example.h"

static const char* Text[]={"Tod␣des␣Objects", "Tod␣des␣Interfaces", "Test1", "Test2

const void* getText(enum TextIndices index){
        return Text[index];
}

DataPrinter::DataPrinter() : DataPrinterInterface(), out(cout){
}

DataPrinter::~DataPrinter(){
        const char* text=(const char*)getText(Object)
        cout <<  text << endl;
}

DataPrinterInterface::~DataPrinterInterface(){
        cout << *(Text+Interface) << endl;
}

ostream& DataPrinter::operator<<(Data &d){
        out << d.s << endl;
}

ostream& DataPrinter::operator<<(const void* address){
        out <<  hex << address << endl;
}
```

Beispiel Projekt: example.cc II

```
void DataPrinter::printOne(){
        this->out << "one: " << return1() << endl;
}

        string output;
        const void* ptr=reinterpret_cast<const void*>(output);

        DataUnion u;

int main()
{
        DataPrinter print;

        output="Test1";

        u.ptr=Text[Test1];
        print << u.data;

        cout << "&(" << output << "): " <<  ptr << endl;

        output=reinterpret_cast<const char*>(getText(Test2));
        cout << "&(" << output << "): " << ptr << endl;
```