

(Abgaben,) Debugging und Stack

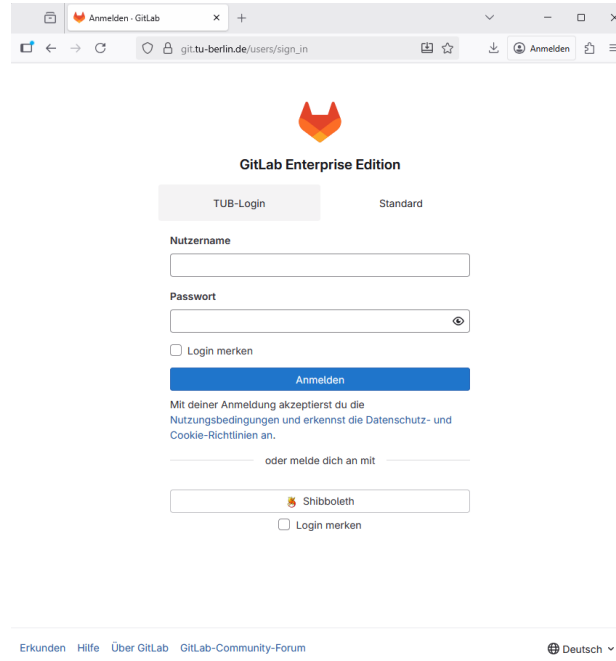
Uwe A. Kuehn | Open Distributed Systems | Einführung in die Programmierung, WS 25/26

Rückblick

- VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
- VL 1 „Hello World“: „Lebenswichtiges“, Programtablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
- VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
- VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
- VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
- VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
- VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
- VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
- VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
- VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git
- VL 10 „Debugging und Stack“: Fehlverhalten/Bugs, Fehlersuche Strategien und Werkzeuge**

Abgaben

- Zuerst müssen Sie auf gitlab.tu-berlin.de Ihren Account akkreditieren.



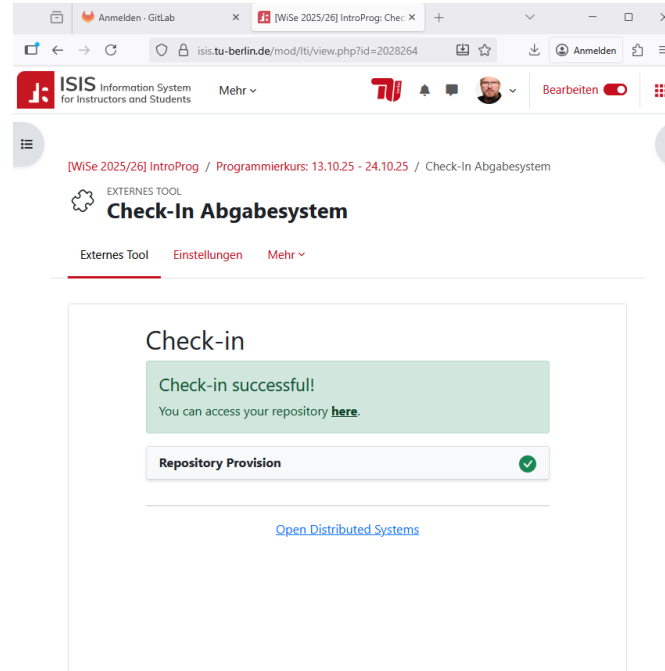
The screenshot shows a web browser window with the address bar displaying "git.tu-berlin.de/users/sign_in". The page title is "Anmelden - GitLab". The main content area features the GitLab logo (an orange fox head) and the text "GitLab Enterprise Edition". Below this, there are two tabs: "TUB-Login" (selected) and "Standard". The "TUB-Login" tab contains a login form with the following elements:

- A "Nutzername" (Username) input field.
- A "Passwort" (Password) input field with a toggle for visibility.
- A checkbox labeled "Login merken" (Remember login).
- A blue "Anmelden" (Login) button.
- A message: "Mit deiner Anmeldung akzeptierst du die Nutzungsbedingungen und erkennst die Datenschutz- und Cookie-Richtlinien an." (By logging in, you accept the terms of use and acknowledge the privacy and cookie policies).
- A separator line with the text "oder melde dich an mit" (or log in with).
- A "Shibboleth" login option with a small logo.
- A checkbox labeled "Login merken" (Remember login).

At the bottom of the page, there is a footer with links: "Erkunden", "Hilfe", "Über GitLab", and "GitLab-Community-Forum". On the right side of the footer, there is a language selector set to "Deutsch".

Abgaben

- Führen Sie den Check-In im ISIS-Kurs durch.



Abgaben

- Führen Sie die Abgaben im ISIS-Kurs durch.

The screenshot shows a web browser window with the URL `isis.tu-berlin.de/mod/lti/view.php?id=2028351`. The page header includes the ISIS logo and navigation links. The main content area displays the title 'Abgabe PKurs Blatt 10' under the heading 'EXTERNES TOOL'. Below the title, there are tabs for 'Externes Tool', 'Einstellungen', and 'Mehr'. The 'Externes Tool' tab is active, showing a submission card for 'Assignment: Abgabe PKurs Blatt 10'. The card includes a code icon, the branch name 'pkurs-b10', the due date '2025-11-14 20:00:00', and the number of hand-ins '5'. A red 'X' button is visible on the right side of the card. Below the card, there is a section titled 'Hand-ins' with a table showing a submission with the ID '7ac20c5f', a timestamp of '2025-10-08 13:51:56', and a status of 'FINISHED'.

Assignment: Abgabe PKurs Blatt 10
Branch: pkurs-b10 Due date: 2025-11-14 20:00:00 Hand-ins: 5

Hand-ins

7ac20c5f	2025-10-08 13:51:56	FINISHED	X
--------------------------	---------------------	----------	---

Abgaben

- Demo

Debugging

- Nötig während der Entwicklung
 - Hinweis: Code sollte Stück für Stück entwickelt, getestet und debugged werden!
- Nötig, wenn Code nicht compiliert
- Nötig, wenn Software „sich anders verhält als erwartet“
 - Ergebnis unterscheidet sich von Spezifikation
- Typischerweise wegen eines kleinen inkorrekten Codefragments (Bug)
 - Bug: Codefragment, das seiner Spezifikation nicht entspricht

- Konsequenzen von Bugs:
 - Compiler gibt Hinweise auf syntaktische/semantische Fehler
 - Programm hält mit Laufzeit Fehler (Run-time Error)
 - Programm hält nie an
 - Programm läuft vollständig, aber gibt inkorrekte Resultate
 - Programm läuft vollständig, aber gibt manchmal(?) inkorrekte Resultate

Wiederholung: Syntaktische Fehler

- Vorgehen, wenn der Compiler syntaktische Fehler ausgibt
- Zum **ersten Fehler** gehen (wegen möglicher Folgefehler)
 - In die entsprechende Zeile im Code gehen (siehe Fehlermeldung des Compilers)
 - Fehler verstehen
 - Fehler beheben
 - Kompilieren
 - Fehler behoben?
 - Wenn ja gegebenenfalls nächsten Fehler beheben
 - Wenn nein, versuchen, Fehler zu verstehen und zu beheben...

Debugging: Fehlerlokalisierung

- Bug: Codesegment mit nicht beabsichtigten Aktionen.

Man muss nachvollziehen:

- Was das Programm **machen sollte**
- Was das Programm **wirklich macht**

- **Problem:** Zu viel Informationen!
- **Lösung:** Einkreisen des Problems!

Debugging: Suchstrategie

Ausnutzung typischer Strukturen eines Programms

- Fokus auf verdächtigen Datenstrukturen
 - Nach Initialisierung
 - An „strategischen Punkten“ während der Programmausführung
 - Am Ende des Programms
- Strategische Punkte?
 - Nach der ersten, zweiten, mittleren Iteration einer Schleife
 - Nach einem Tastendruck, der ein interaktives Programm zum Absturz bringt
 - Nach dem Punkt, an dem das Programm die letzte korrekte Ausgabe geliefert hat

Debugging: Verstehen des Fehlers

- Vorgehen, nachdem eine fehlerhafte Stelle gefunden wurde:
 - Was ist der Zustand des Programms vor Ausführung des fehlerhaften Codes?
 - Was ist der Zustand des Programms nach der Ausführung?
 - Was ist der erwartete Zustand? (entsprechend Spezifikation)
- Was ist der **Zustand eines Programms**?
 - Namen und Werte aller aktiven Variablen
 - Z.B. **x** = 3, **y** = 7, ...

Programmzustandsuntersuchung mit `printf`

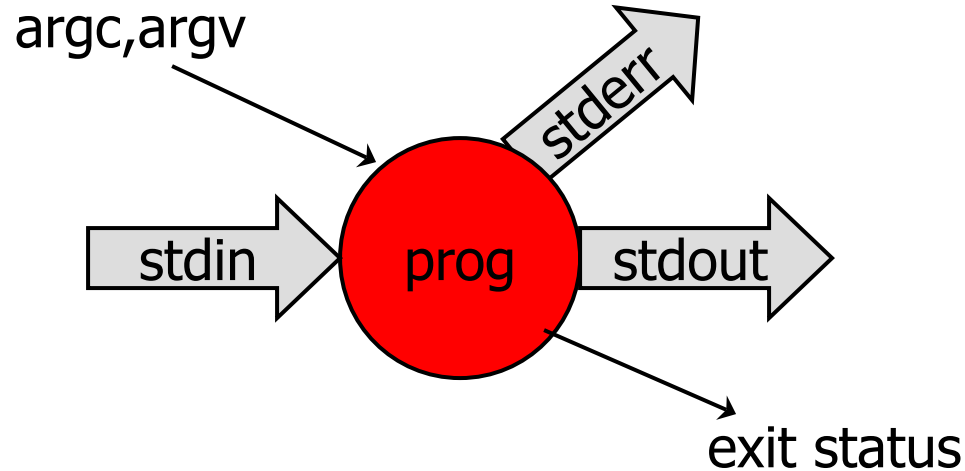
- `printf` von „suspekten“ Variablen
- Probleme:
 - Ändert das Programm
 - Raten bezüglich der suspekten Variablen
 - Ausgabe vieler Daten notwendig

Debugger

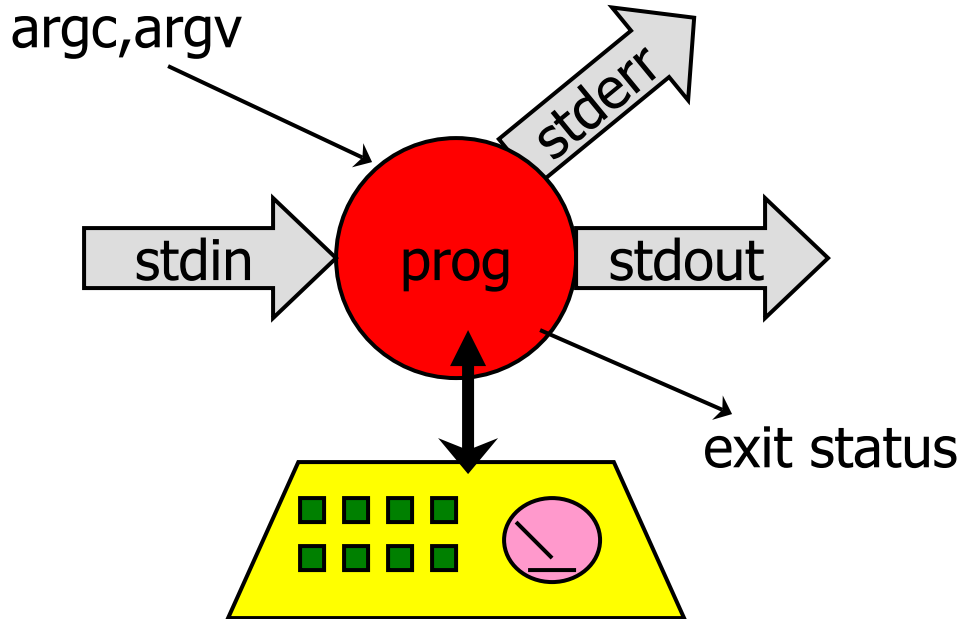
Programmzustandsuntersuchung mit Debugger

- Debugger
 - Kann ein Programm an einem bestimmten Punkt unterbrechen
 - Kann bei Unterbrechung den Zustand des Programms anzeigen
- Zusätzlich
 - Ermöglicht, den Zustand von Programmen, die durch „Run-time“ Fehler (Laufzeitfehler) abgestürzt sind, anzuzeigen
 - Führt oft leichter zu dem Punkt, wo der Fehler auftritt

Normale Programmausführung



Programmausführung mit Debugger



Debugger

- Demo Integrated Development Environment

Debugger: Werkzeuge

- **gdb** und **lldb** sind Kommandozeilen-basierte Debugger für C, C++, ...
- Funktionalität:
 - Kontrollierte Programmausführung
 - Anzeige des Zustands des Programms
 - Zusätzlich: Änderung von Variablen, Speicher, ...

gdb: Start

- Kompilieren mit `-g` als Argument
- Aufruf ohne core file:
 - `gdb prog`
- Aufruf mit core file:
 - `gdb prog core`
- Debugger als Shell zum Kontrollieren und Beobachten eines Programms

gdb: Befehle #1

- `quit` – verlässt gdb
- `help [CMD]` – on-line Hilfe für Kommando CMD
- `run ARGS` – Ausführen des Programms mit Argumenten ARGS,
z.B. aus dem Befehl
`./prog arg1 arg2`
wird im Debugger
`run arg1 arg2`

gdb: Befehle #2

- **break [PROC|LINE]** – Setzen eines Haltepunkts (breakpoint). Wenn das Programm die Funktion PROC (oder die Linie LINE) erhält, wird die Ausführung des Programms unterbrochen und die Kontrolle an **gdb** übergeben
- **next** – single step (over procedures): Ausführen des nächsten Statements. Falls das Statement ein Funktionsaufruf ist, ausführen des gesamten Funktionskörpers
- **step** – single step (into procedures): Ausführen des nächsten Statements. Falls das Statement ein Funktionsaufruf ist, halte beim ersten Statement in der Funktion

gdb: Befehle #3

- `bt/backtrace` – gibt Aufrufkette (Stack-trace) aus
 - Mit core dump: Finden, welche Funktion das Programm ausgeführt hat, als es abgestürzt ist
 - Bei Unterbrechung: Ausgabe der Aufrufkette

Exkurs: Aufrufketten

Aufrufkette: Beispiel

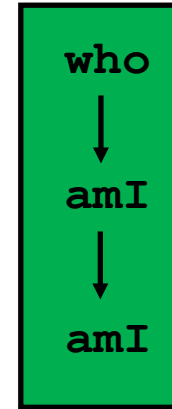
- Code Struktur

```
who (...)  
{  
    .  
    .  
    amI () ;  
    .  
    .  
}
```

Funktion **amI** rekursiv

```
amI (...)  
{  
    .  
    .  
    amI () ;  
    .  
    .  
}
```

Aufrufkette



Aufrufkette

- Recap: Fakultät berechnen
- Beispiel: Potenzen 2^n für $n \geq 0$ berechnen

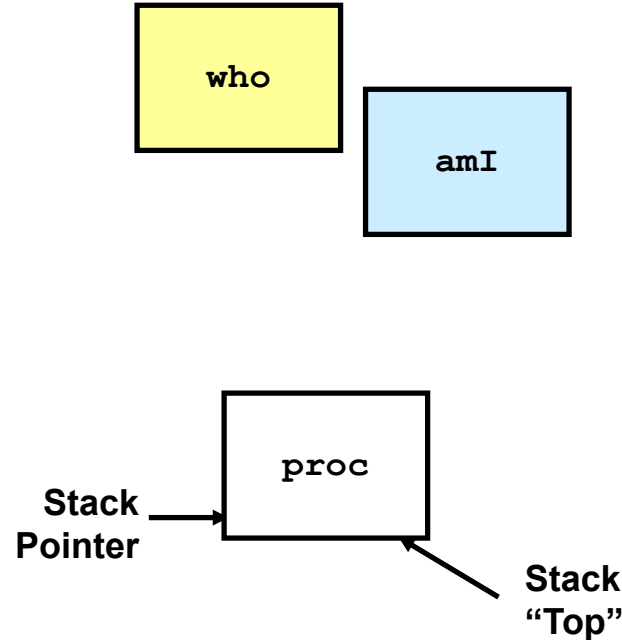
$$2^n = \begin{cases} 1, & \text{falls } n \leq 1 \\ 2 \cdot 2^{(n-1)} & \text{sonst} \end{cases}$$

Video

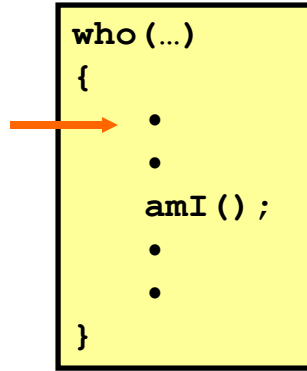


Stack-Frames

- Inhalt
 - Lokale Variablen
 - Rückkehrinformation
 - Parameter
- Speicherverwaltung
 - Speicher allokiert beim Eintritt in die Funktion
 - Freigegeben bei der Rückkehr
- Hilfsmittel
 - Stack-Pointer



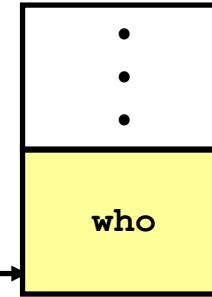
Stack-Operationen



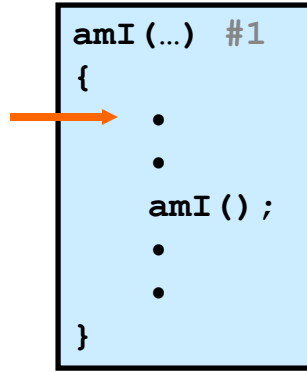
Aufrufkette

who

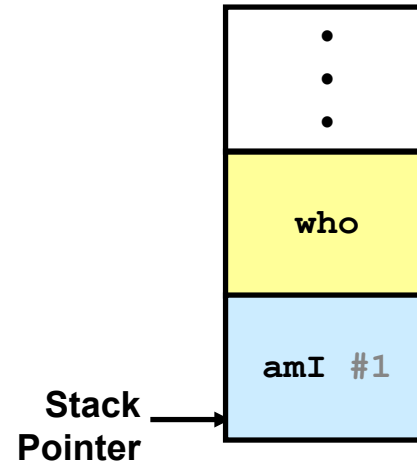
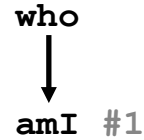
Stack
Pointer



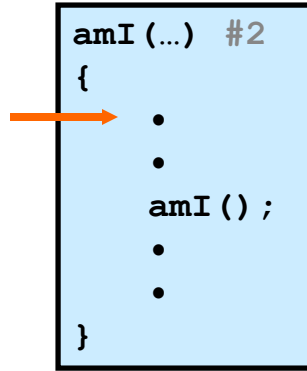
Stack-Operationen



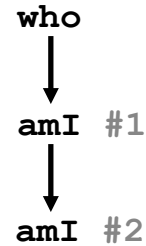
Aufrufkette



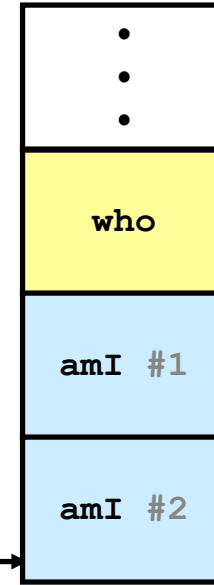
Stack-Operationen



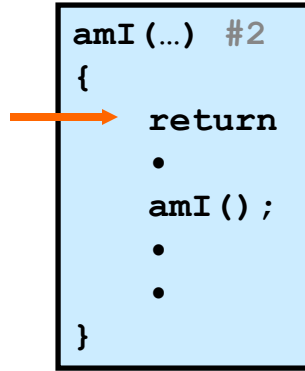
Aufrufkette



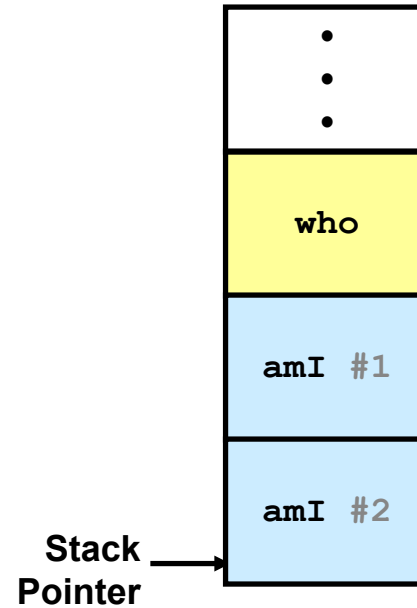
Stack
Pointer



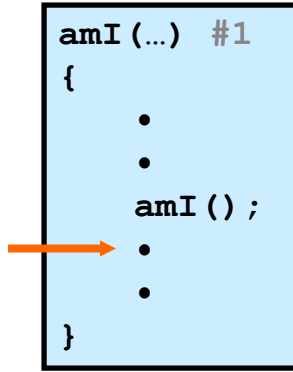
Stack-Operationen



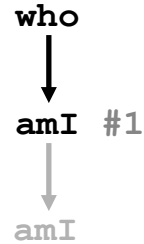
Aufrufkette



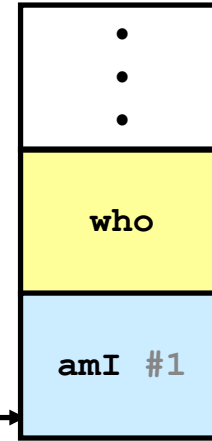
Stack-Operationen



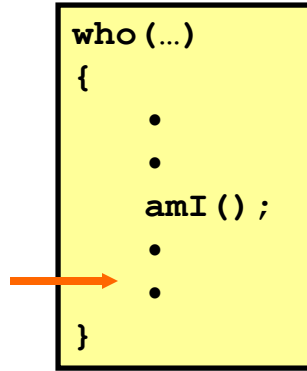
Aufrufkette



Stack
Pointer



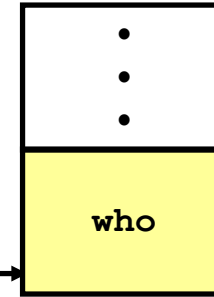
Stack-Operationen



Aufrufkette



Stack
Pointer



Zurück zum Debuggen

gdb: Befehle #3

- `bt/backtrace` – gibt Aufrufkette (Stack-trace) aus
 - Mit core dump: Finden, welche Funktion das Programm ausgeführt hat, als es abgestürzt ist
 - Bei Unterbrechung: Ausgabe der Aufrufkette

gdb: Befehle #4

- **up [N]** – Wechseln des Kontexts eine Ebene höher im Stack; Ändert den Rahmen (Scope) einer bestimmten Funktion im Stack
- **down [N]** – Wechseln des Kontexts eine Ebene niedriger
- **list [LINE/PROC]** – Anzeigen des Programmcodes; zeigt 5 Zeilen Code vor und nach dem momentanen Statement
- **print EXPR** – zeige die Werte der Expression EXPR

DEMO 1 `simple_stack.c`

Workflow

- Am häufigsten nach einem Laufzeitfehler
- Starten von `gdb` mit core Datei, anzeigen mit `bt`, welche Programmzeile zum Absturz geführt hat
- Wo das Programm abgestürzt ist, ist häufig ein erster Hinweis auf den Ort des Fehlers
- Allerdings nur ein erster Hinweis: Der Fehler kann viel früher aufgetreten sein.

- Wenn man eine Idee hat, wo der Fehler sein kann
 - Setzen eines **Breakpoints** kurz vorher im Code
 - **Laufen** lassen des Programms (mit denselben Daten)
 - Ausführen des Programms in **Einzelschritten** (single-step) durch die suspekte Region (Nach dem Breakpoint)
 - **Ansehen** der Werte der suspekten Variablen nach jedem Schritt
- Hierdurch sollte man feststellen können, welche Variable den falschen Wert hat

Workflow

- Nachdem man herausgefunden hat, dass der Wert einer Variablen (z.B. x) falsch ist, muss man herausfinden, **warum** der Wert falsch ist.
- Es gibt zwei Möglichkeiten:
 - Das Statement, das x den Wert zuweist, ist falsch
 - Die Werte der anderen Variablen im Statement sind falsch
- Beispiel
 - `if (c > 0) { x = a + b; }`
 - Falls wir wissen, dass x falsch ist und dass die Bedingung und der Ausdruck richtig implementiert sind, dann müssen wir finden, wo a und b gesetzt werden

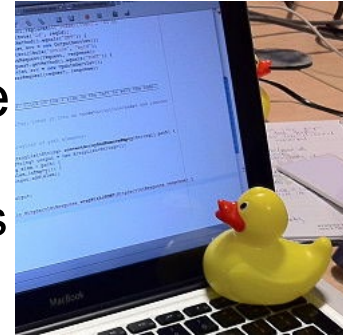
Debugging Ideen

- Debugging Beispiel: „**wissenschaftliches Vorgehen**“
 - Entwicklung einer Hypothese
 - Datensammlung zur Verifikation der Hypothese
 - Änderung der Hypothese, falls neue Beweismittel vorliegen
- Die Hypothese ist:
 - „Ich denke, der Bug liegt in diesem Statement ...“

Gesetze des Debuggens

(Zoltan Somogyi, Melbourne University) Sorry for Denglish

- Before you can fix it, you must be able to break it (consistently)
 - Nicht reproduzierbare Bugs ... Heisenbugs ... sind schwierig
- If you can't find a bug where you're looking, you're looking in the wrong place
 - Eine Pause machen und später weitermachen, ist im Allgemeinen eine gute Idee
- It takes two people to find a subtle bug, but only one of them needs to know the program
 - Die zweite Person stellt Fragen, um die Annahmen des Debuggers in Frage zu stellen



Bugs: Beispiel

Beispiel eines fehlerhaften(?) Programms, das einen Speicherbereich mit den Zahlen 256...1 füllen soll:

```
int main() {  
    int n = 256;  
    int buf[n];  
    unsigned int i;  
    for (i = n - 1; i >= 0; i--) {  
        buf[i] = n - i;  
    }  
}
```

Debugger

- Demo ohne IDE `reverse_buf.c`

Fehler in der Speicherverwaltung finden

Typische Speicherfehler

- Memory Leaks
 - Speicher wird angefordert, aber nicht freigegeben
 - Fatal bei lang laufenden Prozessen
- Uninitialisierter Speicher
 - Speicher wird gelesen, ohne dass vorher geschrieben wurde
 - Fehler treten oft unerwartet auf, abhängig vom Speicherinhalt
 - Kann Informationen preisgeben (z.B. Crypto-Schlüssel)
- Use after free
 - Speicher wird gelesen, nachdem er freigegeben wurde
 - Fehler treten auf, wenn der Speicher neu vergeben wird

... ist ein Tool zum Auffinden von Speicherfehlern.

- Speicherleaks `--leak-check=full`
- Uninitialisierter Speicher `--undef-value-errors=yes`
`--malloc-fill=0xfe`
- Use after free `--track-origins=yes`

➤ Wird auch von den automatischen Tests verwendet, um Speicherfehler zu finden

clang Sanitizer

Oder falls valgrind nicht verfügbar, Address Sanitizer:

- Aktivieren `-fsanitize=address`
- Für bessere Strack-Traces `-fno-omit-frame-pointer`

Oder Memory Sanitizer (uninitialisierter Speicher):

- Aktivieren `-fsanitize=memory`
- Ursprung `-fsanitize-memory-track-origin`
- Für bessere Strack-Traces `-fno-omit-frame-pointer`

DEMO 3 `wrong_alloc.c`

Beispiel: Bufferoverflows

String-Library Code

Implementation der Unix Funktion gets

- Keine Möglichkeit die Anzahl, der zu lesenden Zeichen anzugeben

```
/* Get string from stdin */
char *gets(char *dest) {
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

- Ähnliche Probleme auch bei anderen Unix-Funktionen
 - scanf, fscanf, sscanf, mit %s Konvertierungsspezifikation

Angreifbarer Buffer-Code

```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Way too small! */  
    gets(buf);  
    puts(buf);  
}
```

```
int main()  
{  
    printf("Type a string:");  
    echo();  
    return 0;  
}
```

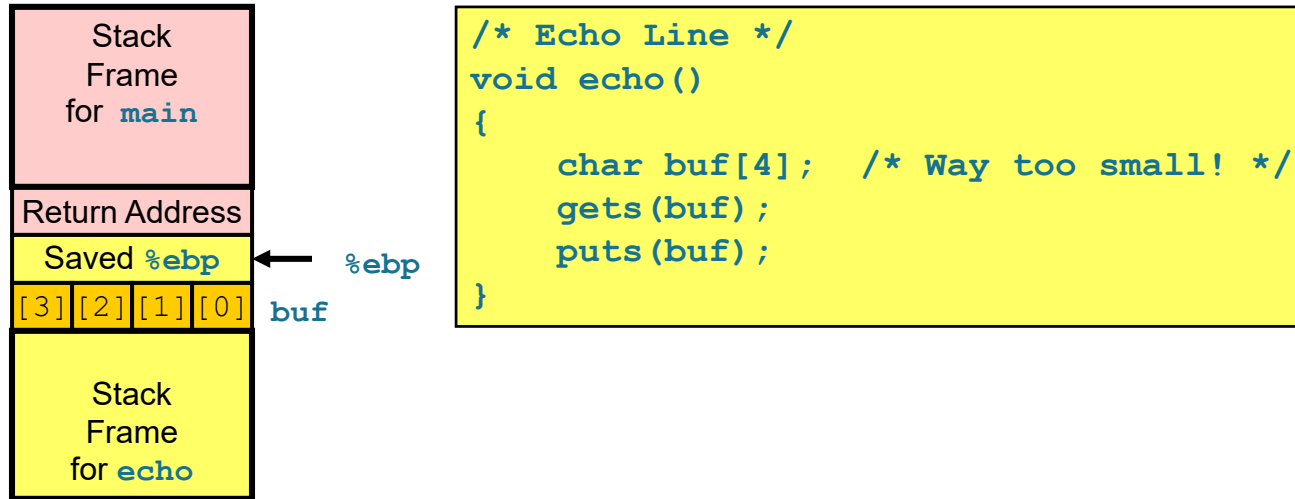
Beispiel eines Bufferoverflows

```
unix>./bufdemo  
Type a string:123  
123
```

```
unix>./bufdemo  
Type a string:1234  
Segmentation Fault
```

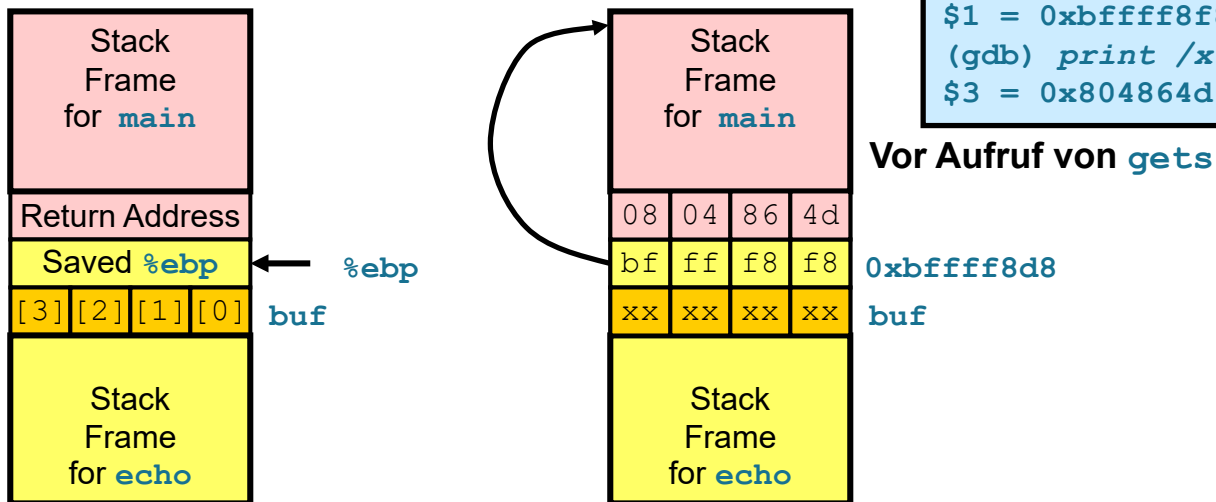
```
unix>./bufdemo  
Type a string:12345678  
Segmentation Fault
```

Stack während Bufferoverflow: #1



Stack während Bufferoverflow: #2

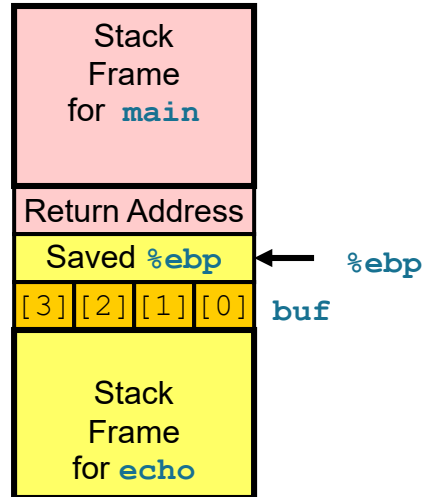
```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```



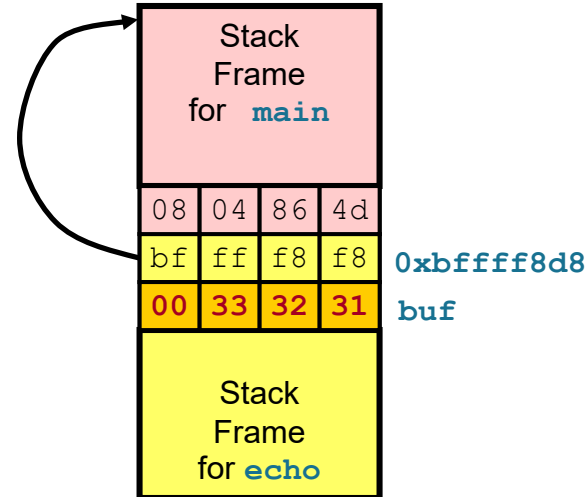
Vor Aufruf von `gets`

Stack während Bufferoverflow: #3

Vor dem Aufruf von `gets`

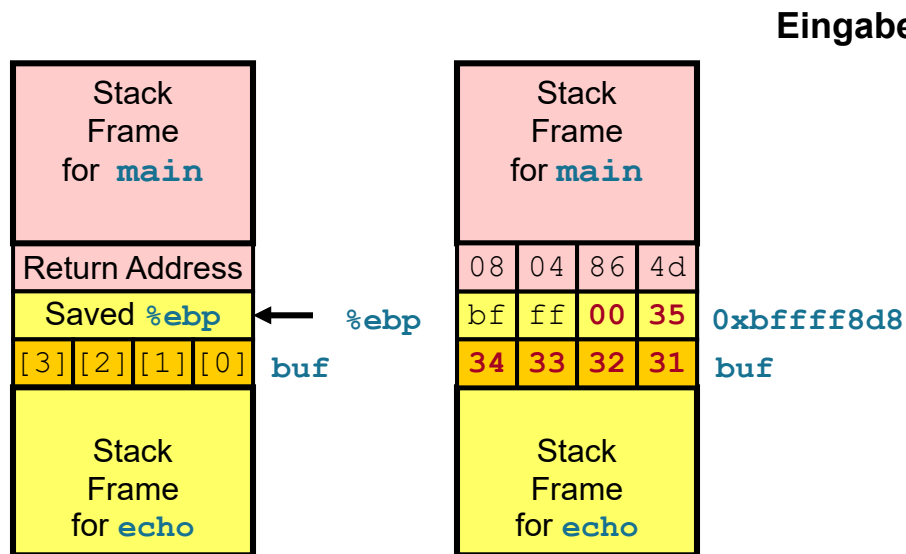


Eingabe: "123"



Kein Problem

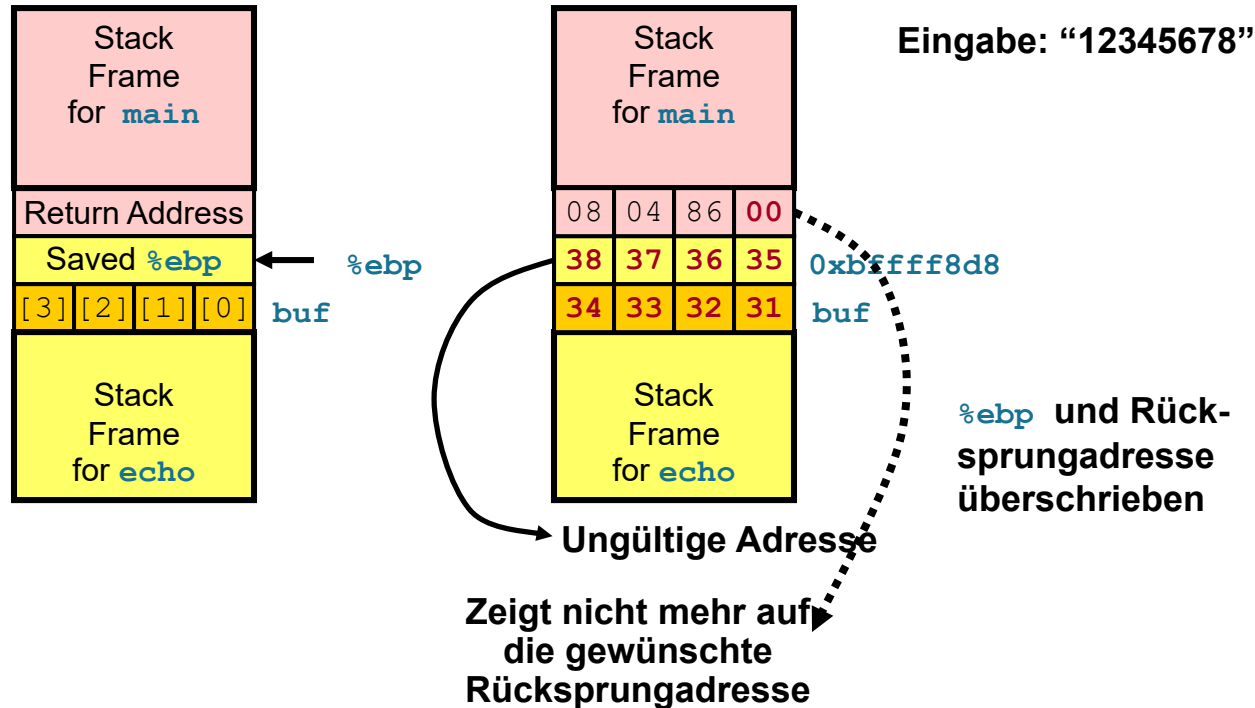
Stack während Bufferoverflow: #4



Gespeicherter Wert
von `%ebp` wird auf
`0xbffff0035`
gesetzt

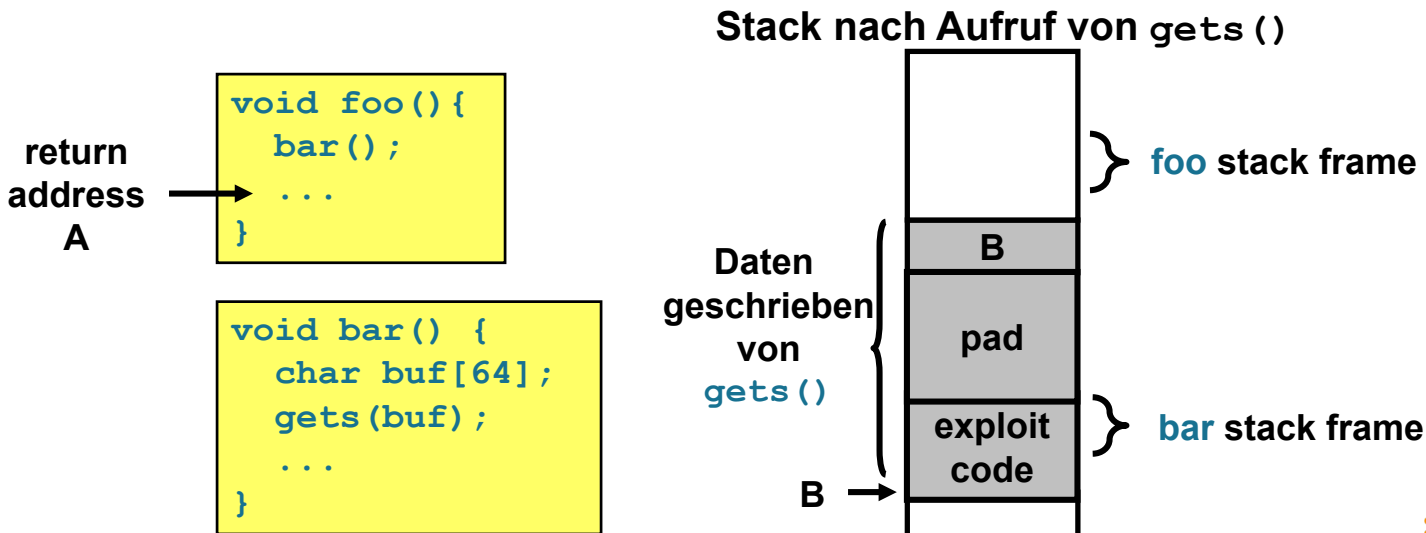
Schlechte Nach-
richt, wenn später
versucht wird, den
Wert von `%ebp`
wiederherzustellen

Stack während Bufferoverflow: #5



Ausnutzung von Bufferoverflows

- Eingabe String enthält binär Codierung des ausführbaren Codes
- Überschreibt Rücksprungadresse mit Adresse des Buffers
- Wenn bar() return ausführt, erfolgt der Sprung zum Exploit (exploit code)



Ausblick

- VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
- VL 1 „Hello World“: „Lebenswichtiges“, Programtablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
- VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
- VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
- VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
- VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
- VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
- VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
- VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
- VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git
- VL 10 „Debugging und Stack“: Fehlverhalten/Bugs, Fehlersuche Strategien und Werkzeuge