Übung 07.1

Debugging and Testing

0. Organisatorisches

- Abgabe des 6. Übungsblatt wurde bis **08.12.16** verlängert
- 7. Übungsblatt kommt demnächst raus und ist bis zum **15.12.16** abzugeben
- **02.12.16 (heute):** Einführung in Debugging mit gdb
- **09.12.16:** Besprechung des 6. Übungsblattes
- **16.12.16:** Besprechung des 7. Übungsblattes
- Ab **19.12.16** vorlesungsfreie Zeit

Einführung in: Debugging mit GDB

http://greeneggmedia.com/images/uploads/blog/Debugging_Comic.gif

1. Was ist Debugging und wozu benötigt man es?

http://i.makeagif.com/media/11-18-2016/GbdZAy.gif

1. Was ist Debugging und wozu benötigt man es?

- Handwerkzeug zum Auffinden von Fehlern im Programmcode
- Warum nicht einfach cout oder System.out.println benutzen?
  - Bläht Programmcode unnötig auf
  - Kann in größeren Projekten leicht vergessen werden
  - Kann ebenfalls Fehler enthalten
- Mit ihm kann Programmcode Zeile für Zeile durchgegangen und die Werte der Variablen ausgelesen werden
- Wo befindet man sich derzeit im Backtrace (wie ist man genau an diese eine Stelle gelangt?)

2. Was ist GDB?

- GDB = The GNU Debugger
- Verfügbar für Windows, OSX und Linux: https://www.sourceware.org/gdb/
- Dokumentation: https://sourceware.org/gdb/current/onlinedocs/gdb/
- Unterstützte Sprachen: C, C++, Object C, D, Go, Ada, Pascal, Fortran, OpenCL C, Rust, Modula-2
- Es besitzt eine interaktive Shell:
  - Recall history über Pfeiltasten
  - Vervollständigt Wörter mit TAB
  - Viele weitere Funktionen

3. Aller Anfang ist schwer

- Normal werden C++ Programme so kompiliert:

```
g++ [flags] <source files> -o <output file>
```

- Um GDB zu benutzen muss das Programm wie folgt kompiliert werden:

```
g++ [flags] -g <source files> -o <output file>
```

- `-g` generiert Source-Level Debugginginformationen, die wir für GDB benötigen

- Start einer GDB-Sitzung:

```
gdb <output file>
```

4. Breakpoints, Watchpoints, Catchpoints

- Eine der meistgenutzten Funktionen eines jeden Debuggers
- Sie stoppen das Programm unter bestimmten Voraussetzungen um ein bestimmtes Problem zu untersuchen
- Man kann so viele Breakpoints wie möglich haben – ihr entscheidet wo ihr welche benötigt
- Die Zeile mit einem Breakpoint wird ausgeführt, nachdem man einen Schritt weiter gegangen ist

4.1 Breakpoints

- Stoppen das Programm immer, wenn ein Punkt im Programm erreicht wurde
- Sie können Conditions enthalten, die eine feinere Kontrolle über das Stoppen ermöglichen
- Um einen Breakpoint zu setzen benutzt man entweder: `break` oder `b`
- Bsp.: Main-Funktion startet in Zeile 10:

```
(gdb) break 10
(gdb) b main.cpp:10
(gdb) b main.cpp:10 if (argc == 0)
(gdb) b main.cpp:main
(gdb) b main
```

- All diese Befehle setzen einen Breakpoint in der 10. Zeile der main.cpp-Datei
- Es ist ebenfalls möglich einen Breakpoint in jeder anderen Datei oder an den Anfang einer Funktion zu setzen

https://sourceware.org/gdb/current/onlinedocs/gdb/Set-Breaks.html#Set-Breaks

4.2 Watchpoints

- Spezielle Breakpoints, die das Programm stoppen, wenn der Wert eines Ausdruckes (z.B. einer Variable) sich ändert
- Definiert keinen besonderen Ort im Programm (= Data Breakpoint)
- Abhängig vom Scope der Variable relativ zur aktuellen Programmposition wird entschieden, welche Variable beobachtet wird

```
(gdb) watch variable
```

https://sourceware.org/gdb/current/onlinedocs/gdb/Set-Watchpoints.html#Set-Watchpoints

4.3 Catchpoints

- Spezielle Breakpoints, die das Programm stoppen, sobald ein bestimmtes Event eingetreten ist, wie z.B. das Werfen einer C++ Exception oder das Laden einer Shared Library

```
(gdb) catch <event>
```

- <event> = throw, catch, exception, assert, signal, ...
- Es können zusätzliche Parameter mit angegeben werden

- Beispiel:

```
(gdb) catch signal SIGSEGV
```

https://sourceware.org/gdb/current/onlinedocs/gdb/Set-Catchpoints.html#Set-Catchpoints

4.4 Löschen und Deaktivieren der *points

- Manchmal ist es nützlich bestimmte Breakpoints zu löschen oder sie temporär zu deaktivieren
- Löschen

```
(gdb) clear [<location>]
```

```
(gdb) delete [breakpoints]
```

- Deaktivieren

```
(gdb) disable [breakpoints]
```

```
(gdb) enable [breakpoints]
```

https://sourceware.org/gdb/current/onlinedocs/gdb/Delete-Breaks.html#Delete-Breaks
https://sourceware.org/gdb/current/onlinedocs/gdb/Disabling.html#Disabling

5. Running, Continuing & Stepping

- Wir wissen jetzt, wie wir Breakpoints in unser Programm setzen können
- Jetzt wollen wir das Programm laufen lassen und durchsteppen
- In jedem Debugger, die Mechanismen sind alle ziemlich ähnlich, um durch den Code durchzugehen
  - `continue`
  - `step`
  - `next`
  - `finish`
  - `until`
- Der gleiche Befehl kann mit `ENTER` wiederholt werden (sehr hilfreich fürs Durchlaufen des Porgramms mit `c, s, n,` ...)

https://sourceware.org/gdb/current/onlinedocs/gdb/Continuing-and-Stepping.html#Continuing-and-Stepping

5.1 run || r

- So einfach wie es klingt, um ein Programm in GDB laufen zu lassen, tippe:

```
(gdb) run [arg1, arg2, ...]
```

- Wenn kein Breakpoint gesetzt wurde, läuft das Programm bis zum return 0 der main oder bis eine Exception aufgefangen wird
- Ansonsten stoppt es beim ersten Breakpoint

- Beispiel:

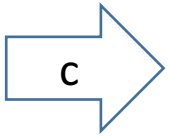
```
(gdb) run
Starting program: <Path/To/Executable>
Breakpoint 1, main () at main.cpp:6
```

5.2 continue || c

- Setzt die Ausführung des Programmes beim letzten Breakpoint fort und stoppt bis ein neuer Breakpoint erreicht wird

```
(gdb) continue
```

```
1   int main() {
2       printf("Hello!");
3       func1();
4       printf("Goodbye!");
5   }
6   void func1() {
7       int i = 1;
8       func2(i);
9       i = 42;
10  }
11  void func2(int a) {
12     int b = a;
13  }
```

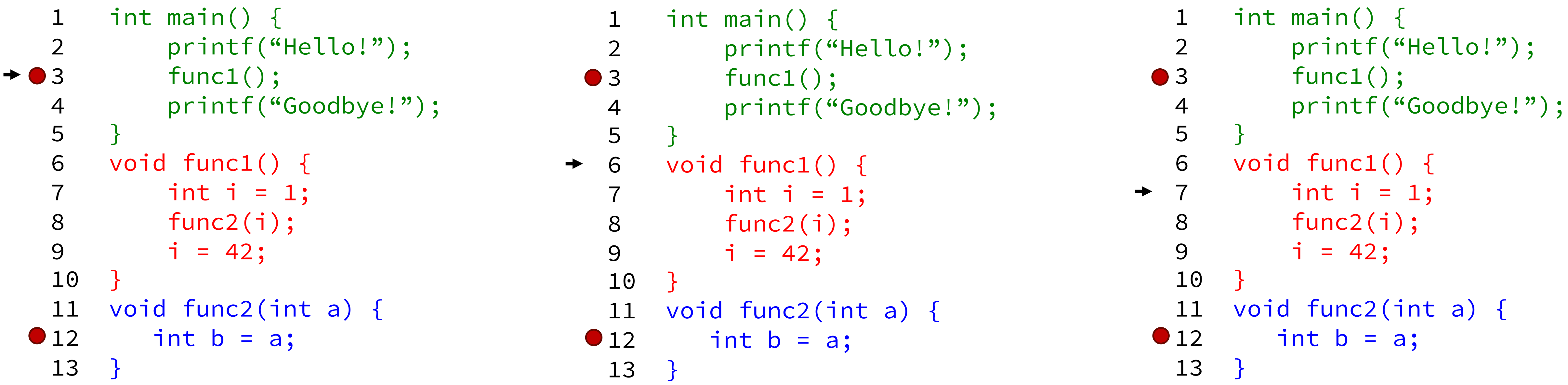



```
1   int main() {
2       printf("Hello!");
3       func1();
4       printf("Goodbye!");
5   }
6   void func1() {
7       int i = 1;
8       func2(i);
9       i = 42;
10  }
11  void func2(int a) {
12     int b = a;
13  }
```

5.3 step || s

- Tritt in jede Funktion innerhalb der Zeile ein
- Durchläuft alle Schleifendurchläufe, ausgenommen Funktionen, die nicht mit Debugging-Informationen kompiliert wurden (dafür benutzt man `stepi`)

```
(gdb) step
```



```
int main() {
    printf(“Hello!”);
    func1();
    printf(“Goodbye!”);
}
void func1() {
    int i = 1;
    func2(i);
    i = 42;
}
void func2(int a) {
   int b = a;
}
```

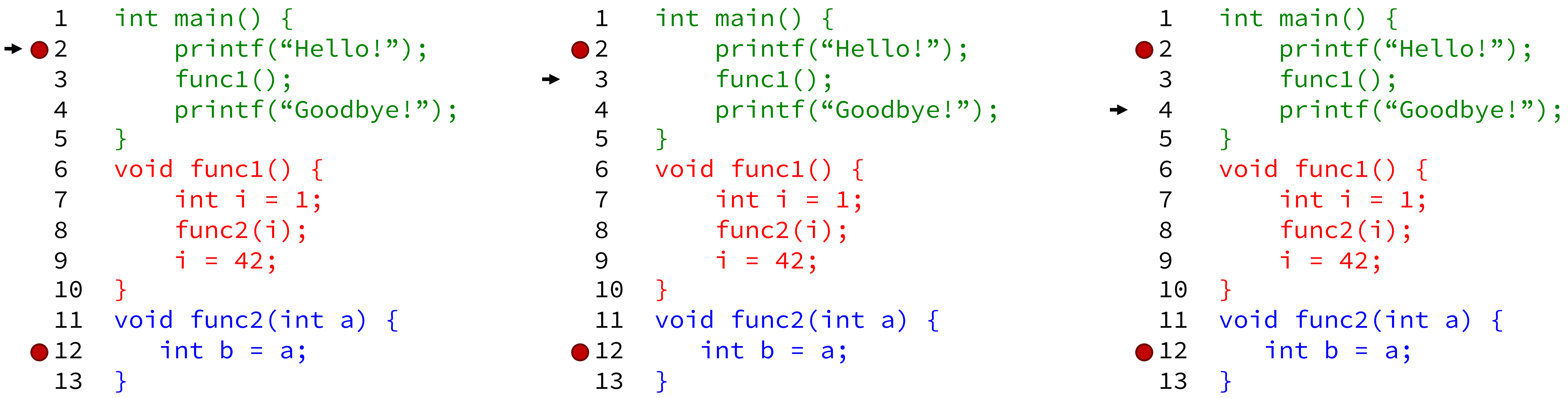
```
int main() {
    printf(“Hello!”);
    func1();
    printf(“Goodbye!”);
}
void func1() {
    int i = 1;
    func2(i);
    i = 42;
}
void func2(int a) {
   int b = a;
}
```

```
int main() {
    printf(“Hello!”);
    func1();
    printf(“Goodbye!”);
}
void func1() {
    int i = 1;
    func2(i);
    i = 42;
}
void func2(int a) {
   int b = a;
}
```

5.4 next || n

- Geht weiter zur nächsten Codezeile im aktuellen (innermost) Stack Frame
- Durchläuft alle Schleifendurchläufe, ausgenommen Funktionen, die nicht mit Debugging-Informationen kompiliert wurden

```
(gdb) next
```



```
int main() {
    printf(“Hello!”);
    func1();
    printf(“Goodbye!”);
}
void func1() {
    int i = 1;
    func2(i);
    i = 42;
}
void func2(int a) {
   int b = a;
}
```

```
int main() {
    printf(“Hello!”);
    func1();
    printf(“Goodbye!”);
}
void func1() {
    int i = 1;
    func2(i);
    i = 42;
}
void func2(int a) {
   int b = a;
}
```

```
int main() {
    printf(“Hello!”);
    func1();
    printf(“Goodbye!”);
}
void func1() {
    int i = 1;
    func2(i);
    i = 42;
}
void func2(int a) {
   int b = a;
}
```

5.6 finish || fin

- Setzt das Programm bis genau nach der Funktion-return im ausgewählten Stack Frame fort

```
(gdb) finish
```

```
1   int main() {
2       printf("Hello!");
3       func1();
4       printf("Goodbye!");
5   }
6   void func1() {
7       int i = 1;
8       func2(i);
9       i = 42;
10   }
11  void func2(int a) {
12     int b = a;
13  }
```

fin

```
1   int main() {
2       printf("Hello!");
3       func1();
4       printf("Goodbye!");
5   }
6   void func1() {
7       int i = 1;
8       func2(i);
9       i = 42;
10   }
11  void func2(int a) {
12     int b = a;
13  }
```

5.7 until || u

- Ähnlich zu `next`, außer wenn ein Sprung erkannt wird. Hier wird die Ausführung des Programmes automatisch weitergeführt bis der Programmzähler größer als die Adresse des Sprungs ist – WHAT?!
- Es ist ebenso möglich das Programm bis zu einer bestimmten Zeile laufen zu lassen, zum Beispiel mit „`u 20`“

```
(gdb) until
```

```
1   int main() {
2       printf("Hello ");
3       for(int i = 0; i < 10; ++i){
4           printf("World" + i);
5       }
6       printf("Goodbye!");
5   }
```

3*u

```
1   int main() {
2       printf("Hello ");
3       for(int i = 0; i < 10; ++i){
4           printf("World" + i);
5       }
6       printf("Goodbye!");
5   }
```

6. Informationen ausgeben

- Wir können nun das Programm laufen lassen (`run`), unterbrechen (`break`) und durchschreiten (`step`, ...), aber ohne Informationen welchen Wert und Typ eine Variable hat und ohne Angabe, wo wir uns derzeit befinden, wird es schwierig Bugs zu finden
- GDB erlaubt es verschiedene Informationen auf der Konsole auszugeben
- Wenn Probleme mit GDB auftreten oder man einen Befehl nicht weißt, benutze:

```
(gdb) help
```

6.1 print || p

- Man stelle sich vor, dass ein Programm aus einem struct Person besteht und zwei Personen in der main erstellt werden
- Wenn das Programm unterbrochen wird, welchen Wert haben `a`, `b` und `s`?

```
struct person {
    person(string n, string ln, int a):
        name{n}, lastname{ln}, age{a} {}
    string name;
    string lastname;
    int age;
};
```

```
int main() {
    person  a {"Peter", "Lustig", 45};
    person* b = new person("Albert",
                "Einstein", 12);
    string s = "Hello World";
}
```

6.1 print || p

```
(gdb) p a
$1 = {name = "Peter", lastname = "Lustig", age = 45}
(gdb) p b
$2 = (person *) 0x100300290
(gdb) p *b = {name = "Albert", lastname = "Einstein", age = 12}
(gdb) p a.name
$6 = "Peter”
(gdb) p b -> name
$5 = "Albert”
(gdb) p s
$6 = "Hello World"
```

- Mit `print` kann man die Speicheraddresse ($2), den Wert einer Variable ($6) und sogar den Inhalt einer Klasse oder einem Struct sich ausgeben lassen
- Es gibt noch mehr Ouput-Formate für verschiedene Arten von Variablen

https://sourceware.org/gdb/onlinedocs/gdb/Output-Formats.html

6.2 display

- Wenn man den Wert eines Ausdruckes häufig ausgegeben haben möchte (z.B. weil man sehen will, wie er sich verändert), sollte man ihn zu einer *automatischen Anzeigenliste* hinzufügen
- Jedes Mal, wenn das Programm gestoppt wird, wird die Liste an Ausdrücken angezeigt.

```
(gdb) display s
1: s = "Hello World"
```

- Weitere Funktionen: `undisplay`, `disable display`, `enable display`

https://sourceware.org/gdb/current/onlinedocs/gdb/Auto-Display.html#Auto-Display

6.3 list || l

- Wenn man die nächsten (10) Codezeilen sehen möchte, benutzt man:

```
(gdb) list
```

- Ist man an einer bestimmten Region interessiert?

```
(gdb) list <filename>:<linenum>
```

- Oder ist man am Programmcode einer Funktion interessiert?

```
(gdb) list <filename>:<function>
```

https://sourceware.org/gdb/current/onlinedocs/gdb/List.html#List

6.4 ptype & whatis

- Ist man sich nicht ganz sicher, welchen Typ eine Variable hat, kann mit `ptype` die Definition oder mit `whatis` der Datentyp ausgegeben werden

```
(gdb) ptype a
type = struct person {
    std::__1::string name;
    std::__1::string lastname;
    int age;
  public:
    person(std::__1::string, std::__1::string, int);
}
```

```
(gdb) whatis a
type = person
```

https://sourceware.org/gdb/onlinedocs/gdb/Symbols.html

6.5 backtrace || bt || where || info s

- Möchte man wissen, wo man sich exakt im Programm befindet, kann man `backtrace <n>` für die n-innermost Stack-Frames verwenden
- Zusätzlich können lokale Variablen mit `bt full` angezeigt werden

```
(gdb) bt
#0  func2 (a=1) at main.cpp:38
#1  0x0000000100000c77 in func1 () at main.cpp:33
#2  0x0000000100000c0f in main (argc=1, argv=0x7fff5fbffb48) at main.cpp:28
```

```
(gdb) bt full 1
#0  func2 (a=1) at main.cpp:38
No locals.
(More stack frames follow...)
```

https://sourceware.org/gdb/current/onlinedocs/gdb/Backtrace.html#Backtrace

6.6 frame || f

- Auswählen und Ausgeben eines Stack-Frames
- Ohne Argument gibt es den gerade ausgewählten Stack-Frame aus

```
(gdb) frame
#2  0x0000000100000c0f in main (argc=1, argv=0x7fff5fbffb48) at main.cpp:28
28        func1();
```

- Ein Argument spezifiziert den auszuwählenden Frame

```
(gdb) frame 1
#1  0x0000000100000f97 in func1 () at main.cpp:33
33          func2(i);
```

https://sourceware.org/gdb/current/onlinedocs/gdb/Backtrace.html#Backtrace

6.7 info || i

- `info` ist ein generischer Befehl um Informtionen des debuggten Programmes auszugeben
- Liste aller info-Befehle:

```
(gdb) help info
```

- Liste aller Breakpoints im Programm

```
(gdb) info breakpoints
```

- Status des Programms

```
(gdb) info program
```

6.7 info || i

- Lokale Variablen im aktuellem Stack-Frame

```
(gdb) info locals
```

- Informationen zu Argumentvariablen des aktuellem Stack-Frames

```
(gdb) info args
```

- Informationen zum aktuellem Stack-Frame, oder Frame im ADDR

```
(gdb) info frame
```

- Status eines spezifischen Watchpoints (oder allen bei keinem Argument)

```
(gdb) info watchpoints
```

- Ausgabe der derzeit bekannten Threads

```
(gdb) info threads
```

7. GDB beenden: quit || q

- Schließen von GDB (auch durch `Strg+D` möglich)

```
(gdb) quit
```

- Terminierung jeglicher GDB-Kommandos über `Strg+C`

8. Pretty Prints

- Abhängig vom benutzten System, die Ausgabe der STL-Container kann mit unter so aussehen:

```
(gdb) p string
$1 = {<std::__1::__basic_string_common<true>> = {<No data fields>},  __r_ =
{<std::__1::__libcpp_compressed_pair_imp<std::__1::basic_string<char, std::__1::char_traits<char>,
std::__1::allocator<char> >::__rep, std::__1::allocator<char>, 2>> = {<std::__1::allocator<char>> = {<No
data fields>},
        __first_ = {{__l = {__cap_ = 6278140392020199446, __size_ = 1684828783,
                __data_ = 0x0}, __s = {{__size_ = 22 '\026', __lx = 22 '\026'},
                __data_ = "Hello World", '\000' <repeats 11 times>}, __r = {
                __words = {6278140392020199446, 1684828783,             0}}}}}, <No data fields>},
static npos = 18446744073709551615}
```

- Dies kann mit einem Python-Skript behoben werden, um diesen Mist ordentlich auszugeben
- https://sourceware.org/gdb/wiki/STLSupport
- Linux: https://gist.github.com/skyscribe/3978082
- Mac: https://github.com/koutheir/libcxx-pretty-printers

http://www.hwk-chemnitz.de/fileadmin/user_upload/ausbildung/Pruefungen/Gesellenpruefungen/Fragezeichen.JPG