B4B33PSY: Počítačové systémy

Lekce 06. Debugging

Jakub Dupák

dupakjak@fel.cvut.cz

27. 10. 2025

Cíle přednášky

Co si z této přednášky odnést?

✅ Praktické dovednosti

- Číst a interpretovat chybové hlášky kompilátoru
- Používat `-Wall -Wextra` a opravovat varování
- Používat automatické nástroje pro kontrolu používání paměti
- Použít debugger ve VS Code pro nalezení chyby
- Vybrat správný nástroj podle typu problému

📚 Připraveni na

- Praktické cvičení s debugováním ve VS Code
- Samostatné řešení problémů v domácích úkolech
- Efektivnější ladění programů v PSY a PRP druhé polovině semestru

Osnova

- Metodologie debugování
- Co je to vlastně bug?
- Druhy bugů (a nástroje na jejich hledání)
- Manuální debugování
- VS Code debugger

Metodologie debugování

Systematický přístup k debugování

1. Reprodukovat	Zajistit, že chybu umíte spolehlivě vyvolat
2. Izolovat	Najít nejmenší případ, který chybu způsobí
3. Lokalizovat	Určit přesné místo v kódu
4. Opravit	Implementovat řešení
5. Ověřit	Zkontrolovat, že oprava funguje

Tip

Rubber duck debugging: Vysvětlete svůj kód gumové kachničce (kolegovi).
Často při vysvětlování sami najdete chybu!

Co je to bug?

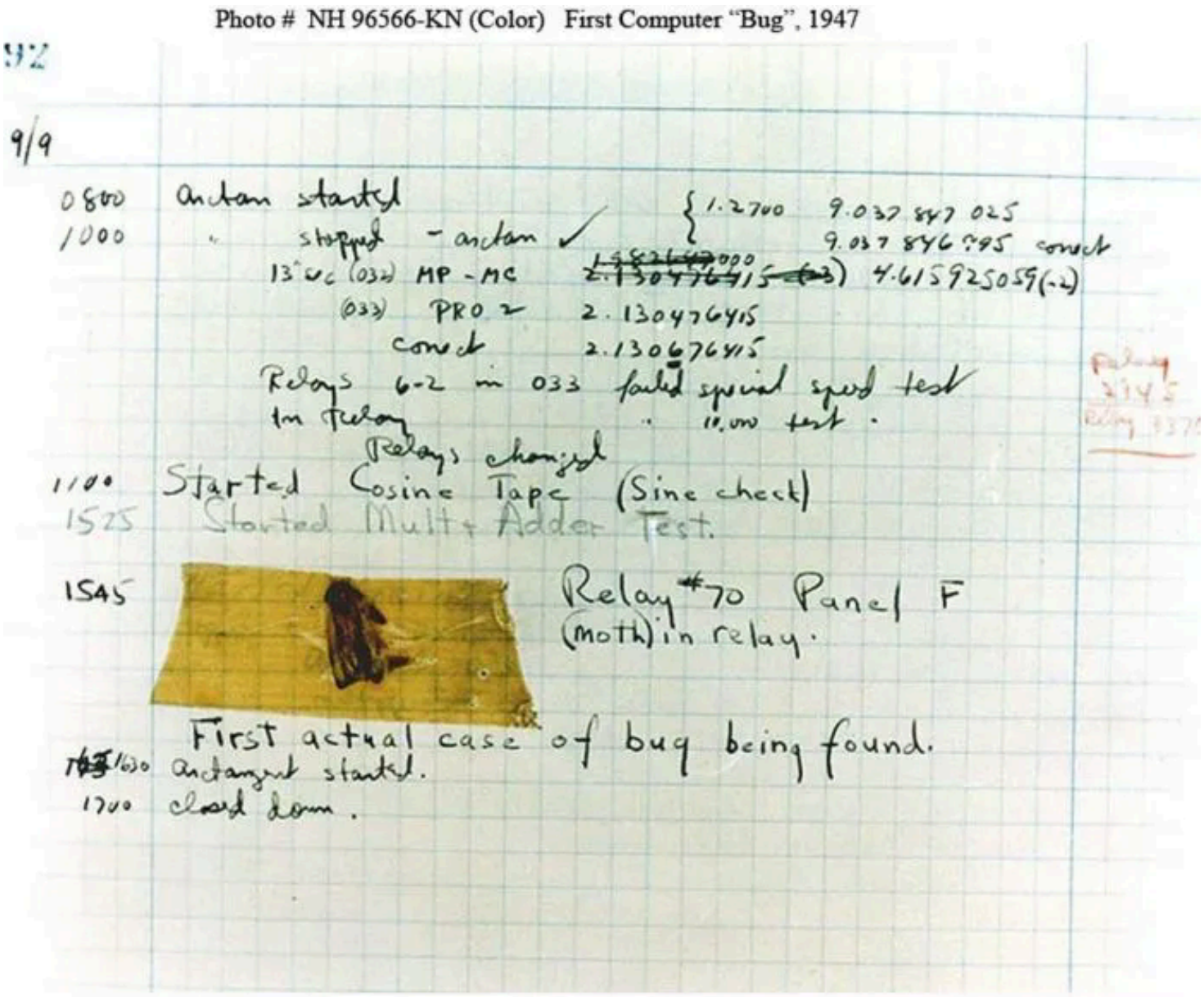


Photo # NH 96566-KN (Color) First Computer "Bug", 1947
9/9
0800 Arctan started
1000 " stopped - arctan ✓
1.2700 9.037 847 025
9.037 846 795 correct
13 u/c (032) MP - MC 4.615925059(-2)
(033) PRO 2 2.13047 6415
correct 2.13067 6415
Relays 6-2 in 033 failed special speed test
In relay . 110.00 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1700 closed down.

Co je to bug?

Bug je rozdíl mezi **návrhem** a **implementací**.

- Příklady:
  - *Program padá (segmentation fault).*
  - *Program (někdy) dává špatné výsledky.*
  - *Program se nikdy nezastaví.*
  - *Program špatně formatuje výstup.*

- Druhy bugů (*neformální*, nebudeme zkoušet):
  - Logický bug
  - Technický bug

Technické bugy

Definice

Porušení daných pravidel programovacího jazyka, nástroje nebo knihovny.

- Charakteristiky:
  - Jasně vymezené - často je lze automaticky detekovat
  - Nejčastěji v `C` špatná práce s pamětí a ukazateli (pointery)
- Příklady:
  - *“Null pointer nesmí být dereferencován.”*
  - *“K paměti uvolněné funkcí `free` nesmí být přistupováno.”*
  - *“Pole musí být přistupováno pouze v rámci jeho velikosti.”*
  - *“Funkce `memcpy` nesmí být použita s překrývajícími se paměťovými oblastmi.”*

Logické bugy

- “Všechny ostatní chyby”
- Charakteristiky:
  - Často chybí jasná specifikace - pak je těžké zjistit, co je vlastně chyba
  - Vyžadují pochopení záměru programu
- Příklady:
  - *“Neošetření okrajových případů (prázdný vstup, nula, záporná čísla).”*
  - *“Chyba v algoritmu (špatná podmínka v cyklu, chybný výpočet).”*
  - *“Nepochopení zadání (program dělá něco jiného, než bylo požadováno).”*
  - *“Chybějící validace vstupů (program spadne na neočekávaný vstup).”*
  - *“Off-by-one error (cyklus běží o jednu iteraci víc nebo míň).”*
- Oblíbené rčení programátorů: *“It’s not a bug, it’s a feature.”*[1]

[1] To není chyba, to je vlastnost/funkcionalita.

Základní debugovací přístupy

- **Čtení kódu a přemýšlení o něm**
- **Výpisy za běhu**
  - *“printf debugging”*
  - Porovnáváme výpisy s naším očekáváním ze čtení kódu.
  - `printf("GOT HERE");`
  - `printf("value=%d", x);`
- **Kontrola za běhu**
  - `if (!(x > 0)) { fprintf(stderr, "x musí být kladné\n"); exit(1); }`

Pokročilejší přístupy pro debugování

- Čtení kódu → **Nástroje na analýzu kódu**
  - technické bugy
  - nástroje: *kompilátor, statická analýza*
- Výpisy za běhu → **Zastavení programů a inspekce stavu**
  - Sledujeme program krok po kroku.
  - logické i technické bugy
  - nástroje: *debugger*
- Kontrola za běhu → `assert`
  - logické i technické bugy
  - `assert(x > 0);`
    - Výraz který musí platit.
    - Vygeneruje výpis obsahující *výraz* a místo v kódu.
  - Pro technické bugy je často dokážeme **generovat automaticky**.
    - Kontrola práce s pamětí, nedefinované chování, vlákna…
  - nástroje: *Valgrind, Memory Sanitizer…*

Pokročilejší přístupy pro debugování

- Varování kompilátoru
- Assert
- Valgrind & Address Sanitizer
- VS Code debugger

Varování kompilátoru

Varování kompilátoru

```c
int factorial(int n) {
    int result;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

Command

```
clang -Wno-everything factorial.c
```

- Jaký je výsledek této funkce pro `n = 5`?

Varování kompilátoru

```c
int factorial(int n) {
    int result;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

Command

```
clang -Wno-everything factorial.c
```

- Jaký je výsledek této funkce pro `n = 5`?
- Program se přeloží a spustí. **Ale výsledek je náhodný**!

Varování kompilátoru

```
int factorial(int n) {
    int result;
    for (int i = 1; i <= n; i++) { ... }
    return result;
}
```

Běh 1:

Výstup programu

```
result = 0
```

Běh 2:

Výstup programu

```
result = 1438953472
```

- Proměnná `result` obsahuje **náhodnou** předcházející hodnotu z paměti.
  - Hodnota závisí na ostatních funkcích, optimalizacích kompilátoru, atd.
- Proměnná může být také *“podmíněně inicializovaná”*, pokud jenom některé větve programu inicializují hodnotu.

Varování kompilátoru

Command

```
clang -Wall factorial.c
```

Výstup programu

```
factorial.c:3:5: warning: variable 'result' is uninitialized when
used here [-Wuninitialized]
    3 |          result = result * i;
      |          ^~~~~~
factorial.c:2:16: note: initialize the variable 'result' to silence
this warning
    2 |     int result;
      |                ^
      |                 = 0
```

Kompilátor našel problém! A dokonce navrhuje řešení.

Varování kompilátoru

Toto je **záludná chyba**, která je velmi těžké najít!

- Program může často běžet “správně”, ale občas spadne
- Chování závisí na náhodném stavu paměti
- Program se může chovat jinak v debuggeru než při běžném spuštění

Varování kompilátoru

Kompilátor je první obranná linie proti bugům.

Nikdy neignorujte varování, pokud nechápete, co znamenají!

Doporučené flagy kompilátoru

Vždy používejte minimálně:

```
-Wall -Wextra
```

- `-Wall` = základní varování
- `-Wextra` = další užitečná varování

Command

```
clang -Wall -Wextra -o program program.c
```

Co `-Wall` a `-Wextra` zachytí?

🔴 Kritické - způsobují pády/náhodné chování

- `-Wuninitialized`: neinicializované proměnné
- `-Wmaybe-uninitialized`: podmíněně neinicializované proměnné
- `-Wreturn-type`: chybějící return v ne-void funkci
- `-Wformat`: špatné formátovací řetězce v `printf/scanf`
- `-Wimplicit-function-declaration`: volání funkce bez deklarace/`#include`

🟡 Důležité - způsobují špatné výsledky

- `-Wparentheses`: záměna = a == v podmínkách
- `-Warray-bounds`: přístup mimo pole (pokud je staticky detekovatelný)
- `-Wsign-compare`: porovnání signed/unsigned (může způsobit neočekávané chování)
- `-Wempty-body`: středník za `if/while/for`

Čtení chybových hlášek

Anatomie chybové hlášky

```
program.c:15:10: error: use of undeclared identifier 'x'
    ^       ^    ^       ^
    |       |    |       +-- Popis problému
    |       |    +-- Typ (error/warning/note)
    |       +-- Sloupec
    +-- Soubor a řádek
```

Čtení chybových hlášek

Anatomie chybové hlášky

```
program.c:15:10: error: use of undeclared identifier 'x'
    ^       ^    ^       ^
    |       |    |       +-- Popis problému
    |       |    +-- Typ (error/warning/note)
    |       +-- Sloupec
    +-- Soubor a řádek
```

- **První chybu opravte první!**
- Další chyby mohou být jen důsledkem té první.

Čtení chybových hlášek

Anatomie chybové hlášky

```
program.c:15:10: error: use of undeclared identifier 'x'
    ^       ^    ^       ^
    |       |    |       +-- Popis problému
    |       |    +-- Typ (error/warning/note)
    |       +-- Sloupec
    +-- Soubor a řádek
```

- **První chybu opravte první!**
- Další chyby mohou být jen důsledkem té první.

Nejčastější runtime chyby

- `Segmentation fault` – přístup do neplatné paměti
- `Floating point exception` – dělení nulou
- `Stack overflow` – příliš hluboká rekurze

Chybějící return

Funkce, která má vracet hodnotu, ale nevrací ji ve všech případech.

```
int calculate_grade(int points) {
    if (points >= 90) return 1;
    else if (points >= 75) return 2;
    else if (points >= 60) return 3;
    // Co když points < 60?
}  // warning: non-void function does not return a value
```

Chybějící return

Funkce, která má vracet hodnotu, ale nevrací ji ve všech případech.

```
int calculate_grade(int points) {
    if (points >= 90) return 1;
    else if (points >= 75) return 2;
    else if (points >= 60) return 3;
    // Co když points < 60?
}  // warning: non-void function does not return a value
```

Správně:

```
int calculate_grade(int points) {
    if (points >= 90) return 1;
    else if (points >= 75) return 2;
    else if (points >= 60) return 3;
    else return 4;  // Vždy musí být pokryty všechny cesty!
}
```

Assignment vs. Comparison

```c
int main() {
    int score = 85;
    if (score = 100) { printf("Perfektní skóre!\n"); }
    printf("Skóre je nyní: %d\n", score);  // Vždy 100!
    return 0;
}
```

Výstup programu

```
warning: using the result of an assignment as a condition without
parentheses [-Wparentheses]
   10 |     if (score = 100) {
      |         ~~~~~~^~~~~
note: use '==' to turn this assignment into an equality comparison
   10 |     if (score = 100) {
      |               ^ ==
```

Assignment vs. Comparison

```c
int main() {
    int score = 85;
    if (score = 100) { printf("Perfektní skóre!\n"); }
    printf("Skóre je nyní: %d\n", score);  // Vždy 100!
    return 0;
}
```

Výstup programu

```
warning: using the result of an assignment as a condition without
parentheses [-Wparentheses]
   10 |     if (score = 100) {
      |         ~~~~~~^~~~~
note: use '==' to turn this assignment into an equality comparison
   10 |     if (score = 100) {
      |               ^ ==
```

Správně: `if (score == 100)`

Prázdné tělo cyklu

```c
int main() {
    int i = 0;

    // Středník za while je častá chyba!
    while (i < 10);  // <-- pozor na středník!
    {
        printf("i = %d\n", i);
        i++;
    }
    // Nekonečná smyčka!
    return 0;
}
```

Výstup programu

```
warning: while loop has empty body [-Wempty-body]
   11 |     while (i < 10);
      |                    ^
note: put the semicolon on a separate line to silence this warning
```

String comparison

V jazyce C operátor `==` porovnává **adresy**, ne obsah řetězců.

```
void f(char* a) {
    if (a == "hello") { /* ... */ }
}
```

Výstup programu

```
warning: result of comparison against a string literal is
unspecified
(use an explicit string comparison function instead)
[-Wstring-compare]
    8 |     if (a == "hello") {
      |            ^  ~~~~~~~
```

String comparison

V jazyce C operátor `==` porovnává **adresy**, ne obsah řetězců.

```
void f(char* a) {
    if (a == "hello") { /* ... */ }
}
```

Výstup programu

```
warning: result of comparison against a string literal is
unspecified
(use an explicit string comparison function instead)
[-Wstring-compare]
    8 |     if (a == "hello") {
      |            ^  ~~~~~~~
```

Správně:

```
if (strcmp(a, "hello") == 0) { /* ... */ }
```

Špatně formátovaný řetězec

```c
int main() {
    int x = 42;
    printf("%s\n", x);  // %s očekává string, ne int!
    return 0;
}
```

Výstup programu

```
warning: format specifies type 'char *' but the argument has type
'int' [-Wformat]
    3 |     printf("%s\n", x);
      |             ~~     ^
      |             %d
```

Špatně formátovaný řetězec

```c
int main() {
    int x = 42;
    printf("%s\n", x);  // %s očekává string, ne int!
    return 0;
}
```

Výstup programu

```
warning: format specifies type 'char *' but the argument has type
'int' [-Wformat]
    3 |     printf("%s\n", x);
      |             ~~     ^
      |             %d
```

Správně: `printf("%d\n", x);`

Chybějící `#include`

```c
int main() {
    int len = strlen("hello");  // Zapomněl #include <string.h>
    printf("Length: %d\n", len);
    return 0;
}
```

Výstup programu

```
error: call to undeclared library function 'strlen' with type
'unsigned long (const char *)'; ISO C99 and later do not support
implicit function declarations [-Wimplicit-function-declaration]
    8 |     int len = strlen("hello");
      |               ^
note: include the header <string.h> or explicitly provide a
declaration for 'strlen'
```

Správně:

```c
#include <string.h>
#include <stdio.h>
```

Dynamická analýza

Dynamická analýza

- Analyzovat program bez spuštění je těžké.
- Většina bugů hledá za běhu.

- Najde jen chyby, které se opravdu stanou pro daný vstup.
  - Jak vybrat dostatek vhodných vstupů se dozvíte na příští přednášce o testování.
- Automatické nástroje pro technické bugy.
  - Paměť: **Valgrind, Address & Memory Sanitizer**
  - Nedefinované chování: **Undefined Behavior Sanitizer**
    - Například přetečení.
    - Pozor: přetečení unsigned int je v C definované chování!
  - Vlákna: **Thread Sanitizer**

Assert: Kontrola předpokladů

Co je `assert`?

Makro z `<assert.h>` ověřující, že platí daný vnitřní předpoklad programu. Při selhání: vypíše výraz, soubor a řádek a ukončí program (rychlá detekce chyby).

Proč ho používat?

- Dokumentuje invarianty a předpoklady přímo v kódu
- Zastaví běh hned u zdroje problému (ne až u špatného výsledku dál)
- Zrychluje debugování – jasná zpráva místo tichého rozbití stavu
- Lze vypnout v produkci pomocí `-DNDEBUG`

Assert: Kontrola předpokladů

```c
#include <assert.h>
int factorial(int n) {
    assert(n >= 0); // předpoklad: vstup musí být nezáporný
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Výstup programu

```
Assertion failed: (n >= 0), function factorial, file main.c, line 3
```

Kdy NEassertovat?

- Validace uživatelského vstupu (tam má být ošetření, ne pád)
- Zotavitelné chyby (chyba I/O, nedostupný soubor, síť)

Assert: Kontrola předpokladů

Command

```
clang -DNDEBUG test.c  # assert se neprovede
```

- Asserty se dají vypnout při kompilaci pomocí `-DNDEBUG` pro produkční kód.

Valgrind a Address Sanitizer

- Use after free (dangling pointer dereference)
- Double free
- Buffer overflow
- Use after scope
- Use of uninitialized memory
- Memory leaks

Valgrind a Address Sanitizer

Valgrind Memcheck

- Nevyžaduje překompilování programu
- Provádí překlad a vložení kontroly za běhu (JIT binary translation)
- Sleduje program s velkou přesností
- Nenajde chyby na stacku
- **Výrazné zpomalení** při běhu
  - typicky 10x-300x

Command

```
valgrind --leak-check=full ./
program
```

Address & Memory Sanitizer

- Integrovaný do překladače
  - Přidává kontroly při překladu
- Rychlejší (1.5x-2.5x zpomalení)
  - Použitelný i pro grafické aplikace
- Detekuje chyby na stacku
- Sleduje paměť s menší přesností

Command

```
clang -fsanitize=address \
      -fno-omit-frame-pointer \
      -g -o program program.c

./program
```

Ukázka: Buffer Overflow

```c
int main() {
    int arr[3] = {1, 2, 3};
    arr[3] = 42;  // Přístup mimo pole!
    return 0;
}
```

Výstup programu

```
=================================================================
ERROR: AddressSanitizer: stack-buffer-overflow on address
0x7ffd1234567c at pc 0x4a8b9c bp 0x7ffd12345630 sp 0x7ffd12345628
WRITE of size 4 at 0x7ffd1234567c thread T0
    #0 0x4a8b9b in main 08_buffer_overflow.c:7:12

This frame has 1 object(s):
  [32, 44) 'arr' (line 6) <== Memory access at offset 44 overflows
  this variable
```

Ukázka: Use After Free

```c
int main() {
    int *ptr = malloc(sizeof(int));
    *ptr = 42;
    free(ptr);
    *ptr = 100;  // Použití uvolněné paměti!
    return 0;
}
```

Ukázka: Use After Free

Výstup programu

```
=================================================================
ERROR: AddressSanitizer: heap-use-after-free on address
0x602000000010 at pc 0x4a8c1d bp 0x7ffd12345630 sp 0x7ffd12345628
WRITE of size 4 at 0x602000000010 thread T0
    #0 0x4a8c1c in main 09_use_after_free.c:11:10

freed by thread T0 here:
    #0 0x7f8b1234abcd in free
    #1 0x4a8bef in main 09_use_after_free.c:10:5

previously allocated by thread T0 here:
    #0 0x7f8b1234abcd in malloc
    #1 0x4a8b9b in main 09_use_after_free.c:8:16
```

Ukázka: Memory Leak

```c
int main() {
    for (int i = 0; i < 100; i++) {
        int *ptr = malloc(1024);
        // Zapomněli jsme na free(ptr)!
    }
    return 0;
}
```

Výstup programu

```
=================================================================
ERROR: LeakSanitizer: detected memory leaks

Direct leak of 102400 byte(s) in 100 object(s) allocated from:
    #0 0x7f8b1234abcd in malloc
    #1 0x4a8b9b in main 10_memory_leak.c:9:20

SUMMARY: AddressSanitizer: 102400 byte(s) leaked in 100
allocation(s).
```

Manuální debugování

`printf` debugging

```
int fib(int n) {
    if (n == 0) {
        return n;
    };
    return fib(n - 1) + fib(n - 2);
}
```

`printf` debugging

```
int fib(int n) {
    if (n == 0) {
        return n;
    };
    return fib(n - 1) + fib(n - 2);
}
```

Výstup programu

```
Segmentation fault
```

`printf` debugging

Step 1: Valgrind

Výstup programu

```
==118776==   Stack overflow in thread #1:
             can't grow stack to 0x1ffe801000
==118776==
==118776==   Process terminating with default action
             of signal 11 (SIGSEGV)
==118776==   Access not within mapped region at address
             0x1FFE801FF8

==118776==   Stack overflow in thread #1: can't grow stack to
             0x1ffe801000

==118776==   at 0x10913D: fib (printf-debug.c:3)
```

`printf` debugging

Step 2: Přidání jednoduchých výpisů

```
int fib(int n) {
    printf("running fib"); // <---
    if (n == 0) {
        return n;
    };
    return fib(n - 1) + fib(n - 2);
}
```

Výstup programu

```
running fib
running fib
running fib
(174411x)
running fib
running fib
Segmentation fault
```

`printf` debugging

Step 3: Kontrola “base case” pomocí `printf`

```
int fib(int n) {
    if (n == 0) {
        printf("got here!\n"); // <---
        return n;
    };
    return fib(n - 1) + fib(n - 2);
}
```

Výstup programu

```
got here!
Segmentation fault
```

`printf` debugging

Step 4: Výpis argumentů funkcí

```
int fib(int n) {
    printf("fib(%d)\n", n); // <---
    if (n == 0) {
        return n;
    };
    return fib(n - 1) + fib(n - 2);
}
```

Výstup programu

```
fib(3)
fib(2)
fib(1)
fib(0)
fib(-1)
fib(-2)
...
```

`printf` debugging

Step 5: Výpis dosažení base case

```c
int fib(int n) {
    printf("entering fib(%d)\n", n);
    if (n == 0) { printf("exiting fib(0)\n"); return n; }
    int left = fib(n - 1);
    int right = fib(n - 2);
    return left + right;
}
```

Výstup programu

```
entering fib(3)
 entering fib(2)
  entering fib(1)
   entering fib(0)
   exiting fib(0)
  entering fib(-1)
```

Závěr

Tip

Best practices pro printf debugging:

- Používejte `stderr` pro debug výpisy: `fprintf(stderr, "DEBUG: x=%d\n", x);`
- Přidejte `__FILE__` a `__LINE__` pro lokaci: `printf("[%s:%d] x=%d\n", __FILE__, __LINE__, x);`
- Vždy flushujte buffer: `fflush(stderr);` (při pádu se nemusí vypsat)

- Museli jsme měnit kód a pokaždé ho překompilovat.
- Není lepší způsob?

Debugger

- Nástroj na krokování programu.

Debugger engines

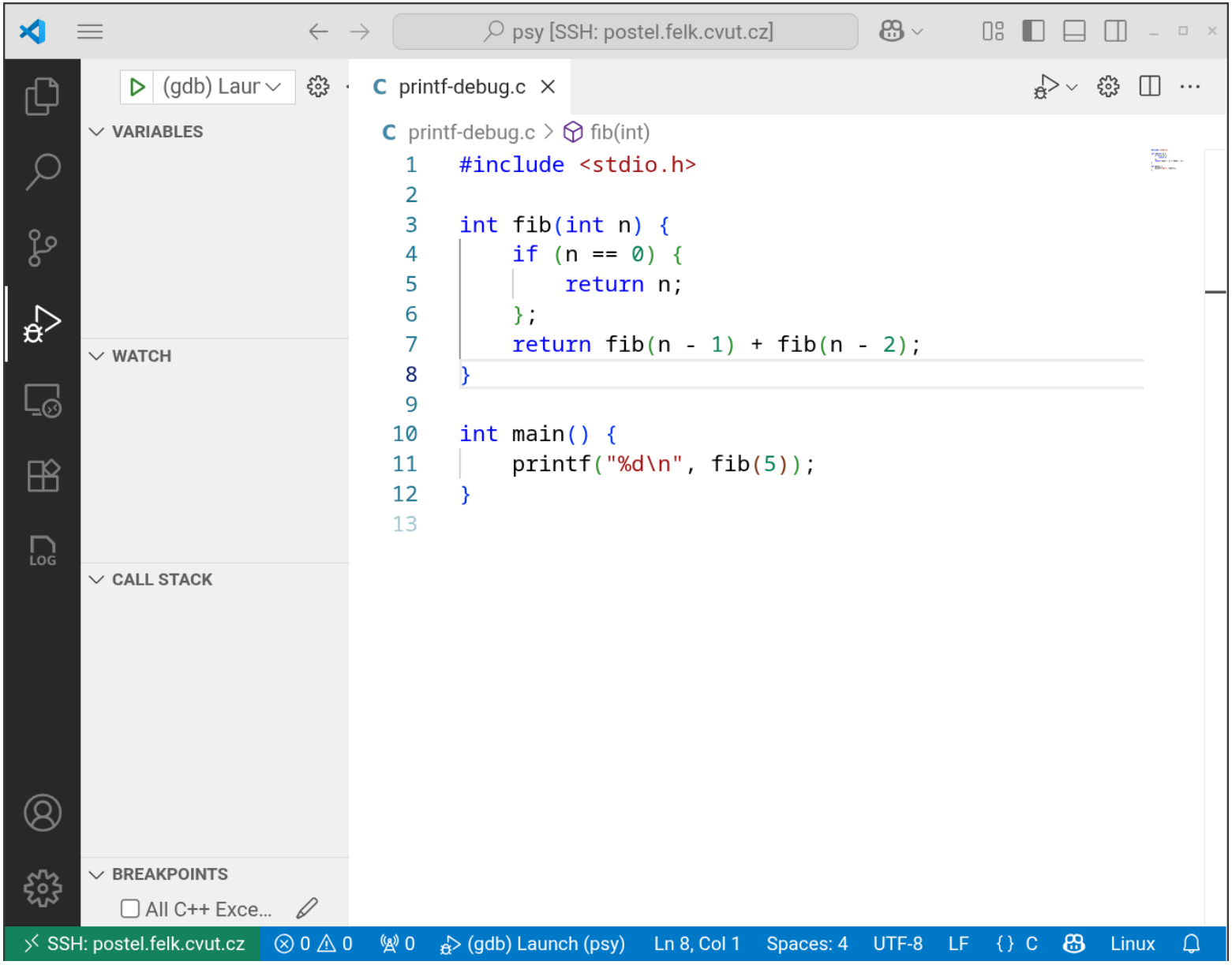
- **LLDB** (LLVM)
- **GDB** (GNU)
- **WinDbg** (Windows)
- **Visual Studio Debugger** (Windows)

Grafické rozhraní

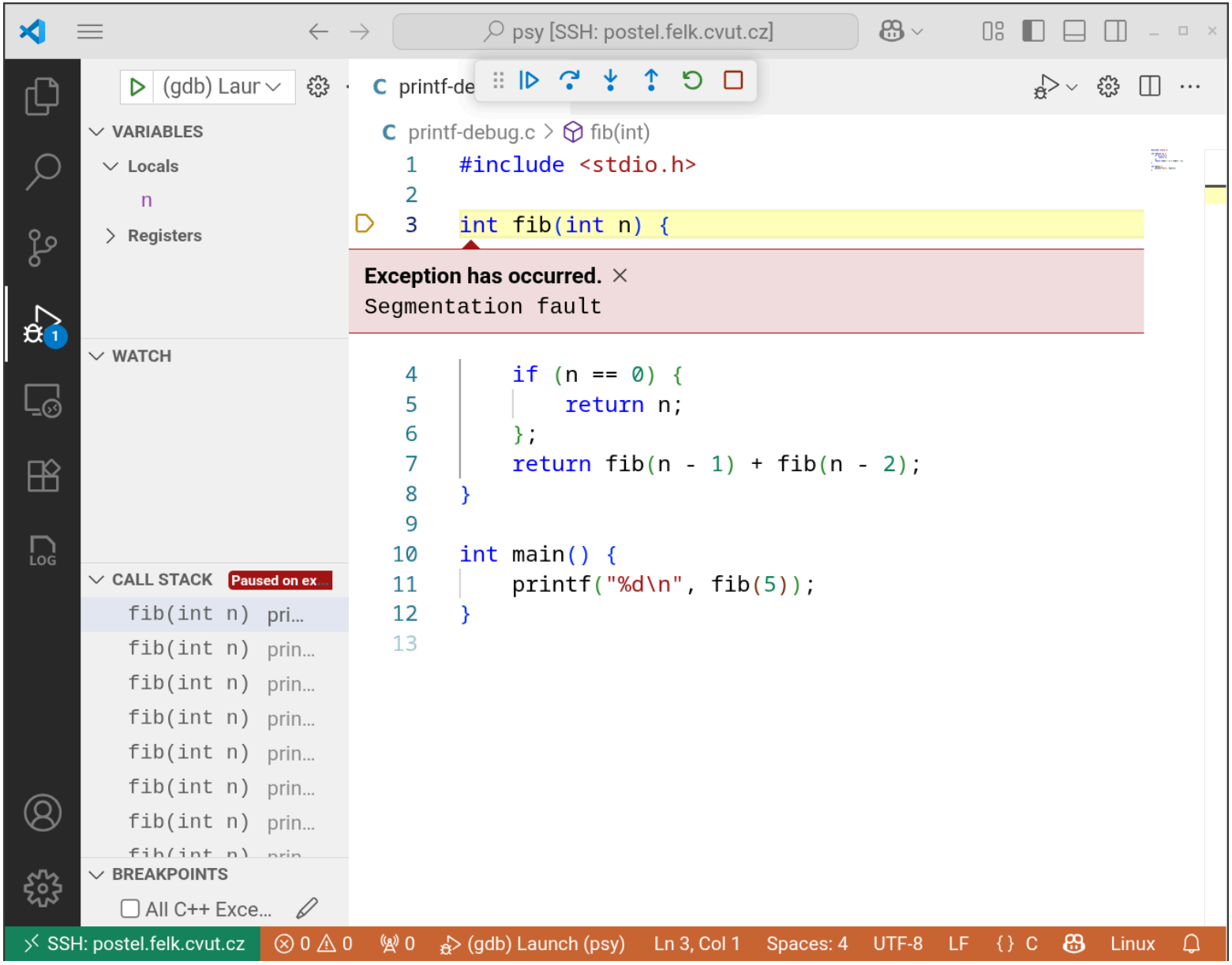
- **VS Code Debugger**
- **CLion Debugger**
- **GDB TUI**, **DDD**, **gdbgui**…
- WinDbg, Visual Studio…

VS Code Debugger (gdb)

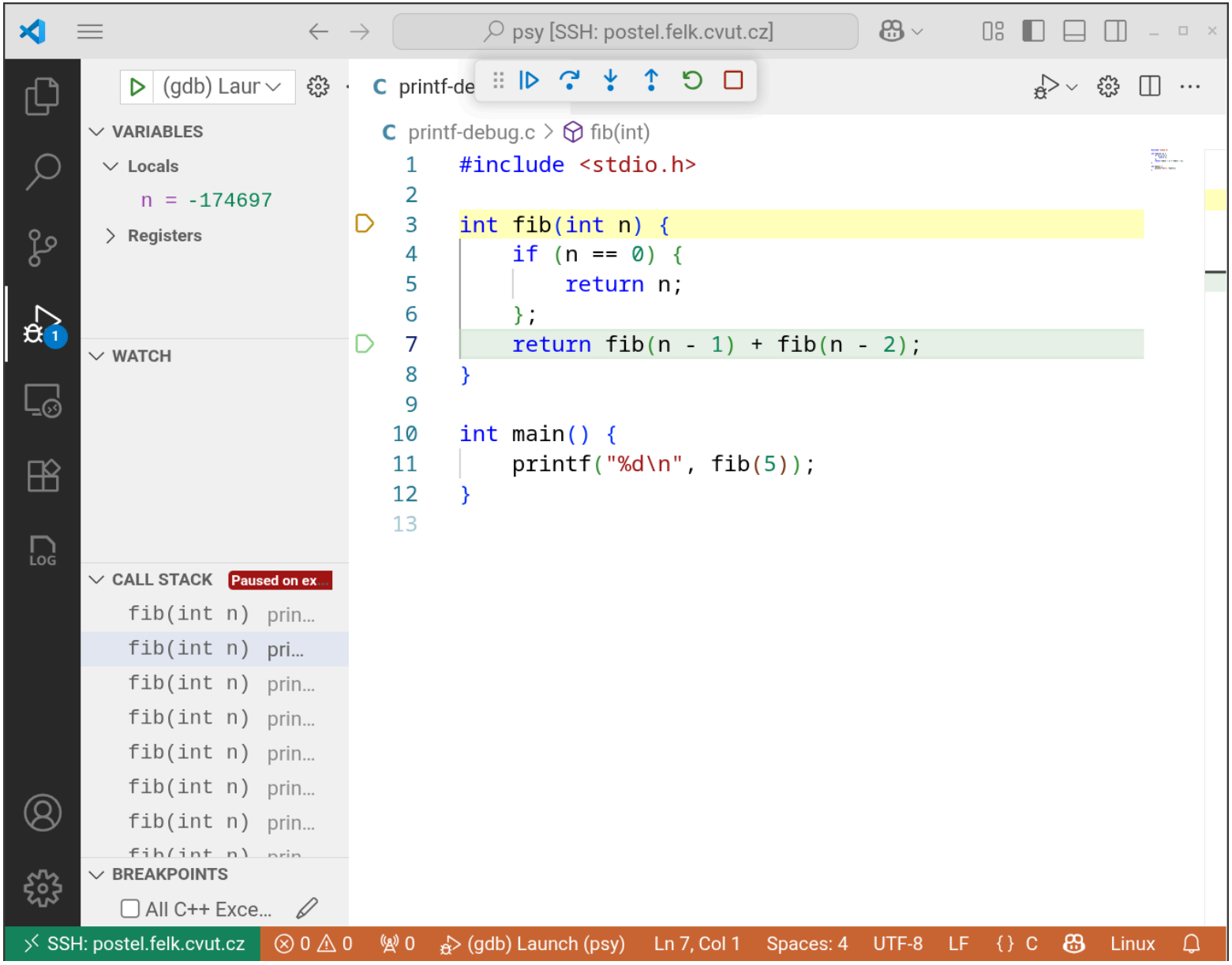
Demo: Hledání chyby v `fib`



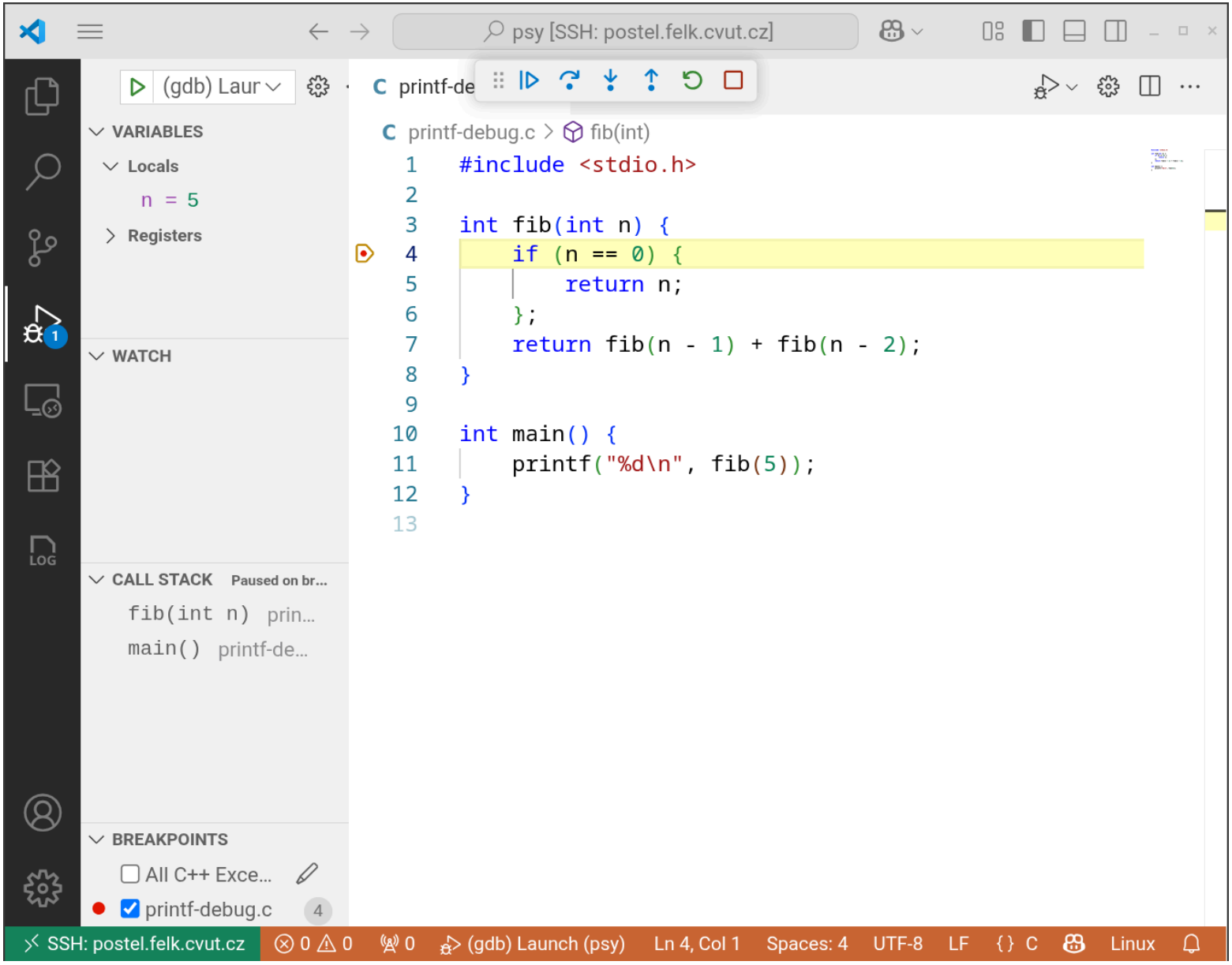
Demo: Hledání chyby v `fib`



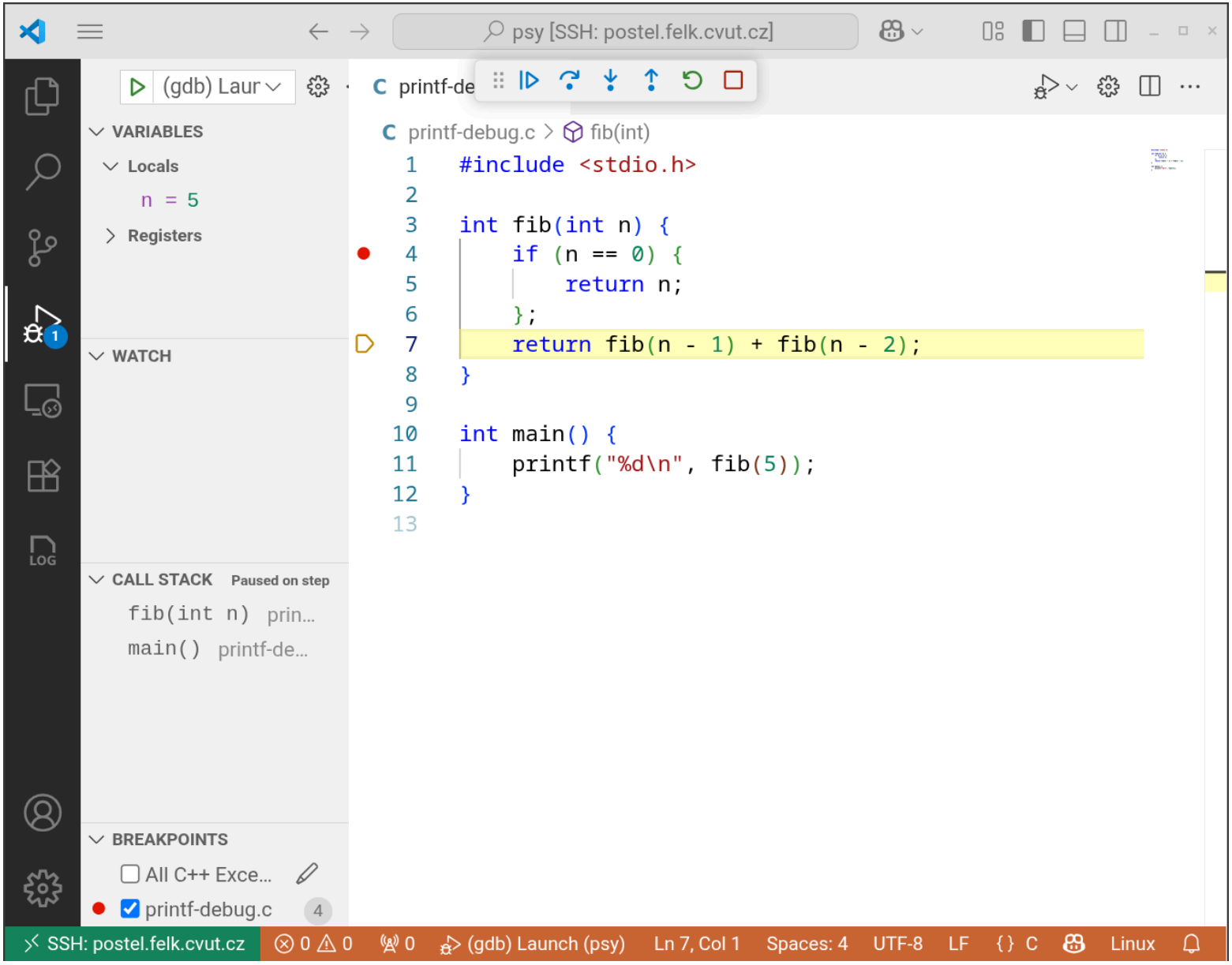
Demo: Hledání chyby v `fib`



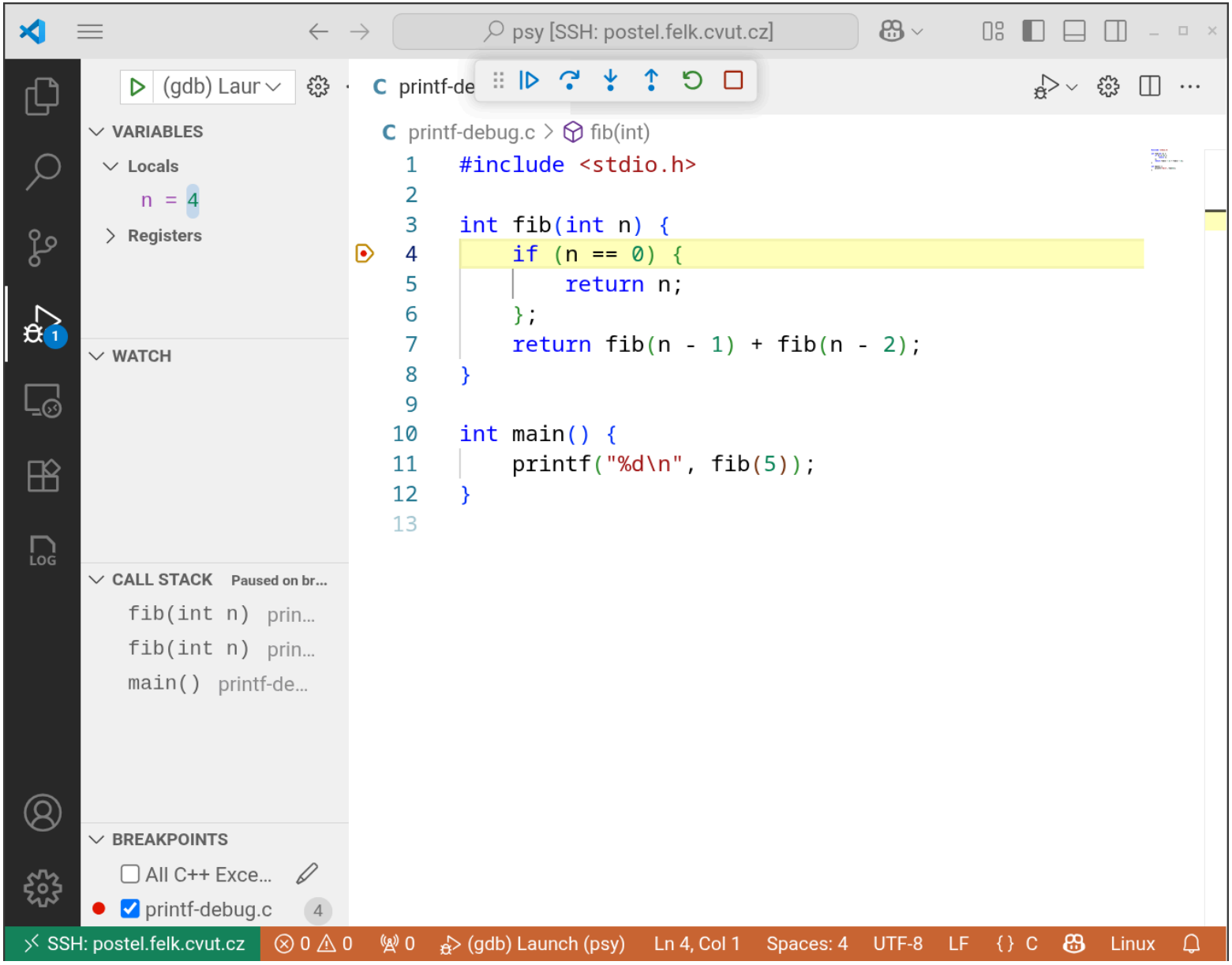
Demo: Hledání chyby v `fib`



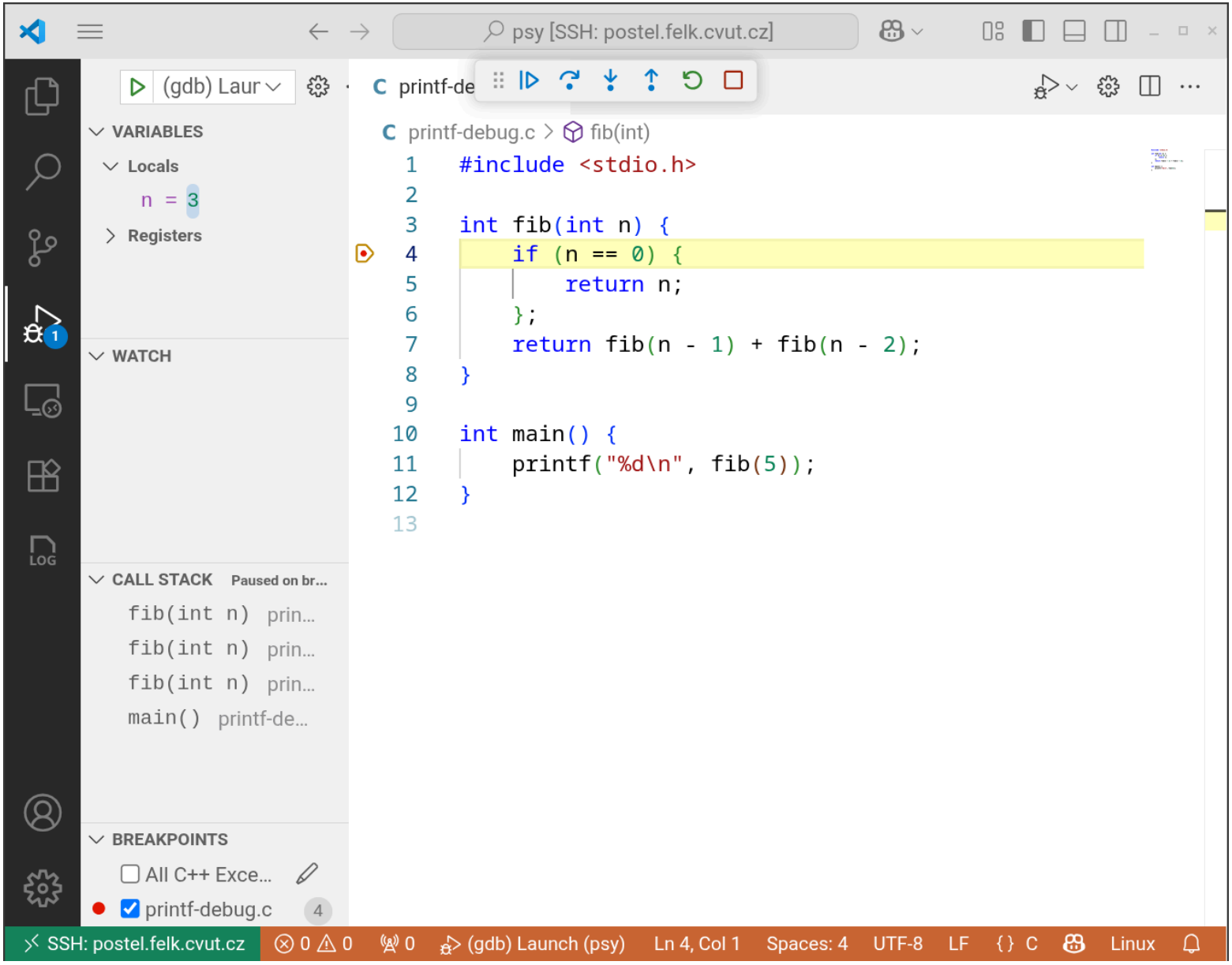
Demo: Hledání chyby v `fib`



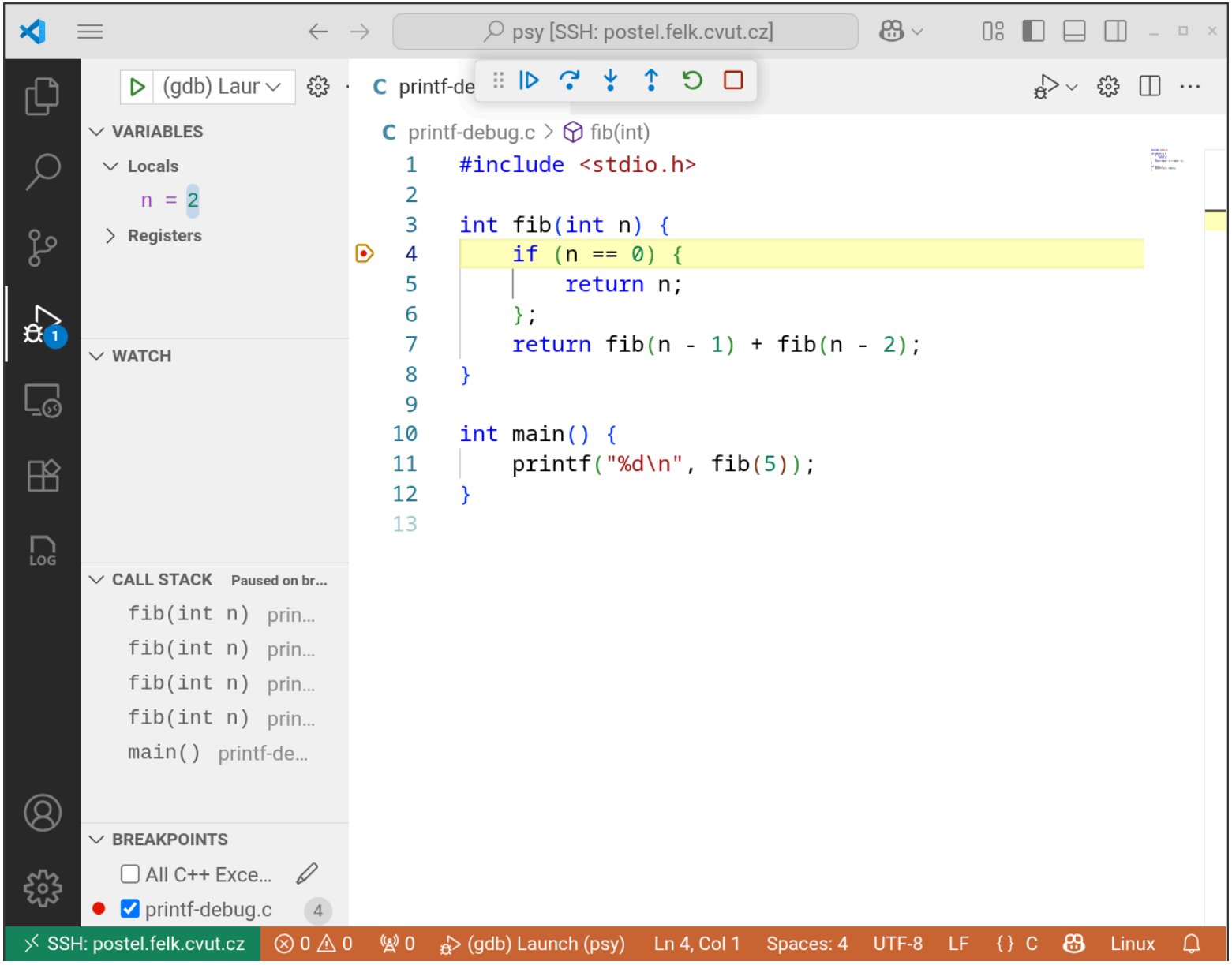
Demo: Hledání chyby v `fib`



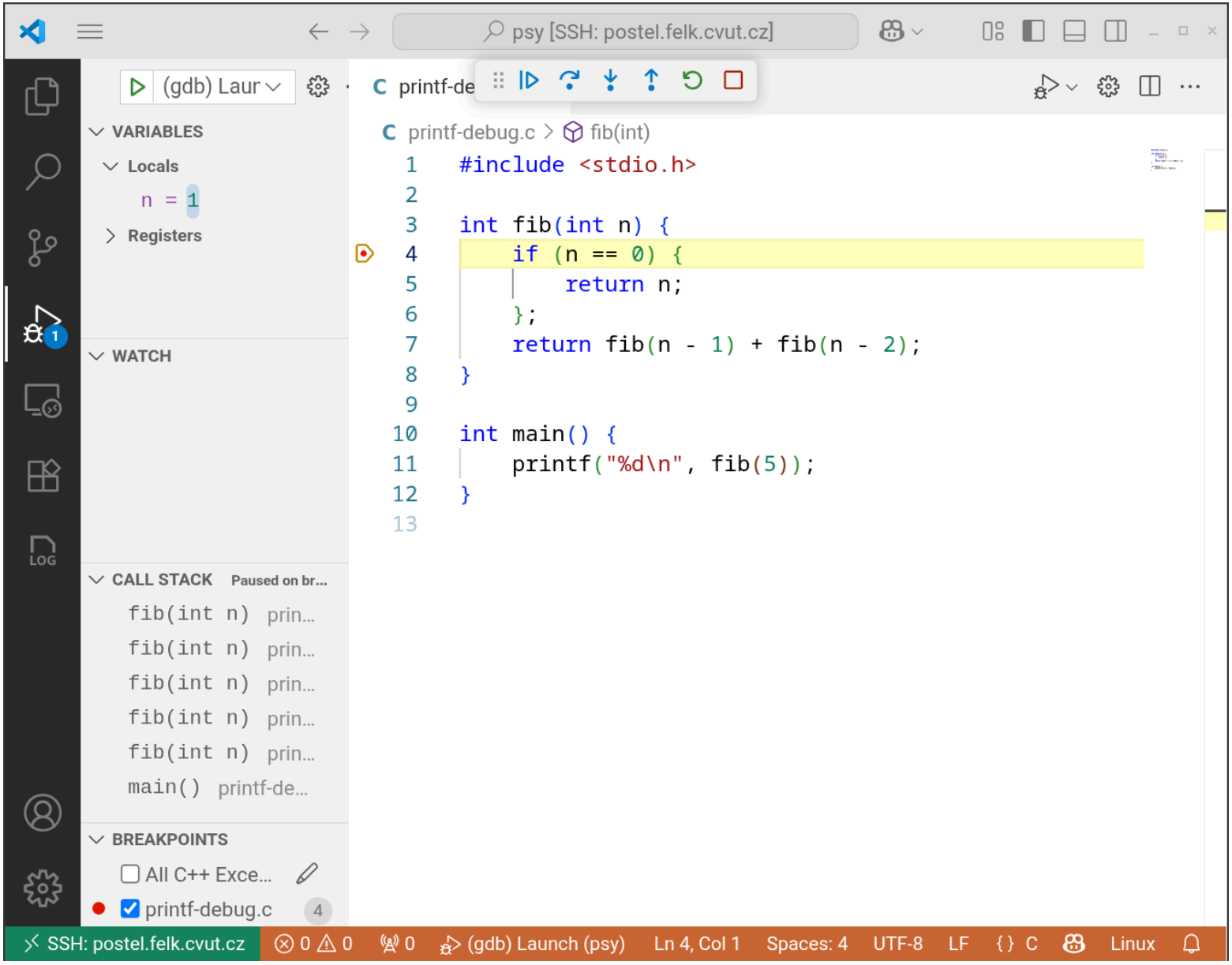
Demo: Hledání chyby v `fib`



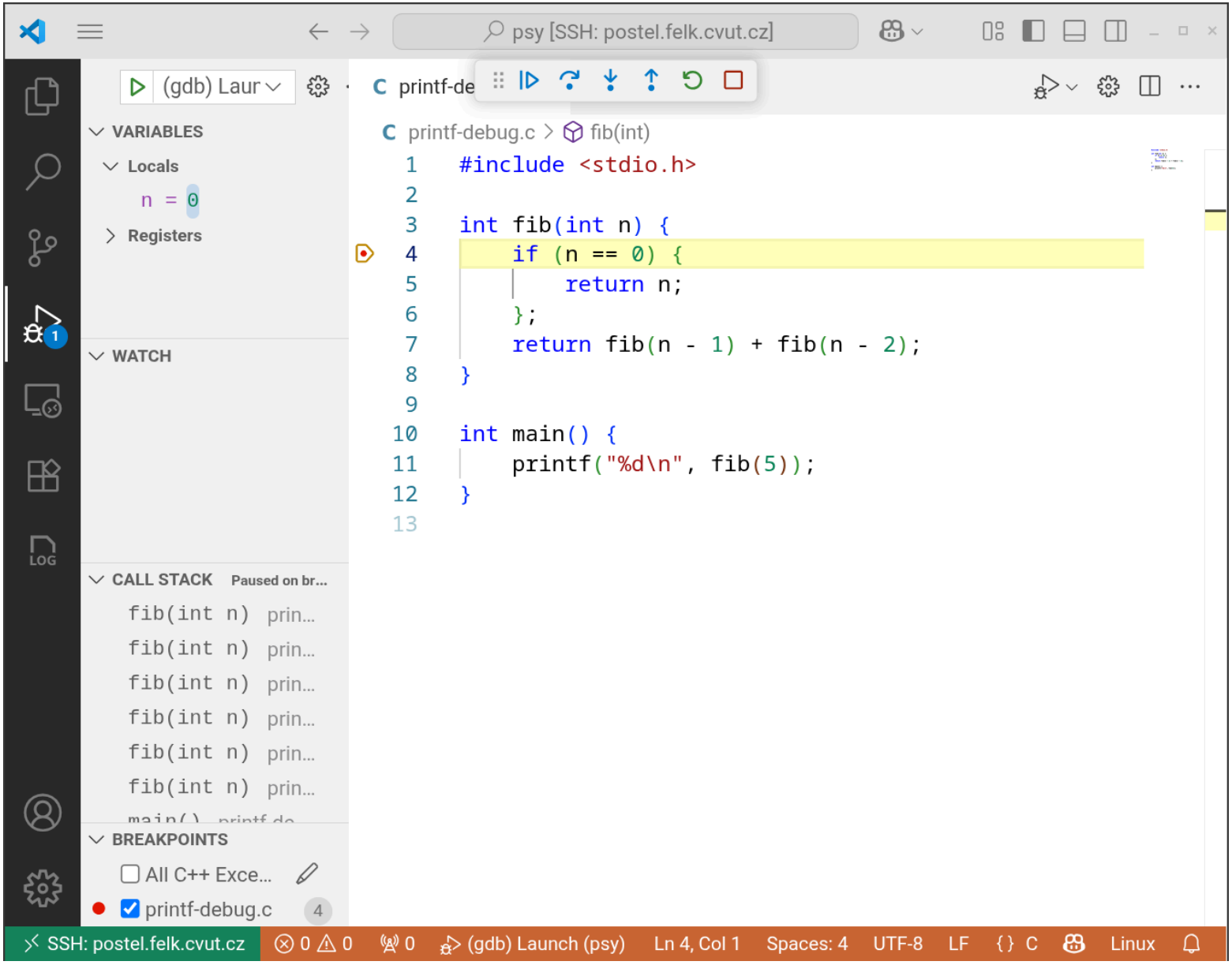
Demo: Hledání chyby v `fib`



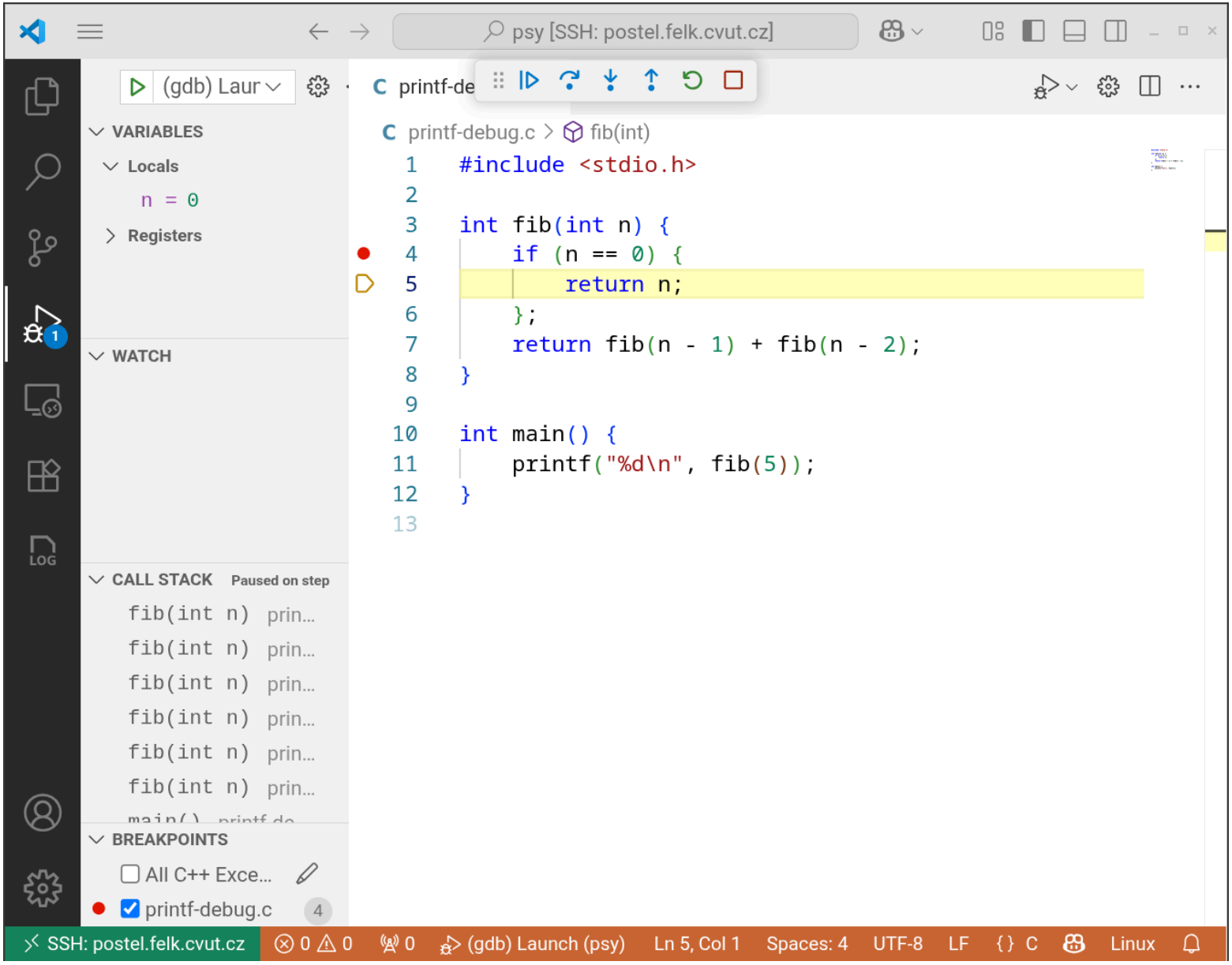
Demo: Hledání chyby v `fib`



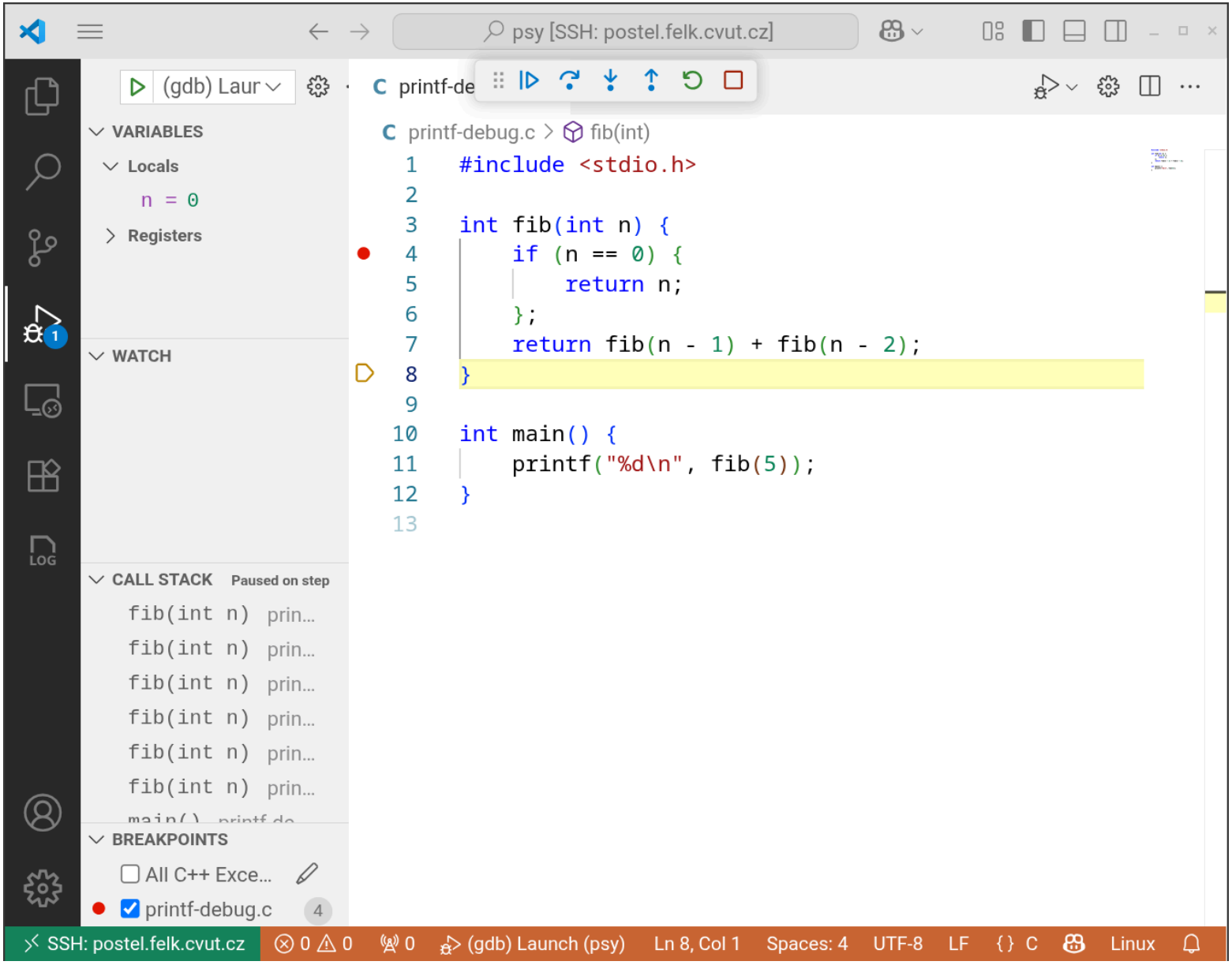
Demo: Hledání chyby v `fib`



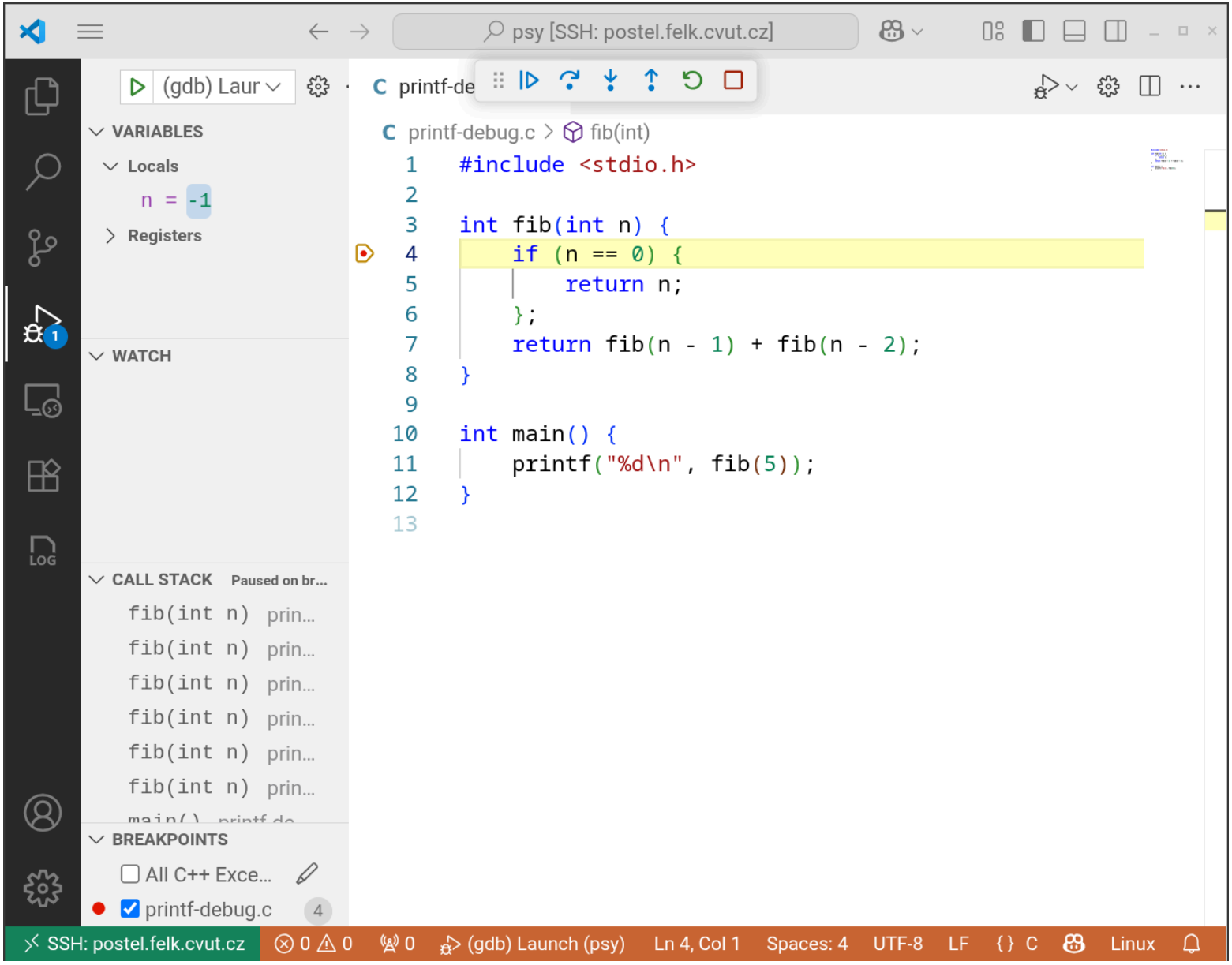
Demo: Hledání chyby v `fib`



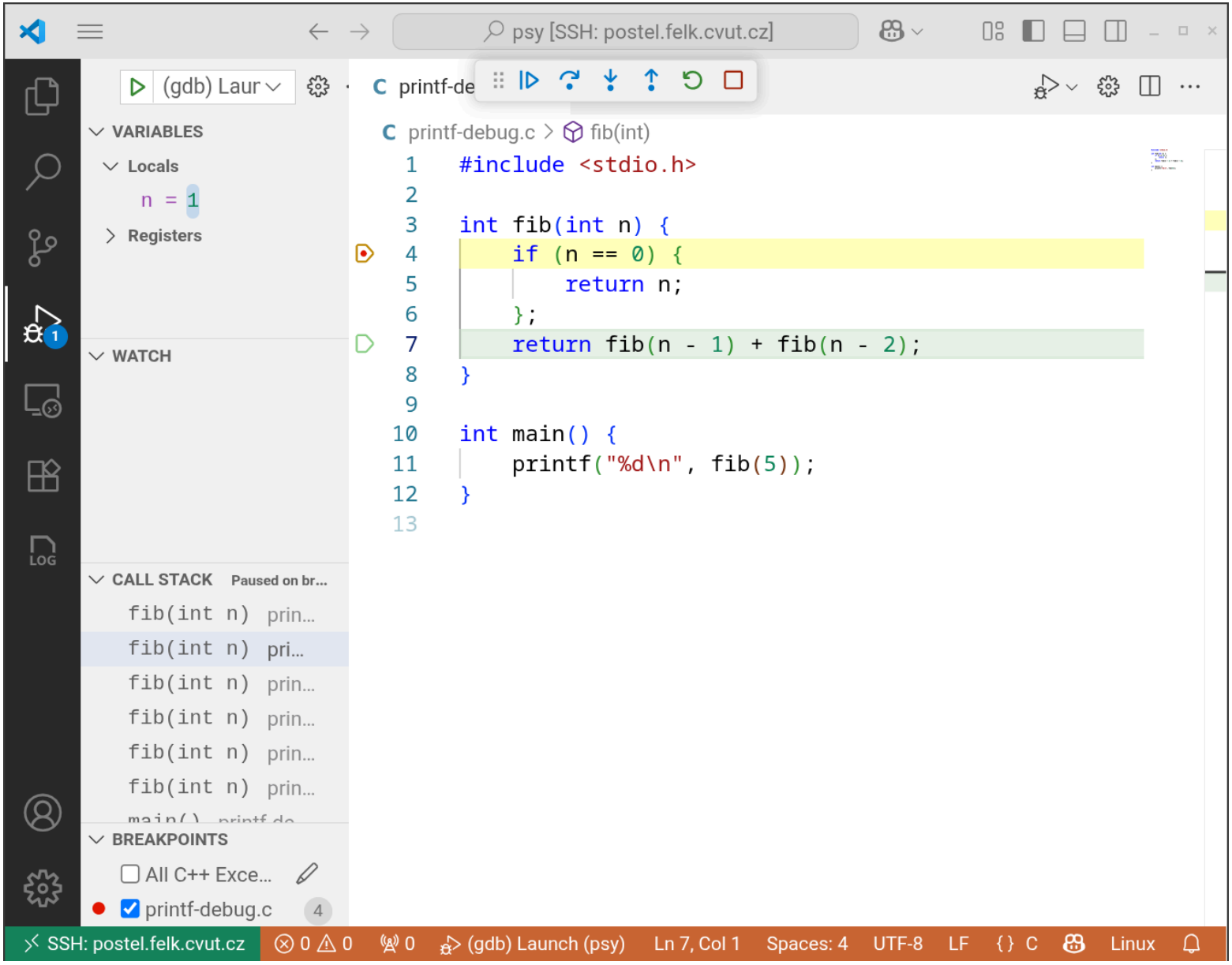
Demo: Hledání chyby v `fib`



Demo: Hledání chyby v `fib`



Demo: Hledání chyby v `fib`



Demo: Hledání chyby v `fib`

- Výhody:
  - Nemuseli jsme měnit kód.
  - Vidíme všechny proměnné.
  - Vidíme, jak jsme se na dané místo v kódu dostali.

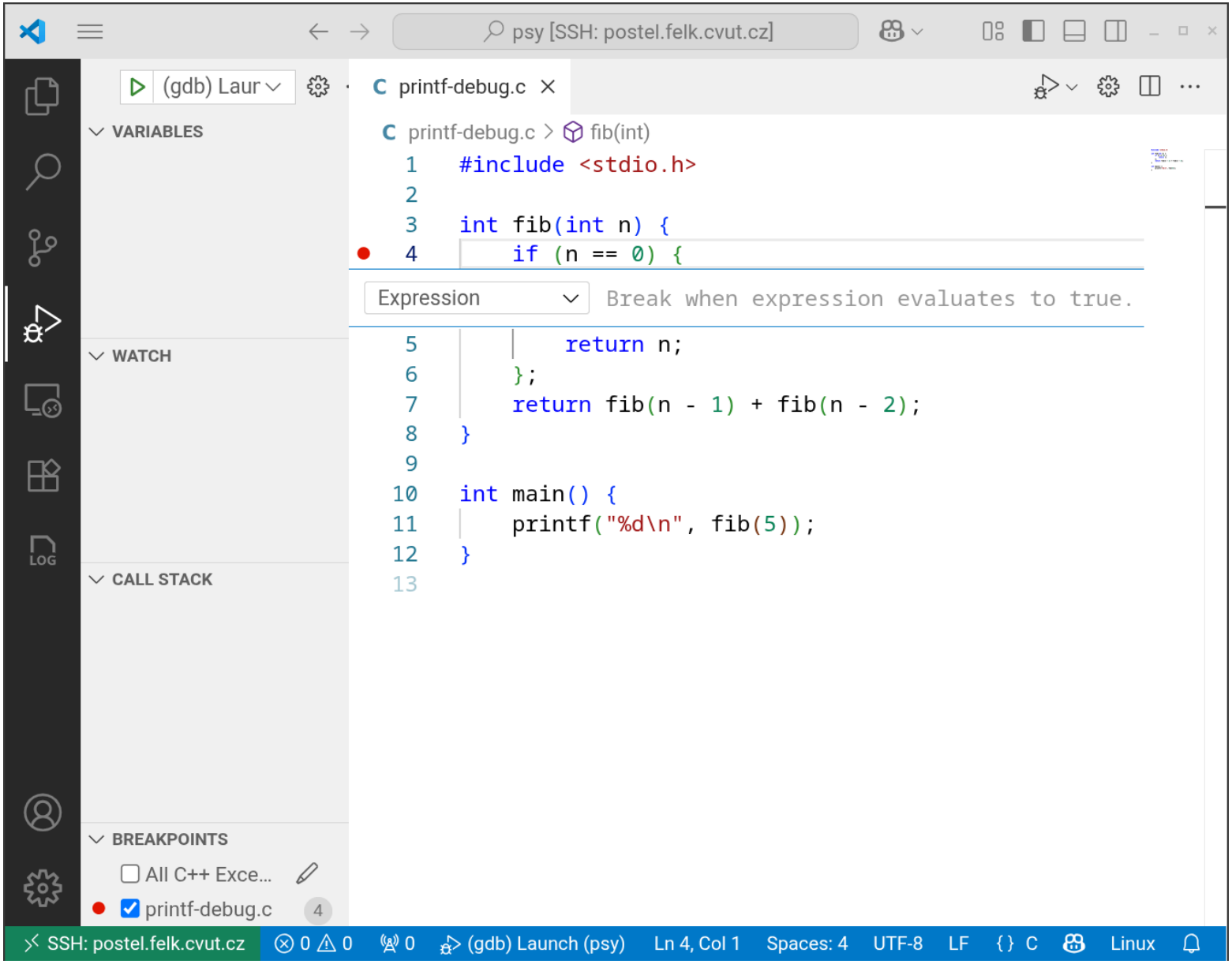
- Co kdybychom ale začínali s `n=100`?

Demo: Hledání chyby v `fib`

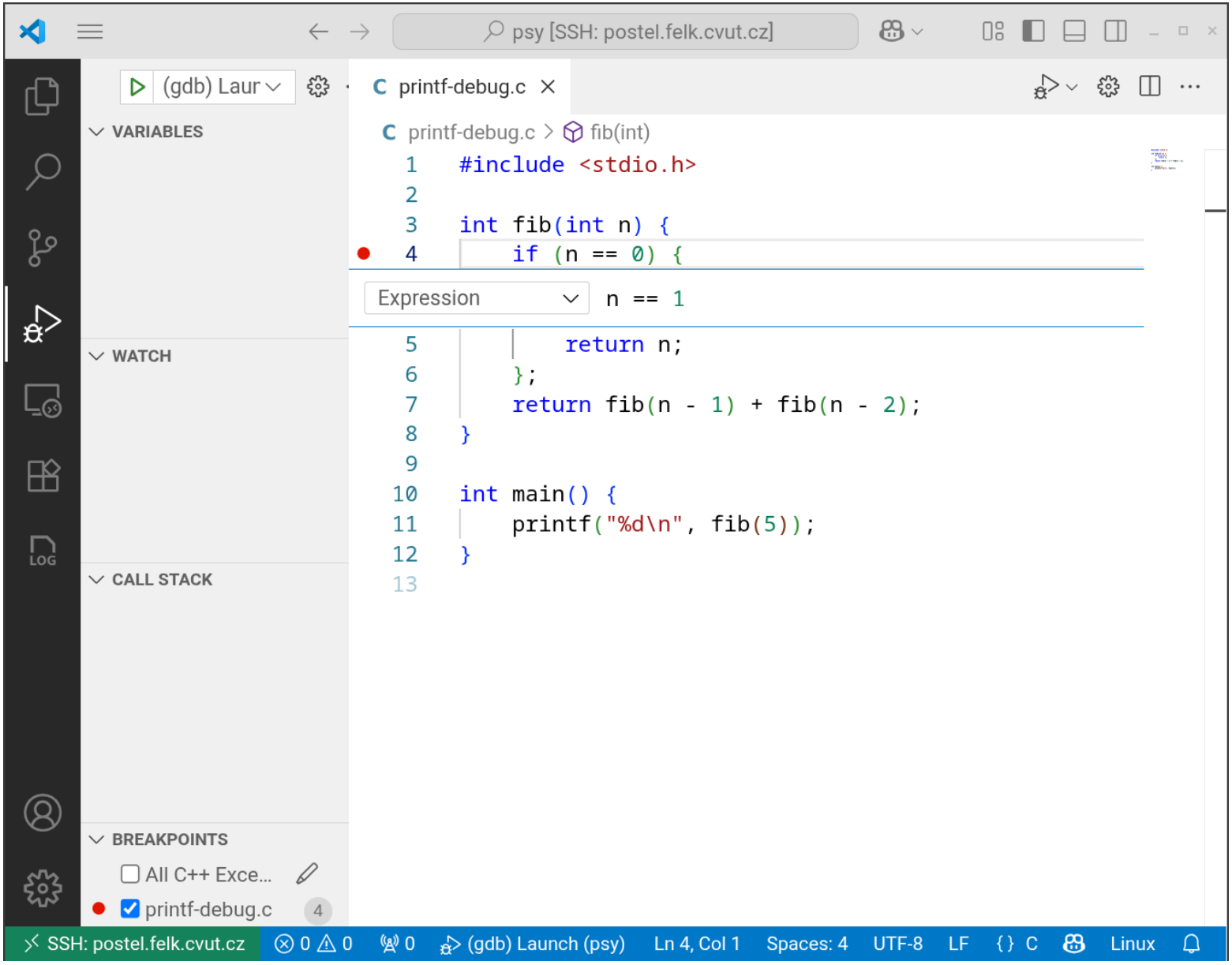
- Výhody:
  - Nemuseli jsme měnit kód.
  - Vidíme všechny proměnné.
  - Vidíme, jak jsme se na dané místo v kódu dostali.

- Co kdybychom ale začínali s `n=100`?
- **Podmíněný breakpoint**

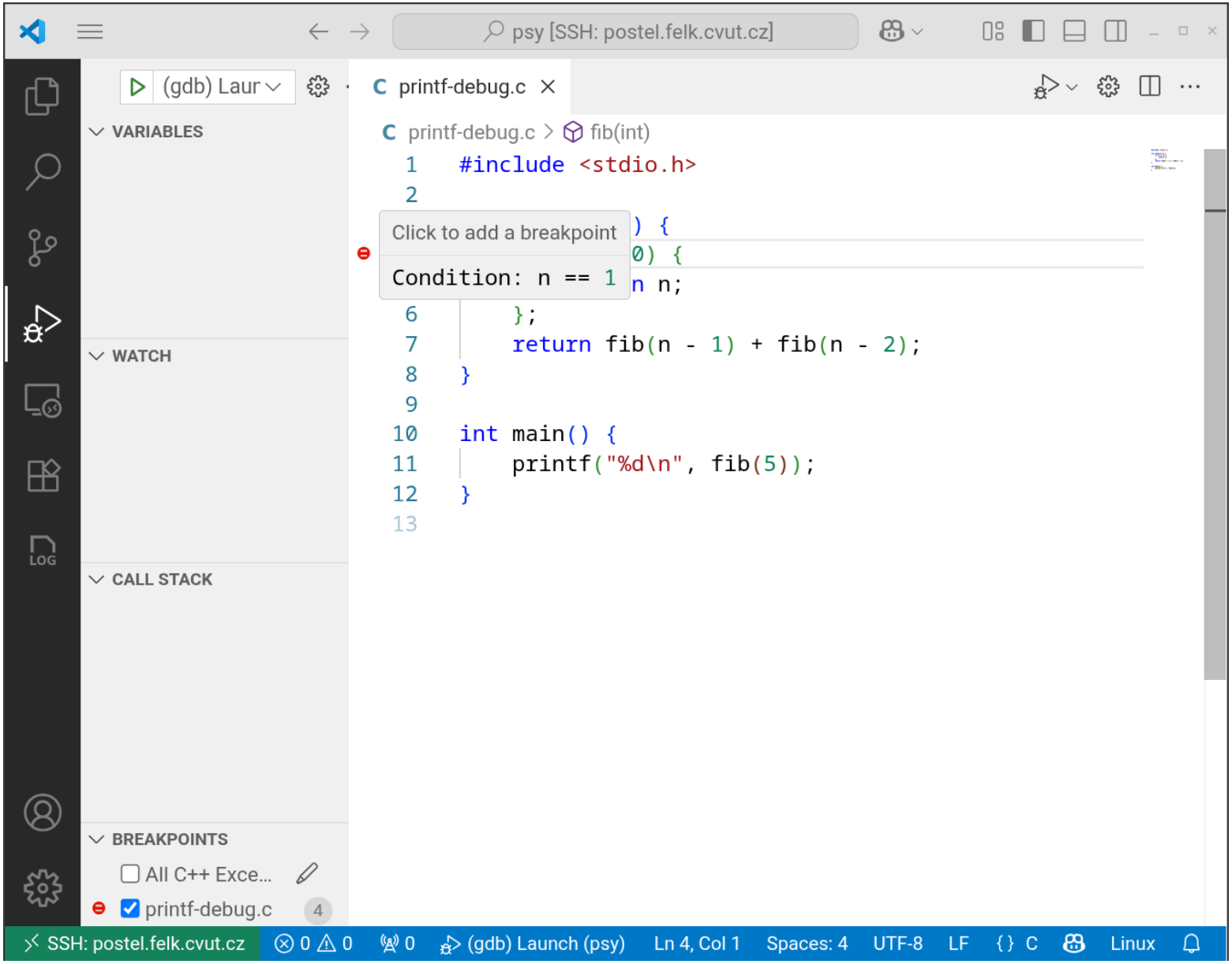
Demo


psy [SSH: postel.felk.cvut.cz]
(gdb) Laur
printf-debug.c
VARIABLES
WATCH
CALL STACK
BREAKPOINTS
All C++ Exce...
printf-debug.c
4
printf-debug.c > fib(int)
1 #include <stdio.h>
2
3 int fib(int n) {
4 if (n == 0) {
Expression
Break when expression evaluates to true.
5 return n;
6 };
7 return fib(n - 1) + fib(n - 2);
8 }
9
10 int main() {
11 printf("%d\n", fib(5));
12 }
13
SSH: postel.felk.cvut.cz
0 0
0
(gdb) Launch (psy)
Ln 4, Col 1
Spaces: 4
UTF-8
LF
C
Linux

Demo


psy [SSH: postel.felk.cvut.cz]
(gdb) Laur
printf-debug.c
VARIABLES
WATCH
CALL STACK
BREAKPOINTS
All C++ Exce...
printf-debug.c
4
printf-debug.c > fib(int)
1 #include <stdio.h>
2
3 int fib(int n) {
4 if (n == 0) {
Expression
n == 1
5 return n;
6 };
7 return fib(n - 1) + fib(n - 2);
8 }
9
10 int main() {
11 printf("%d\n", fib(5));
12 }
13
SSH: postel.felk.cvut.cz
0 0
0
(gdb) Launch (psy)
Ln 4, Col 1
Spaces: 4
UTF-8
LF
{} C
Linux

Demo


psy [SSH: postel.felk.cvut.cz]
(gdb) Laur
printf-debug.c
VARIABLES
WATCH
CALL STACK
BREAKPOINTS
All C++ Exce...
printf-debug.c
4
printf-debug.c > fib(int)
#include <stdio.h>
Click to add a breakpoint
Condition: n == 1
) {
0) {
n n;
};
return fib(n - 1) + fib(n - 2);
}
int main() {
printf("%d\n", fib(5));
}
SSH: postel.felk.cvut.cz
0 0
0
(gdb) Launch (psy)
Ln 4, Col 1
Spaces: 4
UTF-8
LF
C
Linux

Assert

```c
#include <assert.h>

int factorial(int n) {
    assert(n >= 0);
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int factorial(int n) {
    if (!(n >= 0)) {
      fprintf(...);
      exit(EXIT_FAILURE);
    }
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

Základní koncepty debugování

Klíčové funkce debuggeru

- **Krokování**
  - Step Over, Step Into, Step Out, (Step Back)
- **Lokální proměnné**
  - Zobrazení aktuálních hodnot
- **Watch window & vyhodnocení výrazů**
  - Sledování a vyhodnocování výrazů
- **Call Stack**
  - Zásobník volání funkcí
- **Breakpoints**
  - Základní, podmíněné, dočasné, …

Debugger ve VS Code

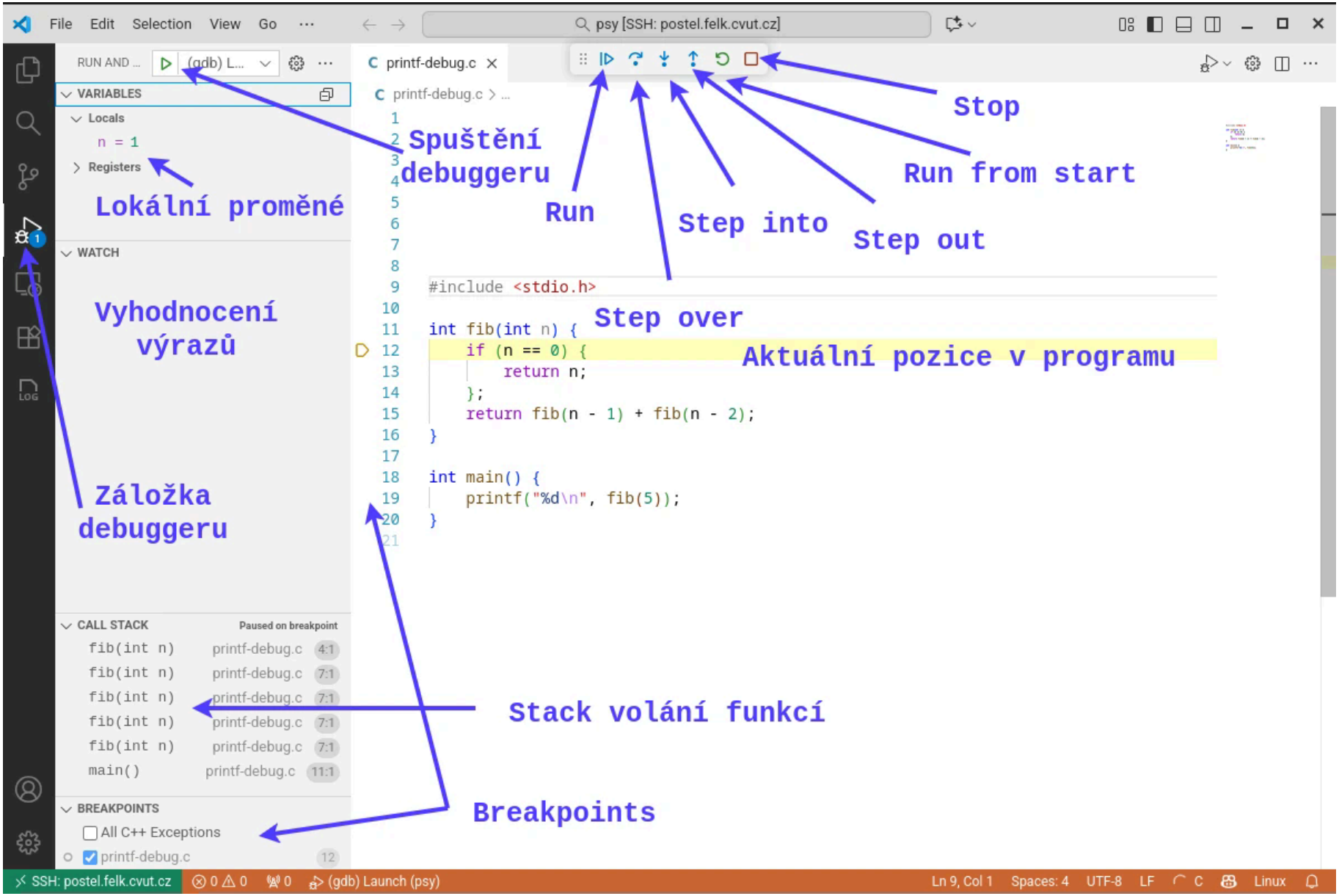
Command

```
ext install ms-vscode.cpptools
ext install vadimcn.vscode-lldb // alternativa
```

Soubor `.vscode/launch.json`

```
{
    "configurations": [{
          "name": "C++ Launch",
          "type": "cppdbg",
          "request": "launch",
          "program": "${workspaceFolder}/build/program",
          "args": [],
          "stopAtEntry": false,
          "cwd": "${workspaceFolder}",
          "MIMode": "gdb"  // nebo "lldb" na macOS
      }]
}
```

Debugger ve VS Code



Debugger v terminálu: GDB/LLDB

Proč se učit CLI debugger?

- Dostupný všude (i přes SSH na serveru)
- Rychlý a mocný, pokud znáte příkazy
- Základ pro všechny grafické debuggery (i ten ve VS Code)

Command

```
gdb ./program
(gdb) b main
(gdb) run
(gdb) next
(gdb) step
(gdb) print i
(gdb) continue
(gdb) bt
(gdb) quit
```

Základní příkazy:

- `b <line|func>`: breakpoint
- `run`: spustit program
- `next`: další řádek
- `step`: další řádek
- `print <var>`: vytiskne `var`
- `c` | `continue`: pokračuje v běhu
- `bt`: backtrace (zásobník volání)
- `q`: ukončit GDB

Kompilace pro debugging

Debug symboly

Compiler potřebuje “přibalit” informace navíc[1][2]:

- Které instrukce procesoru odpovídají řádkům C programu
- Kde v paměti a/nebo v registrech jsou které proměnné
- Použijte flag `-g`

Command

```
gcc -g -Wall -Wextra -o program program.c
```

[1] Na Linuxu jsou součástí spustitelného souboru. Na Windows ve zvláštním souboru `*.pdb`

[2] https://en.wikipedia.org/wiki/DWARF

Problém: Debugging a optimalizace

Co se stane s `-O2` nebo `-O3`?

Kompilátor agresivně mění kód, aby byl rychlejší:

- **Proměnné zmizí:** Jsou uloženy jen v registrech nebo úplně odstraněny.
- **Kód se přeuspořádá:** Instrukce se přesouvají pro lepší výkon.
- **Funkce se inlinují:** Tělo malé funkce se vloží přímo do volajícího.

Důsledek: Vztah mezi zdrojovým kódem a strojovým kódem je nejasný.

Jak to vypadá v debuggeru?

- `print x` nefunguje: `variable 'x' has been optimized out`
- Krokování (step/next) skáče nepředvídatelně po kódu.
- Breakpointy se nemusí trefit na správný řádek.
- Hodnoty proměnných se mohou jevit jako nesmyslné.

Problém: Debugging a optimalizace

Doporučení:

Vždy debugujte s vypnutými optimalizacemi (`-O0`).
Optimalizujte (`-O2`, `-O3`) až finální, otestovanou verzi.

Je `printf` debugging vždy špatný?

Ne! Má své místo - říká se mu **logging**.

Kdy je printf debugging vhodný?

- Program v produkci (na serveru zákazníka)
- Embedded systémy (kde debugger není dostupný)
- Real-time systémy (kde debugger by změnil timing)
- Vícevláknové programy (debugger může změnit race conditions)
- Dlouhodobé sledování chování programu

Který nástroj použít?

Rozhodovací strom

1. Program se nepřeloží? → Čtěte chyby kompilátoru

2. Program se přeloží s varováními? → Opravte varování (`-Wall -Wextra`)

3. Program padá (Segmentation fault)?

→ Zkuste Address Sanitizer nebo Valgrind

→ Pokud stále nejasné, použijte debugger

4. Program dává špatný výsledek?

→ Debugger s breakpointy

→ Podmíněné breakpointy pro specifické hodnoty

5. Program běží pomalu/zasekne se?

→ Debugger: pozastavte běžící program

6. Chyba se objevuje jen občas?

→ Zkuste Address Sanitizer nebo Valgrind

Závěrečné myšlenky

Moudra o debugování

- “Každý program obsahuje alespoň jednu chybu a dá se zkrátit o alespoň jeden řádek. Z toho vyplývá, že každý program lze zredukovat na jediný řádek, který nefunguje.”

Moudra o debugování

- “Debugging je dvakrát těžší než psaní kódu. Pokud tedy napíšete kód tak chytře, jak jen dovedete, nejste podle definice dost chytří na to, abyste ho debugovali.”

Bonus: Post-mortem debugging

Výstup programu

```
Segmentation fault (core dumped)
```

Bonus: Post-mortem debugging

Bonus: Post-mortem debugging

Výstup programu

```
Segmentation fault (core dumped)
```

Command

```
coredumpctl list
coredumpctl debug
```

Výstup programu

```
Core was generated by `./13_assert_example'.
Program terminated with signal SIGABRT, Aborted.
Downloading 4.48 K source file /usr/src/debug/glibc/glibc/nptl/
pthread_kill.c
#0  __pthread_kill_implementation (threadid=<optimized out>,
signo=signo@entry=6, no_tid=no_tid@entry=0) at
pthread_kill.c:44
44              return INTERNAL_SYSCALL_ERROR_P (ret) ?
INTERNAL_SYSCALL_ERRNO (ret) : 0;
```

Bonus: Post-mortem debugging

Výstup programu

```
(gdb) bt
#0  __pthread_kill_implementation (threadid=<optimized out>,
signo=signo@entry=6, no_tid=no_tid@entry=0) at pthread_kill.c:44
#1  0x00007799a9098a13 in __pthread_kill_internal
(threadid=<optimized out>, signo=6) at pthread_kill.c:89
#2  0x00007799a903e410 in __GI_raise (sig=sig@entry=6) at ../
sysdeps/posix/raise.c:26
#3  0x00007799a902557a in __GI_abort () at abort.c:77
#4  0x00007799a90254e3 in __assert_fail_base (fmt=<optimized out>,
assertion=<optimized out>, file=<optimized out>, line=<optimized
out>,
    function=<optimized out>) at assert.c:118
#5  0x000057b6e54b9182 in factorial (n=-1) at 13_assert_example.c:10
#6  0x000057b6e54b91f3 in main () at 13_assert_example.c:17
(gdb)
```