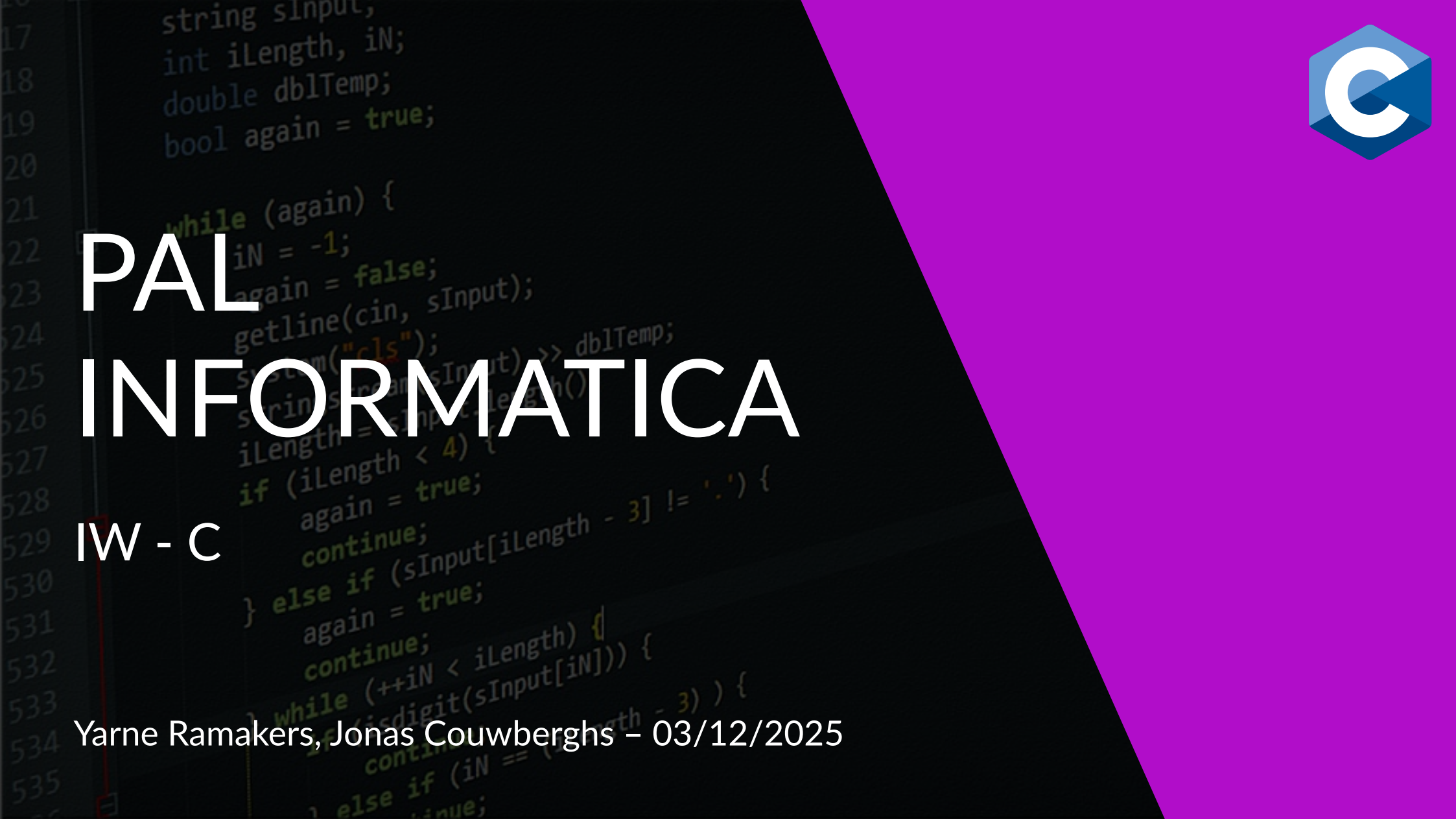


PAL INFORMATICA

IW - C

Yarne Ramakers, Jonas Couwberghs – 03/12/2025

Inhoud

COMPILER VS INTERPRETER

1. Waarom C?

1. Low level controle over machine en geheugen
2. Voorspelbaar, high performance (native)
3. Fundamenteel voor systems software en embedded systems
4. Veel runtimes en libraries (Python, Node, Postgres) geïmplementeerd in C
5. Maar: with great power, comes great responsibility

“I have yet to see a language that comes even close to C, when it comes to interacting with hardware.” – Linus Torvalds

2. Compiled vs Interpreted

- Compiled:
  - source -> assembly -> machine code -> CPU
- Interpreted:
  - source -> bytecode -> VM of interpreter
- JIT (hybride):
  - compile at runtime

2. Compiled vs Interpreted

Gevolgen

- Native speed
- Kleinere runtime
- Errors tijdens compile time
- Ander debugging- en deploymentmodel
- Compiler optimalisaties
- Vaak statisch getypeerd
- Compiler assumet dat er geen undefined behavior is

2. Compiled vs Interpreted

Statisch vs Dynamisch getypeerd

- Statisch getypeerd `(C, C++, Rust, Java)`
  - expliciet types aangeven (compiler kan vaak “inferen”)
  - types gecontroleerd tijdens compilatie
  - variabele en functietypes gekend voor het programma draait
  - fouten vroeg gevonden
- Dynamisch getypeerd `(Python, JS, Ruby, Perl)`
  - types tijdens runtime bepaald
  - meer flexibiliteit (minder boilerplate)
  - typefouten verschijnen pas bij het uitvoeren

2. Compiled vs Interpreted

Compilatie stappen

1. C source code (main.c)
2. Pre-processor (main.i)
3. Compiler (main.s)
   - C omzetten naar een IR[1] en dan naar assembly
4. Assembler (main.o)
   - placeholder voor adressen
5. Linker (executable)
   - vult placeholders in
   - linkt main.o met libraries (.so en .dll)

[1] voorstelling tussen C en assembly

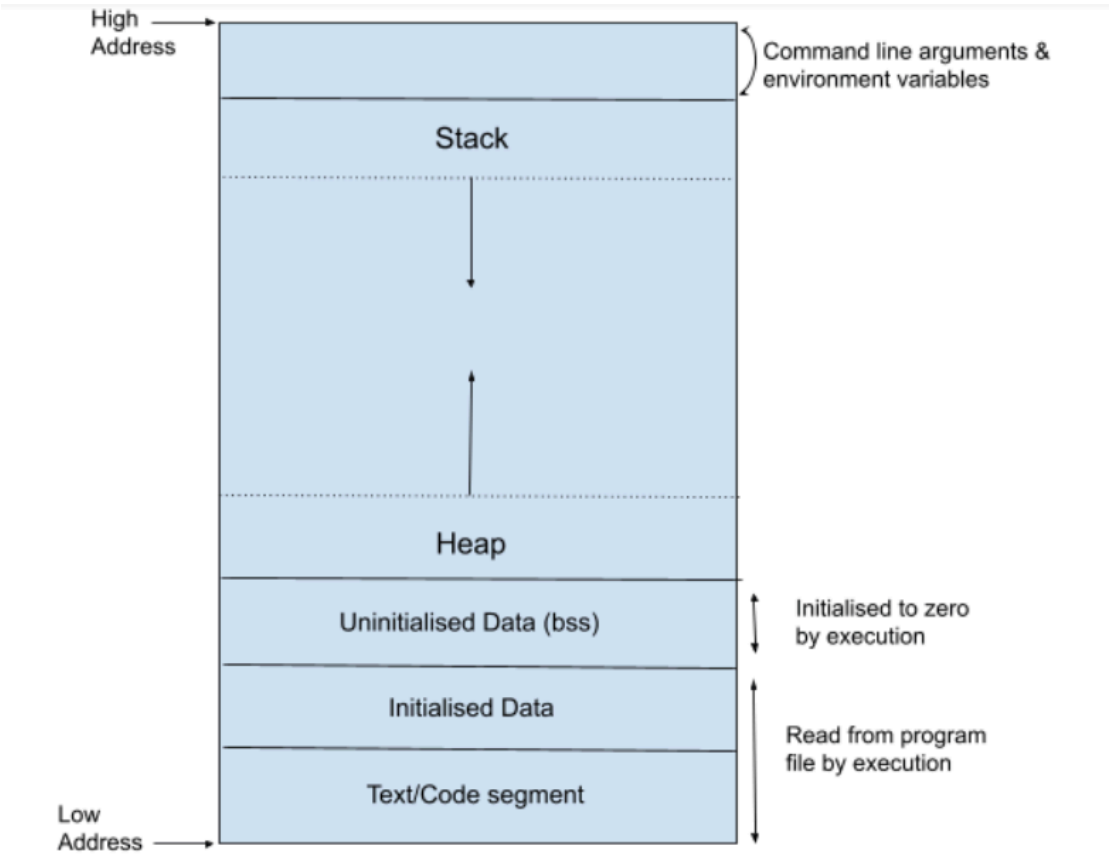
2. Compiled vs Interpreted

Compilatie stappen

- Preprocessor
  - includes, macros
  - conditionele compilatie
- Compiler
  - type checking
  - vertalen naar assembly
- Assembler
  - machine code, relocations
  - object files
- Linker
  - resolves symbols
  - combineert object files met libraries

HET C GEHEUGENMODEL

Het C Geheugenmodel



1. Text segment

Bevat de feitelijke instructies van het programma die worden uitgevoerd

- read only
- ingeladen in geheugen als je programma opstart
  - = Von Neumann

2.Data (static)

Globale en statische variabelen, vaak opgedeeld in twee delen

- Geïnitializeerd data segment
  - waarden staan in executable
  - waarden automatisch ingesteld door OS

```
int x = 100;
```

- Ongeïnitializeerd data segment (BSS)
  - waarden op 0 gezet door OS

```
int y;
```

3. Heap

Dynamisch geheugen, zelf beheren

- alloceren met `malloc`, `calloc`, `realloc`
- groeit omhoog
- geheugen vrijgeven met `free()` na gebruiken
  - anders memory leak!
  - ownership: wie malloc doet, doet ook free
- hier zet je dingen waarvan de compiler niet kan weten hoe groot het is
- data op heap aanspreken met pointers
- meestal trager dan stack/data segment

```c
int* ptr = (int*) malloc(sizeof(int) * 5); // allocate array of 5 ints on heap
free(ptr);                                 // free memory
```

4. Stack

Dynamisch geheugen, beheerd door de compiler

- functie call frames, lokale variabelen en functie parameters
- automatisch gealloceerd wanneer functies stoppen en opgeroepen worden
  - als functie stopt, verdwijnen de lokale variabelen (let op met pointers!)
- snelle access via offset
- grootte van data moet gekend zijn (vandaar pointers!)

```c
void foo() {
    int x = 10;     // local variable, stored on stack
}
```

- stackoverflow als er te veel geheugen op de stack zit (vb oneindige recursie)

5. Valkuilen

- pointer gebruiken na de data out of scope is (dangling)
- pointer gebruiken na het gefreet is (use after free)
- pointer vs array lifetime
- een buffer overvullen op de stack kan frames corrupteren
  - kan misbruikt worden om malicious code uit te voeren

```
int* p;
{
    int x = 5;
    p = &x; // dangling pointer
}
// *p invalid!
```

- segfault
  - geheugen opvragen dat niet van jou is

C SYNTAX

1. Basis structuur van een programma

C

```c
#include <stdio.h>

int main(void) {
  printf("Hello, world!\n");
  return 0;
}
```

vs.

Python

```python
print("Hello, world!")
```

C gebruikt een `main` entry point

Python runt top-level code direct.

2. Variabelen en Types

C (static typing)

```c
int i = 5;          // 32bit int
unsigned u = 10;    // 32bit uint
long l = 132;       // 32/64bit int
float pi = 3.14f; // 32bit float
char c = 'A';       // char
...
```

vs.

Python (dynamic typing)

```python
i = 5       # int
u = 10      # int
l = 132     # int
pi = 3.14   # float
c = 'A'     # str
...
```

Variabelen in C hebben een vast type dat geweten is tijdens het compileren

Python bepaalt het type van de variabele tijdens de runtime

Volledige lijst van C primary types: https://www.programiz.com/c-programming/c-data-types

3. Pointers in C

C

```c
int x = 42;
// ptr slaat het adres van x op
int *ptr = &x;

printf("x = %d\n", x);
// print "x = 42"
printf("*ptr = %d\n", *ptr);
// print "*ptr = 42" (dereferencing)

// verander x via de pointer
*ptr = 100;
printf("x = %d\n", x);
// print "x = 100"
```

vs.

Python

```python
x = 42
# y verwijst naar hetzelfde object als x
y = x

print("x =", x)
# print "x = 42"
print("y =", y)
# print "y = 42"

# x verandert niet
y = 100
print("x =", x)
# print "x = 42"
```

3. Pointers in C

Uitleg

C

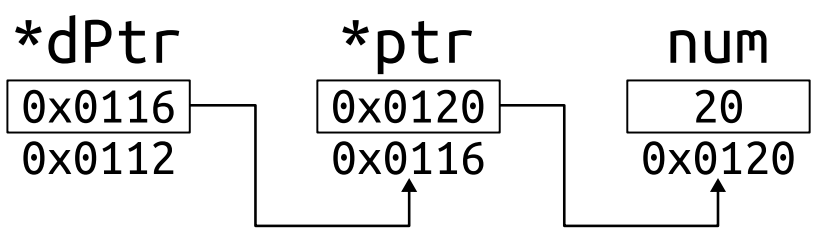
- `&` vraagt het adres van een variabele op
- `*` dereferenceert een pointer om de waarde te lezen of te schrijven

vs.

Python

- Geen expliciete pointers
- Variabelen zijn referenties naar objecten

```c
int num = 20;
int *ptr = &num;
int **dPtr = &ptr;
```



4. Dynamic Memory: malloc / free

C

```c
#include <stdlib.h>

int main(void) {
  // Statisch gealloceerd
  int arr[5] = {1, 2, 3, 4, 5};
  // Dynamisch gealloceerd
  int *dynArr = malloc(5 * sizeof(int));
  if(!dynArr) return 1;
  for (int i = 1; i ⩽ 5; i++) dynArr[i] = i;

  // Geheugen van dynArr vrijgeven
  // wanneer het nietmeer nodig is
  free(dynArr);
  return 0;
}
```

vs.

Python

```python
# Dynamisch gealloceerd
arr = [1, 2, 3, 4, 5]

# NIET (ZOMAAR)
# MOGELIJK IN C
arr.append(10)

# Garbage collection
#  → Geen free nodig
```

4. Dynamic Memory: malloc / free

Uitleg

Statisch gealloceerd:

- Opgeslagen op de **stack**
- Automatisch aangemaakt bij uitvoeren van functie
- Automatisch gefreed op einde van de functie
- Vaste grote bij compiletime bekend

vs.

Dynamisch gealloceerd

- Opgeslagen op de **heap**
- Wordt aangemaakt door `malloc` op te roepen
- Bestaat tot `free` opgeroepen wordt
- Grote kan dynamisch bepaald worden

5. Strings in C

C strings (null-terminated char arrays):

```c
// opgeslagen als een array met een null ('\0') als laatste teken
char s[] = "Hello"; // opgeslagen als {'H','e','l','l','o','\0'}

// dynamische allocatie voorbeeld
char *name = malloc(6); // alloceer de maximum lengte van de string + 1 (voor '\0')
if (name) {
  strcpy(name, "Alice");
  printf("%s %s!\n", hello, name); // print "Hello Alice!"
  free(name);
}
```

5. Strings in C

Nuttige functies voor strings

- `strcmp(char *a, char *b)`
  - Vergelijkt string `a` met string `b`, returnt 0 als de strings gelijk zijn
- `strncopy(char *dest, char *src, size_t n)`
  - Kopieert maximaal `n` tekens van `src` naar `dest`
- `strncat(char *dest, char *src, size_t n))`
  - Voegt maximaal `n` tekens van `src` toe aan het einde van `dest`
- `strlen(char *s)`
  - Returnt de lengte van de string
- `strchr(char *s, char c)`
  - Geeft de index van het eerste voorkomen van teken `c` in `s` terug of −1
- `strstr(char *s, char *sub)`
  - Geeft de index van het eerste voorkomen van substring `sub` in `s` terug of −1

6. Structs

- Groepering van verschillende datatypes in één geheel
- Velden of members: individuele onderdelen van de struct
- Creëert een nieuw type voor variabelen of pointers
- Toegang van velden via . (variabele) of -> (pointer)
- Kan genest worden of gebruikt in arrays
- Handig om gerelateerde data overzichtelijk te bewaren

```c
struct point_s {
  int x;
  int y;
} point_t;

struct line_s {
  point_t *begin;
  point_t *end;
} line_t;

struct list_s {
  int value;
  struct list_s *next;
} list_t;
```

6. Structs

Voorbeeld

```c
struct point_s {
  int x;
  int y;
} point_t;
```

```c
// Statisch gealloceerd
point_t p1 = {10, 20};
point_t p2;
p2.x = 5;
p2.y = 15;

// Dynamisch gealloceerd
point_t *p3 = malloc(sizeof(point_t))
if (!p3) return 1;
p3→x = 12; // (hetzelfde als: (*p3).x = 12)
p3→y = 6; // (hetzelfde als: (*p3).y = 6)

printf("p1 = (%d, %d)\n", p1.x, p1.y);
printf("p2 = (%d, %d)\n", p2.x, p2.y);
printf("*p3 = (%d, %d)\n", p3→x, p3→y);

free(p3);
```

7. De ++ operator

Op integers:

```c
int a = 5;
printf("a = %d\n", a); // Print 5
a++; // (zelfde als: a = a + 1)
printf("a = %d\n", a); // Print 6
```

vs.

Op pointers:

```c
int arr[5] = {5, 4, 3, 2, 1};
// p wijst naar het eerste element
int *p = arr;

printf("*p = %d\n", *p); // Print 5
// p verschuift naar het volgende element
p++; // (zelfde als: p = p + size_of(int))
printf("*p = %d\n", *p); // Print 4
```

8. Out-of-bound access

```c
int arr[3] = {10, 20, 30};

// !!! Index 3 bestaat niet
arr[3] = 40;
printf("%d\n", arr[3]);  // Onvoorspelbaar gedrag
```

- Waarden van andere variabelen in de stack worden overschreven
  - Bij dynamische allocatie kan het andere waarden op de heap overschrijven
- Het programma kan crashen `(core dumped)`
- Of het lijkt te “werken”, maar veroorzaakt later rare fouten

GCC

Wat is GCC

- GNU Compiler Collection
- Compiler voor C, C++, etc
- Converteert `.c` naar een executable
- Kan ook: warnings geven, optimaliseren, preprocessen, linken, etc

Compileren

```
$ gcc main.c -o main
$ ./main
```

- `-o main` stelt outputnaam in
- zonder `-o` standaard `a.out`
- compileert en linkt in één stap

Compilatie stappen

```
$ gcc -E main.c -o main.i   # preprocess
$ gcc -S main.c -o main.s   # compile naar asm
$ gcc -c main.c -o main.o   # assemble
$ gcc main.o -o main        # link
```

- handig om elke fase te bekijken
- GGC is een driver voor alle fasen

Waarschuwingen

```
$ gcc -Wall -Wextra -Wpedantic -Werror main.c -o main
```

- `Wall` meest voorkomende errors, zoals:
  - niet gebruike variabelen
  - ontbrekende return statements
- `Wextra` meer aggressieve warnings, zoals:
  - ongebruikte functie parameters
  - vreemde pointer arithmetic
- `Wpedantic` geen compiler extensies gebruiken
  - GGC heeft extra features die niet deel zijn van de C-standaard
- `Werror` zet warnings om in errors

Optimalisaties

```
$ gcc -O0   # geen optimalisaties (debuggen)
$ gcc -O1   # basic optimalisaties
$ gcc -O2   # goede default
$ gcc -O3   # agressieve optimalisaties
```

- `O0` compileert code exact zoals het geschreven is
- optimizen maakt code sneller en korter
  - compileren kan langer duren
  - bij agressieve optimalisaties kan binary veel groter worden!
- let op: undefined behavior kan tot rare resultaten leiden
  - compiler gaat ervan uit dat er geen UB is!

Typisch compileren met:

```
gcc -O2 -Wall -Wextra -pedantic
```

DEMO

GDB

Wat is GDB

- GNU Debugger
- Inspecteer programma terwijl het draait
- Bekijk variabelen, geheugen, stack frames
- Debug segfaults en UB-gedrag

GDB gebruiken

Compileer eerst een debug build:

```
gcc -g -O0 main.c -o main
```

- `-g` compileer met debug symbolen
- `$O0` geen optimalisaties

Start de debugger

```
gdb ./main
```

Commando's

```
run          # start het programma
break main   # breakpoint op een functie
break 12     # breakpoint op een lijn
next         # volgende lijn, skip functies
step         # step in een functie
continue     # ga verder met uitvoeren
```

- `break` (of `b`) is de belangrijkste
- `next` vs `step` is een groot verschil

Inspecteren

```
print x     # print een variabele
print *ptr  # dereference een pointer
backtrace   # toon de call stack
info locals # lokale variabelen in de huidige frame
info args   # argumenten van de functie
```

- `backtrace` (of `bt`) helpt om segfaults op te sporen
  - laat zien in welke functie je crasht

Segfault debuggen

1. Compileer met debug symbolen
2. Run in GDB
3. Laat het crashen
4. Gebruik:

```
bt        # waar? (callstack)
frame n   # inspecteer een specifieke frame
print var # inspecteer een variabel
```

- GDB gaat naar de lijn waar het programma crasht
- backtrace toont hoe je daar bent gekomen

Condities

- Het kan zijn dat bugs pas “soms” gebeuren

```
break 23 if x > 10
```

Memory tracken

```
watch x
watch *ptr
```

- pauzeert wanneer een variabele verandert
  - of wanneer de waarde op die pointer verandert
  - `rwatch` of `awatch` pauzeren wanneer variabel gelezen (of gelezen en geschreven) wordt
- perfect om memory corruptie op te sporen

Handige commandos

```
set print pretty on/off   # mooiere weergave structs en arrays
disas                     # laat assembly zien
info registers            # toont CPU-registers
```

DEMO

VOORBEELD OPGAVE

Terug te vinden op GitHub

De voorbeeld opgave kan je samen met de slides terugvinden op GitHub:

https://github.com/informatica-kul-PAL/IW-C-Slides/