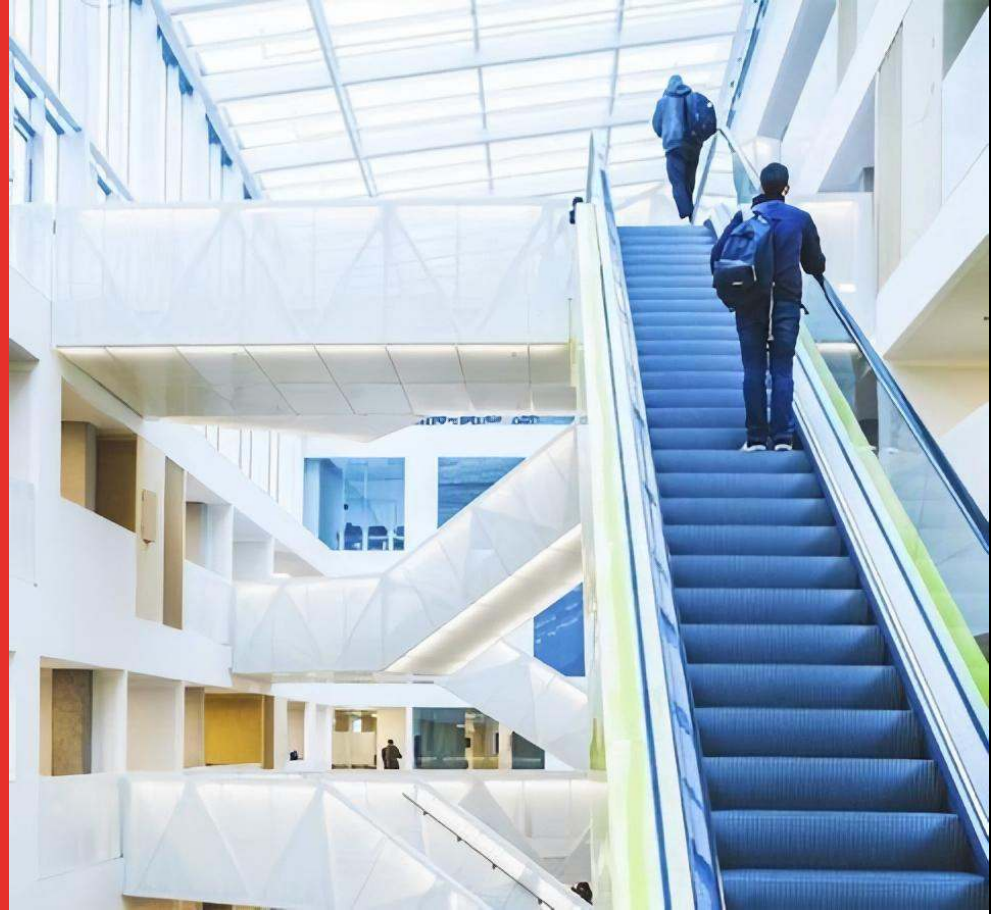
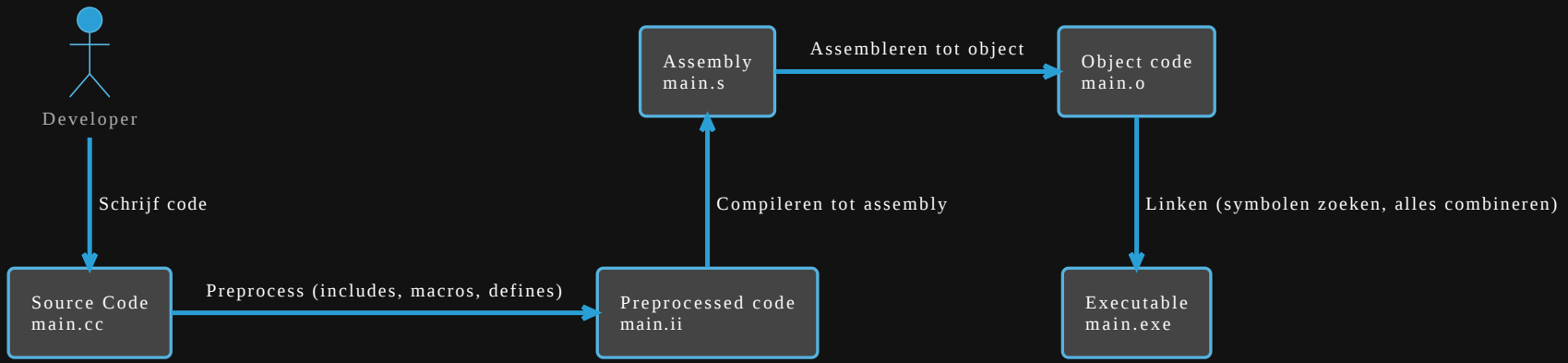
EEN APPLICATIE BOUWEN

- Buildproces in C++: Preprocessen, Compileren, Assembleren & Linken
- Devcontainers en ontwikkelomgeving
- Introductie tot CMake
- Basis debug technieken



OVERZICHT BUILDPROCES

- Preprocessen: Macro-expansie, include-bestanden en conditionele compilatie
- Compileren: Vertalen van preprocessed code naar assembly
- Assembleren: Omzetten van assembly code naar objectcode
- Linken: Samenvoegen van objectbestanden tot een uitvoerbaar bestand



PREPROCESSING

Wat doet de preprocessor?

- Vervangt macro's en constante expressies
- Verwerkt `#include` statements: voegt header-bestanden in
- Voert voorwaardelijke compilatie uit via `#ifdef`, `#ifndef` etc.
- Verwijdert commentaar
- Eigenlijk alles waar een `#` voor staat heeft met de preprocessor te maken

```
1  #include <iostream>
2  #define SQUARE(x) ((x) * (x))
3
4  int main() {
5      std::cout << "4 kwadraat is " << SQUARE(4) << "\n";
6      return 0;
7  }
```

PREPROCESSING: MACRO'S

Macro's in C++:

- Gedefinieerd met `#define`
- Vervangen codefragmenten tijdens de preprocessing
- Worden vaak gebruikt voor herbruikbare code en constante waarden
- Zijn een slecht idee!
- Maar je moet wel weten wat ze zijn!

Voorbeeld:

```
1
2
3  int main() {
4      std::cout << "4 kwadraat is " << ((4) * (4)) << "\n";
5      return 0;
6  }
```

WAAROM IS DIT EEN SLECHT IDEE?

- Ze zijn geen onderdeel van de taal
- Je kan ze niet debuggen
- Vervelende side effects
- Macros hebben geen namespace

WAT DAN WEL?

```
1  #define X = 5
2
3  #define SQUARE(x) ((x) * (x))
4
5  #import <iostream>
```

```
1  constexpr static int x = 5;
2
3  constexpr auto square(auto x) {return x * x;}
4
5  import std; // toekomstmuziek
```

VOORWAARDELIJKE COMPILATIE

Wat is voorwaardelijke compilatie?

- Wordt toegepast met `#ifdef`, `#ifndef`, `#if`, `#else` en `#endif`
- Laat toe dat bepaalde stukken code alleen worden gecompileerd als aan een voorwaarde is voldaan
- Veelgebruikt voor debug builds of platform-specifieke code

```
1  int main() {
2      std::cout << "Debug mode aan" << "\n";
3      return 0;
4  }
```

Je kunt de macro `DEBUG` ook definiëren via de command line (of je buildsysteem), zonder deze in de code te zetten

```
1  g++ -DDEBUG -E main.cc -o main.ii
```

VOORWAARDELIJKE COMPILATIE: VOORBEELD MET LOGGING

```cpp
#ifdef DEBUG
  #define LOG(msg) std::cout << "DEBUG: " << msg << "\n";
#else
  #define LOG(msg)
#endif

int berekenFactorial(int n) {
    LOG("Start berekenFactorial met n = " << n);
    int result = 1;
    for (int i = 1; i ≤ n; ++i) {
         result *= i;
         LOG("Intermediate result at i = " << i << " is " << result);
    }
    LOG("Eind berekenFactorial met result = " << result);
    return result;
}

int main() {
    std::cout << "Factorial van 5 is " << berekenFactorial(5) << "\n";
    return 0;
}
```

INCLUDE STATEMENTS

Wat doet `#include` ?

- Het voegt de inhoud van een opgegeven bestand in op de plaats van het `#include` statement
- Maakt code modulair en herbruikbaar door declaraties uit headers te importeren

```
1   #include "myheader.hh"
```

INCLUDE TRANSFORMATIE: VOORBEELD

Stel je hebt een headerbestand `myheader.hh` :

```
1  // myheader.hh
2  #ifndef MYHEADER_H
3  #define MYHEADER_H
4
5  #include <iostream>
6  void printMessage() {
7      std::cout << "Hello from myheader!\n";
8  }
9
10 #endif // MYHEADER_H
```

Originele code:

```
1  #include <iostream>
2  #include "myheader.hh"
3
4  int main() {
5      printMessage();
6      return 0;
7  }
```

INCLUDE TRANSFORMATIE: VOORBEELD

Stel je hebt een headerbestand `myheader.hh` :

Na Preprocessing (vereenvoudigd):

```cpp
#include <iostream> // let op wat hier gebeurd

// Inhoud van myheader.h
#ifndef MYHEADER_H
#define MYHEADER_H

#include <iostream> // let op wat hier gebeurd
void printMessage() {
    std::cout << "Hello from myheader!\n";
}

#endif // MYHEADER_H

int main() {
    printMessage();
    return 0;
}
```

HANDMATIG PREPROCESSEN

Hoe laat je de compiler alleen pre-processen?

- Gebruik de `-E` flag met GCC of Clang.
- Hiermee zie je de output na preprocessing zonder de code te compileren.

```
1  g++ -E main.cc -o main.ii
```

COMPILEREN

Wat gebeurt er bij compileren?

- De compiler zet de preprocessed code om in assembly-code
- Optimalisaties worden toegepast
- Er ontstaan waarschuwingen en fouten indien er syntaxfouten in de code zitten

Codevoorbeeld:

```
1  int add(int a, int b) {
2      return a + b;
3  }
```

Compiler opdracht (GCC):

```
1  g++ -S main.cc -o main.s
```

COMPILEREN

```cpp
int add(int a, int b) {
    return a + b;
}
```

```asm
add(int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     edx, DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-8]
        add     eax, edx
        pop     rbp
        ret
```

- Bekijk in Compiler Explorer

ASSEMBLEREN

Wat gebeurt er bij assembleren?

- De assembly-code wordt omgezet in objectcode (van text naar machinecode)
- Het resultaat is een objectbestand (.o of .obj) dat nog niet zelfstandig uitvoerbaar is
- Een object bestand bevat informatie over welke functies en variabele deze aanbied en nodig heeft

Voorbeeld:

Na compilatie krijg je een bestand `main.s` . De assembler zet dit om naar `main.o`

```
1  nasm -f elf64 -o main.o main.s
```

VOORBEELD

```
1   global _start
2
3   section .text
4
5   _start:
6     mov rax, 1          ; schrijf functie
7     mov rdi, 1          ; naar stdout
8     mov rsi, msg        ; adres van string
9     mov rdx, msglen     ; lengte van string
10    syscall             ; roep kernel aan
11
12    mov rax, 60         ; exit functie
13    mov rdi, 0          ; 0 = succes
14    syscall             ; roep kernel aan
15
16  section .rodata
17    msg: db "Hello, world!", 10
18    msglen: equ $ - msg
```

```
1   objdump -D hello.o
```

```
1   Disassembly of section .text:
2
3   0000000000000000 <_start>:
4      0:   b8 01 00 00 00          mov    eax,0x1
5      5:   bf 01 00 00 00          mov    edi,0x1
6      a:   48 be 00 00 00 00 00    movabs rsi,0x0
7     11:   00 00 00
8     14:   ba 0e 00 00 00          mov    edx,0xe
9     19:   0f 05                   syscall
10    1b:   b8 3c 00 00 00          mov    eax,0x3c
11    20:   bf 00 00 00 00          mov    edi,0x0
12    25:   0f 05                   syscall
```

LINKING

Wat gebeurt er bij linken?

- Meerdere objectbestanden en bibliotheken worden samengevoegd tot één uitvoerbaar bestand
  - Statische objecten (.o, .a, .lib)
  - Dynamische objecten (.so, .dll)
- De linker zorgt voor het oplossen van symbolen (functies en variabelen die in verschillende bestanden of zelfs programmeertalen gedefinieerd zijn)
- Geeft alles een adres
- Zet alles in de juiste sectie
  - text / data / bss

Voorbeeld:

```
1  ld -o hello hello.o
2  objdump -D hello
```

NAME MANGLING

C / Assembly: Geen overloading, interne naam == externe naam

```c
// C
void print_char(char c) {...}
void print_int(int x) {...}
void print_string(const char *s) {...}
```

```asm
; Assembler
print_char: ...
print_int: ...
print_string: ...
```

C++: Overloading, interne naam == mangle(externe naam)

```c
// C++
void print(char c) {...}
void print(int x) {...}
void print(const char *s) {...}
```

```asm
; Assembler
_Z5printc: ...
_Z5printi: ...
_Z5printPKc: ...
```

C++ met extern "C": Schakelt overloading en mangling uit

```c
// C++
extern "C" void print_char(char c) {...}
extern "C" {
    void print_int(int x) {...}
    void print_string(const char *s) {...}
}
```

```asm
; Assembler
print_char: ...

print_int: ...
print_string: ...

```

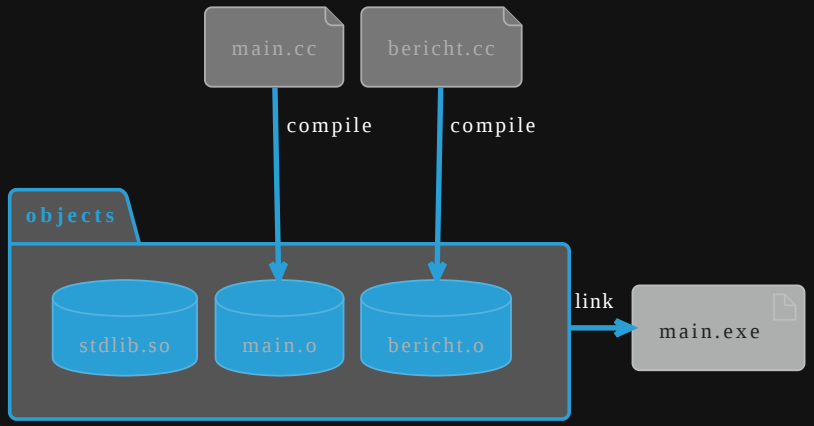
LINKING

main.cc

```cpp
void print_bericht();
char bericht[] = "Hello";
int main() {
    std::cout
        << "Main: [" << bericht << "]"
        << " @ " << (size_t) bericht << "\n";
    print_bericht();
}
```

bericht.cc

```cpp
char bericht[] = "Hello";
void print_bericht() {
    std::cout
        << "Func: [" << bericht << "]"
        << " @ " << (size_t) bericht << "\n";
}
```



stdout

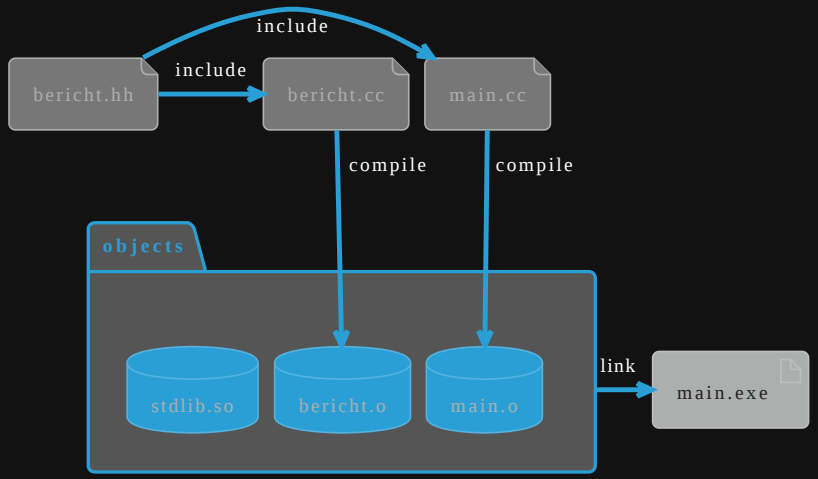
```
Main: [Hello] @ 102032131358752
Func: [Hello] @ 102032131358758
```

- Twee kopieën in de sources
- Twee kopieën in de executable

LINKING

main.cc

```cpp
#include "bericht.hh"
void print_bericht();
int main() {
    std::cout
        << "Main: [" << bericht << "]"
        << " @ " << (size_t) bericht << "\n";
    print_bericht();
}
```



bericht.hh

```cpp
char bericht[] = "Hello";
```

bericht.cc

```cpp
#include "bericht.hh"
void print_bericht() {
    std::cout
        << "Func: [" << bericht << "]"
        << " @ " << (size_t) bericht << "\n";
}
```

stdout

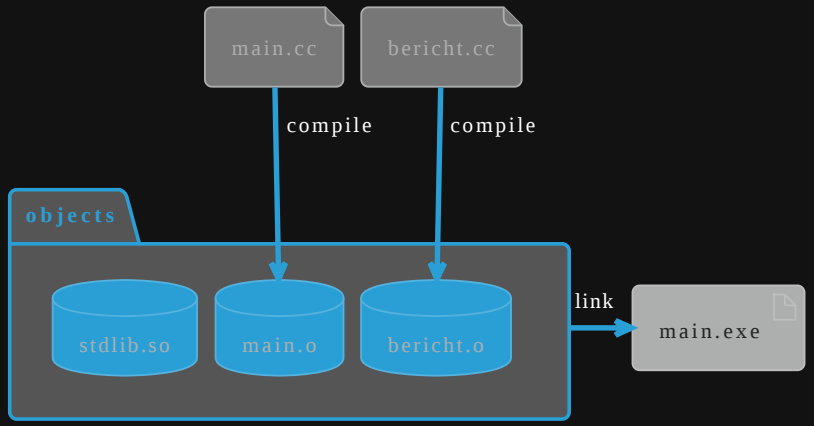
```
Main: [Hello] @ 102032131358752
Func: [Hello] @ 102032131358752
```

- Een kopie in de sources
- Twee kopieën in de executable

LINKING

main.cc

```cpp
void print_bericht();
extern char bericht[];
int main() {
    std::cout
        << "Main: [" << bericht << "]"
        << " @ " << (size_t) bericht << "\n";
    print_bericht();
}
```



bericht.cc

```cpp
char bericht[] = "Hello";
void print_bericht() {
    std::cout
        << "Func: [" << bericht << "]"
        << " @ " << (size_t) bericht << "\n";
}
```

stdout

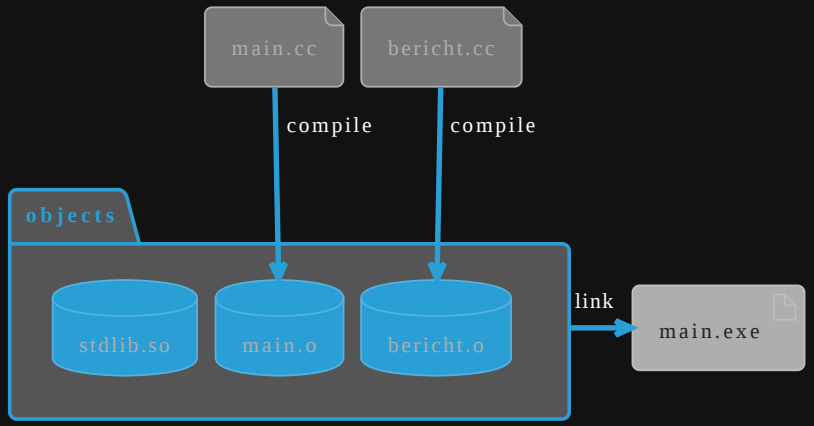
```
Main: [Hello] @ 102032131358752
Func: [Hello] @ 102032131358752
```

- Een kopie in de sources
- Een kopie in de executable
- Maar...

LINKING

main.cc

```
1  void print_bericht();
2  extern char bericht[];
3  int main() {
4      std::cout
5          << "Main: [" << bericht << "]"
6          << " @ " << (size_t) bericht << "\n";
7      print_bericht();
8  }
```



bericht.cc

```
1  int bericht = 42;
2  void print_bericht() {
3      std::cout
4          << "Func: [" << bericht << "]"
5          << " @ " << (size_t) bericht << "\n";
6  }
```

stdout

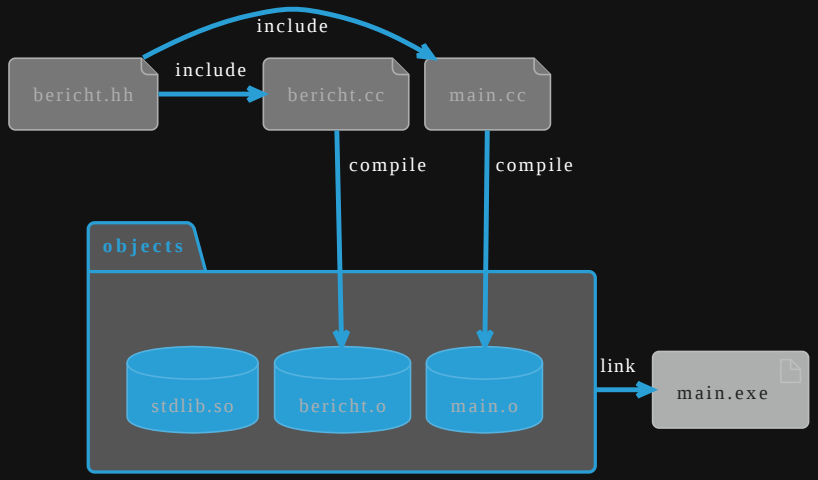
```
1  Main: [
2
3  Program terminated with signal: SIGSEGV
```

- Voor de linker is een label maar een geheugenplekje

LINKING

main.cc

```
1  #include "bericht.hh"
2  void print_bericht();
3  int main() {
4      std::cout
5          << "Main: [" << bericht << "]"
6          << " @ " << (size_t) bericht << "\n";
7      print_bericht();
8  }
```



bericht.hh

```
1  char bericht[]; // declaratie (makkelijk)
```

bericht.cc

```
1  #include "bericht.hh"
2  char bericht[] = "Hello" // definitie
3  void print_bericht() {
4      std::cout
5          << "Func: [" << bericht << "]"
6          << " @ " << (size_t) bericht << "\n";
7  }
```

stdout

```
1  Main: [Hello] @ 102032131358752
2  Func: [Hello] @ 102032131358752
```

- Een kopie in de sources
- Een kopie in de executable
- Geen onenigheid!

DEVCONTAINERS & ONTWIKKELOMGEVING

Wat zijn devcontainers?

- Docker containers die als complete ontwikkelomgeving fungeren.
- Bieden een consistente setup voor alle teamleden.
- Verminderen "it works on my machine" problemen.

Hoe werken ze?

- Gebruik de Remote - Containers extensie in VSCode.
- Start een devcontainer via de command palette.
- Gebruik de "Dev Container: New C++ Project" template als basis.

CMAKE

Waarom CMake?

- Cross-platform build systeem.
- Beheert buildprocessen via `CMakeLists.txt`.
- Integreert met VSCode voor betere code-completion en debugging.

Voorbeeld van een CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.10)
project(HelloWorld)

# Zorgt voor het genereren van compile_commands.json voor IntelliSense
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

add_executable(hello main.cc)
```

CONFIGURATIE CMAKE VOOR AUTOCOMPLETE

Stap-voor-stap:

1. CMakeLists.txt aanpassen: Voeg `set(CMAKE_EXPORT_COMPILE_COMMANDS ON)` toe.

2. Build directory maken:

```
mkdir build && cd build
cmake ..
```

3. VSCode instellen:
   Open de Command Palette (`Ctrl+Shift+P`) en kies `C/C++: Edit Configurations (UI)` om het pad naar `build/compile_commands.json` in te stellen.

DEBUGGEN IN C++

Basisprincipes:

- Breakpoints: Stop de uitvoering op een specifieke regel.
- Step Over/Into: Voer code regel voor regel uit.
- Variabelen inspecteren: Bekijk de waarde van variabelen tijdens runtime.

Voorbeeldcode:

```cpp
#include <iostream>

int main() {
    int a = 5;
    int b = 0;
    // Zet hier een breakpoint
    int c = a / b; // Fout: deling door nul
    std::cout << "Result: " << c << "\n";
    return 0;
}
```

\0