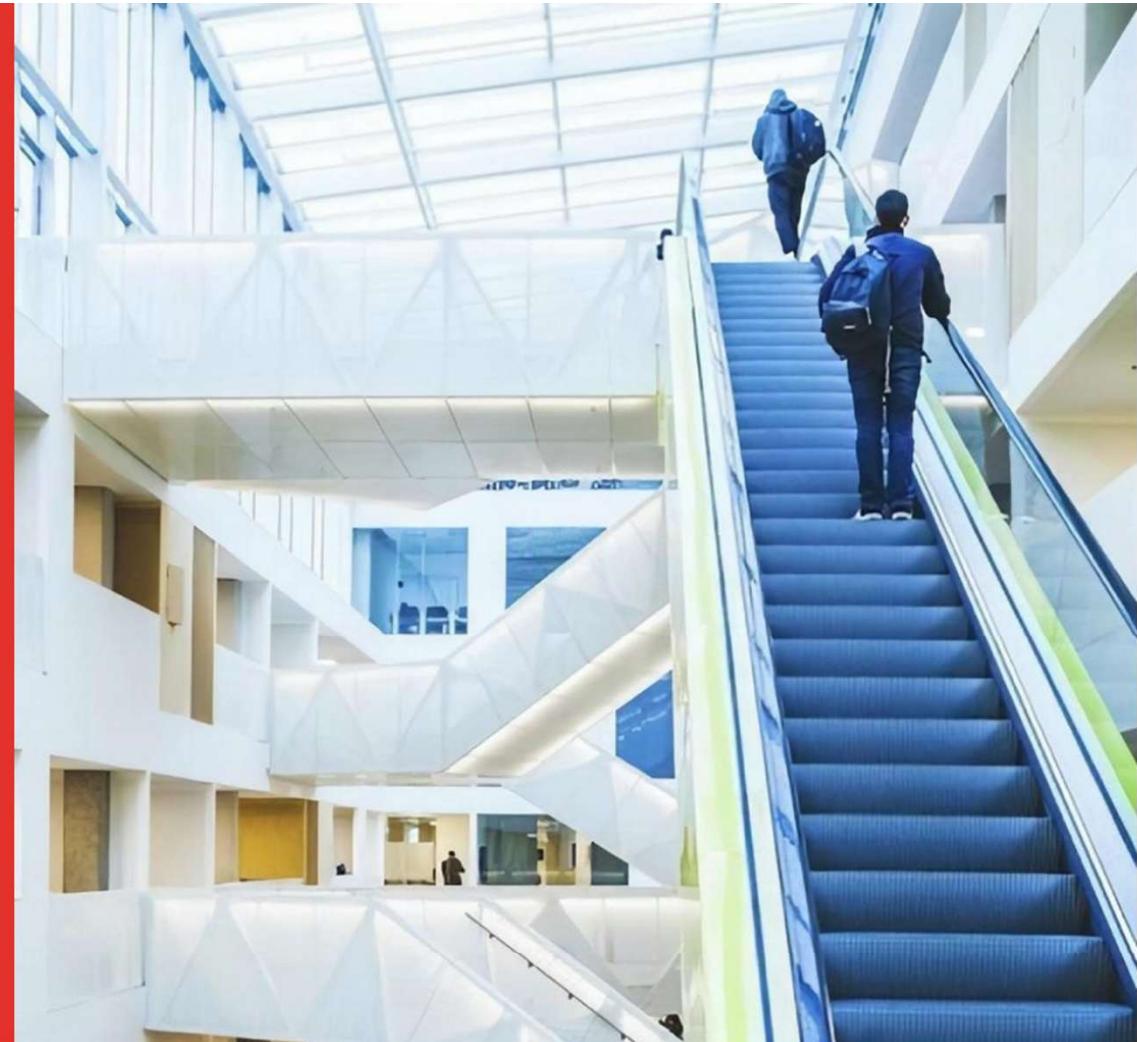


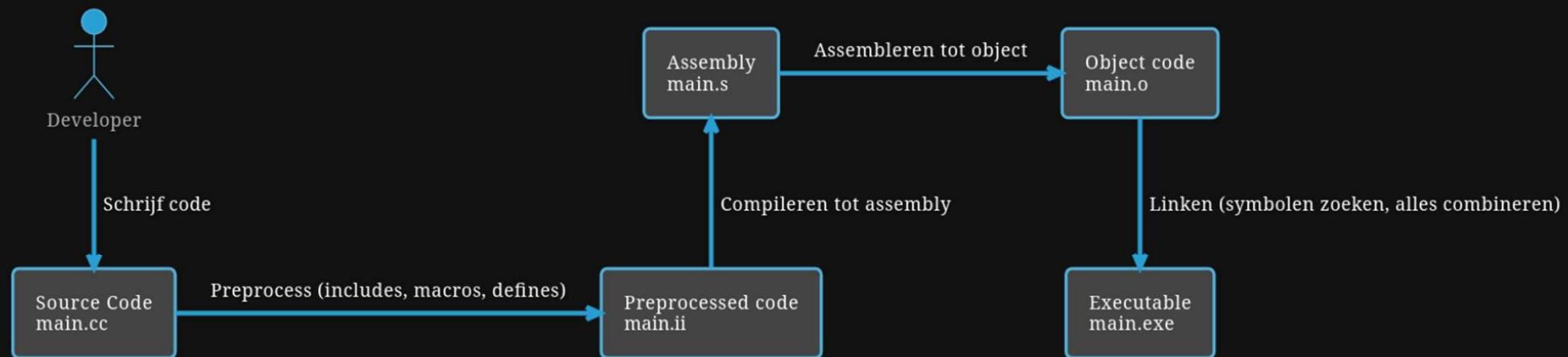
EEN APPLICATIE BOUWEN

- 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.
- Verwijdt 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

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!

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!

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 #define SQUARE(x) ((x) * (x))
```

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

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?

WAAROM IS DIT EEN SLECHT IDEE?

Ze zijn geen onderdeel van de taal

WAAROM IS DIT EEN SLECHT IDEE?

Ze zijn geen onderdeel van de taal

Je kan ze niet debuggen

WAAROM IS DIT EEN SLECHT IDEE?

Ze zijn geen onderdeel van de taal

Je kan ze niet debuggen

Vervelende side effects

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

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

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 #define DEBUG
2
3 int main() {
4 #ifdef DEBUG
5     std::cout << "Debug mode aan" << "\n";
6 #else
7     std::cout << "Release mode aan" << "\n";
8 #endif
9     return 0;
10 }
```

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

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

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 buildsystem), zonder deze in de code te zetten

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

VOORWAARDELIJKE COMPILATIE: VOORBEELD MET LOGGING

```
1 #ifdef DEBUG
2     #define LOG(msg) std::cout << "DEBUG: " << msg << "\n";
3 #else
4     #define LOG(msg)
5 #endif
6
7 int berekenFactorial(int n) {
8     LOG("Start berekenFactorial met n = " << n);
9     int result = 1;
10    for (int i = 1; i ≤ n; ++i) {
11        result *= i;
12        LOG("Intermediate result at i = " << i << " is " << result);
13    }
14    LOG("Eind berekenFactorial met result = " << result);
15    return result;
16 }
17
18 int main() {
19     std::cout << "Factorial van 5 is " << berekenFactorial(5) << "\n";
20     return 0;
21 }
```

INCLUDE STATEMENTS

Wat doet `#include` ?

- Het voegt de inhoud van een opgegeven bestand in op de plaats van het `#include` statement
- Maakt code modular 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):

```
1 #include <iostream> // let op wat hier gebeurd
2
3 // Inhoud van myheader.h
4 #ifndef MYHEADER_H
5 #define MYHEADER_H
6
7 #include <iostream> // let op wat hier gebeurd
8 void printMessage() {
9     std::cout << "Hello from myheader!\n";
10 }
11
12 #endif // MYHEADER_H
13
14 int main() {
15     printMessage();
16     return 0;
17 }
```

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

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

```
1 add(int, int):  
2     push    rbp  
3     mov     rbp, rsp  
4     mov     DWORD PTR [rbp-4], edi  
5     mov     DWORD PTR [rbp-8], esi  
6     mov     edx, DWORD PTR [rbp-4]  
7     mov     eax, DWORD PTR [rbp-8]  
8     add     eax, edx  
9     pop     rbp  
10    ret
```

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
2
3 Disassembly of section .text:
4
5 0000000000000000 <_start>:
6     0: b8 01 00 00 00      mov    eax,0x1
7     5: bf 01 00 00 00      mov    edi,0x1
8     a: 48 be 00 00 00 00 00  movabs rsi,0x0
9     11: 00 00 00
10    14: ba 0e 00 00 00      mov    edx,0xe
11    19: 0f 05                syscall
12    1b: b8 3c 00 00 00      mov    eax,0x3c
13    20: bf 00 00 00 00      mov    edi,0x0
14    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

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

```
1 ; Assembler
2 print_char: ...
3 print_int: ...
4 print_string: ...
```

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

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

```
1 ; Assembler
2 _Z5printc: ...
3 _Z5printi: ...
4 _Z5printPKc: ...
```

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

```
1 // C++
2 extern "C" void print_char(char c) {...}
3 extern "C" {
4     void print_int(int x) {...}
5     void print_string(const char *s) {...}
6 }
```

```
1 ; Assembler
2 print_char: ...
3
4 print_int: ...
5 print_string: ...
6
```

LINKING

main.cc

```

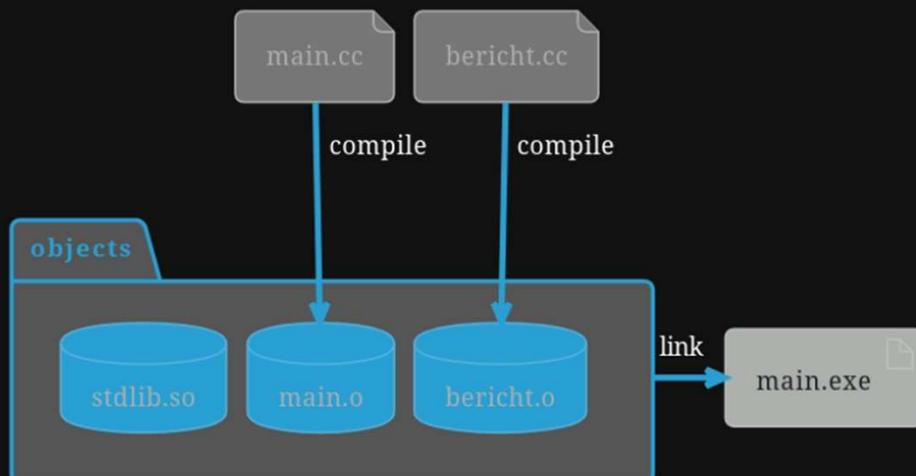
1 void print_bericht();
2 char bericht[] = "Hello";
3 int main() {
4     std::cout
5         << "Main: [" << bericht << "]"
6         << " @ " << (size_t)bericht << "\n";
7     print_bericht();
8 }
```

bericht.cc

```

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

stdout



```

1 Main: [Hello] @ 102032131358752
2 Func: [Hello] @ 102032131358758

```

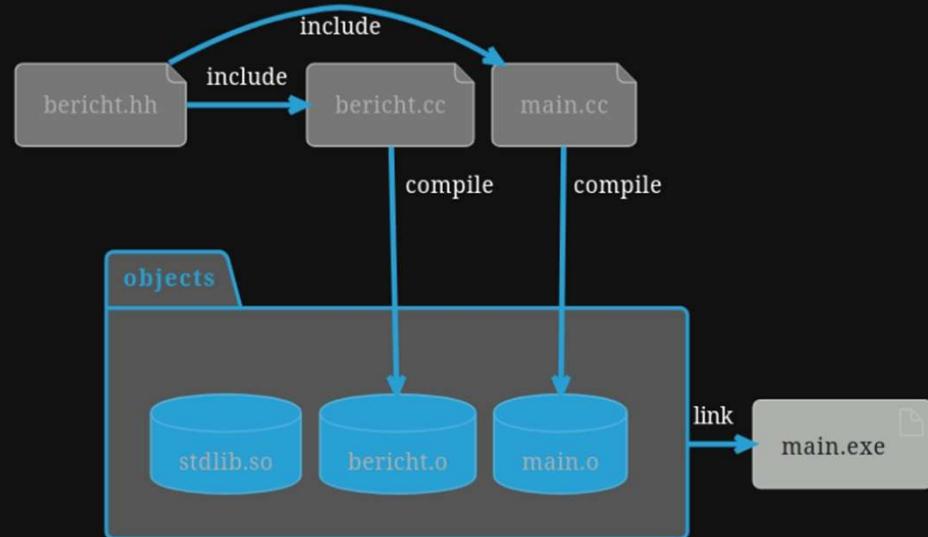
- Twee kopieën in de sources
- Twee kopieën in de executable

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[] = "Hello";
```

bericht.cc

```

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

stdout

```

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

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

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

```

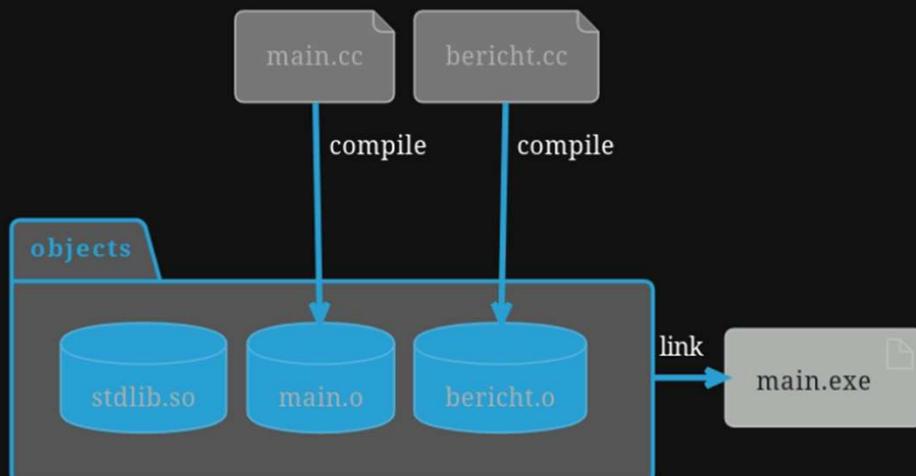
1 char bericht[] = "Hello";
2 void print_bericht() {
3     std::cout
4         << "Func: [" << bericht << "]"
5         << " @ " << (size_t)bericht << "\n";
6 }
```

stdout

```

1 Main: [Hello] @ 102032131358752
2 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

```

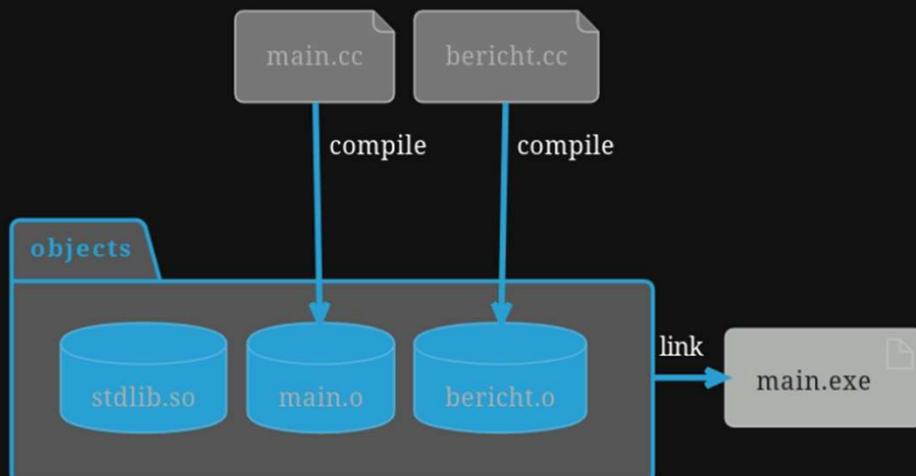
1 int bericht = 42;
2 void print_bericht() {
3     std::cout
4         << "Func: [" << bericht << "]"
5         << " @" << (size_t)bericht << "\n";
6 }
```

stdout

```

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

- Voor de linker is een label maar een geheugenplekje

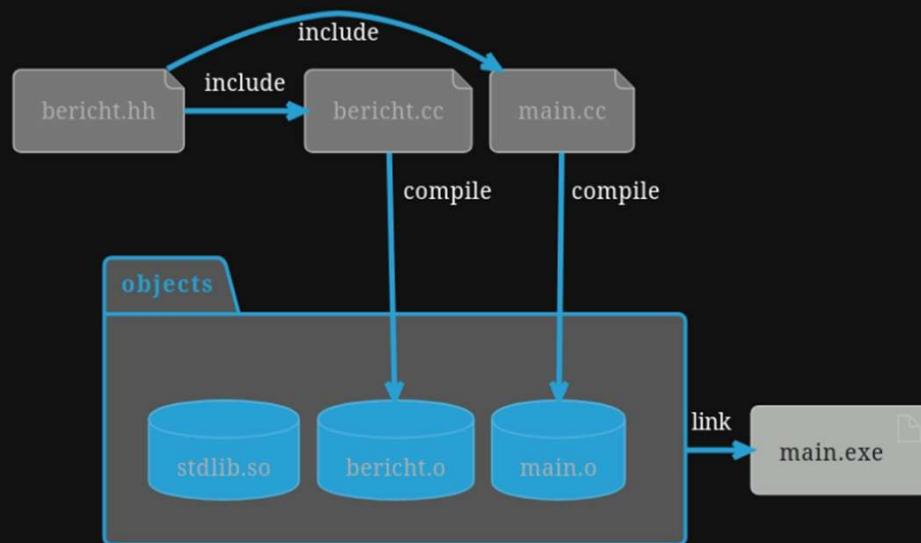


LINKING

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`:

```
1 cmake_minimum_required(VERSION 3.10)
2 project(HelloWorld)
3
4 # Zorgt voor het genereren van compile_commands.json voor IntelliSense
5 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
6
7 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:

```
1 #include <iostream>
2
3 int main() {
4     int a = 5;
5     int b = 0;
6     // Zet hier een breakpoint
7     int c = a / b; // Fout: deling door nul
8     std::cout << "Result: " << c << "\n";
9     return 0;
10 }
```

\0