# C++ 1 Les 1

Introductie

Header/Source Files

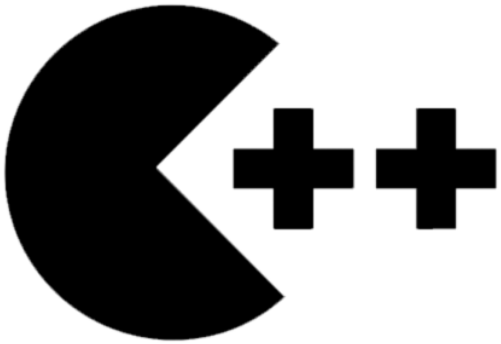
Compilatie

# Leerdoelen

Aan het eind van deze workshop kun je:

- uitleggen hoe deze cursus is opgebouwd
- uitleggen wat het verschil is tussen header en source files
- het process van preprocessen, compileren, en linken beschrijven.
- beschrijven wat de functie is van de door de compiler gegenereerde files
- uitleggen hoe een *linker error* verschilt van een *compile error*

# Introductie



# Waarom C++ ?

- Je wordt er een **betere programmeur** door
- Voor goed C++ heb je inzicht nodig in **geheugen-gebruik**, en moet je dat **zelf managen**
- Modern C++ biedt uitstekende hulpmiddelen om resource management "pijnloos" te doen
- C++ wordt nog steeds **veel gebruikt**, ook voor nieuwe projecten
- C++ is de taal waarmee je de **snelst executerende** software schrijft
- Met C++ heb je **directe toegang tot system calls**

"With great power comes great responsibility"

# Opzet CPPLS1

- **2x** per week hoorcollege van ieder 1 lesuur alleen de eerste 3 weken
  - stukje theorie
- **2x** per week practicum van ieder 2 lesuren alleen de eerste 3 weken
  - werken aan opdrachten (later: eindopdracht, in duo's)
- 1x hoorcollege in week 4 met vooruitblik op Smart Pointers
- 1x practicum in week 4
- eindcijfer volgt uit mondeling assessment, waarin je de eindopdracht laat zien
- CPPLS1 = 3 EC

project van de minor wordt ook in C++ gebouwd

# Periode 1, Blok 13: CPPLS1

- Basisbeginselen C++

- Leren hoe je zelf resource management doet, vooral memory management
- Hoe bouw je correcte C++-classes hiervoor?

- Raw Pointers

(en een vooruitblik op Smart Pointers)

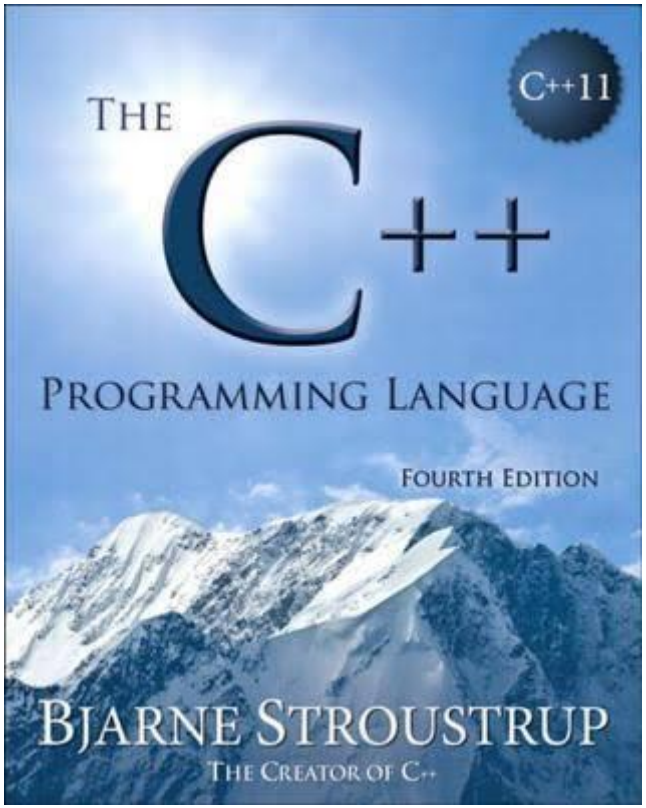
# Opzet CPPLS2

- **1x** per week hoorcollegge van 1 lesuur
  - stukje theorie
- **1x** per week practicum van 2 lesuren
  - werken aan opdrachten (later: eindopdracht, in duo's)
- eindcijfer volgt uit mondeling assessment, waarin je de eindopdracht laat zien
- CPPLS2 = 4 EC

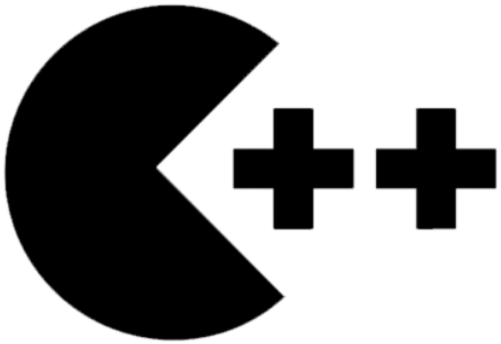
# Periode 2, Blok 14: CPPLS2

- C++ Standard Library
- Geen resource management zelf meer doen, maar overlaten aan utility classes uit de library
- Smart Pointers

# Boek is verplicht!



# Header en Source Files



# Header en Source files

- Header file
  - *.h
  - Ook wel bekend als *Include file*
  - Declaration
- Source file
  - *.cpp
  - Definition (implementatie)

Stop dus niet zoals bij C# of Java één klasse in één file!

# Voorbeeld header file

```
#pragma once //komen we later op terug

class MyClass { //declaratie van een class
public:
	void foo(); //declaratie van een functie

private:
	int bar(int a); //nog een functie declaratie

	int i; //declaratie van een member variable
};
```

# Voorbeeld source file

```
#include “myclass.h” //Deze header file is nodig in
                     //deze source file

void MyClass::foo() //implementatie van eerste functie
{
        i = bar(2);
}

int MyClass::bar(int a) //implementatie van tweede functie
{
        return a + i;
}
```

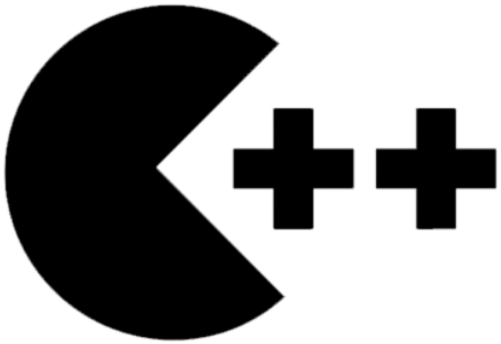
# Voordelen van Header Files

- Het scheidt interface van implementatie

# Nadelen van Header Files

- Complexer
  - 2x zoveel files in je project
  - informatie over een klasse staat verspreid over 2 files (minder overzichtelijk)

# Compilatie



# Compileren: Het hele proces

1. Preprocessen
2. Compilen
   - Precompiled headers
3. Linken
   - Linker errors

# Preprocessing

In een source code file (*.cpp) worden *preprocessor directives* verwerkt:

- macro definitions (#define, #undef)
- Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)
- Error directive (#error)
- Source file inclusion (#include)
- Pragma directive (#pragma)
- Predefined macro names (__LINE__, __FILE__, __DATE__, etc.)
- etc.

# Preprocessing

De #include files (en andere directives) worden “uitgepakt” zodat er één file van gemaakt wordt.

Zo’n file wordt gemaakt per source file (*.cpp).

De ontstane file dient als input voor de compiler.

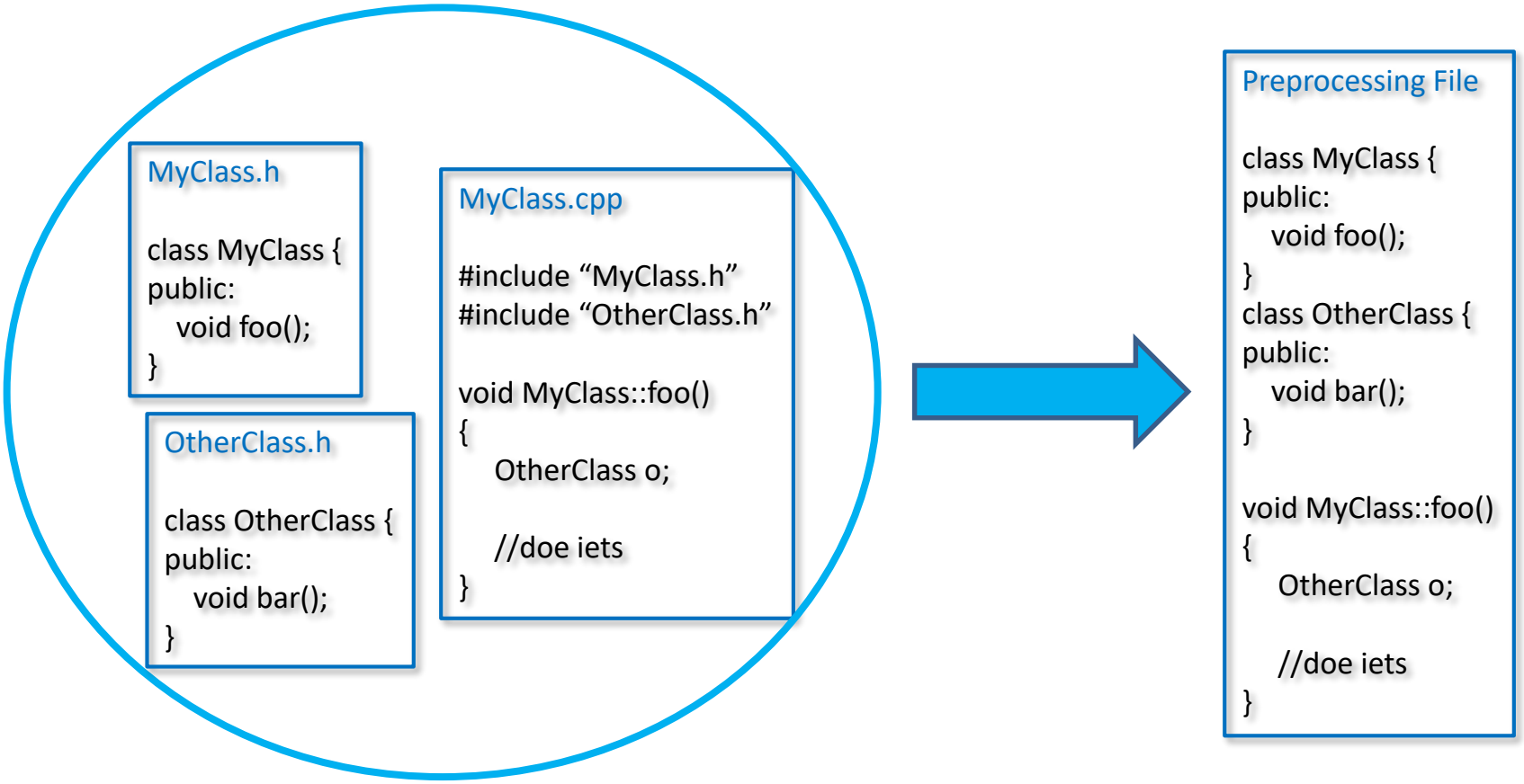
# Preprocessing

Bijvoorbeeld:

Preprocessor output bij Microsoft Visual C++ afhankelijk van compiler opties:

- /E: preprocess to stdout
- /P: preprocess to file
- /EP: preprocess to stdout without #line directives

# Preprocessing



# Compilation

Preprocessing File

```
class MyClass {
public:
   void foo();
}
class OtherClass {
public:
   void bar();
}

void MyClass::foo()
{
    OtherClass o;

    //doe iets
}
```

MyClass.o

# Compilation

De compiler neemt de file van de preprocessor en compileert dit naar een object file.

- *.o voor CLang
- *.o voor GCC
- *.Obj voor MSVC.

Je hebt dus na het compileren een verzameling *.o of *.Obj files die corresponderen met je source files.

# Precompiled headers

MSVC: *.pch (#include “stdafx.h”)

GCC: *.gch

Eerste keer wordt alles in de precompiled headers gecompileerd.

Daarna niet meer, zolang niets in de precompiled headers veranderd is.

Handig bij grote includes die (bijna) nooit veranderen. Zoals Windows.h of de STL.

Compilatie tijd vermindert omdat alles in de precompiled header niet steeds opnieuw gecompileerd hoeft te worden.

# Precompiled headers

Precompilen:

1. Preprocessing
2. Eerste stap van compilation (tokenizen)
   - Er wordt dus geen *.o aangemaakt maar bijvoorbeeld een *.pch file

Compilen (van source file):

1. Preprocessing
2. Eerste stap van compilation
   a. Kijk wat er al in de pch voorkomt
   b. Tokenize de rest wat daar niet in voorkomt
3. Maak compilation af om de *.o file te maken

# Linking

De linker pakt de object files en maakt daarvan:

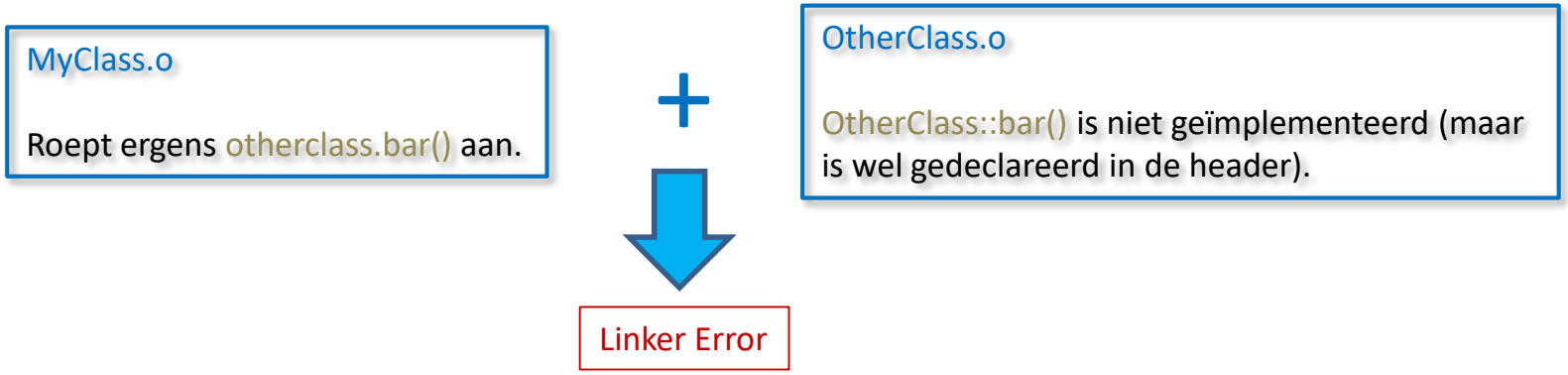
- Executable
- Dynamic (Link) Library
- Static Library

# Linker Errors

De header file “belooft” iets, namelijk dat er een implementatie is van alles wat er in gedeclareerd wordt.

Bij het linken wordt die belofte waar gemaakt: de beloofde implementatie wordt opgezocht in de betreffende object file.

Ontbreekt die implementatie: Linker Error



# Hele process voor CLang

- Preprocessing
  - output: *.i voor C, *.ii voor C++, *.mi voor Objective-C, *.mii voor Objective-C++
- Parsing and Semantic Analysis
  - output: Abstract Syntax Tree (AST)
- Code Generation and Optimization
  - output: Assembly file (*.s)
- Assembler
  - output: Object file (*.o)
- Linker
  - output: Executable of Dynamic Library (a.out, *.dylib, *.so)

Standaard executable naam. Wordt normaalgesproken door de linker hernoemd naar iets anders.