

OBJEKTORIENTIERTES PROGRAMMIEREN MIT C++, SOMMERSEMESTER 2018

Ihr Betreuer:

Thomas Kammerer



thomas.kammerer@th-nuernberg.de

Studium der Daten- und Informationstechnik an der FH Nürnberg.

Langjährige Erfahrung im Bereich Messtechnik, Medizintechnik und SW-Engineering.

- Entwicklung von Software für bildgebende medizinische Systeme,
- Teamleiter Medizinproduktsoftware,
- Abteilungsleiter Forschung & Entwicklung,
- Chief Technology Officer der ASTRUM IT GmbH,
- Berater für Software-Engineering mit Schwerpunkt Medizinprodukte.

Abbildung 1: Experienced developer 😊

HELLO WORLD

Natürlich fangen auch wir mit dem „Hello-World –Programm“ an – jeder fängt damit an: irgendjemand in den Bell Labs schrieb einst „Hello World“ und daraus entwickelten sich die heutigen Programmierer.



Abbildung 3: Very young developer

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello C++ World!" << std::endl;
5      return 0;
6  }
```

Abbildung 2: Very small C++ program

Laden Sie von github https://github.com/thkammerer/cpp_summer_course_2018 herunter und entzippen Sie in ein von Ihnen gewähltes Arbeitsverzeichnis.

Wechseln Sie : z.B. in `cpp_course_summer_2018/000_intro/000_001_hello_world`

Legen Sie dort zwei Unterverzeichnisse an! `/src` und `/bin`

Legen Sie im `src`-Verzeichnis die Datei `hello.cpp` an!

Jetzt müssen wir den Code reinschreiben, können Sie vorher erklären, was der macht?

Erklären Sie den Code in Abbildung 2!

`#include <iostream>`: _____

`int main()`: _____

`std::cout << "Hello C++ World!" << std::endl;` _____

`return 0;` _____

Falls Sie den Code nicht erklären können, dann fragen Sie bitte nach!

Schreiben Sie mit einem Editor ihrer Wahl (am Besten nehmen Sie den wirklich guten „Atom“ Editor (<http://atom.io/>) – leider muss der noch installiert werden) das Programm!

Jetzt müssen wir den Code übersetzen, linkern und dann noch ausführen.

DAS COMPILIEREN VON C++ PROGRAMMEN

Im Praktikum verwenden wir den Gnu-C++-Compiler g++ - auf den Rechner befindet sich irgendwo mingw (mingw steht für „**minimal gnu for windows**“ und im bin-Verzeichnis finden wir auch den Compiler g++.exe.

Suchen Sie ihren auf Ihren Rechner den Gnu C++ Compiler g++!

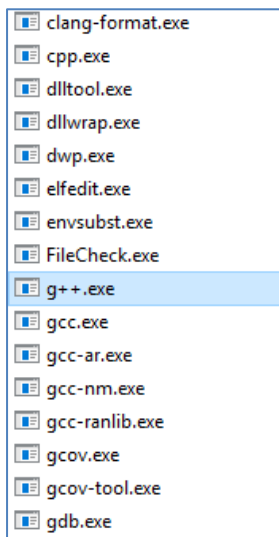


Abbildung 4: g++.exe

Jetzt wollen wir den Compiler auch nutzen! Dazu brauchen wir eine Shell. Bei Windows nutzen wir das sog. „Command Prompt“ (oder MSYS bzw. Cygwin die eine Unix/Linux-Umgebung auf Windows schaffen). Schauen wir doch mal, ob wir den Gnu-Compiler aufrufen können:

Tippen Sie: `g++ --version`

Bei mir sieht das ungefähr so aus:

```
C:\>g++ --version
g++ (Rev1, Built by MSYS2 project) 6.3.0
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

C:\>
```

Abbildung 5: g++ --version

Wahrscheinlich geht das bei Ihnen nicht. Das liegt daran, dass die PATH-Umgebungsvariable nicht auf den Pfad für den Compiler zeigt. Da wir jetzt nur kurz bisschen rumspielen wollen, setzen wir die Umgebungsvariable lokal im offenen Command Prompt.

Dazu stellen wir im Command Prompt ein paar Sachen ein:

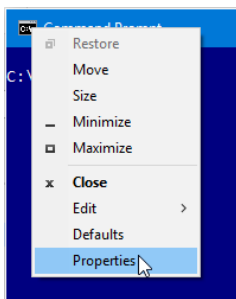


Abbildung 6: Cmd-Properties

Rufen Sie die Properties ihres Command-Prompts auf:

Unbedingt anhaken:

- Quick-Edit Mode
- Insert Mode

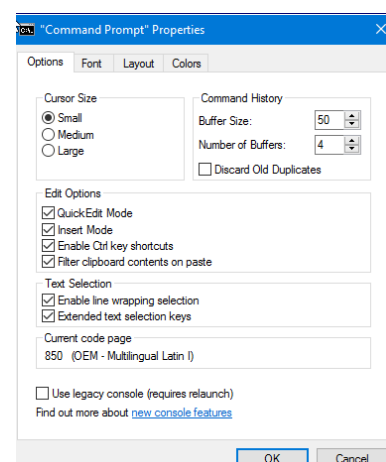


Abbildung 7: Cmd Shell Properties

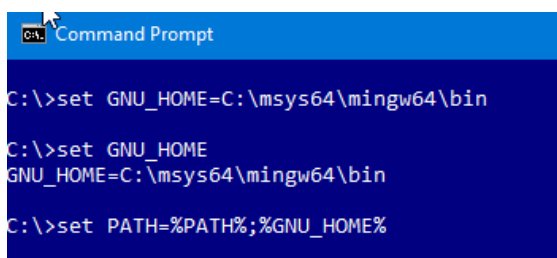
Jetzt die PATH-Variable erweitern:

Schauen Sie sich den Inhalt der PATH-Variable an: Tippen Sie hierzu set path!

Holen Sie sich den Pfad zum Gnu-Compiler (bei mir ist das z.B. C:\msys64\mingw64\bin) in die Zwischenablage.

Setzen Sie eine Umgebungsvariable GNU_HOME auf den Pfad.

Hierzu tippen Sie set GNU_HOME="....ihr Pfad..."



```
C:\>set GNU_HOME=C:\msys64\mingw64\bin
C:\>set GNU_HOME
GNU_HOME=C:\msys64\mingw64\bin
C:\>set PATH=%PATH%;%GNU_HOME%
```

Abbildung 8: Erweiterung der PATH Variablen

Schauen Sie, ob das Setzen geklappt hat: set GNU_HOME gibt jetzt den von Ihnen gesetzten Pfad aus

Erweitern Sie jetzt die PATH-Variable: set PATH=%PATH%;%GNU_HOME%

Achten Sie auf die %-Zeichen! Überprüfen Sie jetzt, ob Ihr PATH auch erweitert wurde!

Jetzt sollte g++ --version klappen!

Welche Version haben Sie?

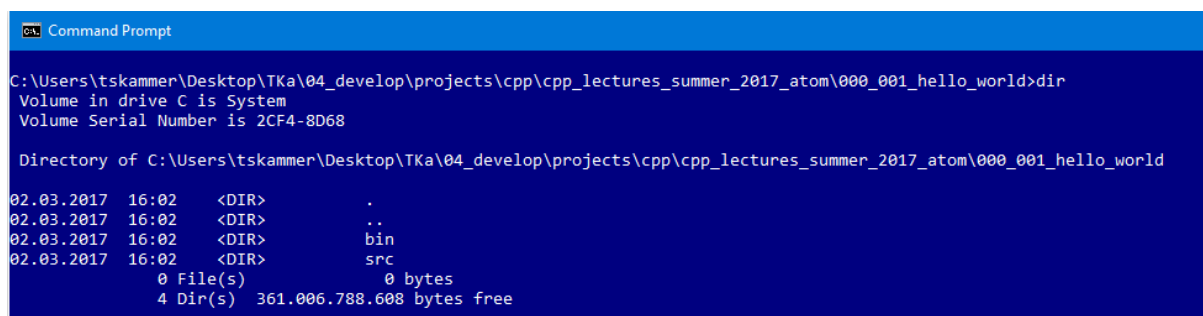
Meine g++ Version: _____

So, nun geht es ans Übersetzen!

Wechseln Sie dazu im Command-Prompt in ihr Projektverzeichnis:

cd <pfad in ihr projektverzeichnis>

danach mit „dir“ den Verzeichnisisinhalt ausgeben lassen



```
C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_001_hello_world>dir
Volume in drive C is System
Volume Serial Number is 2CF4-8D68

Directory of C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_001_hello_world

02.03.2017  16:02    <DIR>        .
02.03.2017  16:02    <DIR>        ..
02.03.2017  16:02    <DIR>        bin
02.03.2017  16:02    <DIR>        src
               0 File(s)              0 bytes
               4 Dir(s) 361.006.788.608 bytes free
```

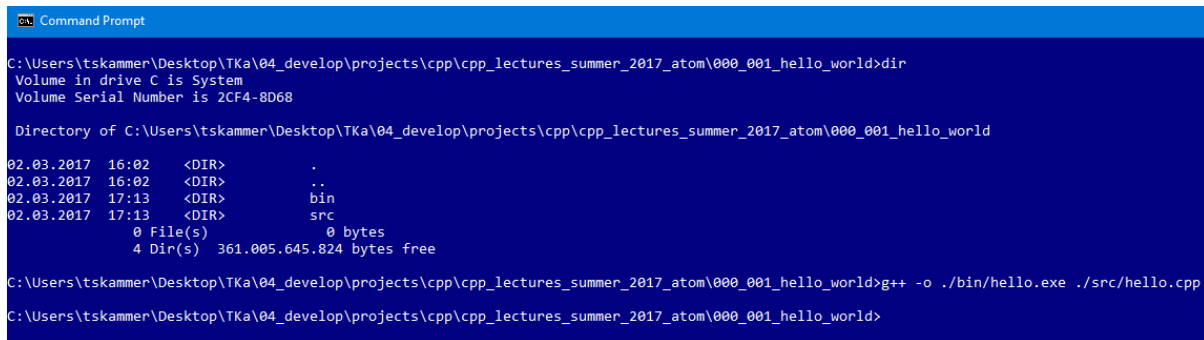
Abbildung 9: Command-Prompt im Projektverzeichnis

Jetzt übersetzen wir die Datei `./src/hello.cpp` und schreiben das Ergebnis `hello.exe` ins `bin`-Verzeichnis:

Führen Sie aus:

```
g++ -o ./bin/hello.exe ./src/hello.cpp
```

die Option `-o` weist den Compiler an, wo er das Output-File (hier `hello.exe`) hinschreiben soll.



```
Command Prompt
C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_001_hello_world>dir
Volume in drive C is System
Volume Serial Number is 2CF4-8D68

Directory of C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_001_hello_world
02.03.2017  16:02    <DIR>        .
02.03.2017  16:02    <DIR>        ..
02.03.2017  17:13    <DIR>        bin
02.03.2017  17:13    <DIR>        src
               0 File(s)              0 bytes
               4 Dir(s) 361.005.645.824 bytes free

C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_001_hello_world>g++ -o ./bin/hello.exe ./src/hello.cpp
C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_001_hello_world>
```

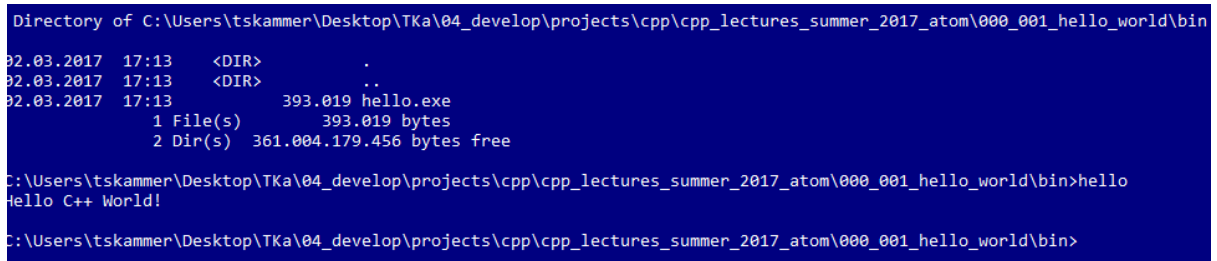
Abbildung 10: kompilieren!

Wechseln Sie jetzt ins bin Verzeichnis!

```
cd ./bin
```

Dort finden Sie - wenn alles geklappt hat – `hello.exe` (dir –Kommando verwenden!)

Führen Sie hello.exe aus: heLLo



```
Directory of C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_001_hello_world\bin
02.03.2017  17:13    <DIR>        .
02.03.2017  17:13    <DIR>        ..
02.03.2017  17:13             393.019 hello.exe
               1 File(s)              393.019 bytes
               2 Dir(s) 361.004.179.456 bytes free

C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_001_hello_world\bin>hello
Hello C++ World!

C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_001_hello_world\bin>
```

Abbildung 11: hello C++ World!

Was ist da alles passiert? Der Befehl `g++ -o ./bin/hello.exe ./src/hello.cpp` macht eine ganze Menge:

1. **Preprocessing:** er führt den Präprozessor auf (includes, compilation instructions, macros)
2. **Compilation:** Er übersetzt den C++-Source-Code in Assembler-Source-Code
3. **Assembly:** Er übersetzt den Assembler-Code in Object-Code (das ist schon Binärcode)
4. **Linking:** Er nimmt einen (oder mehrere) Object-Code-Files und erzeugt eine ausführbare Datei (unter Windows normalerweise eine *.exe)

C++ Program Build Process

for the example: hello.cpp

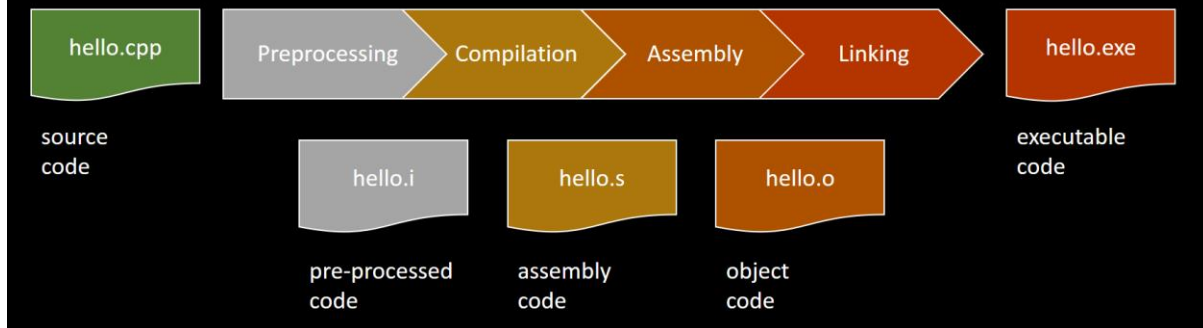


Abbildung 12: Build Process

Das macht der Compiler alles in einem Schritt und er gibt nur die exe aus.

Wenn wir uns die Zwischenergebnisse anschauen wollen können wir die Compiler-Optionen entsprechend setzen:

```
g++ -E -o .\bin\hello.i .\src\hello.cpp -E: Preprocess only; do not compile, assemble or link.
g++ -S -o .\bin\hello.s .\src\hello.cpp -S: compile only; do not assemble or link.
g++ -c -o .\bin\hello.o .\src\hello.cpp -c: Compile and assemble, but do not link.
```

Abbildung 13: Compiler Flags

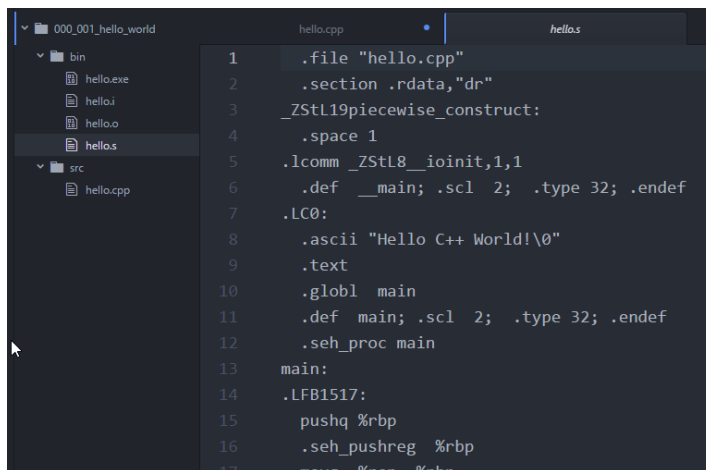


Abbildung 14: Assembler File hello.s

ÜBERSETZEN MIT HEADER-DATEI UND ZWEI CPP-DATEIEN UND MAKE

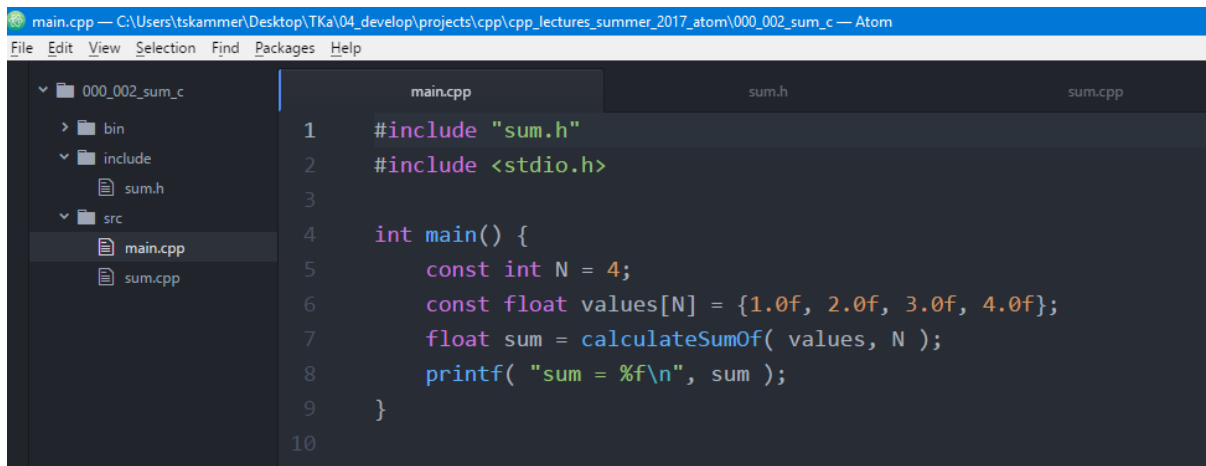
Jetzt wird es etwas spannender: Wir berechnen die Summe aus ein paar Zahlen! ☺

Wir wollen hier gleich drei Dateien anlegen:

1. **main.cpp** für die Ein- und Ausgabe
2. **sum.h** für die **Deklaration** der Summenfunktion
3. **sum.cpp** für die **Definition** (= Implementierung) der Summenfunktion

Damit wir hier mit Tipparbeit Zeit verschwenden, wurde schon mal was vorbereitet:

Im Projektordner „....\cpp_course_summer_2017_students\000_introduction\000_002_sum_c“ finden Sie den Ordner 000_002_sum_c mit Unterordnern bin, include und src. Schauen Sie sich den Inhalt der Dateien an.



The screenshot shows the Atom code editor interface. On the left, a file explorer pane displays the project structure for '000_002_sum_c', including subdirectories 'bin', 'include', and 'src', and files 'main.cpp' and 'sum.cpp'. The main editor area shows the code in 'main.cpp' with line numbers 1 through 10. The code includes headers for 'sum.h' and 'stdio.h', defines a constant 'N' as 4, initializes an array of floats, calls a function 'calculateSumOf', and prints the result.

```
1 #include "sum.h"
2 #include <stdio.h>
3
4 int main() {
5     const int N = 4;
6     const float values[N] = {1.0f, 2.0f, 3.0f, 4.0f};
7     float sum = calculateSumOf( values, N );
8     printf( "sum = %f\n", sum );
9 }
10
```

Abbildung 15: Projektstruktur 000_002_sum und main.cpp

Wenn Sie sich im Ordner „....\cpp_course_summer_2017_students\000_introduction\000_002_sum_c“ übersetzen und Linken Sie in einem Zug mit:

`g++ -I .\include\ -o .\bin\sum.exe .\src\main.cpp .\src\sum.cpp`

Erforschen Sie, was die Option `-I` bedeutet: _____

Was passiert, wenn man die Option `-I .\include` weglässt? _____

Führen Sie nun sum.exe aus!

```
PS C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_002_sum_c\bin> .\sum.exe
sum = 10.000000
PS C:\Users\tskammer\Desktop\TKa\04_develop\projects\cpp\cpp_lectures_summer_2017_atom\000_002_sum_c\bin>
```

Abbildung 16: Ausführung von sum.exe

Versuchen Sie nun ein Assemblerfile zu erzeugen!

In Abbildung 17: Assembler-File sum.s Abbildung 17 sehen Sie zum Beispiel den Anfang von sum.s.

	main.cpp	sum.s	sum.h
1		.file "sum.cpp"	
2		.text	
3		.globl _Z14calculateSumOfPKfi	
4		.def _Z14calculateSumOfPKfi; .scl 2; .type 32; .endef	
5		.seh_proc _Z14calculateSumOfPKfi	
6		_Z14calculateSumOfPKfi:	
7		.LFB0:	
8		pushq %rbp	
9		.seh_pushreg %rbp	
10		movq %rsp, %rbp	
11		.seh_setframe %rbp, 0	
12		subq \$48, %rsp	
13		.seh_stackalloc 48	
14		movaps %xmm6, -16(%rbp)	
15		.seh_savexmm %xmm6, 32	
16		.seh_endprologue	
17		movq %rcx, 16(%rbp)	
18		movl %edx, 24(%rbp)	
19		movl 24(%rbp), %eax	

Abbildung 17: Assembler-File sum.s

MAKE, CMAKE UND ANDERE...

Sie können sich jetzt sicher vorstellen, dass es extrem schwierig wird, die richtigen Compileraufrufe durchzuführen, wenn wir ein größeres Projekt mit vielen Source-Code-Dateien, externen Libraries usw. haben.

Hinzu kommt, dass wir oft mehrere sog. Build-Targets haben. Wir erzeugen z.B. ein Debug-Target, ein Release-Target, ein Test-Target um Unit- und Integrationstest automatisiert auszuführen; oder wir müssen unterschiedliche Prozessorversionen unterstützen. Unser Programm besteht dabei aus mehreren statischen oder dynamischen Bibliotheken (jede .lib oder .dll ist wiederum ein eigenes Build-Target). All das macht das Bauen unserer Software nicht gerade einfach.

Um diese Arbeiten zu unterstützen und die Abhängigkeiten zwischen verschiedenen Build-Targets zu verwalten, gibt es verschiedene Werkzeuge: Build-Management-Tools. Ein sehr bewährtes Werkzeug ist das **make**-Tool, das bereits seit 1977 im Einsatz ist und auch heute noch verwendet wird. Es gehört zum Posix-Standard und ist damit auf Unix- und Linux-Systemen verbreitet.

Inzwischen gibt es eine ganze Reihe von Derivaten und Alternativen:

- aus make abgeleitet:
 - Gnu Make (Linux)
 - nmake (Microsoft)
 - mk
- SCons
- Apache ant, Apache maven
- Jam (Boost)
- Rake
- Meson
- und viele andere..

Besonders bei größeren Programmen werden die Make-Files nicht direkt geschrieben, sondern entweder mittels unterschiedlicher Programme aus einfacheren Regeln erstellt oder es werden mächtigere Systeme verwendet. Eine Methode zur automatischen Erzeugung von komplexen Makefiles ist die Verwendung von Präprozessoren wie den GNU autotools (autoconf und automake) oder auch mkmf, qmake oder CMake.

Wer hier mehr wissen will schaut in die WIKIPEDIA:

https://en.wikipedia.org/wiki/List_of_build_automation_software#Build_script_generation_tools

Natürlich schauen wir uns wenigstens ein (sehr) einfaches make-File an und übersetzen unser Summen-Programm mit dem make-Programm von mingw: **mingw32-make.exe**.

Makefiles bestehen aus Regeln (Rules), die in Form von:

```
target ... : prerequisites
            recipe
```

geschrieben werden.

Ein **target** ist üblicherweise der Name einer Datei, die von einem Programm erzeugt wird. hello.exe oder hello.o wären Kandidaten solcher targets. Weiterhin kann ein Target auch einfach der Name einer Aktion sein, die ausgeführt werden soll, solche Targets nennt man **Phony Targets** – „unechte Targets“. clean oder clean-all, ein Befehl, der alle erzeugten Files löscht, ist ein übliches Phony Target.

Prerequisites sind Dateien, die als Input genutzt werden, um das Target zu erzeugen.

Recipes sind Aktionen, die make ausführen soll. Ein Recipe kann eine oder mehrere Befehle haben; sie werden entweder in der gleichen Zeile aufgeführt oder in mehreren Zeilen. Achtung man muss dann immer ein TAB-Zeichen am Anfang der Zeile machen!

Wir schauen uns ein paar einfache Beispiele an:

LEERES MAKE FILE

Legen Sie in HOME/cpp_summer_2017/000_intro /000_002_sum_c ein File ein File mit dem Namen makefile an!

Im Command-Prompt gehen Sie in dieses Verzeichnis und führen aus: mingw32-make

Wird bei Ihnen

mingw32-make: *** No targets. Stop.

ausgegeben? Dann ist alles o.k.!

MAKE HELLO WORLD

Jetzt machen wir ein makefile, das Hello World auf das Command Prompt ausgiebt:

```
1 .PHONY: print
2 print:
3     @echo Hello World!
4
```

Abbildung 18: Hello World mit make

Schreiben Sie in das leere makefile die obigen Zeilen. Speichern sie das makefile und rufen sie mingw32-make print auf.

Die Regel print: wird ausgeführt: Es wird **Hello World!** ausgegeben. (.PHONY: print informiert make darüber, dass es sich bei print um ein reines Kommandotarget handelt, solange wir keine Dateien haben, die print heißen, könnte man das auch weglassen - aber wer stundenlange Fehlersuche vermeiden will, gewöhnt sich an, das bei PHONYs immer mit anzugeben!)

EIN EINFACHES REZEPT ZUM BAUEN

Wir erstellen jetzt ein einfaches make-Rezept.

Die Variable VPATH teilt make mit, in welchen Ordnern gesucht werden soll, um Dateien zu finden. Hier (Abbildung 19) teilen wir make mit, dass in ./src und ./include Unterverzeichnis gesucht werden soll.

Weiterhin definieren wir fünf Regeln.

all : sum.exe: es ist üblich die erste Regel als all: zu bezeichnen, diese wird immer als erste angestoßen. make prüft über die Abhängigkeiten, ob ein rebuild notwendig ist. das Target **all** ist abhängig von **sum.exe**. make schaut nun, ob es eine Regel für **sum.exe** gibt.

make findet eine Regel für **sum.exe**, für dieses Target ist als Voraussetzung sum.o und main.o angegeben. Bevor make nun das Rezept (g++ -o ./bin/sum.exe) ausführt, sucht es nach Regeln für sum.o und main.o.

```
1  VPATH = ./src ./include
2
3  all : sum.exe
4
5  sum.exe : sum.o main.o
6      g++ -o ./bin/sum.exe ./bin/sum.o ./bin/main.o
7
8  sum.o : sum.cpp sum.h
9      g++ -c -std=c++14 -Wall -I./include -o ./bin/sum.o ./src/sum.cpp
10
11 main.o : main.cpp sum.h
12     g++ -c -std=c++14 -Wall -I./include -o ./bin/main.o ./src/main.cpp
13
14 .PHONY : clean
15 clean:
16     del .\bin\*.exe .\bin\*.o
```

Abbildung 19: ein einfaches makefile

make findet die Regel sum.o, mit den Voraussetzungen sum.cpp und sum.h. Für diese gibt es keine Vorbedingungen, make findet über den Suchpfad (VPATH) die beiden Files, schaut, ob die neuer sind als ein bereits gebautes sum.o und führt ggf. das Rezept aus.

Versuchen Sie die Regeln nachzuvollziehen.

Schreiben Sie das obige makefile und übersetzen sie unser Projekt (mit mingw32-make)

REZEPTE MIT VARIABLEN

Um Redundanzen zu vermeiden, können wir weitere Variablen einführen. Variablen können einfach mit

```
1  OUT_PATH = ./bin
2  INC_PATH = ./include
3  SRC_PATH = ./src
4
5  CXX = g++
6  CXXFLAGS = -std=c++14 -Wall -I$(INC_PATH)
7  VPATH = $(SRC_PATH) $(INC_PATH)
8
9  TARGET = ./bin/sum.exe
10 OBJECTS = ./bin/sum.o ./bin/main.o
11
12 LINK = $(CXX)
13 COMPILE = $(CXX) -c $(CXXFLAGS)
```

Abbildung 20: Variablendefinition in makefile

ABC = xyz definiert werden.

Üblicherweise werden Variablen in Großbuchstaben geschrieben, muss aber nicht so sein abc = xyz klappt auch, allerdings ist ABC eine andere Variable als abc.

Will ich auf den Inhalt der Variablen zugreifen, muss ich \$(ABC) schreiben.

Versuchen Sie das makefile aus Abbildung 19 so umzuschreiben, dass jetzt Variablen, wie in Abbildung 20 angegeben benutzt werden.

AUTOMATIC VARIABLES, EINGEBAUTE REZEPTE UND FUNKTIONEN

make kann noch viel mehr, z.B. existieren eingebaute Rezepte und sog. „automatische Variablen“.



Wer alles über GnuMake (mingw32-make) wissen will klickt auf den stehenden Link:

<http://www.gnu.org/software/make/manual/>

```
1 #####
2 # compiler stuff
3 CXX = g++
4 CXXFLAGS = -std=c++14 -Wall
5
6 #####
7 # source files etc
8 TARGET = sum.exe
9 SRC_FILES = main.cpp sum.cpp
10 INC_FILES = sum.h
11 OBJ_FILES = $(SRC_FILES:.cpp=.o)
12
13 OUT_PATH = .\bin
14 SRC_PATH = .\src
15 INC_PATH = .\include
16
17 OBJ_FILES_WITH_PATH = $(addprefix $(OUT_PATH)/, $(OBJ_FILES) )
18 VPATH = $(SRC_PATH) $(INC_PATH) $(OUT_PATH)
19
20 #####
21 # build commands
22 CXXFLAGS += -I$(INC_PATH)
23 COMPILE = $(CXX) -c $(CXXFLAGS)
24 LINK = $(CXX)
25 OUTPUT_OPTION = -o $(OUT_PATH)\$@
26
27 #####
28 # rules
29
30 all : $(TARGET)
31
32 $(TARGET) : $(OBJ_FILES)
33     $(LINK) -o $(OUT_PATH)\$@ $(OBJ_FILES_WITH_PATH)
34
35 $(OBJ_FILES) : sum.h
36
37 .PHONY : clean clean_objects clean_exe
38 clean_all: clean_objects clean_exe
39
40 clean_objects:
41     del $(OUT_PATH)\*.o
42
43 clean_exe:
44     del $(OUT_PATH)\*.exe
45
46 .PHONY: print
47 print:
48     @echo compiler:          $(CXX)
49     @echo compiler flags:    $(CXXFLAGS)
50     @echo output dir:        $(OUT_PATH)
51     @echo src files:         $(SRC_FILES)
52     @echo obj files:         $(OBJ_FILES)
53     @echo build target:      $(TARGET)
54     @echo output option:     $(OUTPUT_OPTION)
```

Abbildung 21: make file nutzt automatic variables, implicit rules und function addprefix

Sie sehen schon: das ist ganz schön komplex. Viele Entwickler nutzen deshalb mächtigere Werkzeuge, die sie beim Erstellen von makefiles unterstützen. Z.B die Gnu autotools, cmake oder modernere Buildsysteme wie scons oder meson.

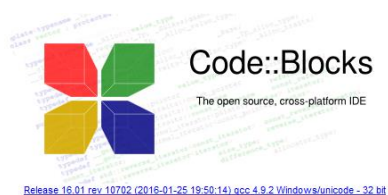
CODE::BLOCKS

Code::Blocks ist eine einfache, aber gute IDE. IDE steht für Integrated Development Enviroment. Viele Entwickler mögen so etwas, andere (vor allem im Linux-Umfeld) schwören auf einen guten, leicht anpassbaren Editor (emacs, vim, heute auch atom) und automatisierte Build-Tools.

Bekannte IDEs sind:

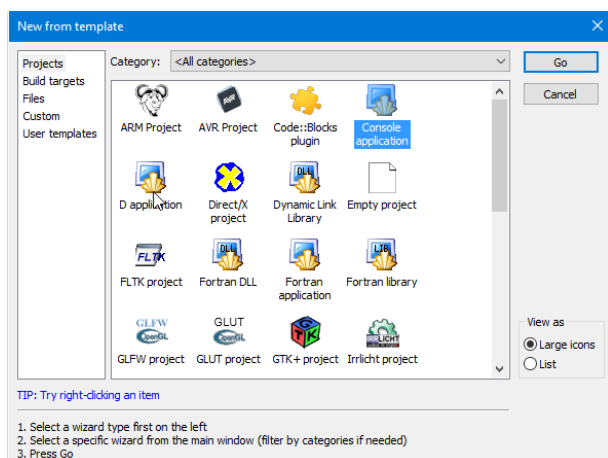
- MS Visual Studio (unter Windows)
- Qt Creator (man muss hier nicht Qt verwenden)
- CodeLite
- Code::Blocks
- eclipse CTD
- NetBeans
- XCode (Apple)
- CLion

Starten Sie jetzt Code::Blocks!



HELLO WORLD MIT CODE::BLOCKS

Legen Sie eine Consolen Applikation mit Code::Blocks an. File/New/Project -> Console application (vgl. **Error! Reference source not found.**) -> Go



Jetzt C++ auswählen und next.

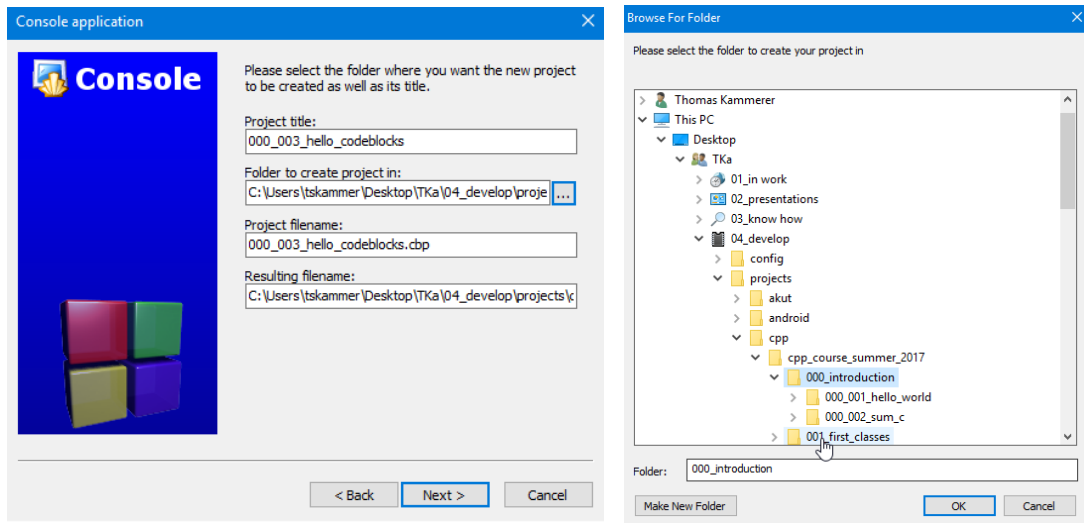


Abbildung 22: neues Projekt anlegen

Wählen Sie als Projekttitel: 000_003_hello_codeblocks

Wählen sie das Verzeichnis: 000_introduction.

Next -> Finish

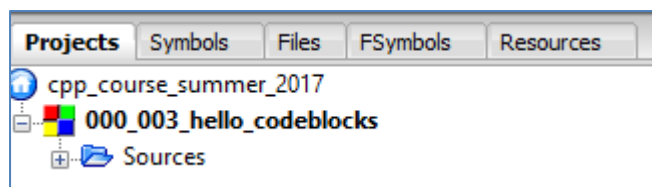


Abbildung 23: project tree

Unter Sources finden Sie main.cpp

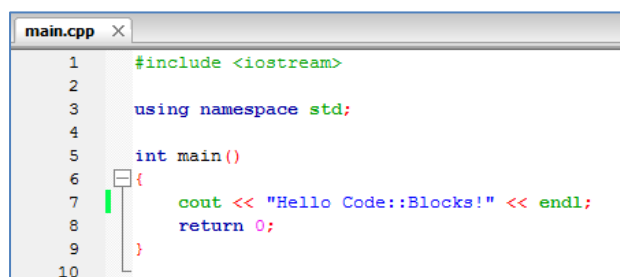


Abbildung 24: main.cpp mit Änderung Hello Code::Blocks

Kompilieren Sie und führen Sie das Programm aus.

Suchen Sie die Befehle und schauen Sie auf das Buildlog.

```

Logs & others
Code:Blocks x Search results x Cccc x Build log x Build messages x CppCheck x CppCheck messages x Cscope x Debugger x DoxyBld x Fortran info x Closed files list

Cleaned "000_003_hello_codeblocks - Debug"

----- Build: Debug in 000_003_hello_codeblocks (compiler: GNU GCC Compiler)-----
g++.exe -Wall -fexceptions -g -O2 -Wall -std=c++14 -IC:\Users\taskammer\Desktop\TKa\04_develop\tools\cpp\catch\include -c C:\Users\taskammer\Desktop\TKa\04_develop\projects\cpp\cpp_course_summer_2017\000_introduction\000_003_hello_codeblocks\main.cpp -o obj\Debug\main.o
g++.exe -o bin\Debug\000_003_hello_codeblocks.exe obj\Debug\main.o
Output file is bin\Debug\000_003_hello_codeblocks.exe with size 2.60 MB
Process terminated with status 0 (0 minute(s), 0 second(s))
0 error(s), 0 warning(s) (0 minute(s), 0 second(s))

```

Abbildung 25: Build log von hello_codeblocks

GLOBAL COMPILER SETTINGS ETC.

Setzen Sie die Compiler Settings, die für alle unsere Programme gelten sollen. Hierzu unter Settings/Compiler die Compiler Flags `-std=c++14` (bei älteren Version `-std=c++11`) und `-Wall` einschalten:

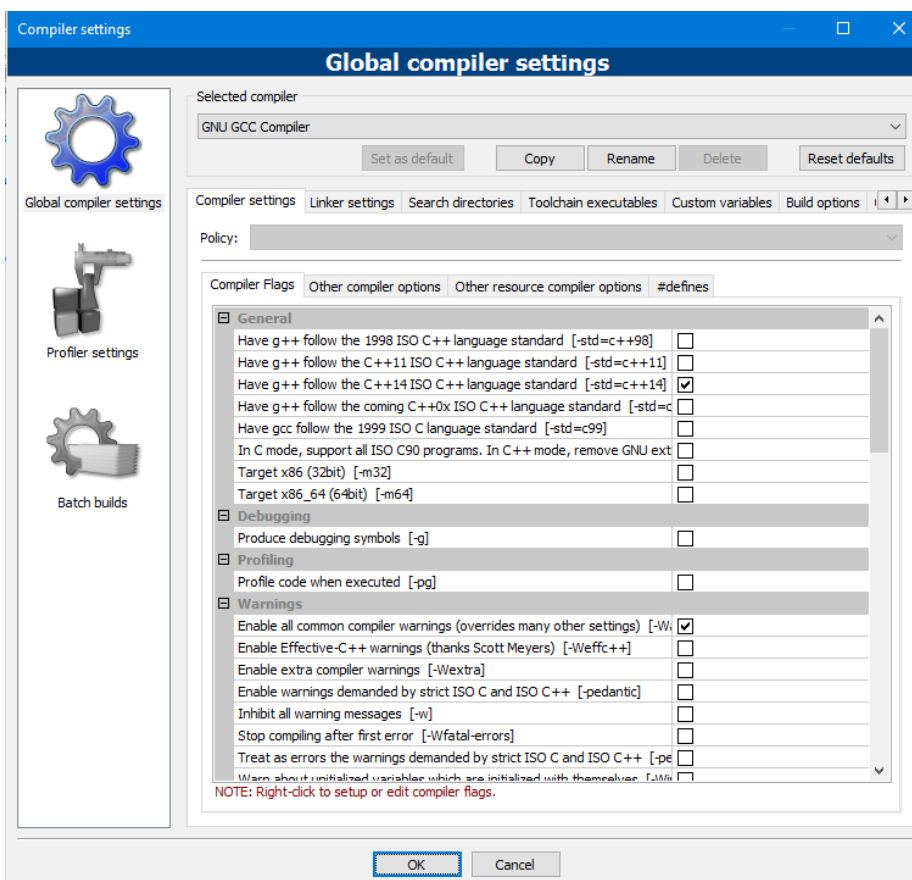


Abbildung 26: Global Compiler Settings

Wenn Sie möchten können Sie auch noch den Code-Style auswählen: Settings/Editor/SourceFormatter.

Ich mag hier Google und Padding/"Insert space padding around paranthesis on the inside"

Weiterhin können noch KeyBoard-Shortcuts ausgewählt werden: Settings/Editor/KeyboardShortcuts:

Ich mag hier:

Strg + Alt + f: für /Plugin/SourceCodeFormater (Asstyle) und

Strg + <: für /Edit/Swap Header Source

SUM.EXE MIT CODE::BLOCKS

Legen Sie unter 000_introduction das Projekt 000_004_sum_cpp an.

Kopieren Sie mit dem File-Explorer aus 000_002_sum_c die Unterverzeichnisse include und src nach 000_004_sum_cpp.

Wählen Sie mit der Maus das neue Projekt im Project-Tree aus. Drücken Sie die rechte Maustaste und wählen Sie Add files...

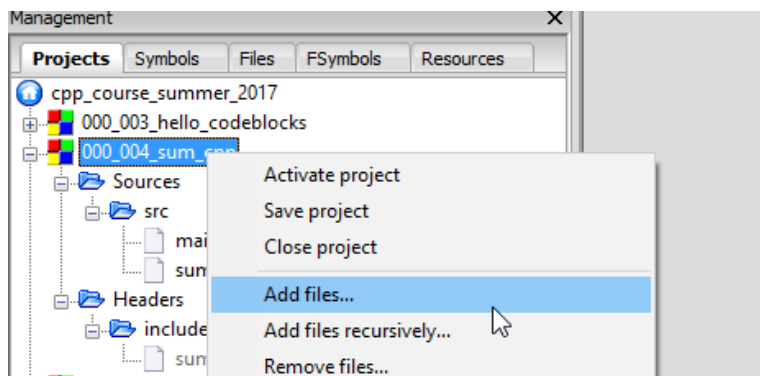


Abbildung 27: Add files to project

und wählen Sie jetzt sum.h, danach main.cpp und sum.cpp aus.

Öffnen Sie main.cpp im Editor von Code::Blocks.

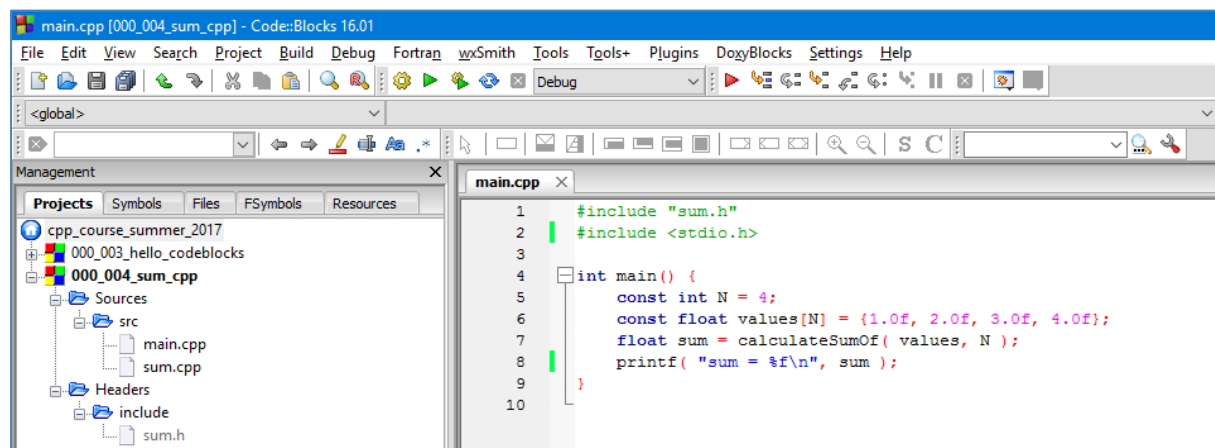


Abbildung 28: Projekt 000_004_sum_cpp mit geöffneten main.cpp in Code::Blocks



Abbildung 29: Werkzeugleiste

Übersetzen Sie und starten Sie das Programm! Nutzen Sie die Werkzeugleiste, Menüpunkt Build oder Shortcuts z.B. F9

Wahrscheinlich gibt es noch einen Fehler, weil der Compiler das Include-File sum.h nicht findet. Wir setzen die Compiler Settings in Code::Blocks, damit auch das Include-File gefunden wird:

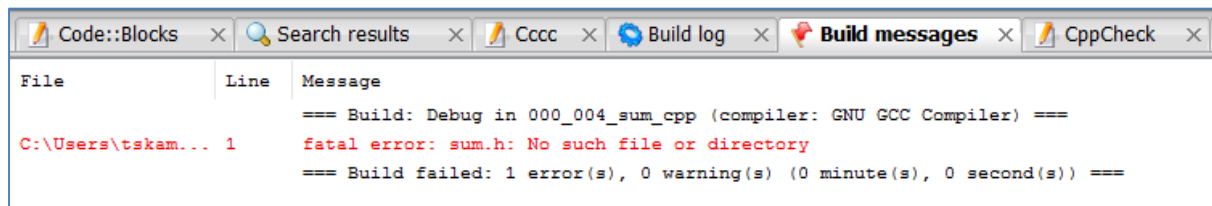


Abbildung 30: fatal error!

Öffnen Sie mit der rechten Maustaste das Build-Options-Menü

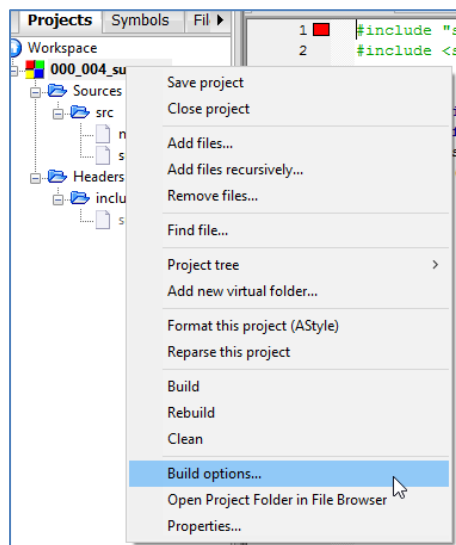


Abbildung 31: Öffnen des Build options Menü

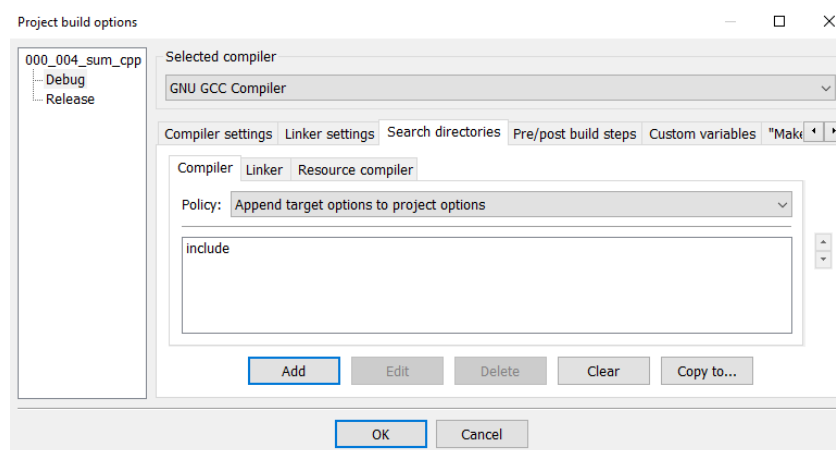


Abbildung 32: Build options Menü - Search directories

Gehen Sie auf die dritten Tab-Karte (Search directories) und fügen Sie (mit Add Button) unter Compiler Ihr include –Verzeichnis hinzu. Jetzt sollte das Compilieren klappen!

Jetzt verbessern wir schrittweise das Programm!

PROGRAMM „SUM“ VERBESSERN

Wir wollen das Programm verbessern und etwas C++-mäßiger gestalten. Ein paar Zahlen zu addieren ist ein sehr einfaches Problem. Verstehen Sie den Code auf Anhieb? Oder müssen Sie lange nachdenken, bis Sie alles verstanden haben? Erkennen Sie die Rekursion? Das ist sehr trickreich und wird vor allem von Funktional-Programmieren (das ist zur Zeit wieder in Mode gekommen) gerne verwendet, da man keine temporären Variablen benötigt.

Identifizieren Sie Stärken und Schwächen im Programm:

STÄRKEN:

SCWÄCHEN:

Jetzt gehen wir schrittweise daran, den Code zu verändern:

Nach jedem Schritt immer prüfen, ob das Programm noch übersetzt werden kann und ob es noch läuft!

1. C++ Streams verwenden für die Ausgabe

Verwenden Sie in main.cpp keine printf-Ausgabe, sondern nutzen Sie iostream
#include <stdio.h> weg dafür
#include <iostream> rein
statt printf jetzt std::cout verwenden!

2. Eliminieren Sie die Rekursion in sum.cpp

Benennen Sie die Argumente der Funktion um, so dass sie einfacher zu verstehen sind.
Passen Sie hier sum.cpp und sum.h an!
Analyse der Rekursion:
*(num_v) ? *v+calculateSumOf(v+1, num_v-1) : 0;*

Welche Stärken und welche Schwächen hat der obige Code?

Stärken: _____

Schwächen: _____

Das ganze iterativ formulieren: for-Schleife verwenden!

3. Statt des c-Arrays einen Container der C++-Standardbibliothek (`std::vector<float>`) verwenden !
in `sum.h` sieht das z.B. so aus:
`float calculateSumOf(const std::vector<float>& summands);`

- a) Informieren Sie sich über `std::vector<>`
nutzen Sie hier:
www.cplusplus.com oder
<http://en.cppreference.com/w/>
- b) passen Sie `sum.h` dementsprechend an!
- c) passen Sie `sum.cpp` an und eliminieren Sie den Parameter, der die Anzahl der Summande angibt!
- d) passen Sie `main.cpp` an, so dass alles wieder läuft!
- e) Erläutern Sie warum `const std::vector<float>& summands` besser ist als
`std::vector summands`

4. Seit C++11 gibt es die sog. Range-based for loop. Wir wollen Sie in `sum.cpp` verwenden!
Seit C++11 gibt es den auto Specifier. Wir wollen ihn in `sum.cpp` verwenden!

Verwenden Sie range based for loop und auto specifier!

Das könnte z.B. so aussehen:

```
for( auto cur_summand : summands ) {
```

...

Informieren Sie sich über auto und range based loop!

5. Interaktives Hauptprogramm

Erweitern Sie ihr `main.cpp` so, dass der Anwender n-Zahlen eingeben kann:

zuerst soll er seinen Namen eingeben

dann soll er eingeben, wie viele Zahlen er addieren möchte,

dann gibt er die Zahlen ein

Die Nutzer-Interaktion könnte dabei z.B. so aussehen:

Wie ist dein Name? Thomas

Hallo Thomas, wie viele Werte soll ich für dich addieren? 4

Wie lautet der 1. Wert? 1

Wie lautet der 2. Wert? 2

Wie lautet der 3. Wert? 3

Wie lautet der 4. Wert? 4

Die Summe über 1 2 3 4 ist 10

Bis bald, Thomas

6. Ergebnispräsentation

Schreiben einen Kommentar in main.cpp der Form:

// @name: <ihr vor und nachname> @mail: <mailadresse von name>

// @name: <ihr vor und nachname> @mail: <mailadresse von name> bei Gruppenarbeit

Beispiel:

// @name: Thomas Kammerer @mail: thomas.kammerer@th-nuernberg.de

Präsentieren Sie den Code im nächsten Praktikum!