5. MODULARITÄT

Inhalt

- Getrennte Kompilierung
  - Header-Datei und Implementierung
  - Eigene Bibliotheken erstellen
  - Mehr zu Makefiles
- Globale Variablen
- Argumentübergabe und Ergebnisrückgabe
- Namespaces

Kompilierung eines C++-Programms

- Bis jetzt haben wir unseren Code folgendermaßen kompiliert

```
> g++ -o ggT main.cpp
```

- Hinter den Kulissen führt ein solcher `g++`-Aufruf **zwei** separate Schritte aus
  - Der **Compiler** übersetzt jede Quelltextdatei (`*.cpp`) in eine binäre Objektdatei (`*.o`)
  - Der **Linker** kombiniert die Objektdateien zu einem ausführbaren Programm (Standardname `a.out`)
- Natürlich lassen sich diese zwei Schritte auch **getrennt** ausführen

```
> g++ -c main.cpp
> g++ -o ggT main.o
```

Compiler

- Der **Compiler** übersetzt alle Funktionen aus der angegebenen Quelltextdatei in Maschinencode

```
> g++ -c main.cpp
```

  - Dieser `g++`-Aufruf erzeugt die Objektdatei `main.o`
  - Eine Objektdatei ist aber noch **kein** lauffähiges Programm
- Mit dem Tool `nm` („Name Mangling“) kann man sich die sogenannte **Symboltabelle** einer Objektdatei ausgeben
  - `T` (Text, Code) wird bereitgestellt
  - `U` (Undefined) wird benötigt
  - Die Optionen `-C` oder `--demangle` zeigen die C++-Namen anstatt der compiler-internen Namen an

Linker

- Der **Linker** fügt vorher kompilierte Objektdateien zu einem ausführbaren Programm zusammen

```
> g++ -o ggT main.o
```

- Wenn mehrere Objektdateien zusammengeführt werden, muss gelten, dass
  - jede Funktion, die in einer Objektdatei benötigt wird, von **genau einer** anderen Objektdatei bereitgestellt wird

```
undefined reference to ...
```
(Funktion nicht gefunden)

```
multiple definition of ...
```
(Funktion mehrmals gefunden)

  - es in **genau einer** der Objektdateien eine `main`-Funktion gibt

```
undefined reference to main
```
(kein `main`)

```
multiple definition of main
```
(mehrere `main`)

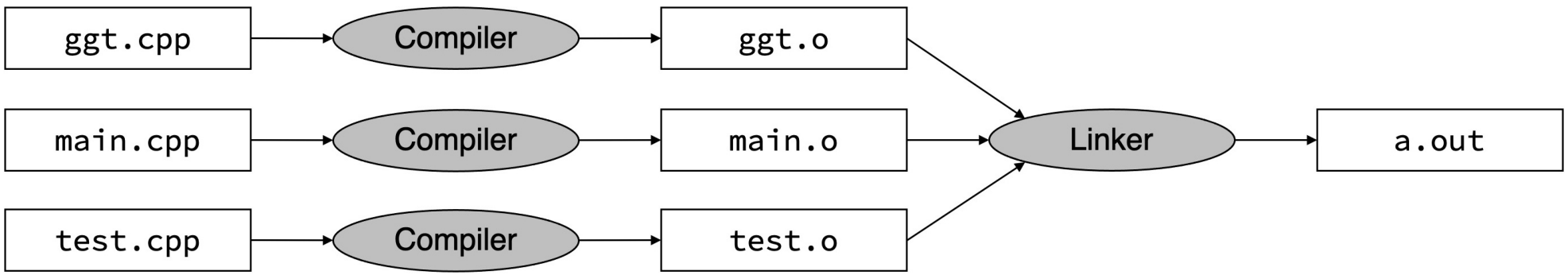
Getrennte Kompilierung

- Mittelgroße bis große Programme unterteilt man oft in mehrere separate **Übersetzungseinheiten**
  - Änderungen an (bestehendem) Code sind oft inkrementell und lokal
  - Man möchte nur die Teile neu kompilieren, die sich geändert haben
  - Insbeondere will man nicht immer die ganze Standardbibliothek neu kompilieren
- Im Prinzip kann man `g++` auch mit mehreren Dateien aufrufen

```
> g++ main.cpp ggt.cpp
```

  - Die Dateien werden eine nach der anderen kompiliert und gelinkt
  - Man kann sogar Quelltextdateien (`*.cpp`) und Objektdateien (`*.o`) mischen (**kein** guter Stil)
  - Das löst aber das Problem nicht, dass immer alles kompiliert wird
- Um **getrennt** zu kompilieren, muss der Compiler Information über Funktionen haben, die ihm nicht im Quelltext vorliegen

Getrennte Kompilierung



- In der ersten Woche hatten wir die `ggT`-Funktion erstellt, die sowohl von `main.cpp` wie auch von `test.cpp` gebraucht wurde
  - Damals hatten wir einfach eine `#include`-Direktive für die Quelltextdatei `ggt.cpp` in beiden Dateien eingefügt
  - Einerseits machte dieses Vorgehen `cpplint` fürchterlich unglücklich

    ```
    Do not include *.cpp files from other packages
    ```

  - Andererseits wird so die `ggT`-Funktion zweimal kompiliert: einmal aus `main.cpp` und einmal aus `test.cpp`
- Um den Zustand in der Abbildung oben zu erreichen, müssen wir die **Deklaration** der Funktion von ihrer **Definition** trennen

Header-Dateien

- In C++ muss eine Funktion entweder **deklariert** oder **definiert** sein, bevor sie benutzt werden kann („declare-before-use“)
  - Funktionsdeklaration `int ggT(int a, int b);`
  - Funktionsdefinition `int ggT(int a, int b) { ... }`
- Somit können wir die Deklaration und die Definition einer Funktion auf **zwei separate** Dateien verteilen
  - `ggt.h` **Header-Datei** (Deklaration)
  - `ggt.cpp` **Quelltextdatei** (Definition)
- Die Header-Datei binden wir nun in `ggt.cpp`, `main.cpp` und `test.cpp` mit der Direktiven `#include "./ggt.h"` ein
- Die Quelltextdatei `ggt.cpp` müssen (und wollen) wir nur noch einmal kompilieren: `g++ -c ggt.cpp`

Header-Dateien

- Kommentare
  - Die Dokumentation von Zweck und Benutzung von Funktionen gehört **ausschließlich** in die Header-Datei, nicht in die Quelltextdatei
  - Dokumentation an **zwei** Stellen führt unweigerlich zu Inkonsistenzen
- Header-Dateien richtig einbinden
  - Jede Quelltextdatei bindet **genau** die Header-Dateien ein, deren Funktionen sie auch wirklich braucht
  - **Nicht** darauf verlassen, dass die eingebundenen Header-Dateien benötigte Funktionen „irgendwie transitiv“ einbinden
- `cpplint`
  - `cpplint` erwartet, dass alle Header-Dateien in einem **eigenen** Verzeichnis abgelegt sind
  - Damit der Ansatz auf der letzten Folie von `cpplint` geschluckt wird, muss ihm die Option `--repository=.` übergeben werden

Include-Wächter

- Eine Header-Datei kann eine **andere** Header-Datei einbinden
  - Bei komplexeren Programmen ist das oft sogar die Regel
  - Dabei muss man **zyklische** Abhängigkeiten verhindern
- **Beispiel**
  - Header-Datei `A.h` bindet (unter anderem) Header-Datei `B.h` ein
  - Header-Datei `B.h` bindet (unter anderem) Header-Datei `C.h` ein
  - Header-Datei `C.h` bindet (unter anderem) Header-Datei `A.h` ein
- Wird nicht verhindert, dass bereits gelesene Header-Dateien nochmals eingelesen werden, dauert die Kompilierung **ewig**!
- Es gibt zwei Lösungen, solche Zyklen mit sogenannten **Include-Wächtern** („Header-Guards“) zu unterbrechen
  - Präprozessor-Makro `#ifndef`
  - Spracherweiterung `#pragma once`

Präprozessor-Makro

- Alle Header-Dateien werden, wie folgt, erweitert

```
#ifndef GGT_H_
#define GGT_H_

int ggT(int a, int b);

#endif  // GGT_H_
```

- Wenn der Compiler die Header-Datei zum ersten Mal sieht, wird ein sogenanntes **Präprozessor-Makro**, hier `GGT_H_`, definiert
- Wenn der Compiler die Header-Datei nochmals sieht, wird ihr Inhalt einfach übersprungen
- **GTK** Der Name des Präprozessor-Makros sollte möglichst **eindeutig** gewählt werden

Spracherweiterung

- Bei dieser Lösung genügt es, die Header-Datei wie folgt zu erweitern

```
#pragma once

int ggT(int a, int b);
```

  - **Vorteil**: Man muss sich keine eindeutigen Namen ausdenken
  - **Nachteil**: Der Compiler muss die Spracherweiterung unterstützen

- GTK Die Spracherweiterung `#pragma once` wird heute von allen gängigen C++-Implementierungen unterstützt
- GTK `cpplint` ist auch mit beiden Lösungen zufrieden

Bibliotheken

- Eine **Bibliothek** („library“) ist prinzipiell nichts anderes als eine Sammlung von Objektdateien mit einem speziellen Namen
  - **Statische** Bibliothek („**a**rchive“) `lib<name>.a`
  - **Dynamische** Bibliothek („**s**hared **o**bject“) `lib<name>.so`
- Im Gegensatz zu einer Objektdatei enthält eine Bibliothek aber auch einen **Index**
  - Eine Bibliothek enthält typischerweise den Code **vieler** Funktionen
  - Der Linker muss den Code einer bestimmten Funktion **schnell** finden
- Mit dem **Google Test Framework** haben wir in der ersten Woche bereits eine Bibliothek kennengelernt
  - Eine Bibliothek kann man genauso linken wie eine Objektdatei

```
> g++ test.o ggt.o libgtest.a
```

  - Normalerweise ist die Bibliothek aber nicht im aktuellen Verzeichnis

```
> g++ test.o ggt.o /usr/lib/libgtest.a
```

Linken einer Bibliothek

- **GTK** Es ist eher selten, dass man beim Linken den Pfad und Dateinamen der Bibliotheken angibt
  - Bibliotheken können auf verschiedenen Systemen in **unterschiedlichen Verzeichnissen** installiert sein
  - Die statische Bibliothek hat eine **andere Dateiendung** als die dynamische Bibliothek
- Um das System entscheiden zu lassen, mit welcher Datei genau gelinkt wird, verwendet man deshalb besser die Option `-l`

```
> g++ test.o ggt.o -lgtest
```

  - So bleibt der Befehl zum Linken auf allen Systemen immer gleich
  - **GTK** Das „`lib`“ lässt man dabei per Konvention weg
- Mit der Option `-L` können Verzeichnisse angegeben werden, in denen auch nach der Bibliothek gesucht werden soll

```
> g++ -L/usr/local/lib test.o ggt.o -lgtest
```

Statische Bibliotheken

- Bei einer **statischen** Bibliothek wird der aus der Bibliothek benötigte Code Teil des ausführbaren Programms
  - **Vorteil**: die Bibliothek wird nur zum Linken, aber nicht zum Ausführen des Programms benötigt
  - **Nachteil**: das ausführbare Programm kann dadurch sehr groß werden
  - **Nachteil**: wenn sich die Implementierung der Bibliothek ändert, muss das Programm neu kompiliert werden
- Eine statische Bibliothek kann, wie folgt, gelinkt werden

```
> g++ -static test.o ggt.o -lgtest
```

- GTK Üblicherweise sind dynamische den statischen Bibliotheken vorzuziehen

Dynamische Bibliotheken

- Bei einer **dynamischen** Bibliothek steht im ausführbaren Programm nur eine Referenz auf die Stelle in der Bibliothek
  - **Vorteil**: das ausführbare Programm wird viel kleiner
  - **Nachteil**: man braucht die Bibliothek auch zur Laufzeit
  - **Nachteil**: es kann zu Versionierungsproblemen kommen
- GTK Nur weil Bibliotheken beim Linken gefunden wurden, werden sie nicht automatisch auch bei der Ausführung gefunden
  - Bibliotheken müssen in einem **Suchpfad** des Systems installiert sein
  - Prüfen, welche Bibliotheken (nicht) gefunden werden

```
> ldd a.out
```

Suchpfade (Linux)

- In Linux und Unix gibt es **zwei** Alternativen, die Suchpfade für dynamische Bibliotheken zu setzen
  - Auf der **Kommandozeile**

    ```
    > export LD_LIBRARY_PATH=./lib
    ```

    Diese Einstellung gilt aber nur in der aktuellen Kommandozeile
  - Den Pfad zu einer der Dateien in `/etc/ld.so.conf.d/` hinzufügen (z.B. `local.conf`) und danach `ldconfig` ausführen
  - **GTK** `ld` ist das Programm, das `g++` hinter den Kulissen zum Linken benutzt, der sogenannte „Linker“
- **GTK** Windows und macOS haben anderen Mechanismen, wie dynamische Bibliotheken zur Laufzeit gefunden werden
  - **macOS**: Dynamically Linked Library (`*.dylib`)
  - **Windows**: Dynamic-Link Libraries (`*.dll`)

Erstellen einer Bibliothek

- Zum Erstellen einer Bibliothek braucht man eine Menge von Objektdateien, die eine Menge von Funktionen enthalten
- Eine **statische** Bibliothek erstellt man folgendermaßen

```
> ar cr lib<name>.a <obj1>.o <obj2>.o ...
```

  - `ar` ist der Name des Programms („archive“)
  - Die Option `cr` erstellt („create“) eine Bibliothek

- Eine **dynamische** Bibliothek erstellt man folgendermaßen

```
> g++ -fPIC -shared -o lib<name>.so <obj1>.o ...
```

  - Die Option `shared` erstellt eine dynamische Bibliothek
  - Die Option `fPIC` erstellt Code mit relativer Adressierung („position-independent code“), der ohne Änderung an einer beliebigen Speicherstelle ausgeführt werden kann

Mehr zu Makefiles

Betrachten wir nochmals unser **ggT-Beispiel** aus der ersten Woche.

- Nehmen wir an, dass wir unsere drei Quelltextdateien bereits zu den folgenden Objektdateien kompiliert haben
  - `main.o` das Hauptprogramm
  - `test.o` das Testprogramm
  - `ggt.o` die Funktion `ggT`
- Nehmen wir nun an, dass wir `main.cpp` wegen eines gravierenden Programmierfehlers ändern müssen
- Anstatt alles nochmals zu kompilieren, genügt es in diesem Fall
  - `main.o` neu zu erzeugen
  - `main` neu zu linken
- Es wäre schön, wenn unser **Makefile** das irgendwie erkennen und so umsetzen würde

Abhängigkeiten

- **Abhängigkeiten** („dependencies“) kann man in Makefiles folgendermaßen angeben

```
<Target>: <Dependency1> <Dependency2> ...
    <Command1>
    <Command2>
    ...
```

- Nun wird bei `make <Target>` erst folgendes gemacht

```
make <Dependency1>
make <Dependency2>
...
```

- Wenn es keine Ziele mit diesem Namen gibt, gibt es eine Fehlermeldung der Art

```
No rule to make target <...> needed by <target>
```

Abhängigkeiten

Nochmals **genauer**: bei `make <Target>` passieren zwei Dinge.

1. Es wird `make <Dependency>` für jede der Abhängigkeiten von <Target> ausgeführt (siehe letzte Folie)
   - Natürlich kann jede der Abhängigkeiten wieder Abhängigkeiten haben
   - Die Rekursion bricht an Zielen ohne Abhängigkeiten ab
   - Bei einem Zyklus bricht `make` an der betreffenden Stelle ab
2. Die Anweisungen eines Ziels <Target> werden nur ausgeführt, wenn mindestens **eine** der folgenden Bedingungen erfüllt ist
   - Es existiert keine Datei mit dem Namen <Target>
   - Es existiert keine Datei mit dem Namen <DependencyX> für ein X
   - Eine der Dateien <DependencyX> ist neuer als die Datei <Target>

Fortgeschrittene Regeln

- **Musterbasierte Regeln**
  - Anstatt für jede Datei einzeln zu spezifizieren, wie `make` sie erzeugen kann, können auch musterbasierte Regeln verwendet werden
  - **Beispiel**: Objektdateien aus Quelltextdateien erzeugen

```
%.o : %.cpp
    @g++ -c $<
```

  - Das Prozentzeichen (`%`) steht für eine beliebige Zeichenkette
  - `$<` steht für den Dateinamen der ersten Abhängigkeit in der Regel
  - `@` bedeutet, dass die Anzeige des `g++`-Befehls unterdrückt wird
- **Suffix-Regeln** (sollte man **nicht** mehr benutzen)
  - Regeln wie die im Beispiel oben werden so oft in Makefiles benötigt, dass `make` diese (und viele andere) automatisch kann
  - Man generiert diese sogenannten Suffix-Regeln, in dem man ganz oben im Makefile z.B. die folgende Zeile einfügt

```
.SUFFIXES: .cpp.o
```

Variablen

- In Makefiles können auch **Variablen** verwendet werden

```
TEXT = Hello World!
hello:
    @echo $(TEXT)
```

- Es gibt in Makefiles auch viele vordefinierte Variablen
- **Beispiel**: Die Suffix-Regel `.cpp.o` auf der letzten Folie generiert hinter den Kulissen die folgende musterbasierte Regel

```
%.o : %.cpp
    $(CXX) $(CPPLAGS) $(CXXFLAGS) -c -o $@ $<
```

  - `CXX` ist der C++-Compiler
  - `CPPFLAGS` sind Optionen für den C-Präprozessor
  - `CXXFLAGS` sind Optionen für den C++Compiler
  - `$@` steht für den Dateinamen des Ziels in der Regel
- **GTK** Musterbasierte Regeln anstatt Suffix-Regeln verwenden!

Unechte Ziele

- Ein **unechtes Ziel** („phony target“) ist ein spezielles Ziel
  - Es gibt keine Datei mit dem Namen des Ziels
  - Die Anweisungen des Ziels erzeugen keine Datei mit diesem Namen
- Unechte Ziele dienen lediglich als **Abkürzung** für eine Abfolge von Anweisungen
- **Beispiel**: Das Ziel `clean` löscht alles, was generiert wurde

```
clean:
    @rm -f *.o ggT test
```

- Unechte Ziele werden im Makefile folgendermaßen deklariert

```
.PHONY: all clean
```

- Wenn ein Ziel unter seinen Abhängigkeiten auch nur ein unechtes Ziel hat, werden die Anweisungen immer ausgeführt
- **GTK** Das folgt aus den Regeln, die wir schon kennen!

Programmieren!

- Schreibe ein `Makefile` für das `ggT`-Beispiel, das
  - das Ziele für `all`, `ggT`, `test` und `checkstyle` definiert
  - ein Ziel `cleanup` spezifiziert, das alle generierten Dateien löscht
- **Ausprobieren**: Was passiert, wenn man `ggt.cpp` ändert und dann `make ggT` aufruft?

Globale Variablen

- Variablen, die außerhalb einer Funktion definiert sind, nennt man **globale Variablen**

```
int x;

void fun() {
  // x kann hier verwendet werden
}
```

- Prinzipiell kann man globale Variablen überall im Code benutzen, auch in anderen Dateien
- Auch globale Variablen müssen vor der Benutzung **deklariert** und in genau einer Quelltext datei **definiert** werden
  - Deklaration mit dem Schlüsselwort `extern`
  - Definition wie gehabt (ohne `extern`)
- Wird eine mit `extern` deklarierte globale Variable beim Linken nicht gefunden, gibt es die gleichen Fehler wie bei Funktionen

Argumentübergabe und Ergebnisrückgabe

- Es gibt viele verschiedene Möglichkeiten, wie Information innerhalb eines Programms übermittelt werden kann
  - **Globale Variablen** (letzte Folie) sind dabei sehr fehleranfällig
  - **Argumentübergabe und Ergebnisrückgabe**
  - **Objekte** (nächste Woche)
- Bei Informationsübermittlung durch Funktionsargumente und -rückgabewerte können folgende Fälle unterschieden werden
  - Die Daten sollen **nicht geteilt** werden
    - Werden die Daten **kopiert**? $\Leftarrow$ Das ist das **Standardverhalten**
    - Werden die Daten **verschoben**?
  - Die Daten sollen **geteilt** werden
    - Sind die Daten **veränderbar**?
    - Sind die Daten **nicht veränderbar**?
- Abhängig davon, ob wir über Argumente oder Rückgabewerte reden, müssen wir für jeden Fall eine **andere Strategie** wählen

Argumentübergabe (Ungeteilte Daten)

- Von der aufrufenden in die aufgerufene Funktion **kopieren**

```
void echo(std::string s) { ... }

int main(void) {
  std::string text("Hello World!");
  echo(text);
  ...
}
```

  - Beim Aufruf wird der Wert der Variablen `text` in die für die Funktion `echo` lokale Variable `s` kopiert
  - Selbst wenn `echo` den Wert von `s` verändert, hat `text` nach dem Funktionsaufruf den gleichen Wert wie davor
  - **GTK** Dieser Art der Argumentübergabe sagt man **Call-by-Value**
- Von der aufrufenden in die aufgerufene Funktion **verschieben**
  - `std::move` aus dem Header `<utility>` verschiebt Werte
  - Im Beispiel können wir das so machen: `echo(std::move(text))`
  - Danach ist der Wert von `text` in `main` **nicht mehr definiert**

Argumentübergabe (Geteilte Daten)

- Bei **großen Werten** ist es oft effizienter, diese über ihre Adresse zu teilen, als den ganzen Wert zu kopieren oder zu verschieben
  - **Call-by-Pointer**: `void p_echo(std::string* s) { ... }`
  - **Call-by-Reference**: `void r_echo(std::string& s) { ... }`
  - GTK In beiden Fällen wird nur eine Adresse (4–8 Bytes) kopiert!
- **In-Out-Paramter**: Funktion kann das Argument verändern

```
std::string text("Hello World!");
p_echo(&text);
r_echo(text);
```

- **In-Parameter**: Funktion kann das Argument **nicht** verändern
  - `const std::string* s` ⇒ der Wert, auf den der Zeiger zeigt, ist konstant
  - `const std::string& s` ⇒ der Wert der Referenz ist konstant

Entscheidungshilfe

- **In-Parameter**
  - Call-by-Value für „kleine“ Werte
  - Call-by-Value für „große“ Werte, von denen man eine Kopie braucht
  - sonst Call-by-Pointer/Call-by-Reference (mit `const`) für „große“ Werte
- **In-Out-Parameter**
  - Call-by-Pointer/Call-by-Reference (ohne `const`) für alle Werte
- **Rückgabewerte**
  - Normale Rückgabe mit `return` für „kleine“ und „große“ Werte
    - Bei großen Werte ist Aufruf von `std::move` nicht notwendig
    - Das geschieht automatisch, manchmal geschieht sogar was Besseres
  - In-Out-Parameter für „große“ Werte (siehe Argumentübergabe)
  - Rückgabe eines Zeigers/Referenz für „große“ Werte

Rückgabe von Zeigern und Referenzen

- Rückgabe eines **Zeigers**

```
std::string* hello() {
  std::string* p = new std::string("Hello World!");
  return p;
}
```

  - **Problem**: Wer gibt den Speicher frei, den `hello()` reserviert?
  - **Lösung**: Smart Pointer `std::unique_ptr` verwenden!

- Rückgabe einer **Referenz**

```
std::string& hello() {
  std::string r("Hello World!");
  return r;
}
```

  - Kompiliert erst gar nicht, da die lokale Variable `r` nach dem Funktionsaufruf von `hello()` **nicht mehr existiert**
  - In Klassen können solche Rückgabewerte aber Sinn machen (Woche 6 und 7)!

Programmieren!

- Schreibe ein C++-Programm `fibsum.cpp`, das
  - eine (ganze, positive) Zahl `n` von der Kommandozeile liest,
  - eine Funktion aufruft, die die ersten `n` Fibonacci-Zahlen in einen `std::vector` einfügt,
  - eine Funktion aufruft, um die Zahlen in einem `std::vector` zusammenzuzählen und
  - die so erhaltene Summe ausgibt

Namespaces

- Neben Funktionen und Klassen sind **Namespaces** eine weitere Möglichkeit, C++-Programme zu strukturieren

```
namespace mathe {
  int ggT(int a, int b);
}

int mathe::ggT(int a, int b) {
  ...
}

int main() {
  return mathe::ggT(3, 45);
}
```

- Namespaces haben wir bis jetzt vor allem in Zusammenhang mit der C++-Standardbibliothek (`std`) gesehen
- **Keine Angst**: im Programmierkurs 1 werden wir nicht so viel Code schreiben, dass wir Namespaces wirklich brauchen