Programmierkurs

Ole Klein Steffen Müthing

Interdisciplinary Center for Scientific Computing, Heidelberg University

November 17, 2017

Standardbibliothek

Variablen und Referenzen

Aufrufkonventionen

Der C++-Kompilierprozess

Mehrdateiprogramme

CMake

C++-Standardbibliothek

- C++ enthält eine umfangreiche Standardbibliothek mit vielen Datenstrukturen und Algorithmen.
- Wenn möglich, ist es **immer** besser, Funktionen aus der Standardbibliothek zu nehmen als eigene.
  - Gut optimiert
  - Umfangreich getestet
- Bestandteile:
  - Datenstrukturen
  - Algorithmen
  - Mathematische Funktionen
  - Input / Output
- Gute Referenz auf `https://cppreference.com`.

Coding Style

- Bei der Verwendung der Standardbibliothek sollte das `std::` normalerweise explizit hingeschrieben werden.
- In besonderen Fällen ist es manchmal erforderlich, eine Funktion ohne `std::` aufzurufen. Diese sollte dann lokal importiert werden:

```
1 void foo()
2 {
3   int a, b;
4   using std::swap;
5   swap(a,b);
6 }
```

- `using namespace std;` hat in einem C++-Programm nichts zu suchen!
  - Schwierig zu sagen, woher eine Funktion kommt.
  - Kann zu sehr schwer zu debuggenden Fehlern führen.

array

Liste, deren Länge zur Compile-Zeit bekannt ist:

```
1  #include <array>
2  #include <iostream>
3
4  int main(int argc, char** argv)
5  {
6    std::array<int,4> a = {{1,2,3,4}};
7    std::cout << a.size() << std::endl; // 4
8    a[2] = 4;
9  }
```

- Besser als C arrays (`int a[4];`).

vector

Dynamisch anpassbare Liste von Objekten:

```
1  #include <vector>
2  #include <iostream>
3
4  int main(int argc, char** argv)
5  {
6    std::vector<int> a; // leere Liste
7    std::vector<int> b = {{1,2,3,4}};
8    std::cout << b.size() << std::endl; // 4
9    b[2] = 4;
10   a = b; // kopiert den Inhalt
11   a.resize(100); // Grösse anpassen
12   a.push_back(1); // Wert 1 hinten anfügen
13   a.pop_back();   // Letztes Element entfernen
14 }
```

- Bei grossen Listen ($>$ 100-1000) immer besser als `std::array`.

Iterieren über Container

Vereinfachter Loop zum Iterieren über Standard-Container:

```
1  #include <vector>
2  #include <iostream>
3
4  int main(int argc, char** argv)
5  {
6    std::vector<int> a = {{1,2,3,4}};
7    for (int i : a)
8      std::cout << i << std::endl;
9  }
```

- Funktionert für alle Standard-Container.
- Besser lesbar.
- Keine Gefahr, Fehler am Ende zu machen ($<$ vs. $\leq$).

Sortieren

- C++ hat einen hochoptimierten, eingebauten Sortier-Algorithmus.
- Der Algorithmus basiert auf *Iteratoren*, was im Moment aber bis auf die Syntax zum Aufrufen egal ist:

```
1  #include <vector>
2  #include <algorithm>
3
4  int main(int argc, char** argv)
5  {
6    std::vector<int> a = .....;
7    // sortiert a nach aufsteigenden Zahlenwerten
8    std::sort(a.begin(),a.end());
9  }
```

Variablen

- Variablen repräsentieren eine Stelle im Arbeitsspeicher, an der Daten eines bestimmten Typs gespeichert sind.
- Jede Variable hat einen Namen und einen Typ.
- Der Speicherbedarf einer Variablen
  - hängt von ihrem Typ ab.
  - kann mit dem Operator `sizeof(var)` oder `sizeof(type)` abgefragt werden.
- Die Stelle im Speicher, an der der Wert einer Variablen gespeichert ist, kann nicht verändert werden.

Konstante Variablen

- Variablen in C++ können mit dem Keyword `const` als konstant (unveränderlich) deklariert werden.
- Eine konstante Variable kann nicht mehr verändert werden, nachdem sie definiert wurde:

```
1 const double pi = 3.1415926535;
2 pi = pi + 1; // compile error
```

- Konstante Variablen können helfen, Programmierfehler zu vermeiden.
- Unter bestimmten Umständen kann der Compiler schnelleren Code generieren, wenn Variablen konstant sind.

Referenzen

- Referenzen sind zusätzliche Namen für existierende Variablen.
- Der Typ einer Referenz ist der Typ der existierenden Variablen gefolgt von `&`. Der Typ einer Referenz auf `int` ist also `int&`.
- Eine Referenz wird **immer** in dem Moment initialisiert, in dem sie definiert wird:

```
1 int x = 4;
2 int& x_ref = x;
3 int& no_ref; // compile error!
```

- Eine Referenz zeigt immer auf die gleiche Variable.
- Die Referenz verhält sich genau so wie die Original-Variable.
- Änderungen an der Referenz verändern auch die Original-Variable und umgekehrt.

Referenzen: Beispiel

```
1  # include <iostream>
2
3  int main ()
4  {
5    int a = 12;
6    int & b = a; // definiert Referenz
7    int & c = b; // Referenz auf Referenz
8    float & d = a; // falscher Typ, Compile-Fehler
9    int e = b;
10   b = 2;
11   c = a * b;
12   std :: cout << a << std :: endl ; // 4
13   std :: cout << e << std :: endl ; // 12
14 }
```

Iterieren über Container (II)

- Wenn man den Inhalt eines Containers beim Iterieren verändern will, muss man für die Variable einen Referenz-Typ verwenden:

```
1  #include <vector>
2  #include <iostream>
3
4  int main(int argc, char** argv)
5  {
6    std::vector<int> a = {{1,2,3,4}};
7    for (int& i : a)
8      i = i * 2;
9  }
```

Call by Value

- Wenn eine Funktion mit einem normalen Parameter aufgerufen wird, erstellt C++ eine Kopie des Parameterwerts, mit dem die Funktion dann arbeitet:

```
1 double square(double x);
```

- Diese Aufrufkonvention heißt *call by value*.
- Bisher haben wir in den Übungen nur call by value verwendet.
- Weil die Funktion eine Kopie der Original-Variablen bekommen hat, wirken sich Änderungen in der Funktion nicht auf die Original-Variable aus.

Call by Reference

- Wenn eine Funktion mit einem Referenz-Parameter aufgerufen wird, übergibt C++ eine Referenz auf die Original-Variable an die Funktion:

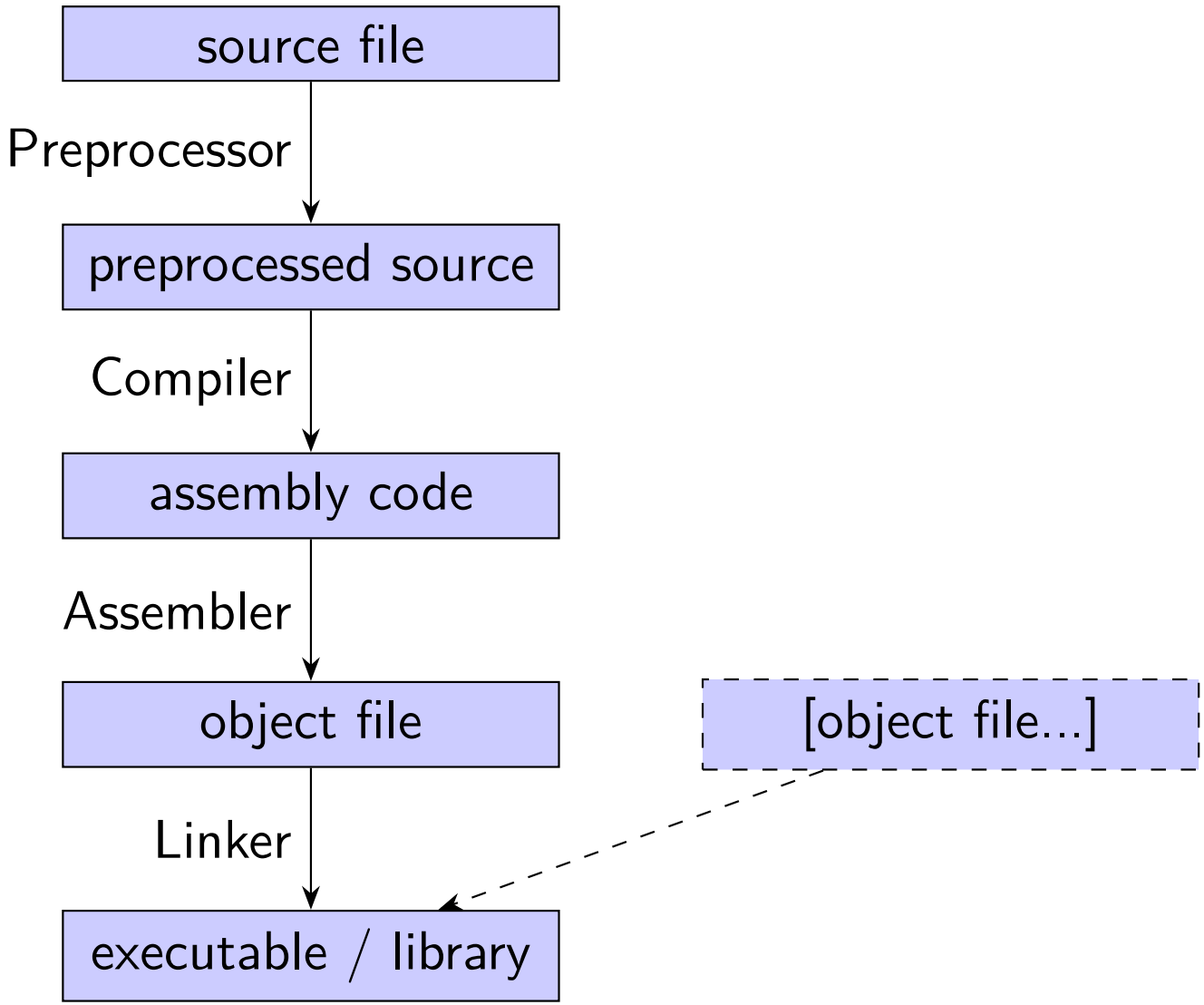
```
1  void sort(std::vector<int>& x);
```

- Diese Aufrufkonvention heißt *call by reference*.
- Funktionen mit dieser Aufrufkonvention können Werte außerhalb der Funktion verändern, ohne dass dies beim Aufruf direkt ersichtlich ist (sie haben *Seiteneffekte*).
- Nicht-triviale Datentypen wie `std::vector` sollten normalerweise per Referenz übergeben werden, weil das Kopieren teuer sein kann.

Dangling References

- Intern sind Referenzen eine spezielle Art von konstanter Variable, die den Speicherort der Original-Variablen enthält.
- Wenn die Original-Variable aufhört zu existieren, wird ihr Speicherort ungültig.
- Referenzen auf die nicht mehr existierende Variable greifen weiter auf den ungültigen Speicherort zu
  $\Rightarrow$ Programm stürzt ab oder liefert falsches Ergebnis!
- Tipps für Referenzen:
  - Referenzen sind oft gut für Funktionsparameter (call by reference).
  - Niemals Referenzen als Rückgabewert von normalen Funktionen verwenden!

Der C++-Kompilierprozess



Der Präprozessor

- Der C++-Präprozessor fügt Header-Dateien in Quellcode ein und expandiert Makros.
- Alle Zeilen, die mit `#` anfangen (sogenannte Direktiven), werden vom Präprozessor verarbeitet.
- Die wichtigsten Direktiven:
  - `#include <header>` fügt den Inhalt der Datei `header` an dieser Stelle ein.
  - `#include "header"` fügt den Inhalt der Datei `header` an dieser Stelle ein, sucht die Datei aber auch im aktuellen Verzeichnis.
  - `#define MACRO REPLACEMENT` definiert ein Makro: Immer, wenn nach dieser Zeile `MACRO` als alleinstehendes Wort auftaucht, wird es durch `REPLACEMENT` ersetzt.
  - Text zwischen `#ifdef MACRO` und `#endif` wird entfernt, wenn das angegebene Makro nicht definiert ist.
- Der Präprozessor kann mit `g++ -E` ausgeführt werden.

Der Compiler

- Der Compiler übersetzt den C++-Code in einfachere Befehle, die der Prozessor verstehen kann.
- Das Resultat dieses Schritts ist Assembly-Code, eine für Menschen lesbare Version der Maschinenbefehle.
- Assembly Code enthält keinerlei Variablennamen oder Schleifen mehr.
- Die Ausgabe des Compilers unterscheidet sich je nach Prozessor (Smartphone-Prozessoren verwenden andere Befehle als PCs).
- Der Compiler kann in diesem Schritt das Programm stark optimieren, wenn aktiviert (Option `-O2` oder `-O3`).
- Die Ausgabe des Compilers kann man mit `g++ -S` oder auf `godbolt.org` anschauen.

Der Assembler

- Der Assembler verwandelt Assembly-Code in die binären Befehlscodes, die der Prozessor versteht.
- Der Assembler produziert sogenannte *object files* mit der Erweiterung `.o`.
- Um ein object file zu erzeugen, muss der Compiler mit der Option `-c` aufgerufen werden.

Der Linker

- Der Linker kombiniert den object code aus einem oder mehreren object files und Programmbibliotheken und erzeugt eine ausführbare Datei.
- Funktionen aus einigen Standardbibliotheken werden vom Linker automatisch gefunden, andere muss man explizit angeben.
- Linkeraufruf, um ein ausführbares Programm aus mehreren object files zu erzeugen:

```
1 g++ -o executable file1.o file2.o ...
```

Programme mit mehreren Dateien

```
1  double cube(double x)
2  {
3    return x * x * x;
4  }
```

- Funktionen, die man mehrfach verwendet, sollte man in eine eigene Datei auslagern.
- Man benötigt meistens zwei Dateien:
  - Ein *header file*, das von anderen Programmen eingebunden werden kann und alle Funktionalität, die wir bereitstellen, *deklariert*.
  - Ein *implementation file*, das die eigentliche Implementierung enthält.

Deklaration vs. Definition

- Bevor man eine Funktion in C++ verwenden kann, muss sie deklariert werden.
- Eine Deklaration sagt dem Compiler nur, dass es eine Funktion mit einer bestimmten Signatur gibt.
- Deklarationen sind Funktionsköpfe, bei denen statt Code ein Semikolon folgt:

```
1 double cube(double x);
```

- Eine Definition enthält den eigentlichen Programmcode, wie bekannt.
- Eine Funktion darf beliebig oft deklariert werden, aber nur einmal definiert (one definition rule).
  - Deklaration → Header
  - Definition → Implementation

Beispiel

cube.hh

```
1 // function for calculating the cube of a double
2 double cube(double x);
```

cube.cc

```
1 #include "cube.hh"
2 double cube(double x)
3 {
4   return x * x * x;
5 }
```

main.cc

```
1 #include <iostream>
2 #include "cube.hh"
3 int main(int argc, char** argv)
4 {
5   std::cout << cube(3.0) << std::endl;
6 }
```

Header Guards

- Echte Programme inkludieren Header oft mehrmals in einer translation unit.
  - langsam
  - Problematisch bei Makro-Definitionen
- Lösung: Header Guard

```
1  #ifndef CUBE_HH
2  #define CUBE_HH
3
4  // function for calculating the cube of a double
5  double cube(double x);
6
7  #endif // CUBE_HH
```

Kompilieren

```
1 g++ -Wall -std=c++14 -c cube.cc
2 g++ -Wall -std=c++14 -c main.cc
3 g++ -Wall -o example cube.o main.o
```

Probleme:

- Viel Tipparbeit
- Welche Datei inkludiert welche andere?
- Was muss ich alles erneut ausführen, wenn ich eine Datei verändere?

$\Rightarrow$ Buildsysteme (make, CMake, qmake, autotools, . . .)

CMake

- CMake ist ein leistungsfähiges Buildsystem für Projekte in C und C++.
- Abhängigkeiten zwischen Programmen, Quell- und Headerdateien werden automatisch erkannt.
- Es können Tests geschrieben werden, um herauszufinden, ob das aktuelle System bestimmte Features hat.
- CMake unterstützt unterschiedliche Konfigurationen (debug, release etc.).

CMakeLists.txt

Um CMake zu konfigurieren, muss man in der Datei `CMakeLists.txt` beschreiben, aus welchen Programmen und Quelldateien das aktuelle Projekt besteht:

```
cmake_minimum_required(VERSION 3.1)
project(ipk-demo LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 14)

add_executable(cube cubemain.cc cube.cc)
add_executable(cuberoot rootmain.cc cuberoot.cc)
add_executable(calculator calcmain.cc basic.cc cube.cc)
```

CMake verwenden

- Im ersten Schritt erzeugt man mit CMake ein Buildsystem für `make`, indem man
  `cmake <pfad-zum-verzeichnis-mit-cmakelists.txt>`
  aufruft.
- Das Buildsystem muss in einem anderem Verzeichnis erzeugt werden als die Quelldateien.
- Eine gute Wahl ist das Unterverzeichnis `build/`:

```
1 mkdir build
2 cd build
3 cmake ..
```

- Wenn das Buildsystem existiert, startet man den eigentlichen Build-Prozess mit dem Befehl `make` im Verzeichnis, in dem man auch `cmake` aufgerufen hat.

CMake-Feintuning

Um genauer zu steuern, wie CMake ein Projekt baut, kann man dem CMake-Aufruf Optionen mitgeben:

```
cmake -DVARIABLE=VALUE <pfad>
```

Wichtige Variablen sind:

- `CMAKE_CXX_COMPILER` : Der C++-Compiler (wichtig im Pool).
- `CMAKE_CXX_FLAGS` : Zusätzliche Flags für den Compiler, z.B. `-Wall`.
- `CMAKE_BUILD_TYPE` : Build-Konfiguration (Release oder Debug).

Weitere Optionen kann man finden, indem man nach `cmake` im Build-Verzeichnis `ccmake .` aufruft, die Taste `t` drückt und dann durch die Liste blättert.