Programmierkurs
Vorlesung 3
Standard Library / Variablen und Referenzen

Andreas Naumann

Institut für Wissenschaftliches Rechnen
Universität Heidelberg

29. März 2023

Organisatorisches

Variablen und Referenzen

Aufrufkonventionen

Standardbibliothek
- Datenstrukturen
- Algorithmen

Buildsysteme
- Der C++-Kompilierprozess
- Mehrdateiprogramme
- CMake

Organisatorisches

- Bitte auf Moodle unter `https://moodle.uni-heidelberg.de/course/view.php?id=16916` eintragen
- Einige Themen wurden zusammen gefasst
- Weitere Möglichkeit zur Projektabgabe und git-übung: `https://edu.ziti.uni-heidelberg.de`
  - Login mittels uni-id
  - Tutoren hinzufügen
  - Merge-Request an Tutoren oder mich

Variablen (Wiederholung)

- 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.
- Der Typ kann modifiziert werden bezüglich
  - erlaubten Zugriff
  - Assoziation

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:

```
const double pi = 3.1415926535;
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:

```
int x = 4;
int& x_ref = x;
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

```cpp
# include <iostream>

int main ()
{
  int a = 12;
  int& b = a; // definiert Referenz
  int& c = b; // Referenz auf Referenz
  float& d = a; // falscher Typ, Compile-Fehler
  int e = b;
  b = 2;
  c = a * b;
  std::cout << a << std::endl; // 4
  std::cout << e << std::endl; // 12
}
```

Call by Value

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

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

- Diese Aufrufkonvention heißt *call by value*.
- 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:

```
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.
- Wenn keine Änderung erfolgt, sollte eine `const`-Referenz übergeben werden:

```
int sum(const std::vector<int>& x);
```

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!

Zeiger

- Adressvariablen werden mit einem * nach dem Typen definiert:
- Der Adressoperator & liefert die Adresse eine Variable

```
int x = 4;
int* x_p = &x;
```

Hinweis

- `int* x, y;` deklariert den Zeiger `x` und die int-variable `y`.
- Korrekt wäre `int *x, *y;`
- Besser zwei Zeilen:

```
int* x;
int* y;
```

Zeiger

- Adressvariablen werden mit einem * nach dem Typen definiert:
- Der Adressoperator & liefert die Adresse eine Variable

```
int x = 4;
int* x_p = &x;
```

- Adressen müssen **NICHT** bei der Initialisierung zugewiesen werden
- Zugriff auf undefinierte Adressen führt zu undefinierten Verhalten.
- Sonderfall: Adresse `nullptr`:
  - oftmals auch der Wert 0
  - C-Äquivalent zu Makro `NULL`
  - Typ: `nullptr_t`

Zeiger

- Adressvariablen werden mit einem $*$ nach dem Typen definiert:
- Der Adressoperator & liefert die Adresse eine Variable
- Adressen müssen **NICHT** bei der Initialisierung zugewiesen werden
- Zugriff auf undefinierte Adressen führt zu undefinierten Verhalten.
- Sonderfall: Adresse `nullptr`
- Gegensatz zu Referenz: Adressen können geändert werden

```
int x = 4;
int y = 6;
int* p = nullptr;
if(x > y) {
  p = &x;
} else {
  p = &y
}
```

Smart Pointers

- Problem bei Referenzen und Pointer: Speicher dahinter kann ungültig werden, ohne den Zeiger zu ändern: (analog dangling)

```
int* produce(int count)
{
  int* result = new int[count];
  ...
  return result;
}

/* sums up the entries in d */
int sum(int* d, int count)
{
  int r = 0;
  for(int i = 0; i < count; ++i) {
    r += d[i];
  }
  return r;
}
```

```
int main()
{
int* r = produce(4);
int* r2 = r; // kopiert Adresse
delete [] r; // löscht das Array
sum(r2, 4);
return 0;
}
```

Smart Pointers

- Problem bei Referenzen und Pointer: Speicher dahinter kann ungültig werden, ohne den Zeiger zu ändern: (analog dangling)
- Ein Smart Pointer reserviert Speicher für ein Objekt auf dem Heap und räumt das Objekt auf, wenn es nicht mehr verwendet wird.
- `unique_ptr` erzeugt das neue Objekt beim Anlegen und gibt es frei, sobald die Pointer-Variable aufhört zu existieren:

```cpp
#include <memory>

std::unique_ptr<int> foo(int i) {
  return std::make_unique<int>(i);
}

int add(int a, int b) {
  auto p = foo(a);
  return *p + b;
} // memory gets freed here
```

- `unique_ptr` kann nur verschoben werden, nicht kopiert

Smart Pointers für geteilte Objekte

- Oft ist es nicht möglich, einen eindeutigen Eigentümer für ein Objekt festzulegen.
- Hierfür gibt es `shared_ptr`.
- Mehrere `shared_ptr` können auf das gleiche Objekt zeigen.
- Das Objekt wird genau dann freigegeben, wenn der letzte `shared_ptr` auf das Objekt zerstört wird.
- Wichtig: `shared_ptr` immer nur mit `make_shared` anlegen oder aus anderen `shared_ptr`n kopieren!

```
#include <memory>

std::shared_ptr<int> foo(int i) {
  return std::make_shared<int>(i);
}

int add(int a, int b) {
  auto p = foo(a);
  auto p2 = p;
  return *p + *p2 + b;
} // memory gets freed here
```

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`

Container (Datenstrukturen)

Container bzw. Datenstrukturen verwalten eine zusammenhängende Menge von Werten mit bestimmten Eigenschaften. C++ liefert einige praktische Container mit:

- `array` für Listen von Werten, deren Anzahl zur Compile-Zeit bekannt ist
- `vector` für Listen von Werten, deren Anzahl zur Compile-Zeit nicht bekannt ist
- `list` für Listen von Werten, bei denen oft in der Mitte Elemente hinzufügt oder entfernt werden
- `(unordered_)map` für das Zuordnen von Werten zu Schlüsseln beliebigen Typs (z.B. für ein Wörterbuch). Es gibt eine Variante, die die Eintrage nach den Schlüsseln sortiert, und eine, die das nicht tut.
- `pair` Ein Paar von Werten, die unterschiedliche Typen haben können
- `tuple` Eine Liste von Werten, die unterschiedliche Typen haben können

array

- Liste, deren Länge zur Compile-Zeit bekannt ist:

```
std::array<Datentyp,Länge>
```

- Einfachster Typ, der das C++-Container-Interface erfüllt
- Beispiel:

```
#include <array>
#include <iostream>

int main(int argc, char** argv)
argc, char** argv)
{
  std::array<int,4> a = {1,2,3,4};
  std::cout << a.size() << std::endl; // 4
  a[2] = 4;
  std::cout << a[3];
}
```

array

- Liste, deren Länge zur Compile-Zeit bekannt ist:

```
std::array<Datentyp,Länge>
```

- Einfachster Typ, der das C++-Container-Interface erfüllt
- Manchmal sieht man auch C arrays:
  - `int a[4];`,
  - können dynamische Größe haben
  - Die Größe des C-Arrays nur in diesem Scope bekannt

Eigenschaften von Containern

- Container sind Objekte (mehr dazu später).
  - Objekte haben member variables und member functions, die zu dem jeweiligen Objekt gehören und darin gespeichert werden oder auf dieses wirken:

```
p.first = 3; // member variable
a.size();    // member function
```

  - Member function werden auch Methoden genannt.
- Container sind Templates. Templates sind parametrisierte Typen, die als Templateparameter in spitzen Klammern andere Typen oder Konstanten übergeben bekommen.
  - Bei Containern wird hierüber z.B. festgelegt, welchen Typ von Werten sie speichern können.
  - Mehr zu Templates später.
- Der Zugriff auf Inhalte eines Containers erfolgt meistens mit eckigen Klammern:

```
my_array[3];
my_map["key"];
```

vector

Dynamisch anpassbare Liste von Objekten:

```
#include <vector>
#include <iostream>

int main(int argc, char** argv)
{
  std::vector<int> a; // leere Liste
  std::vector<int> b = {1,2,3,4}; // {{1,2,3,4}} funktioniert ebenfalls, aber nur bei i
  std::vector<int> c(20); // Liste mit 20 Einträgen
  std::cout << b.size() << std::endl; // 4
  b[2] = 4;
  a = b; // kopiert den Inhalt
  a.resize(100); // Grösse anpassen
  a.push_back(1); // Wert 1 hinten anfügen
  a.pop_back();   // Letztes Element entfernen
}
```

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

Iterieren über Container

```
std::vector<int> v = {1,2,3,4};
for (int i = 0 ; i < v.size() ; ++i)
  std::cout << v[i] << std::endl;
```

- Aufpassen am Ende! Index wird nicht auf Gültigkeit überprüft.
- Prüfen kostet unnötig viel Zeit, wenn für jeden Eintrag nur wenige Operationen benötigt werden, z.B. Summe
- Bei Fehlern: Ersetze `v[i]` durch `v.at(i)`, gleiche Bedeutung, aber prüft Gültigkeit

Iterieren über Container

```
std::vector<int> v = {1,2,3,4};
for (int i = 0 ; i < v.size() ; ++i)
  std::cout << v[i] << std::endl;
```

- Aufpassen am Ende! Index wird nicht auf Gültigkeit überprüft.
- Prüfen kostet unnötig viel Zeit, wenn für jeden Eintrag nur wenige Operationen benötigt werden, z.B. Summe
- Bei Fehlern: Ersetze `v[i]` durch `v.at(i)`, gleiche Bedeutung, aber prüft Gültigkeit

Vereinfachter Loop zum Iterieren über Standard-Container:

```
std::vector<int> v = {1,2,3,4};
for (int entry : v)
  std::cout << entry << std::endl;
```

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

pair und tuple

Kurze Listen, die Werte von unterschiedlichem Typ speichern können.

- pair speichert genau zwei Werte. Oft benutzt, um zwei Werte aus einer Funktion zurückzugeben:

```
#include <utility>

std::pair<std::string,int> nameAndGrade(int matrikelNr) {
  return std::make_pair("Max",2);
}

std::pair<std::string,int> r = nameAndGrade(42);
std::cout << "Name: " << r.first << std::endl
          << "Note: " << r.second << std::endl;
```

pair und tuple

Kurze Listen, die Werte von unterschiedlichem Typ speichern können.

- tuple speichert mehrere Werte. Etwas komplizierterer Zugriff:

```
#include <tuple>

auto /*std::tuple<int,int,int,int>*/ student(int matrikelNr) {
  return std::make_tuple(1,1,1990, matrikelNr);
}

std::tuple<int,int,int> bday(const tuple<int,int,int,int>& stud) {
  return std::make_tuple(get<0>(stud), get<1>(stud), get<2>(stud));
}

std::tuple<int,int,int, int> stud = student(42);
std::cout << "Tag: " << std::get<0>(stud) << std::endl
          << "Monat: " << std::get<1>(stud) << std::endl
          << "Jahr: " << std::get<2>(stud) << std::endl;
          << "Matnummer " << std::get<3>(stud) << std::endl;
auto bDay = bday(stud);
```

Maps: Nachschlagewerke in C++

- `std::array` und `std::vector` speichern Daten kontinuierlich
  - Einträge addressiert über 0-basierten, konsekutiven Index
  - zulässige Indizes beschränkt auf `[0,size()-1]`
- Ungeeignet für
  - Listen mit “Löchern” in der Indexmenge
  - Negative Indizes
  - Assoziieren von Werten mit Schlüsseln, die keine ganzen Zahlen sind

Maps: Nachschlagewerke in C++

- `std::array` und `std::vector` speichern Daten kontinuierlich
  - Einträge addressiert über 0-basierten, konsekutiven Index
  - zulässige Indizes beschränkt auf `[0,size()-1]`
- Ungeeignet für
  - Listen mit "Löchern" in der Indexmenge
  - Negative Indizes
  - Assoziieren von Werten mit Schlüsseln, die keine ganzen Zahlen sind
- Lösung: Maps als Abbildung von Werten mit Typ `Key` auf Werte mit Typ `Value`:
  - `std::map<Key,Value>` speichert Einträge in sortierter `Key`-Reihenfolge.
  - `std::unordered_map<Key,Value>` speichert Einträge in zufälliger Reihenfolge, ist bei vielen Schlüsseln deutlich schneller.

Maps: Syntax I

- Benötigen `#include <map>` bzw. `<unordered_map>`
- Verwendung identisch, im folgenden nur für `std::map` gezeigt.
- Maps werden immer leer angelegt:

```
std::map <std::string,int> shopping_list;
```

- Einträge werden beim ersten Zugriff angelegt:

```
shopping_list["cookies"] = 3; // create or overwrite value
shopping_list.insert({"cookies",3}); // unsuccessful if key "cookies" exists
```

- Fehlende Einträge werden beim Abfragen mit dem Standardwert (bei Zahlen: 0) initialisiert:

```
shopping_list["biscuits"]; // returns 0
```

- Die Grösse der Map kann wieder mit `size()` bestimmt werden:

```
shopping_list.size(); // returns 2 (cookies and biscuits)
```

Maps: Syntax II

- Testen, ob ein Eintrag in der Map enthalten ist:

```
// returns 1, as there is 1 entry for key biscuits
shopping_list.count("biscuits");
// returns 0, as there is no entry for key crisps
shopping_list.count("crisps");
```

  Hier wird eine Anzahl zurückgegeben, weil die Bibliothek auch Multi-Maps enthält, die einem Schlüssel mehrere Einträge zuordnen können.

- Eintrag löschen:

```
// returns 1, because 1 element removed
shopping_list.erase("biscuits");
// returns 0, because 0 elements removed
shopping_list.erase("crisps");
```

- Map komplett leeren:

```
shopping_list.clear();
```

Iterieren über Container mit Referenzen

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

```
#include <vector>
#include <iostream>

int main(int argc, char** argv)
{
        std::vector<int> v = {1,2,3,4};
        for (int& value : v)
        value *= 2;
}
```

Iterieren über Maps (mit Referenzen)

- Beim Iterieren möchten wir auf Schlüssel und zugeordneten Wert zugreifen können.
- Verbesserter `for`-Loop liefert Referenz auf `std::pair<const Key,Value>` zurück:

```
for (std::pair<const std::string,int>& entry : shopping_list)
std::cout << entry.first << ": "
<< entry.second << std::endl;
```

- Bei Verwendung von `std::map` werden die Einträge nach aufsteigender `Key`-Reihenfolge sortiert abgelaufen.
- Bei Verwendung von `std::unordered_map` ist die Reihenfolge der Einträge zufällig.
- Für Fortgeschrittene: `std::map` basiert auf einem sortierten Binärbaum, `std::unordered_map` auf einer Hashtable.

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:

```
#include <vector>
#include <algorithm>

int main(int argc, char** argv)
{
  std::vector<int> a = .....;
  // sortiert a nach aufsteigenden Zahlenwerten
  std::sort(a.begin(),a.end());
}
```

Fragen?

C++-Projekte jenseits kleiner Übungen

Echte C++-Projekte sind um ein vielfaches grösser als unsere bisherigen Programme in den Übungen. Dies bringt neue Herausforderungen mit sich:

- Code-Strukturierung: Der Quellcode ist auf mehrere Dateien aufgeteilt, die jeweils zusammenhängende Funktionen enthalten.
- Code-Reuse I: Für viele Funktionen wird auf externe Bibliotheken jenseits der Standardbibliothek zurückgegriffen (grafische Oberflächen, Netzwerk, . . .).
- Code-Reuse II: Wiederverwendbare Teile des eigenen Programms sollen als Bibliothek zur Verfügung stehen.

Wichtig hierfür:

C++-Projekte jenseits kleiner Übungen

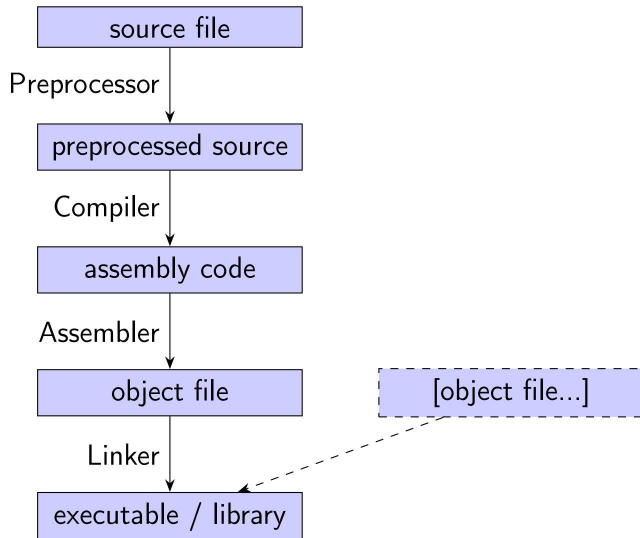
Echte C++-Projekte sind um ein vielfaches grösser als unsere bisherigen Programme in den Übungen. Dies bringt neue Herausforderungen mit sich:

- Code-Strukturierung: Der Quellcode ist auf mehrere Dateien aufgeteilt, die jeweils zusammenhängende Funktionen enthalten.
- Code-Reuse I: Für viele Funktionen wird auf externe Bibliotheken jenseits der Standardbibliothek zurückgegriffen (grafische Oberflächen, Netzwerk, . . .).
- Code-Reuse II: Wiederverwendbare Teile des eigenen Programms sollen als Bibliothek zur Verfügung stehen.

Wichtig hierfür:

- Verständnis des Kompilier-/Buildprozesses
- Aufteilung von Code auf mehrere Dateien
- Verwaltung der Abhängigkeiten zwischen Dateien

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.
  - Text zwischen `#ifndef MACRO` und `#endif` wird entfernt, wenn das angegebene Makro 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 `https://godbolt.org` anschauen.

Der Assembler

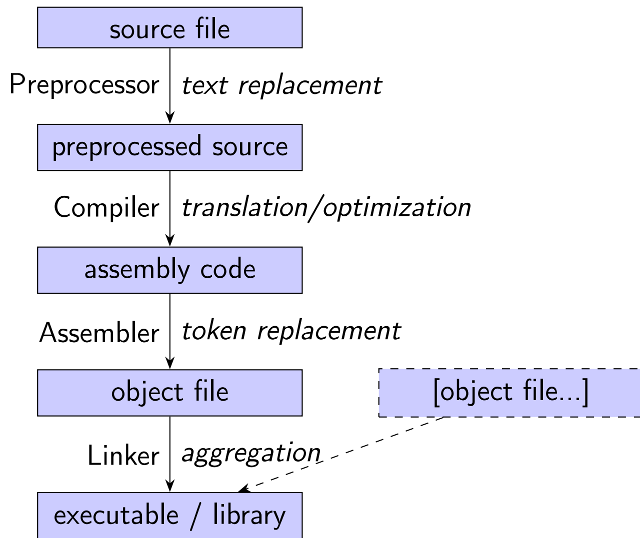
- Der Assembler verwandelt Assembly-Code in die binären Befehlscodes, die der Prozessor versteht.
- Der Assembler produziert sogenannte *object files* mit der Erweiterung `.o`.
- Achtung: das hat nichts mit den "Objekten" aus der objektorientierten Programmierung zu tun!
- Object files enthalten *object code*, das Endprodukt des Kompiliervorgangs im engeren Sinne.
- 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 das Endprodukt des Build-Prozesses:
  - Ausführbare Dateien (executables), die von der Kommandozeile aufgerufen werden können.
  - Bibliotheken (libraries), die Funktionen enthalten und diese für andere Programme / Bibliotheken zur Verfügung stellen.
- 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:

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

Der C++-Kompilierprozess Revisited



Programme mit mehreren Dateien

```
double cube(double x)
{
  return x * x * x;
}
```

- Funktionen, die man mehrfach verwendet, sollte man in eine eigene Datei auslagern:
  - Einfache Wiederverwendbarkeit.
  - Bessere Übersichtlichkeit bei grösseren Programmen.
- Man benötigt meistens zwei Dateien:
  - Immer ein *header file*, das von anderen Dateien eingebunden werden kann und alle Funktionalität, die wir bereitstellen, *deklariert*.
  - Ein *implementation file*, das die eigentliche Implementierung enthält. Dies kann in manchen Situationen (siehe Templates) entfallen.

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:

```
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 (`.hh`-Datei, der Inhalt taucht in jeder `.cc`-Datei auf, die den Header inkludiert).
  - Definition → Implementation (`.cc`-Datei).

Beispiel

cube.hh

```
// function for calculating the cube of a double
double cube(double x); // <-- declaration
```

cube.cc

```
#include "cube.hh" // preprocessor: replaced by declaration
double cube(double x) // <-- definition
{
  return x * x * x;
}
```

main.cc

```
#include <iostream>
#include "cube.hh" // also replaced by declaration
int main(int argc, char** argv)
{ // meaning of "cube" is clear thanks to declaration
  std::cout << cube(3.0) << std::endl;
}
```

Header Guards

- Echte Programme inkludieren Header oft mehrmals in einer Translation Unit (`.cc`-Datei).
  - langsam
  - problematisch bei Makro-Definitionen
- Lösung: Header Guard

```
#ifndef CUBE_HH
#define CUBE_HH

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

#endif // CUBE_HH
```

- Für den Namen des Guard-Makros nimmt man am besten den Dateinamen des Headers als Vorlage.

Header Guards

- Echte Programme inkludieren Header oft mehrmals in einer Translation Unit (.cc-Datei).
  - langsam
  - problematisch bei Makro-Definitionen
- Lösung: Header Guard

```
#ifndef CUBE_HH
#define CUBE_HH

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

#endif // CUBE_HH
```

- Für den Namen des Guard-Makros nimmt man am besten den Dateinamen des Headers als Vorlage.

Hinweis

Alle Header-Dateien, die Sie in diesem Kurs schreiben, müssen einen Header-Guard haben!

Kompilieren des Projekts

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

Probleme:

- Viel Tipparbeit
- Probleme bei späteren Änderungen:
  - Welche Datei inkludiert welche andere?
  - Was muss ich alles erneut ausführen, wenn ich eine Datei verändere?

Kompilieren des Projekts

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

Probleme:

- Viel Tipparbeit
- Probleme bei späteren Änderungen:
  - Welche Datei inkludiert welche andere?
  - Was muss ich alles erneut ausführen, wenn ich eine Datei verändere?

⇒ Automatisierung des Prozesses durch 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.
- CMake kann testen, ob das Betriebssystem bestimmte Features hat, und den Buildprozess daran anpassen.
- CMake unterstützt unterschiedliche Build-Konfigurationen:
  - Debug (für Entwicklung und Fehlersuche)
  - Release (generiert schnellere Programme für die spätere Nutzung: aktiviert Optimierung)
- CMake trennt sauber zwischen
  - Quellcode-Verzeichnis (enthält `.cc`-Dateien etc.)
  - Build-Verzeichnis (enthält alles, was automatisch generiert wird, z.B. Programme)
- CMake ist streng genommen ein Build System Generator, es erzeugt eine Konfiguration für andere Build Systems, die dann für das eigentliche Bauen verwendet werden.

CMakeLists.txt

- CMake wird über Dateien mit dem festen Namen `CMakeLists.txt` konfiguriert.
- Dateien beschreiben, wie das Projekt konfiguriert werden soll und welche Programme und Bibliotheken aus welchen `.cc`-Dateien gebaut werden sollen.
- Minimalbeispiel:

```
# Set minimum required CMake version
cmake_minimum_required(VERSION 3.5)
# Start project and set its name to ipk-demo
project(ipk-demo LANGUAGES CXX)

# Force compiler to run in C++14 mode
set(CMAKE_CXX_STANDARD 14)

# Create executable programs
add_executable(cube cubemain.cc cube.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/`:

```
mkdir build
cd build
cmake ..
```

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

CMake-Feintuning

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

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

Wichtige Variablen sind:

- `CMAKE_CXX_COMPILER` Der gewünschte C++-Compiler (g++, clang++, etc.)
- `CMAKE_CXX_FLAGS` Zusätzliche Flags für den Compiler, z.B. `-Wall` etc.
- `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.

Wichtige CMake-Befehle

- Eine ausführbare Datei anlegen:

```
add_executable(<name> <.cc-Datei>...)
```

- Eine Bibliothek anlegen:

```
add_library(<name> <.cc-Datei>...)
```

- Target (executable oder library) gegen eine Bibliothek linken:

```
target_link_libraries(<target> PUBLIC <library>...)
```

- Unterstützung für automatische Tests aktivieren:

```
enable_testing()
```

- Test anlegen:

```
add_executable(calculator_test calculator_test.cc...)
add_test(NAME calculator_test COMMAND calculator_test)
```

Tests mit CMake

- Tests sind ein essentieller Bestandteil guter Programmierung!
- Tests prüfen, ob eine Funktion für bestimmte Inputs das erwartete Resultat produziert.
- In CMake ist ein Test ein normales Programm, das sich an folgende Konvention hält:
  - Wenn der Test erfolgreich war, gibt `main()` 0 zurück.
  - Wenn der Test fehlgeschlagen ist, gibt `main()` $0 < n < 127$ zurück.
- Vor dem Anlegen von Tests muss man `enable_testing()` aufrufen.
- Mit dem Befehl `ctest` führt CMake alle Tests aus und zeigt die Resultate auf der Konsole an.