Erste Schritte mit C++

Vorkurs C/C++, Olaf Bergmann

Was wird benötigt?

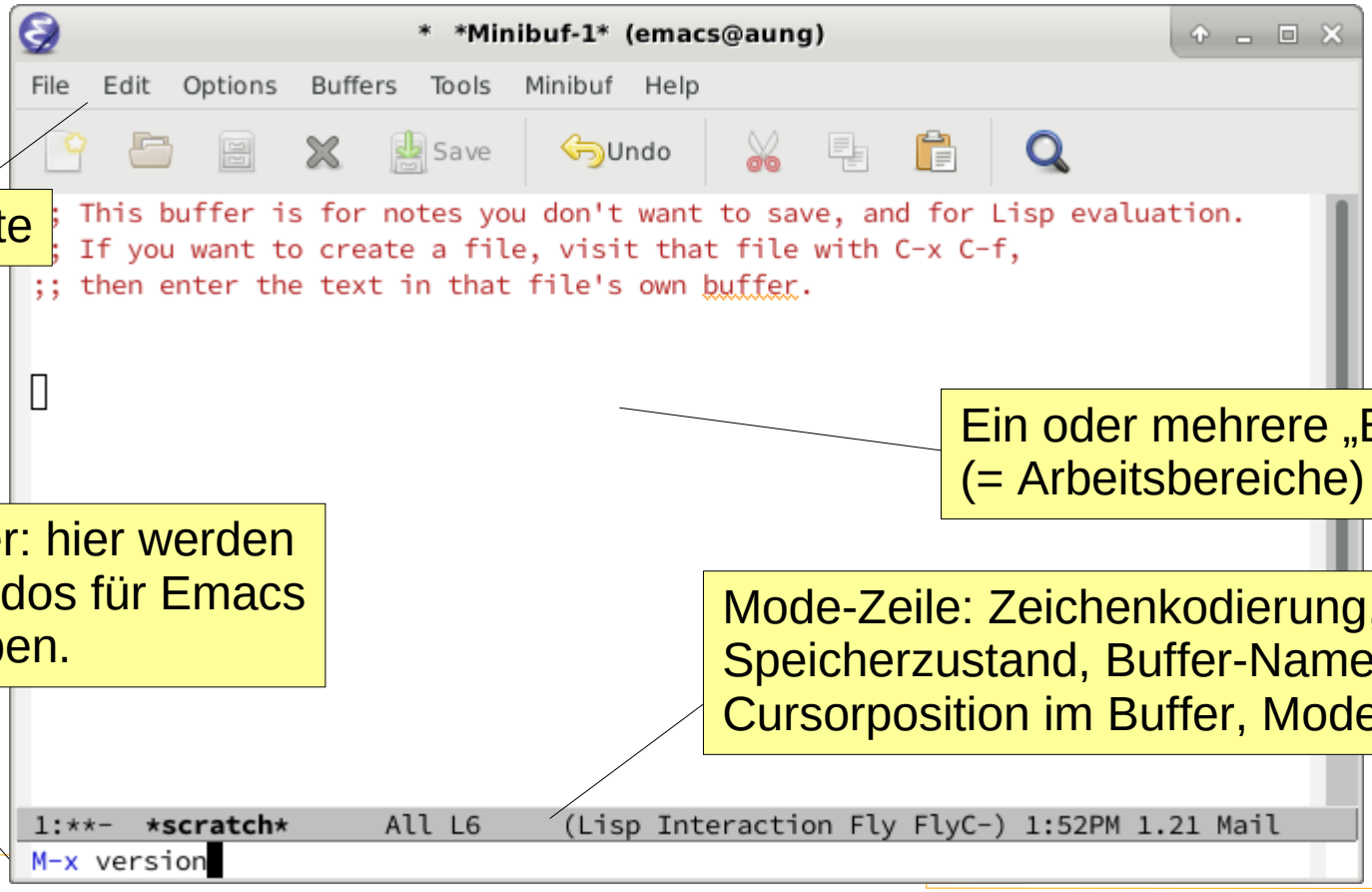
- Entwicklungsumgebung, z. B. Linux/WSL
  - Editor (z. B. **Emacs**, vi, atom, Eclipse)
  - C/C++-Compiler (gcc/g++, clang/clang++)
  - Debugger (gdb)
  - Make
- Optional Cygwin mit Emacs, gcc/g++ usw.
  - für Programmiertest verwendet

- Umgang mit der Kommandozeile

Der Editor Emacs

```
user@c++:~$ emacs &
```

Startet Emacs im Hintergrund, Arbeitsverzeichnis ist aktuelles Verzeichnis (/home/user)



Menüleiste

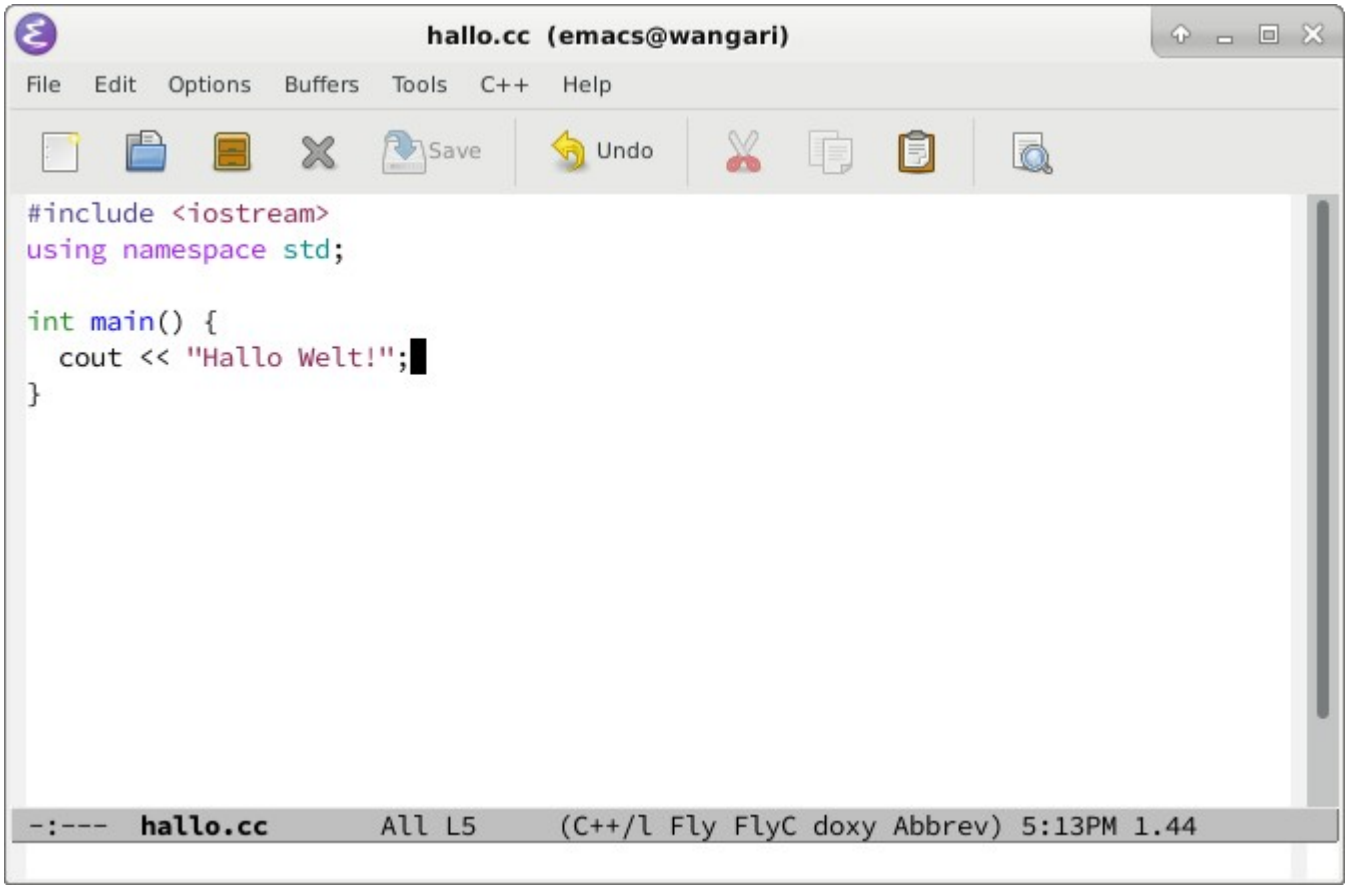
Ein oder mehrere „Buffer“ (= Arbeitsbereiche)

Minibuffer: hier werden Kommandos für Emacs eingegeben.

Mode-Zeile: Zeichenkodierung, Speicherzustand, Buffer-Name, Cursorposition im Buffer, Mode...

Ein C-Programm im Emacs

```
user@c++:~$ emacs hallo.cc &
```



Einige Emacs-Kommandos

- C-_: rückgängig
- C-g: Abbruch
- C-x C-c: Emacs beenden
- C-x C-f: Datei laden/erzeugen
- C-x C-s: Datei speichern
- C-x C-w: Datei unter anderem Namen speichern
- C-x b: Buffer wechseln
- C-x C-b: Bufferliste
- C-x k: aktuellen Buffer löschen
- C-x 0: aktuelles Fenster löschen
- C-x 1: anderes Fenster löschen
- C-x o: Fenster wechseln

M: „Meta“ (ESC oder linke Alt-Taste)
C: „Ctrl“ (linke Steuerungstaste)
S: „Shift“ (linke Umstelltaste)

M-x: Kommando ausführen
(z. B. M-x version)

- C-s: Vorwärtssuche im Buffer
- C-k: aktuelle Zeile löschen
- C-Leertaste: Blockanfang markieren
- M-w: Block kopieren
- C-w: Block ausschneiden
- C-y: Block einfügen

Erste Schritte

```
#include <iostream>

using namespace std;

int main() {
  cout << "Hallo Welt!";
}
```

- Speichern in `hallo.cc`
- Übersetzen und zusammenbinden
  `c++ -o hallo hallo.cc`
- Ausführen
  `./hallo`
  → `Hallo Welt!`

C++-Compiler

Default-Compiler: c++

```
ssh m04 "c++ -v"
→ Apple clang version 14.0.3 …
```

```
ssh x06 "c++ -v"
→ gcc version 12.3.0 (Ubuntu 12.3.0-
1ubuntu1~22.04)
```

Im Zweifel g++ aufrufen

Im Folgenden
immer g++
(Mac OS: `clang++`)

C++98, C++03, C++11, . . ., C++20

- C++98 manchmal noch Grundeinstellung

```
std::vector<double> v = { 1, 5.7, 3, 8.5 }
```

```
→ g++     i.cc   -o i
error: in C++98 'v' must be initialized by constructor, not by '{...}'
```

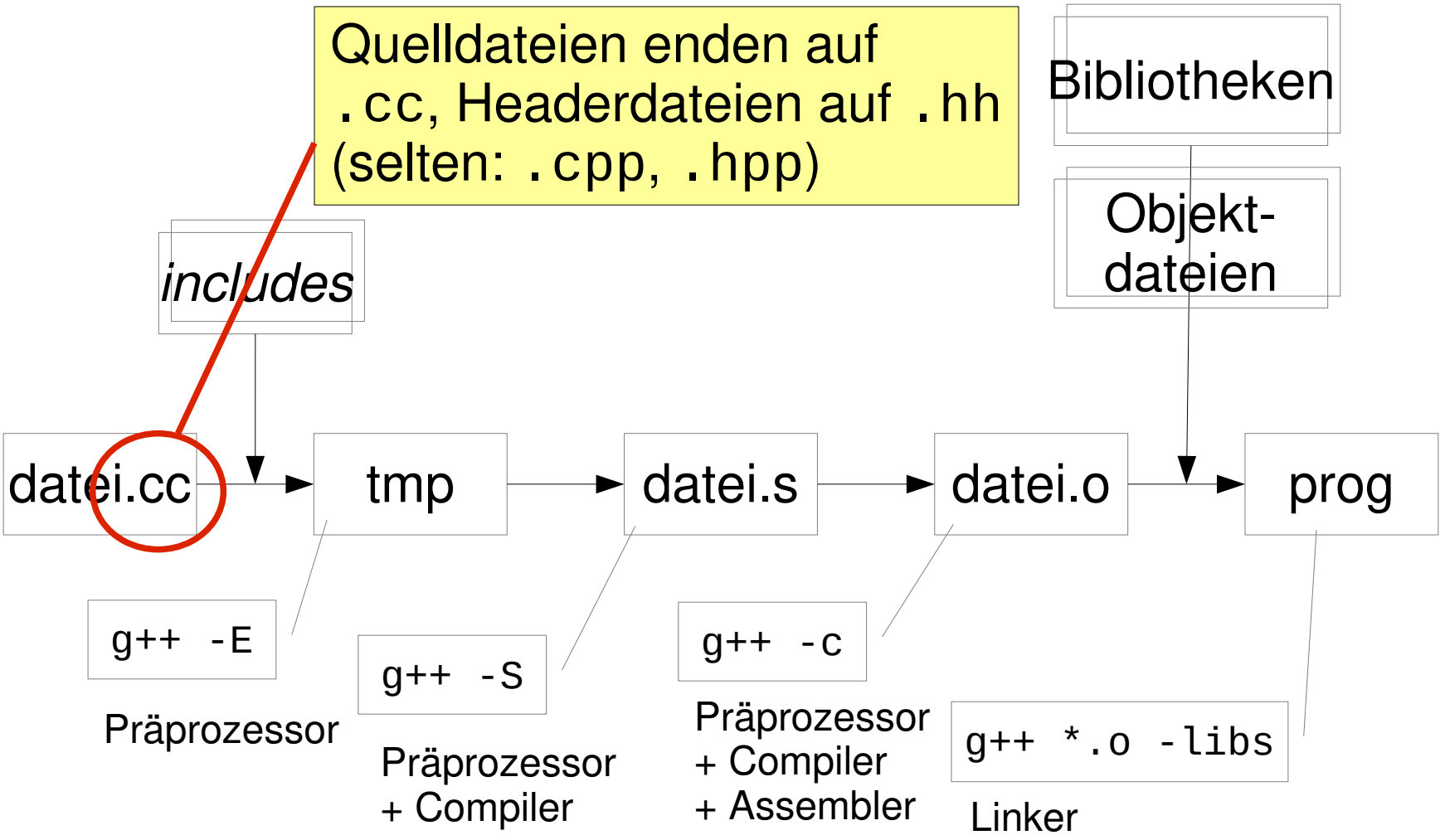
- Nach Möglichkeit immer C++20

`g++ -std=c++20` (ältere Compiler: `g++ -std=c++2a`)

  - `g++ --help=c++`

Fallback: C++17
-std=c++17
oder
-std=c++1z

Compilerpässe in C++



gcc-Flags

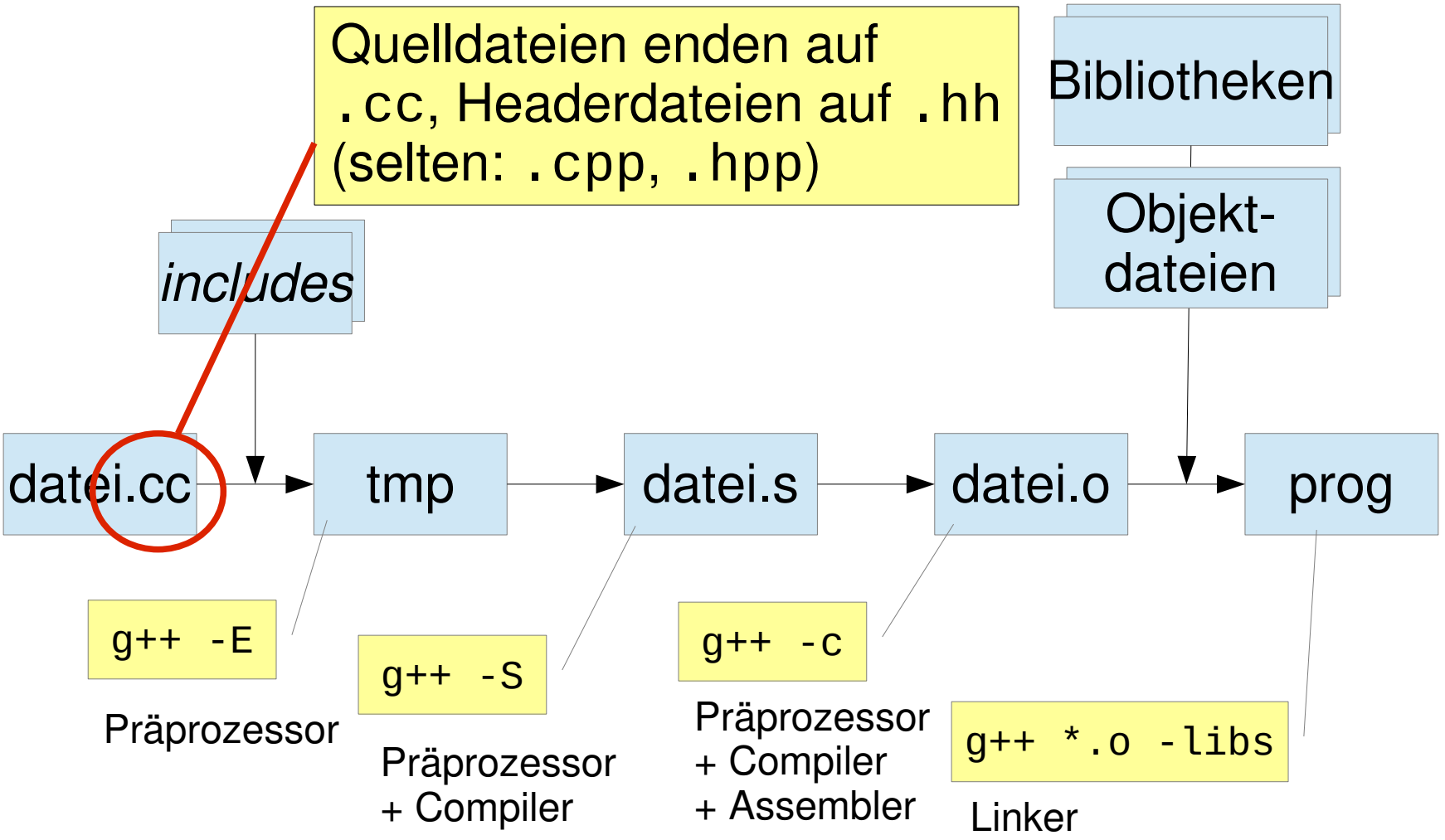
–E	Präprozessor
–S	Präprozessor und Compiler
–c	Präprozessor und Compiler und Assembler
–O	(„groß ooh“) Optimizer einschalten (üblich: –O2)
–v	Einzelne Pässe, Optionen und Argumente anzeigen
–Wall	Viele Warnungen ausgeben
–Wextra	Weitere wichtige Warnungen
–g	Debug-Symbole einfügen (→ gdb)
–o xxx	Linker: Ausgabedatei xxx anlegen (sonst a.out)
–lxxx	(„klein ell“) Linker: mit Bibliothek xxx linken
–Ixxx	(„groß iih“) Präprozessor: Verzeichnis xxx in den Suchpfad aufnehmen
-Lxxx	Linker: Verzeichnis xxx in den Suchpfad aufnehmen

Werkzeugkasten: make

Vorkurs C/C++, Olaf Bergmann

Compilerpässe in C++

Wdh.



gcc-Flags

Wdh.

–E	Präprozessor
–S	Präprozessor und Compiler
–c	Präprozessor und Compiler und Assembler
–O	Optimizer einschalten (üblich: –O2)
–v	Einzelne Pässe, Optionen und Argumente anzeigen
–Wall	Viele Warnungen ausgeben
–Wextra	Weitere wichtige Warnungen
–g	Debug-Symbole einfügen (→ gdb)
–o xxx	Linker: Ausgabedatei xxx anlegen (sonst a.out)
–lxxx	Linker: mit Bibliothek xxx linken
–Ixxx	Präprozessor: Verzeichnis xxx in den Suchpfad aufnehmen
-Lxxx	Linker: Verzeichnis xxx in den Suchpfad aufnehmen

Exkurs: Make

- Wer merkt sich die ganzen Compilerflags?
  `g++ -std=c++20 -Wall -Wextra -Isubdir -Llibs ...`

- make!
  - `make CXXFLAGS=-std=c++20 CPPFLAGS=...`

Makefile:

```
CXXFLAGS=-std=c++20
CPPFLAGS=-Wall -Wextra -Isubdir
LDLIBS=-lm
```

```
-> make i
g++ -std=c++20 -Wall -Wextra -Isubdir   i.cc  -lm -o i
```

Exkurs: Makefiles

- Make hilft bei getrennter Übersetzung

Makefile:

```
CXXFLAGS=-std=c++20
CPPFLAGS=-Wall -Wextra -Isubdir
LDLIBS=-lm
LINK.o=$(LINK.cc)       # use C++ linker for .o

# i depends on i.o and j.o (from i.cc and j.cc)
i:: i.o j.o

# the first target is the default
all: i
```

```
-> make
g++ -std=c++20 -Wall -Wextra -Isubdir -c -o i.o i.cc
g++ -std=c++20 -Wall -Wextra -Isubdir -c -o j.o j.cc
g++ i.o j.o -lm -o i
```

mit Compilerangabe, z. B.:
make CXX=clang++

Exkurs: Makefiles

```
-> make
g++ -std=c++20 -Wall -Wextra -Isubdir -c -o i.o i.cc
g++ -std=c++20 -Wall -Wextra -Isubdir -c -o j.o j.cc
g++ i.o j.o -lm -o i
```

```
-> make
make: `i' is up to date.
```

Keine Änderungen, keine neue Übersetzung

```
-> touch j.cc
-> make
g++ -std=c++20 -Wall -Wextra -Isubdir -c -o j.o j.cc
g++ i.o j.o -lm -o i
```

j.cc wurde geändert
neu übersetzen und alles linken

Exkurs: Makefiles

- Regeln, Rezepte und Variablen

Lesen und nie wieder ohne make aus dem Haus gehen:
http://www.gnu.org/software/make/manual/

Makefile:

```
CXXFLAGS=-std=c++20
CPPFLAGS=-Wall -Wextra -Isubdir
LDLIBS=-lm
LINK.o=$(LINK.cc)
PACKAGE?=c++-kurs

# i depends on i.o and j.o (from i.cc and j.cc)
i:: i.o j.o

# the first target is the default
all: i

dist: all
    zip -r $(PACKAGE).zip *.cc subdir Makefile

%.pdf: %.tex
    for f in $$(seq 1 3); do pdflatex $*; done
```

Exkurs: Makefile mit Bedingung

- Aufrufen mit `make GTK_VERSION=3.0`

Makefile:

```
CXXFLAGS=-std=c++14
CPPFLAGS=-Wall -Wextra -Isubdir
LDLIBS=-lm
LINK.o=$(LINK.cc)

gtkmm_config = $(shell pkg-config gtkmm-$(GTK_VERSION) $1)

ifneq (${GTK_VERSION}, )
CPPFLAGS += $(call gtkmm_config,--cflags)
LDFLAGS += $(call gtkmm_config,--libs-only-L)
LDLIBS += $(call gtkmm_config,--libs-only-l)
endif

# ...
```

Container und Iteratoren: Strings

Vorkurs C/C++, Olaf Bergmann

String

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string name = "Olaf";
  name = name + ' ';
  name += "Bergmann";

  cout << "Hallo, " << name << endl;
}
```

Stringoperationen

```
string s;
s = string("Olaf") + string("Bergmann");

string::size_type pos = s.find("Berg");
if (pos != string::npos) {
  s.insert(pos, " ");
}

cout << s << endl;
```

Iteratoren

```
#include <string>
#include <cctype>
...
using namespace std;

string s("Hello");
string up;

for (string::iterator i = s.begin(); i != s.end(); ++i) {
  up += toupper(*i);
}
```

Verweis auf erstes Element

Erzeugt geeigneten Iterator, weitere:
`const_iterator`
`reverse_iterator`

Verweis hinter das letzte Element (darf nicht dereferenziert werden!)
== und != sind immer definiert

value_type

```
#include <string>
#include <cctype>
...
using namespace std;

string s("Hello");

for (string::const_iterator i=s.begin(); i!=s.end(); ++i) {
  string::value_type c = *i;
  std::cout << c;
}
```

Vector

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


using namespace std;


int main() {
  vector<string> zoo = { "Giraffe", "Panther",
                         "Leopard", "Affe",
                         "Gepard", "Puma" };

  for (vector<string>::const_iterator i=zoo.begin();
       i != zoo.end(); i++) {
    cout << *i << endl;
  }
}
```

Typinferenz: auto

Schlüsselwort `auto` als Abkürzung:

```
vector<string> zoo = { "Giraffe", ... };

for (vector<string>::const_iterator i = zoo.begin(); i!=zoo.end(); ++i) {
  vector<string>::value_type tier = *i;
  /* ... */
}
```

…::value_type
ergibt sich aus *i

auto:
Typ zur zur Übersetzungszeit ermitteln

```
Typ::iterator i;
auto v = *i;     → Typ::value_type
```

```
vector<string> zoo = { "Giraffe", ...

for (vector<string>::const_iterator i = zoo.begin(); i!=zoo.end(); ++i) {
  auto tier = *i;
  /* ... */
}
```

Syntaktischer Zucker:

```
for (auto tier : zoo) {
  cout << tier << endl;
}
```

Stringoperationen

```
string text = "Eine Zeichenkette";

for (size_t i = 0; i < text.size(); i++) {
  cout << text[i] << ",";
}
cout << endl;
```

Zugriff auf Element an Position i

liefert Anzahl der Elemente

```
string text = "Eine Zeichenkette";

for (auto c : text) {
  cout << c << ",";
}
cout << endl;
```

Iteriert über Container text,
c repräsentiert aktuelles Element

Eingabe- und Ausgabeströme

Vorkurs C/C++, Olaf Bergmann

Vordefinierte Ausgabestreams

- `cout`: Standard-Ausgabe (flush nach Newline)
- `cerr`: Standard-Fehlerkanal (jedes Zeichen sofort)
- `clog`: Logs (üblicherweise = `cerr`)

```
if (code == ok) {
  cout << "Ausgabe" << endl;
} else {
  cerr << "Error" << get_error(code) << endl;
}
```

```
cout << "Ausgabe ohne Newline" << flush;
```

Vordefinierter Eingabestream

- `cin`: Standard-Eingabe
  - liest Daten vom Unix-Eingabekanal (meistens Tastatur)
  - Nutzung mit Operator >>

```
int zahl;

cin >> zahl;

if (cin)
  cout << "Zahl war: " << zahl << endl;
```

Strings einlesen

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


using namespace std;


int main() {
  vector<string> zoo;

  while (cin) {
    string tier;
    if (cin >> tier)
      zoo.push_back(tier)
  }

  cout << "**** Tiere im Zoo ****" << endl;
  for (auto tier : zoo) {
    cout << tier << endl;
  }
}
```

Einlesen einer ganzen Zeile mit `getline(cin, buf)`

Gleitkommazahlen einlesen

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


using namespace std;


int main() {
  vector<float> zahlen;

  while (cin) {
    float wert;
    if (cin >> wert)
      zahlen.push_back(wert)
  }

  cout << "**** Statistik ****" << endl;
  for (auto z : zahlen) {
    cout << z << ", ";
  }
  cout << endl;
}
```

Container 2: Algorithmen

Vorkurs C/C++, Olaf Bergmann

Algorithmen auf Containern

```
#include <algorithm>
...
int main() {
  vector<float> zahlen;
  while (cin) {
    float wert;
    if (cin >> wert)
      zahlen.push_back(wert)
  }


  sort(zahlen.begin(), zahlen.end());

  cout << "**** Statistik ****" << endl;
  for (auto z : zahlen) {
    cout << z << ", ";
  }
  cout << endl;
}
```

Algorithmen auf Containern

```
#include <algorithm>
...
int main() {
  vector<string> zoo;
  while (cin) {
    string tier;
    if (cin >> tier)
      zoo.push_back(tier)
  }

  sort(zoo.begin(), zoo.end());

  cout << "**** Tiere im Zoo ****" << endl;
  for (auto tier : zoo) {
    cout << z << endl;
  }
}
```

Algorithmen auf Containern

```
void print(float z) {
  cout << z << ' ';
}

int main() {
  vector<float> zahlen;
  while (cin) {
    float wert;
    if (cin >> wert)
      zahlen.push_back(wert)
  }

  cout << "**** Statistik ****" << endl;
  for_each(zahlen.begin(), zahlen.end(), print);

  cout << endl;
}
```

Funktionsargument muss kompatiblen Typ haben

Einige C++-Container

`vector`	Folge von Elementen (flexibles Array)
`list`	doppelt verkettete Liste
`queue`	FIFO-Warteschlange (als Wrapper von `deque`)
`deque`	Beidseitig lesbare/schreibbare Warteschlange
`map`	Abbildung Schlüssel → Wert
`multimap`	wie `map`, aber Schlüssel können mehrfach auftreten
`unordered_map` **`unordered_multimap`**	`map` bzw. `multimap` mit Hash-Tabelle
`set`	geordnete Menge von Schlüsseln
`multiset`	wie `set`, aber Schlüssel können mehrfach auftreten
`unordered_set`	`set` mit Hash-Tabelle
`array`	Array mit fester Größe

Beispiel: map

```
#include <iostream>
#include <string>
#include <map>

int main() {
  map<unsigned int, string> plz;

  plz[28359] = "Bremen-Horn";
  plz[27283] = "Verden (Aller)";

  for (auto stadt : plz) {
    cout << get<0>(stadt) << " " << get<1>(stadt)
         << endl;
  }
}
```

plz enthält Paare aus unsigned int und string

Elemente hinzufügen

alternativ: `stadt.first`

erstes Element aus Tupel (Zahl, String) ermitteln

alternativ: `stadt.second`

zweites Element aus Tupel

Beispiel: map

```
#include <iostream>
#include <string>
#include <map>

int main() {
  map<unsigned int, string> plz;

  plz[28359] = "Bremen-Horn";
  plz[27283] = "Verden (Aller)";

  cout << plz[12345] << endl;

  for (auto stadt : plz) {
    cout << get<0>(stadt) << " " << get<1>(stadt)
         << endl;
  }
}
```

Achtung: [ ] fügt ein neues Element hinzu

map: Elemente finden

```
#include <iostream>
#include <string>
#include <map>

int main() {
  map<unsigned int, string> plz;

  plz[28359] = "Bremen-Horn";
  plz[27283] = "Verden (Aller)";

  auto element = plz.find(12345);
  if (element != plz.end()) {
    cout << element->second << endl;
  }
  for (auto stadt : plz) {
    cout << get<0>(stadt) << " " << get<1>(stadt)
         << endl;
  }
}
```

element "zeigt" auf (Zahl, String) oder plz.end()

Aliase für Datentypen

```
#include <iostream>
#include <string>
#include <map>

int main() {
  using Postleitzahlen = map<unsigned int, string>;
  Postleitzahlen plz;

  plz[28359] = "Bremen-Horn";
  plz[27283] = "Verden (Aller)";

  for (auto stadt : plz) {
    cout << get<0>(stadt) << " " << get<1>(stadt)
         << endl;
  }
}
```

Initialisierung

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

int main() {
  using Postleitzahlen = map<unsigned int, string>;
  Postleitzahlen plz = {
    { 24147, "Kiel"},
    { 30167, "Hannover" }
  };

  plz[28359] = "Bremen-Horn";
  plz[27283] = "Verden (Aller)";

  for (auto stadt : plz) {
    cout << get<0>(stadt) << " " << get<1>(stadt)
         << endl;
  }
}
```

Typische Operationen auf Containern

Alle Container-Typen

begin() rbegin()
end() rend()

==, !=, <, <=, >, >=

= swap() empty()

size(), max_size()

insert(), erase(), clear()

Assoziativ

set, multiset, map, multimap

find(), count()

lower_bound()

equal_range()

key_comp()
value_comp()

Sequentiell

list, vector, deque, array

front(), back()

assign()

push_back()
pop_back()

resize()

+ weitere

[ ]
at()

+ weitere

push_front()
pop_front()

splice()

reverse() ...

+ weitere

Membertypen von Containern

`value_type`	Elementtyp
`size_type`	Typ für Index, Anzahl Elemente usw.
`difference_type`	Typ für Iterator-Abstände
`iterator`	Iterator-Typ
`const_iterator`	Konstanter Iterator-Typ
`reverse_iterator`	Iterator-Typ (iteriert in umgekehrter Reihenfolge)
`const_reverse_iterator`	Konstanter Rückwärts-Iterator
`reference`	`value_type &`
`const_reference`	`const value_type &`
`pointer`	`value_type *`
`const_pointer`	`const value_type *`
`key_type`	Schlüsseltyp bei assoziativen Containern
`mapped_type`	Bildtyp bei assoziativen Containern
`key_compare`	Typ für Vergleichskriterium bei ass. Containern
`allocator_type`	Speicherallokator

Klassen und Vererbung 1

Vorkurs C/C++, Olaf Bergmann

Klassen

- Klassen als benutzerdefinierte Typen
  - Ermöglicht objektorientierte Programmierung
  - Garantierte Initialisierung und Finalisierung
  - Steuerbare implizite Typumwandlung
  - Dynamische Typung (Bindung)
- Volle Typüberprüfung

```
class Point {
public:
  double x, y;
  void draw();
};
```

Memberfunktionen („Methoden“)

- Inline (vgl. Java)

```
class Point {
public:
  double x, y;
  void draw() { … }
};
```

- Trennung Interface/Implementierung

```
class Point {
public:
  double x, y;
  void draw();
};
```

```
void Point::draw() {

  ...

}
```

struct vs class

- `struct`: Struktur + Memberfunktionen
  - Logische Erweiterung von C-Strukturen
  - Information hiding?
- Drei Zugriffsebenen
  - public: alle
  - private: nur das Objekt
  - protected: Objekt und Instanzen abgeleitete
- `public` ist Default für `struct`
- Besser: alles verbieten, was nicht ausdrücklich erlaubt ist
  - `private` ist Default für `class`

```
struct Point {
  double x, y;
  void draw();
};
```

```
class Point {
private:
  double x, y;
public:
  void draw();
};
```

Vererbung

```
class Point {
private:
  double x, y;
public:
  void draw();
};
```

```
class NewPoint : public Point {
public:
  void draw();
};
```

```
NewPoint np;
Point p;

np.draw();
p.draw();
```

NewPoint::draw()
Point::draw()

Referenzen

Vorkurs C/C++, Olaf Bergmann

Referenzen

- „selbstdereferenzierende Pointer“
  - Initialisierung mit Objekt
  - Danach ist „Referenz“ konstant
  - „dangling Pointer“ schwieriger herzustellen

```
int i = 17, j = 28;
int &falsch;     // FEHLER: uninitialisiert
int &ir = i;     // ir jetzt Alias für i
ir = 12;         // i ist jetzt 12
ir = j;          // i ist jetzt 28
int *ip = &ir;   // ip ist jetzt &i
int &ir2 = ir;   // ir2 ist jetzt Alias für i
```

auto &

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

using namespace std;

int main() {
  vector<float> zahlen;
  float wert;

  while (cin >> wert) {
      zahlen.push_back(wert);
  }

  for (auto  z : zahlen) {
    z = z * z;
  }
}
```

Frage: Was steht in `zahlen`?

auto &

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

using namespace std;

int main() {
  vector<float> zahlen;
  float wert;

  while (cin >> wert) {
      zahlen.push_back(wert);
  }

  for (auto &z : zahlen) {
    z = z * z;
  }
}
```

z ist Referenz auf Element in `zahlen`, d. h. schreibender Zugriff verändert das Element

Call-by-reference

- Parameterübergabe ist auch Initialisierung

```
void swap(int& i, int& j) {
  int t = i;
  i = j;
  j = t;
}
int n, m;
swap(n, m);
```

in C nur mit Pointern möglich:

```
void swap(int *i, int *j) {  ... }

int zahl1, zahl2;
swap(&zahl1, &zahl2);
```

- Referenzparameter für
  - Ausgabe- und Eingabeparameter
  - Übergabe von großen Daten
  - `const`-Deklaration macht Eingabe deutlich

```
struct Huge {
  double my_data[1048576];
};
double mean(const Huge&);
```

Return-by-reference

- Auch Rückgabewerte können Referenzen sein

```
int& index(int p[], int i) {
  return p[i];
}
int a[10];
index(a, 6) = i;
```

- Konstante Referenzen: z. B. für Rückgabe größerer Objekte ohne Schreibberechtigung

```
const Huge& find_right_huge();
```

Klassen und Vererbung 2

Vorkurs C/C++, Olaf Bergmann

Polymorphie

```
class Point {
private:
  double x, y;
public:
  void draw();
};
```

```
class NewPoint : public Point {
public:
  void draw();
};
```

```
NewPoint np;
Point &p = np;

np.draw();
p.draw();
```

Erzeugt eine Referenz auf np

NewPoint::draw()
Point::draw()

Vererbung: virtuelle Methoden

```
class Point {
private:
  double x, y;
public:
  virtual void draw();
};
```

ermöglicht Polymorphie

```
class NewPoint
  : public Point {
public:
  void draw();
};
```

```
NewPoint np;
Point &p = np;

np.draw();
p.draw();
```

NewPoint::draw()
NewPoint::draw()

Rein virtuelle (= pure virtual) Methoden

```
class Point {
private:
  double x, y;
public:
  virtual void draw() = 0;
};
```

Muss von abgeleiteten Klassen implementiert werden.

```
Point p;          /* FEHLER */

NewPoint np;      /* OK */
Point &pp = np;   /* OK */

pp.draw();        /* NewPoint::draw() */
```

Das Schlüsselwort const

Vorkurs C/C++, Olaf Bergmann

Das Schlüsselwort const

- getypte Konstanten definieren:

```
const double pi = 3.14159265359;
const int off = 0, running = 1;
```

- Konstante Referenzen → Daten sind auch konstant

```
int i = 12;
int &ir = i;
const int &cir = i;
cir = 19;                  // Fehler
ir = 19;                   // auch cir jetzt 19

const int ci = 4711;
int &ir3 = ci;             // Fehler: ci konstant
```

Das Schlüsselwort const

- Funktionsargumente:
  ```
  int f(const Point &point);
  ```
- Rückgabewerte:
  ```
  const Point &Point::f(void) { return *this; }
  ```
- konstante Memberfunktionen:
  ```
  void Point::f(void) const;
  ```
- und natürlich beliebige Kombinationen daraus...

```
for (auto const &x : container) {
  /* x darf nicht lvalue sein */
}
```

Funktionen überladen

Vorkurs C/C++, Olaf Bergmann

Funktionen überladen

```
int power(int, int);
double power(double, double);
```

- Signatur entscheidet, welche Funktion gewählt wird
  - Name + Parameter
  - Rückgabewert gehört nicht zur Signatur

```
power(2, 10)          /* erste Form */

power(2.0, 5.0);      /* zweite Form */

power(2.0, 5);        /* Fehler: mehrdeutig */
```

Operatoren überladen

- Existierende Operatoren „erweitern“:

```
struct complex {
  double re, im;
};

complex operator+(complex a, complex b) {
  complex c;
  c.re = a.re + b.re;
  c.im = a.im + b.im;
  return c;
}

/* ... */

complex nw = {-1, 1};
complex sw = {4, -4};
complex sum = nw + sw;
```

Objekte ausgeben

- `printf()`: beliebt für skalare Datentypen in C, aber nicht geeignet für komplexe C++-Objekte
  - Keine nutzerdefinierten Ausgabeformate
  - Keine Unterstützung von Polymorphie
- C++: Stream-basierte Ausgabe (und Eingabe)

```
#include <iostream>

cout << obj << endl;
```

„portables Newline“

`cout` ist globale Variable vom Typ `ostream`

```
ostream& operator<< (ostream& os,
                     const string& str);
```

I/O-Manipulatoren

```
#include <string>
#include <iostream>
#include <iomanip>

using namespace std;

cout << setw(14) << fixed << setprecision(4)
     << number << endl;

cout << setbase(16) << hexvalue << endl;

string s;
cout << quoted(s) << endl;
```

Eigene Objekte ausgeben

```
class Something {
  int thing;
public:
  Something(int t = 0) : thing(t) {}

  friend ostream &operator<<(ostream &os,
                             const Something &s);
};

ostream &operator<<(ostream &os, const Something &s) {
  return os << s.thing;
}



cout << Something(35) << endl;
```

Default-Argumente

- Überladen nicht immer elegant

```
void clear_screen(Screen s, char fill);
void clear_screen(Screen s) { clear_screen(s, ' '); }
```

- Default-Wert in Deklaration angeben

```
void clear_screen(Screen s, char fill = ' ');
```

  - zulässig für ein Vorkommen in der Datei
  - nur am Ende erlaubt
    „triviale“ Parameter ans Ende stellen
  - Compiler fügt Wert automatisch hinzu

- Default-Argumente lassen sich kumulieren (innerhalb einer Datei)

```
Screen my_screen;

void clear_screen(Screen s = my_screen, char fill);

clear_screen();
```