Übungen zu
Parallele Programmierung mit C++
Einführung zu C++
SS 2019

Andreas F. Borchert
Universität Ulm

17. Mai 2019

- Bjarne Stroustrup startete sein Projekt *C with Classes* im April 1979 bei den Bell Laboratories nach seinen Erfahrungen mit Simula und BCPL.
- Sein Ziel war es, die Klassen von Simula als Erweiterung zur Programmiersprache C einzuführen, ohne Laufzeiteffizienz zu opfern. Der Übersetzer wurde als Präprozessor zu C implementiert, der *C with Classes* in reguläres C übertrug.
- 1982 begann ein Neuentwurf der Sprache, die dann den Namen C++ erhielt. Im Rahmen des Neuentwurfs kamen virtuelle Funktionen (und damit Polymorphismus), die Überladung von Operatoren, Referenzen, Konstanten und verbesserte Typüberprüfungen hinzu.

- 1985 begann Bell Laboratories mit der Auslieferung von *Cfront*, der C++ in C übersetzte und damit eine Vielzahl von Plattformen unterstützte.
- 1990 wurde für C++ bei ANSI/ISO ein Standardisierungskomitee gegründet.
- Vorschläge für Templates in C++ gab es bereits in den 80er-Jahren und eine erste Implementierung stand 1989 zur Verfügung. Sie wurde 1990 vom Standardisierungskomitee übernommen.
- Analog wurden Ausnahmenbehandlungen 1990 vom Standardisierungskomitee akzeptiert. Erste Implementierungen hierfür gab es ab 1992. Namensräume wurden erst 1993 in C++ eingeführt.
- Im September 1998 wurde mit ISO 14882 der erste Standard für C++ veröffentlicht.

- Mit C++11 erfolgte eine sehr umfangreiche Revision, die u.a. folgende Features einführte:
  - Rvalue-Referenzen zusammen mit *move constructors*,
  - **constexpr** zur Berechnung von Ausdrücken und Funktionen zur Übersetzzeit,
  - automatische Ableitung eines Typen (Typinferenz),
  - beliebig viele Parameter für Templates,
  - Lambda-Ausdrücke,
  - Threads und
  - *smart pointers*.
- Diese Erweiterungen wurden in C++14 und C++17 weiter ausgebaut und abgerundet. Der aktuelle Standard wurde im Dezember 2017 veröffentlicht und wird kurz C++17 genannt.

- In C++ sind Übersetzungseinheiten Dateien mit Programmtext, die dem C++-Übersetzer unmittelbar auf der Kommandozeile zum Übersetzen angegeben werden. Als Dateiendung wird hier gerne „.cpp“ benutzt – es sind aber auch viele andere üblich wie etwa „.C“ oder „.cc“.
- Mit Hilfe der **#include**-Direktive des Präprozessors können noch sehr viel mehr Programmtexte indirekt hinzukommen.
- Eine Übersetzungseinheit wird normalerweise direkt in Maschinencode für eine ausgewählte Plattform übersetzt (normalerweise die lokale, ein *cross compiler* kann auch für andere Plattformen übersetzen). Diese Resultate werden auch Objekte genannt (hat nichts mit den OO-Konzepten zu tun).
- Mit Hilfe des *ld* (*linkage editor*) können mehrere Objekte zusammen mit den Bibliotheken zu einem ausführbaren Programm zusammengefügt werden.

⟨translation-unit⟩ ⟶ [⟨declaration-seq⟩]

⟨declaration-seq⟩ ⟶ ⟨declaration⟩

⟶ ⟨declaration-seq⟩ ⟨declaration⟩

⟨declaration⟩ ⟶ ⟨block-declaration⟩

⟶ ⟨nodeclspec-function-declaration⟩

⟶ ⟨function-definition⟩

⟶ ⟨template-declaration⟩

⟶ ⟨deduction-guide⟩

⟶ ⟨explicit-instantiation⟩

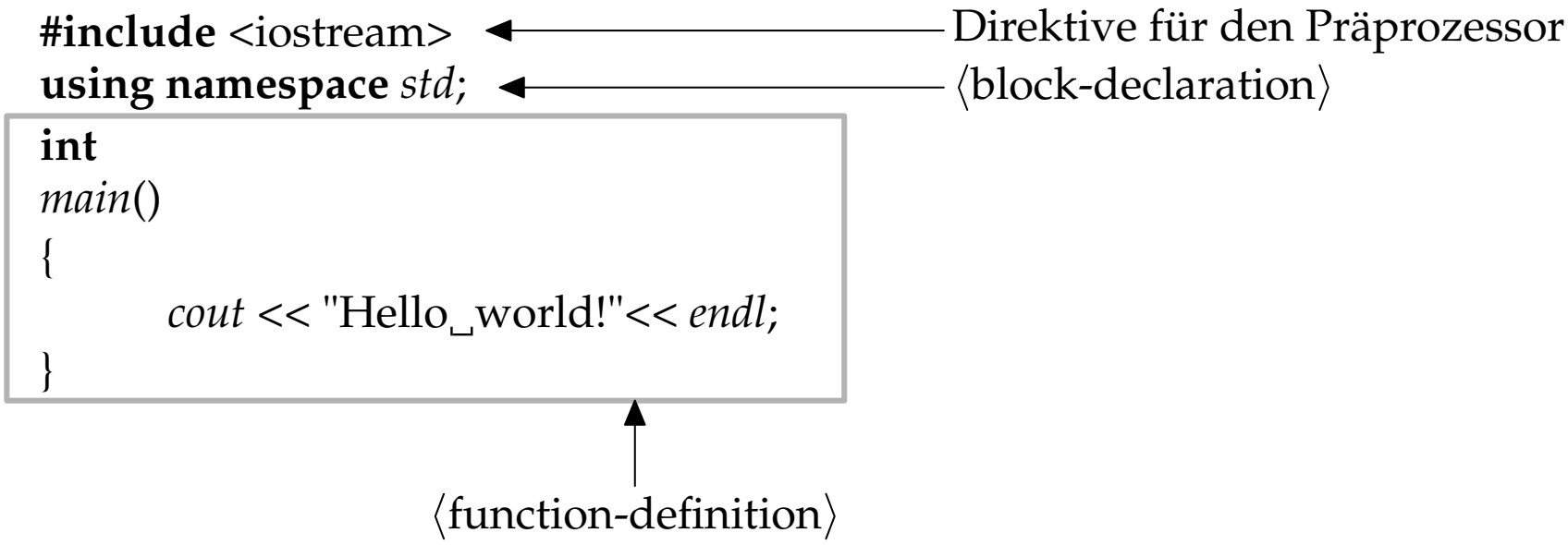
⟶ ⟨explicit-specialization⟩

⟶ ⟨linkage-specification⟩

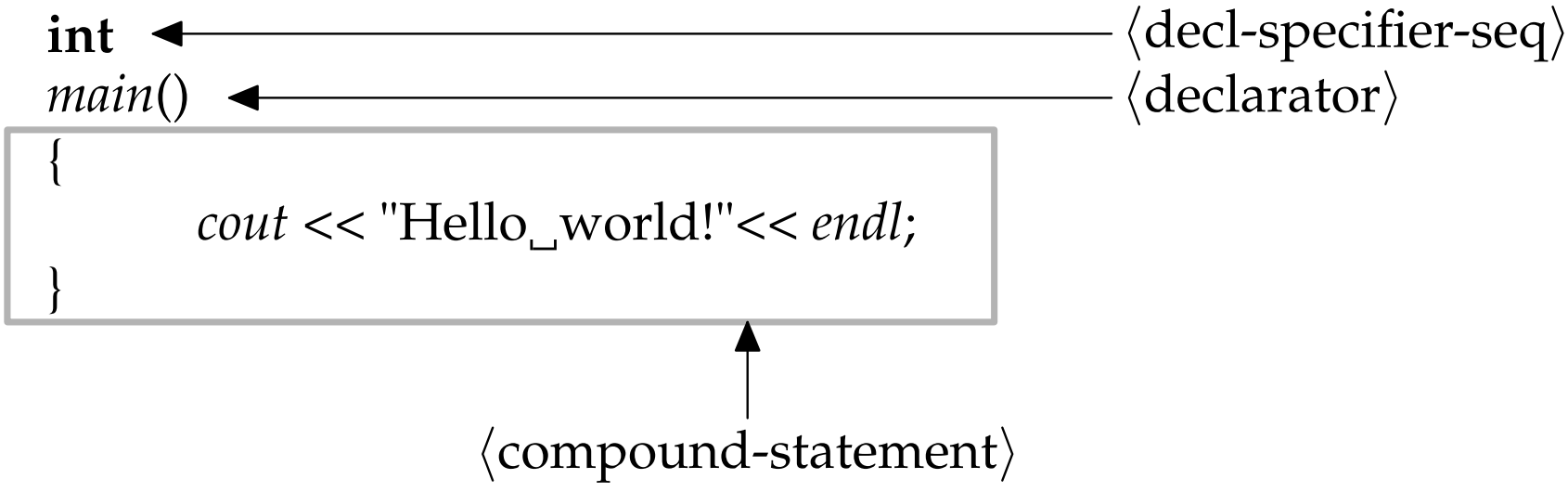
⟶ ⟨namespace-definition⟩

⟶ ⟨empty-declaration⟩

⟶ ⟨attribute-declaration⟩



- Präprozessor-Anweisungen betten sich nicht in die C++-Syntax ein. Durch den Präprozessor werden sie durch Text ersetzt, der der C++-Syntax entsprechend sollte. Bei **#include**-Direktiven ist dies der Inhalt der gegebenen Datei (hier *iostream*, die standardmäßig zur Verfügung steht).
- Mit **using namespace** *std* lässt sich alles aus dem *std*-Namensraum ohne Qualifikation verwenden, also etwa *cout* anstelle von *std::cout*.



⟨function-definition⟩ ⟶ [⟨attribute-specifier-seq⟩]
[⟨decl-specifier-seq⟩]
⟨declarator⟩ [⟨virt-specifier-seq⟩]
⟨function-body⟩

⟨function-body⟩ ⟶ [⟨ctor-initializer⟩]
⟨compound-statement⟩
⟶ ⟨function-try-block⟩
⟶ „=“ **default** „;“
⟶ „=“ **delete** „;“

- Normalerweise ist nur die erste Variante interessant.
- Der ⟨function-try-block⟩ erlaubt eine saubere Lösung der Ausnahmenbehandlung bei Konstruktoren mit Sub-Konstruktoren, die möglicherweise Ausnahmenbehandlungen auslösen.
- Die letzteren beiden Varianten betreffen nur einige standardmäßig unterstützte Methoden – wir kommen darauf noch zurück.
- *ctor* steht für *constructor* – entsprechend kann ein ⟨ctor-initializer⟩ nur bei Konstruktoren vorkommen.

$$
\begin{aligned}
\langle \text{block-declaration} \rangle \quad &\longrightarrow \quad \langle \text{simple-declaration} \rangle \\
&\longrightarrow \quad \langle \text{asm-definition} \rangle \\
&\longrightarrow \quad \langle \text{namespace-alias-definition} \rangle \\
&\longrightarrow \quad \langle \text{using-declaration} \rangle \\
&\longrightarrow \quad \langle \text{using-directive} \rangle \\
&\longrightarrow \quad \langle \text{static\_assert-declaration} \rangle \\
&\longrightarrow \quad \langle \text{alias-declaration} \rangle \\
&\longrightarrow \quad \langle \text{opaque-enum-declaration} \rangle
\end{aligned}
$$

- Blockdeklarationen können in C++ sowohl auf globaler Ebene als auch innerhalb eines Blocks (d.h. inmitten regulären Programmtexts) erfolgen.

⟨simple-declaration⟩ ⟶ ⟨decl-specifier-seq⟩
[⟨init-declarator-list⟩] „;“
⟶ ⟨attribute-specifier-seq⟩
⟨decl-specifier-seq⟩
⟨init-declarator-list⟩ „;“
⟶ [⟨attribute-specifier-seq⟩] ⟨decl-specifier-seq⟩
[⟨ref-qualifier⟩]
„[“ ⟨identifier-list⟩ „]“ ⟨initializer⟩ „;“

- Die Mehrzahl der Deklarationen in C++ fällt unter die Rubrik der ⟨simple-declaration⟩. Dazu gehören u.a. Variablen- und Klassendeklarationen.
- Die wichtigsten Teile einer Deklaration sind die ⟨decl-specifier⟩, die den Grundtyp spezifizieren und der ⟨declarator⟩, der einen Namen mit einer möglicherweise abgeleiteten Variante des Grundtyps assoziiert.

⟨decl-specifier-seq⟩ ⟶ ⟨decl-specifier⟩
[⟨attribute-specifier-seq⟩]
⟶ ⟨decl-specifier⟩
⟨decl-specifier-seq⟩
⟨decl-specifier⟩ ⟶ ⟨storage-class-specifier⟩
⟶ ⟨defining-type-specifier⟩
⟶ ⟨function-specifier⟩
⟶ **friend**
⟶ **typedef**
⟶ **constexpr**
⟶ **inline**

$$
\begin{aligned}
\langle \text{defining-type-specifier} \rangle &\longrightarrow \langle \text{type-specifier} \rangle \\
&\longrightarrow \langle \text{class-specifier} \rangle \\
&\longrightarrow \langle \text{enum-specifier} \rangle \\
\langle \text{type-specifier} \rangle &\longrightarrow \langle \text{simple-type-specifier} \rangle \\
&\longrightarrow \langle \text{elaborated-type-specifier} \rangle \\
&\longrightarrow \langle \text{typename-specifier} \rangle \\
&\longrightarrow \langle \text{cv-specifier} \rangle
\end{aligned}
$$

⟨simple-type-specifier⟩ ⟶ [⟨nested-name-specifier⟩] ⟨type-name⟩
⟶ ⟨nested-name-specifier⟩ **template** ⟨simple-template-id⟩
⟶ [⟨nested-name-specifier⟩] ⟨template-name⟩
⟶ **char** | **char16_t** | **char32_t** | **wchar_t**
⟶ **bool** | **short** | **int** | **long**
⟶ **signed** | **unsigned**
⟶ **float** | **double**
⟶ **void** | **auto**
⟶ ⟨decltype-specifier⟩

- ⟨simple-type-specifier⟩ schließt alle elementaren Datentypen ein. **auto** zwingt den Übersetzer, den Datentyp selbst zu deduzieren.

$$
\begin{array}{rcl}
\langle \text{type-name} \rangle & \longrightarrow & \langle \text{class-name} \rangle \\
& \longrightarrow & \langle \text{enum-name} \rangle \\
& \longrightarrow & \langle \text{typedef-name} \rangle \\
& \longrightarrow & \langle \text{simple-template-id} \rangle \\
\langle \text{decltype-specifier} \rangle & \longrightarrow & \textbf{decltype} \text{ „(“ } \langle \text{expression} \rangle \text{ „)“} \\
& \longrightarrow & \textbf{decltype} \text{ „(“ } \textbf{auto} \text{ „)“}
\end{array}
$$

- Mit **decltype** kann ein Datentyp von einem Ausdruck abgeleitet werden.

⟨class-name⟩ ⟶ ⟨identifier⟩
⟶ ⟨simple-template-id⟩
⟨class-specifier⟩ ⟶ ⟨class-head⟩
„{“ [⟨member-specification⟩] „}“
⟨class-head⟩ ⟶ ⟨class-key⟩ [⟨attribute-specifier-seq⟩]
⟨class-head-name⟩ [⟨class-virt-specifier⟩]
[⟨base-clause⟩]
⟶ ⟨class-key⟩ [⟨attribute-specifier-seq⟩]
[⟨base-clause⟩]
⟨class-key⟩ ⟶ **class**
⟶ **struct**
⟶ **union**

- Bei **class** sind alle Felder und Methoden per Voreinstellung **private**, bei **struct** sind sie (in Kompatibilität zu C) per Voreinstellung **public**. Bei einer **union** werden sämtliche Datenfelder übereinander gelegt.

Deklarationen innerhalb einer Klasse

⟨member-specification⟩ ⟶ ⟨member-declaration⟩
[⟨member-specification⟩]
⟶ ⟨access-specifier⟩ „:“
[⟨member-specification⟩]

⟨member-declaration⟩ ⟶ [⟨attribute-specifier-seq⟩]
[⟨decl-specifier-seq⟩]
[⟨member-declarator-list⟩] „;“
⟶ ⟨function-definition⟩
⟶ ⟨using-declaration⟩
⟶ ⟨static_assert-declaration⟩
⟶ ⟨template-declaration⟩
⟶ ⟨deduction-guide⟩
⟶ ⟨alias-declaration⟩
⟶ ⟨empty-declaration⟩

Greeting.hpp

```
#ifndef GREETING_H
#define GREETING_H

class Greeting {
   public:
      void hello();
}; // class Greeting

#endif
```

- Klassendeklarationen (mitsamt allen öffentlichen und auch privaten Datenfeldern und Methoden) sind in Dateien, die mit „.hpp“, „.hh“ oder „.h“ enden, unterzubringen. Hierbei steht „.h“ allgemein für eine Header-Datei bzw. „.hh“ oder „.hpp“ für eine Header-Datei von C++.
- Alle Zeilen, die mit einem # beginnen, enthalten Direktiven für den Makro-Präprozessor. Dieses Relikt aus Assembler- und C-Zeiten ist in C++ erhalten geblieben. Die Konstruktion in diesem Beispiel stellt sicher, dass die Klassendeklaration nicht versehentlich mehrfach in den zu übersetzenden Text eingefügt wird.

Greeting.hpp

```
class Greeting {
   public:
      void hello();
}; // class Greeting
```

- Eine Klassendeklaration besteht aus einem Namen und einem Paar geschweifter Klammern, die eine Sequenz von Deklarationen eingrenzen. Die Klassendeklaration wird (wie sonst alle anderen Deklarationen in C++ auch) mit einem Semikolon abgeschlossen.
- Kommentare starten mit // und erstrecken sich bis zum Zeilenende.

Greeting.hpp

```
class Greeting {
   public:
      void hello();
}; // class Greeting
```

- Die Deklarationen der einzelnen Komponenten einer Klasse, in der C++-Terminologie *member* genannt, fallen in verschiedene Kategorien, die die Zugriffsrechte regeln:

private	nur für die Klasse selbst und ihre Freunde zugänglich
protected	offen für alle davon abgeleiteten Klassen
public	uneingeschränkter Zugang

  Wenn keine der drei Kategorien explizit angegeben wird, dann wird automatisch **private** angenommen.

Greeting.hpp

```
class Greeting {
   public:
      void hello();
}; // class Greeting
```

- Alle Funktionen (einschließlich der Methoden einer Klasse) haben einen Typ für ihre Rückgabewerte. Wenn nichts zurückzuliefern ist, dann kann **void** als Typ verwendet werden.
- In Deklarationen folgt jeweils dem Typ eine Liste von durch Kommata getrennten Namen, die mit zusätzlichen Spezifikationen wie etwa () ergänzt werden können.
- Die Angabe () sorgt hier dafür, dass aus *hello* eine Funktion wird, die Werte des Typs **void** zurückliefert, d.h. ohne Rückgabewerte auskommt.

Greeting.cpp

```
#include <iostream>
#include "Greeting.hpp"

void Greeting::hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
} // hello()
```

- C++-Programme bzw. die Implementierung einer öffentlichen Schnittstelle in einer Header-Datei werden üblicherweise in separaten Dateien untergebracht.
- Als Dateiendung sind „.cpp“, „.cc“ oder „.C“ üblich.
- Letztere Variante ist recht kurz, hat jedoch den Nachteil, dass sie sich auf Dateisystemen ohne Unterscheidung von Klein- und Großbuchstaben nicht von der Endung „.c“ unterscheiden lässt, die für C vorgesehen ist. (Vorsicht ist hier u.a. bei dem *HFS+*-Dateisystem von Apple geboten.)
- Der Übersetzer erhält als Argumente nur diese Dateien, nicht die Header-Dateien.

Greeting.cpp

```
#include <iostream>
#include "Greeting.hpp"

void Greeting::hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
} // hello()
```

- Die Direktive **#include** bittet den Präprozessor um das Einfügen des genannten Textes an diese Stelle in den Eingabetext für den Übersetzer.
- Anzugeben ist ein Dateiname. Wenn dieser in <...> eingeschlossen wird, dann erfolgt die Suche danach nur an Standardplätzen, wozu das aktuelle Verzeichnis normalerweise nicht zählt.
- Wird hingegen der Dateiname in "..." gesetzt, dann beginnt die Suche im aktuellen Verzeichnis, bevor die Standardverzeichnisse hierfür in Betracht gezogen werden.

Greeting.cpp

```
#include <iostream>
#include "Greeting.hpp"

void Greeting::hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
} // hello()
```

- Der eigentliche Übersetzer von C++ liest nicht direkt von der Quelle, sondern den Text, den der Präprozessor zuvor generiert hat.
- Andere Texte, die nicht direkt oder indirekt mit Hilfe des Präprozessors eingebunden werden, stehen dem Übersetzer nicht zur Verfügung.
- Entsprechend ist es strikt notwendig, alle notwendigen Deklarationen externer Klassen in Header-Dateien unterzubringen, die dann sowohl bei den Klienten als auch dem implementierenden Programmtext selbst einzubinden sind.

Greeting.cpp

```
void Greeting::hello() {
   std::cout << "Hello, world!" << std::endl;
} // hello()
```

- Methoden werden üblicherweise außerhalb ihrer Klassendeklaration definiert. Zur Verknüpfung der Methode mit der Klasse wird eine Qualifizierung notwendig, bei der der Klassenname und das Symbol :: dem Methodennamen vorangehen. Dies ist notwendig, da prinzipiell mehrere Klassen in eine Übersetzungseinheit integriert werden können.
- Eine Funktionsdefinition besteht aus der Signatur und einem Block. Ein terminierendes Semikolon wird hier nicht verwendet.
- Blöcke schließen eine Sequenz lokaler Deklarationen, Anweisungen und weiterer verschachtelter Blöcke ein.
- Funktionen dürfen nicht ineinander verschachtelt werden.

Greeting.cpp

```
void Greeting::hello() {
   std::cout << "Hello, world!" << std::endl;
} // hello()
```

- Die Präprozessor-Direktive **#include** <iostream> fügte Deklarationen in den zu übersetzenden Text ein, die u.a. auch *cout* innerhalb des Namensraumes *std* deklariert hat. Die Variable *std*::*cout* repräsentiert die Standardausgabe und steht global zur Verfügung.
- Da C++ das Überladen von Operatoren unterstützt, ist es möglich, Operatoren wie etwa << (binäres Verschieben) für bestimmte Typkombinationen zu definieren. Hier wurde die Variante ausgewählt, die als linken Operanden einen *ostream* und als rechten Operanden eine Zeichenkette erwartet.
- *endl* repräsentiert den Zeilentrenner.
- *cout* << "Hello,␣world!" gibt die Zeichenkette auf *cout* aus, liefert den Ausgabekanal *cout* wieder zurück, wofür der Operator << erneut aufgerufen wird mit der Zeichenkette, die von *endl* repräsentiert wird, so dass der Zeilentrenner ebenfalls ausgegeben wird.

SayHello.cpp

```
#include "Greeting.hpp"

int main() {
   Greeting greeting;
   greeting.hello();
   greeting.hello();
   return 0;
} // main()
```

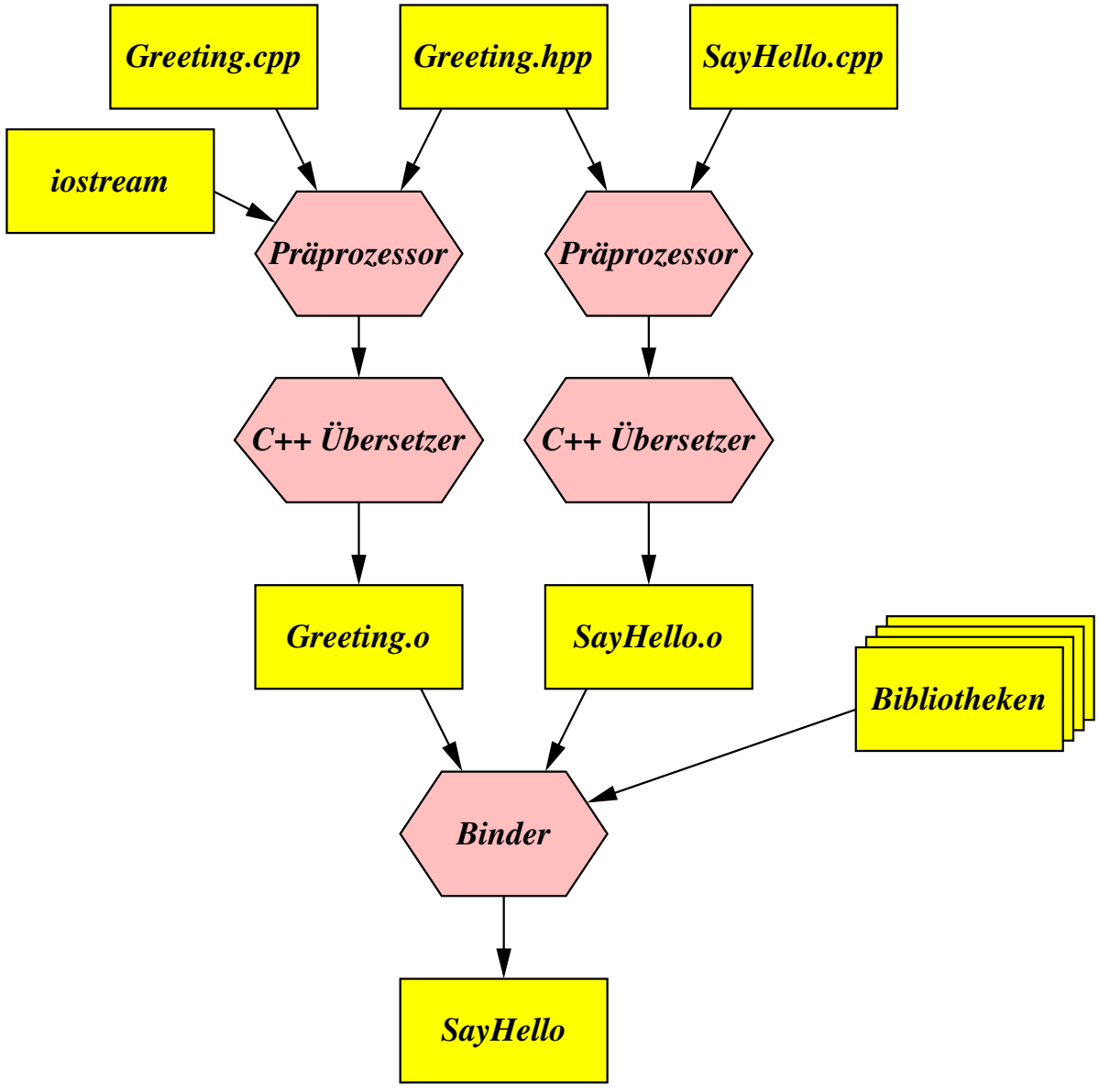
- Dank dem Erbe von C müssen nicht alle Funktionen einer Klasse zugeordnet werden.
- In der Tat darf die Funktion *main*, bei der die Ausführung (nach der Konstruktion globaler Variablen) startet und die Bestandteil eines jeden Programmes sein muss, nicht innerhalb einer Klasse definiert werden.
- Sobald *main* beendet ist, wird das Ende der gesamten Programmausführung eingeleitet.
- Der ganzzahlige Wert, den *main* zurückgibt, wird der Ausführungsumgebung zurückgegeben. Entsprechend den UNIX-Traditionen steht hier 0 für Erfolg und andere Werte deuten ein Problem an.

SayHello.cpp

```
int main() {
   Greeting greeting;
   greeting.hello();
   return 0;
} // main()
```

- Mit *Greeting greeting* wird eine lokale Variable mit dem Namen *greeting* und dem Datentyp *Greeting* definiert. Das entsprechende Objekt wird hier automatisch instantiiert, sobald *main* startet.
- Durch *greeting.hello*() wird die Methode *hello* für das Objekt *greeting* aufgerufen. Die Klammern sind auch dann notwendig, wenn keine Parameter vorkommen.

Bau eines Programms



- Die gängigen Implementierungen für C++ stellen nur eine schwache Form der Schnittstellensicherheit her.
- Diese wird typischerweise erreicht durch das Generieren von Namen, bei denen teilweise die Typinformation mit integriert ist, so dass Objekte gleichen Namens, jedoch mit unterschiedlichen Typen nicht so ohne weiteres zusammengebaut werden.

```
theon$ ls
theon$ curl -s \
> http://www.mathematik.uni-ulm.de/numerik/pp/ss19/Makefile >Makefile
theon$ make depend
gcc-makedepend  SayHello.cpp Greeting.cpp
theon$ make
g++ -Wall -g -std=gnu++17   -c -o SayHello.o SayHello.cpp
g++ -Wall -g -std=gnu++17   -c -o Greeting.o Greeting.cpp
g++ -o SayHello  SayHello.o Greeting.o
theon$ ./SayHello
Hello, fans of C++!
Hello, fans of C++!
theon$ make realclean
rm -f SayHello.o Greeting.o
rm -f SayHello
theon$ ls
Greeting.cpp  Greeting.hpp  Makefile      SayHello.cpp
theon$
```

- *make* ist ein Werkzeug, das eine Datei namens *Makefile* (oder *makefile*) im aktuellen Verzeichnis erwartet, in der Methoden zur Generierung bzw. Regenerierung von Dateien beschrieben werden und die zugehörigen Abhängigkeiten.
- *make* ist dann in der Lage festzustellen, welche Zieldateien fehlen bzw. nicht mehr aktuell sind, um diese dann mit den spezifizierten Kommandos neu zu erzeugen.
- *make* wurde von Stuart Feldman 1979 für das Betriebssystem UNIX entwickelt. 2003 wurde er hierfür von der ACM mit dem Software System Award ausgezeichnet.

```
theon$ curl -s \
> http://www.mathematik.uni-ulm.de/numerik/pp/ss19/Makefile >Makefile
```

- Unter der genannten URL steht eine Vorlage für ein für C++ geeignetes *Makefile* zur Verfügung.
- Das Kommando *curl* kopiert Inhalte von einer gegebenen URL zur Standardausgabe. Alternativ kann auch *wget* verwendet werden.

```
theon$ make depend
```

- Das heruntergeladene *Makefile* geht davon aus, dass Sie den g++ verwenden (GNU C++ Compiler) und die regulären C++-Quellen in „.cpp“ enden und die Header-Dateien in „.hpp“.
- Mit dem Aufruf von „make depend“ werden die Abhängigkeiten neu bestimmt und im *Makefile* eingetragen. Dies muss zu Beginn mindestens einmal aufgerufen werden.
- Wenn Sie dies nicht auf unseren Rechnern probieren, sollten Sie das hier implizit verwendete Skript von Github beziehen. Es ist in Perl geschrieben und sollte mit jeder üblichen Perl-Installation zurechtkommen:
  `https://github.com/afborchert/gcc-makedepend`

Greeting.hpp

```
#ifndef GREETING_H
#define GREETING_H

#include <iostream>

class Greeting {
   public:
      void hello() {
         std::cout << "Hello, fans of C++!" << std::endl;
      }
}; // class Greeting

#endif
```

- Es ist auch möglich, die Methoden innerhalb des Headers direkt zu implementieren.
- Das verlangsamt die Übersetzungszeiten und es ist nicht sichergestellt, dass der Code für die Methodenimplementierungen nur einmal existiert, wenn diese mehrfach per **#include** einkopiert und somit übersetzt werden.
- Diese Vorgehensweise eröffnet dem Übersetzer jedoch ein Optimierungspotential, das er ohne Kenntnis des Programmtexts nicht

Greeting.hpp

```
inline void hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
}
```

- Es ist auch möglich, dem Übersetzer nahezulegen, auf den Methodenaufruf zu verzichten und stattdessen diesen mit der Implementierung der Methode zu ersetzen.
- Ob dies sinnvoll ist, hängt u.a. auch davon ab, wie umfangreich die Methode ist.
- Das ist nicht möglich mit Methoden, deren zugehörige Implementierung zur Laufzeit gesucht wird (dynamischer Polymorphismus).

SayHello.cpp

```
#include "Greeting.hpp"

Greeting greeting1;

int main() {
   greeting1.hello();

   Greeting greeting2;
   greeting2.hello();

   Greeting* greeting3 = new Greeting();
   greeting3->hello();
   delete greeting3;
} // main()
```

- Global erzeugte Objekte wie *greeting1* werden vor dem Aufruf von *main* erzeugt und erst nach dem Verlassen von *main* abgebaut.
- Lokale Variablen wie *greeting2* werden jedesmal erzeugt, wenn der umgebende Block erzeugt wird und beim Verlassen des Blocks automatisch abgebaut.
- Mit **new** kann ein Objekt dynamisch auf dem Heap erzeugt werden. Dieses existiert, bis es explizit mit **delete** wieder abgebaut wird.

Klassen in C++ verhalten sich völlig anders als solche in Java und vielen anderen objekt-orientierten Programmiersprachen:

- Wir haben keine implizite Zeigersemantik.
- Objekte einer Klasse können (wenn nichts anderes bestimmt ist) kopiert und als Wert einer Funktion oder Methode übergeben werden (*call by value*).
- Objekte können nicht nur auf dem Heap leben.
- Objekte werden immer in wohldefinierter Weise abgebaut.

Eine triviale Klasse

vector2d.cpp

```
#include <iostream>

class Vector2D {
   public:
      double x, y;
};

void print_point(Vector2D v) {
   std::cout << "(" << v.x << ", " << v.y << ")";
}

Vector2D add_vectors(Vector2D v1, Vector2D v2) {
   Vector2D result;
   result.x = v1.x + v2.y; result.y = v1.y + v2.y;
   return result;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2; Vector2D b; b.x = 10; b.y = 20;
   Vector2D c = add_vectors(a, b);
   print_point(c); std::cout << std::endl;
}
```

Parameterübergabe mit *call by value*

vector2d.cpp

```
Vector2D add_vectors(Vector2D v1, Vector2D v2) {
   Vector2D result;
   result.x = v1.x + v2.y; result.y = v1.y + v2.y;
   return result;
}
```

- Wenn ein Objekt per Parameter übergeben wird, dann wird per Voreinstellung jede einzelne Variablenkomponente (hier x und y) kopiert. Die Parameter *v1* und *v2* leben dann lokal auf dem Stack.
- Objekte können auch zurückgegeben werden. In diesem Fall wird *result* komponentenweise bei der Rückgabe in c kopiert.

vector2d-scale.cpp

```
void scale_vector(Vector2D v, double factor) {
   v.x *= factor; v.y *= factor;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2;
   scale_vector(a, 10);
   print_point(a); std::cout << std::endl;
}
```

- Diese Variante von *scale_vector* ist sinnlos, da nur die lokale Kopie *v* verändert wird, jedoch nicht der Vektor *a* in *main*.

Es gibt hierzu zwei Möglichkeiten:

- Unter expliziter Verwendung von Zeigern (so wie es auch in C üblich war).
- Unter Verwendung von Referenzen.

Dann lässt sich das auch unnötiges Kopieren vermeiden, das bei größeren Objekten (man denke an sehr große Matrizen) recht teuer werden kann.

Parameterübergabe mit einem Zeiger

vector2d-scale2.cpp

```
void scale_vector(Vector2D* vp, double factor) {
   vp->x *= factor; vp->y *= factor;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2;
   scale_vector(&a, 10);
   print_point(a); std::cout << std::endl;
}
```

- Der Adress-Operator „&“ liefert die Adresse eines Objekts – hier mit dem Datentyp *Vector2D∗*.
- Die Funktion *scale_vector* erwartet hier einen Zeiger und ist entsprechend gezwungen, diesen immer zu dereferenzieren.

Parameterübergabe mit einer Referenz

vector2d-scale3.cpp

```
void scale_vector(Vector2D& vp, double factor) {
   vp.x *= factor; vp.y *= factor;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2;
   scale_vector(a, 10);
   print_point(a); std::cout << std::endl;
}
```

- Referenzen sind implizite Zeiger.
- Entsprechend fällt das explizite Dereferenzieren und die Verwendung des Adress-Operators bei der Übergabe weg.
- Die Umsetzung ist äquivalent zur vorherigen Variante mit expliziten Zeigern.

vector2d-scale3.cpp

```
void print_point(const Vector2D& v) {
   std::cout << "(" << v.x << ", " << v.y << ")";
}
```

- Die Verwendung von Referenzen ist auch dann sinnvoll, wenn das Objekt nicht zu verändern ist, da dann der Kopieraufwand entfällt.
- In diesem Fall ist es sinnvoll, **const** hinzuzufügen. Das sichert zu, dass die so per Referenz übergebene Variable nicht verändert wird.

Eine einfache Klasse

counter.hpp

```
class Counter {
   public:
      // constructors
      Counter() : counter{0} {
      }
      Counter(int counter) : counter{counter} {
      }
      // accessors
      int get() const {
         return counter;
      }
      // mutators
      int increment() {
         assert(counter < INT_MAX);
         return ++counter;
      }
      int decrement() {
         assert(counter > INT_MIN);
         return --counter;
      }
   private:
      int counter;
};
```

counter.hpp

```
private:
   int counter;
```

- Datenfelder sollten normalerweise privat gehalten werden, um den direkten Zugriff darauf zu verhindern. Stattdessen ist es üblich, entsprechende Zugriffsmethoden (*accessors*, *mutators*) zu definieren.

counter.hpp

```
  Counter() : counter{0} {
  }
  Counter(int counter) : counter{counter} {
  }
```

- Eine Klasse kann beliebig viele Konstruktoren anbieten, solange sie sich in ihrer Signatur unterscheiden.
- Wenn kein Konstruktor angegeben ist, generiert der Übersetzer automatisch einen parameterlosen Konstruktor, der, sofern möglich, sämtliche Datenfelder analog ohne Parameter konstruiert.
- Es ist zulässig, die Parameter der Konstruktoren genauso zu nennen wie die entsprechenden Objektvariablen.

⟨ctor-initializer⟩ ⟶ „:“ ⟨mem-initializer-list⟩

⟨mem-initializer-list⟩ ⟶ ⟨mem-initializer⟩ [„...“]

⟶ ⟨mem-initializer-list⟩ „,“ ⟨mem-initializer⟩ [„...“]

⟨mem-initializer⟩ ⟶ ⟨mem-initializer-id⟩ „(“ [⟨expression-list⟩] „)“

⟶ ⟨mem-initializer-id⟩ ⟨braced-init-list⟩

⟨mem-initializer-id⟩ ⟶ ⟨class-or-decltype⟩

⟶ ⟨identifier⟩

⟨braced-init-list⟩ ⟶ „{“ ⟨initializer-list⟩ [„,“] „}“

⟶ „{“ „}“

- Seit C++11 wird die {...}-Notation (⟨braced-init-list⟩) bevorzugt, da sie einen unschönen grammatikalischen Konflikt vermeidet und mehr Möglichkeiten erlaubt.

- Es ist in C++ sehr wichtig, zwischen einer Initialisierung mit Hilfe eines ⟨ctor-initializer⟩ und einer regulären Zuweisung innerhalb des ⟨compound-statement⟩ des Konstruktors zu unterscheiden.
- Bevor das ⟨compound-statement⟩ des Konstruktors ausgeführt wird, müssen alle Teilobjekte konstruiert sein. Wenn die Initialisierung innerhalb des ⟨ctor-initializer⟩ fehlt, dann wird jeweils der parameterlose Konstruktor verwendet.
- Wenn der parameterlose Konstruktor für eines der Teilobjekte nicht zur Verfügung stehen sollte, dann geht es überhaupt nicht ohne die Konstruktion innerhalb des ⟨ctor-initializer⟩.
- Eine implizite automatische Konstruktion in Kombination mit einer späteren Zuweisung kann zu ineffizienteren Code führen. Daher wird grundsätzlich empfohlen, soweit wie möglich alle Teilobjekte innerhalb des ⟨ctor-initializer⟩ zu initialisieren.
- Die Reihenfolge im ⟨ctor-initializer⟩ sollte der der Deklaration der Teilobjekte entsprechen. In letzterer Reihenfolge werden sie ausgeführt.

counter.hpp

```
  Counter() : counter{0} {
  }
  Counter(int counter) : counter{counter} {
  }
```

- Es ist zulässig, die Parameter der Konstruktoren genauso zu nennen wie die entsprechenden Objektvariablen.
- Bei der ⟨mem-initializer-id⟩ wird nur unter den zu initialisierenden Namen der Objektvariablen gesucht bzw. dem Namen der eigenen Klasse oder den Namen der Basisklassen.
- In der ⟨expression-list⟩ bzw. der ⟨initializer-list⟩ werden zuerst die Parameternamen durchsucht, bevor die Namen der Objektvariablen in Betracht gezogen werden.

intinit.cpp

```
int main() {
   cout << "Testing..." << endl;
   int i; // undefined
   cout << "i = " << i << endl;
   int j{17}; // well defined
   cout << "j = " << j << endl;
   int k{}; // well defined: 0
   cout << "k = " << k << endl;
}
```

- Die elementare Datentypen bieten ebenfalls parameterlose Konstruktoren an und einen Konstruktor mit einem Parameter des entsprechenden Typs.
- Wenn jedoch kein Konstruktor explizit aufgerufen wird, erfolgt keine Initialisierung.

```
clonmel$ intinit
Testing...
i = 4927
j = 17
k = 0
clonmel$
```

counter.hpp

```
int get() const {
   return counter;
}
```

- Zugriffsmethoden, die den abstrakten Zustand des Objekts nicht verändern dürfen, werden mit **const** ausgezeichnet.
- Nur diese Methoden dürfen aufgerufen werden, wenn das Objekt in einem Kontext nur lesenderweise zur Verfügung steht.
- Mit **mutable** deklarierte Variablen dürfen auch von **const**-Methoden verändert werden. Dies ist sinnvoll etwa zur Vermeidung sich sonst wiederholender Berechnungen und sollte nicht zu außen sichtbaren Veränderungen führen. (Abstrakter vs. konkreter Zustand eines Objekts.)

```
Counter(const Counter& orig) : counter{orig.counter} {
}
```

- Wenn einer der Konstruktoren genau einen Parameter mit einem Referenztyp der eigenen Klasse hat, dann handelt es sich dabei um einen expliziten Kopierkonstruktor.
- Normalerweise wird dieser mit **const** versehen.
- Der Kopierkonstruktor wird dann ggf. implizit verwendet bei der Parameterübergabe, bei **return** und einer Zuweisung.
- Wenn der Kopierkonstruktor nicht explizit deklariert und nicht ausdrücklich unterbunden wird, dann erzeugt der Übersetzer einen, wobei jedes Teilobjekt entsprechend kopierkonstruiert wird. Das funktioniert nur, wenn dies für alle Teilobjekte geht.

- Klassen, die selbst Ressourcen verwalten wie etwa dynamisch angelegte Speicherbereiche, benötigen sehr viel Sorgfalt in C++, damit mit der impliziten Verwendung von Konstruktoren und Methoden keine Probleme entstehen.
- Es ist dabei insbesondere sicherzustellen, dass dynamische Datenstrukturen nur ein einziges Mal freigegeben werden. D.h. das unbemerkte Kopieren von Zeigern ist ein Problem.
- Das folgende Beispiel illustriert dies an einer sehr einfachen Klasse, die ein **int**-Array verwaltet.

Klassen mit Zeigern

array.hpp

```
class Array {
   public:
      Array() : nof_elements(0), ip(nullptr) {}
      Array(unsigned int nof_elements) : nof_elements{nof_elements},
            ip{new int[nof_elements] {}} {
      }
      Array(const Array& other) : nof_elements{other.nof_elements},
            ip{new int[nof_elements]} {
         for (unsigned int i = 0; i < nof_elements; ++i) {
            ip[i] = other.ip[i];
         }
      }
      Array(Array&& other) : nof_elements{other.nof_elements},
            ip{other.ip} {
         other.nof_elements = 0; other.ip = nullptr;
      }
      ~Array() { delete[] ip; }
      Array& operator=(const Array& other) = delete;
      Array& operator=(Array&& other) = delete;
      unsigned int size() const { return nof_elements; }
      int& operator()(unsigned int i) {
         assert(i < nof_elements); return ip[i];
      }
      const int& operator()(unsigned int i) const {
         assert(i < nof_elements); return ip[i];
      }
   private:
      unsigned int nof_elements; int* ip;
};
```

array.hpp

```
Array(unsigned int nof_elements) : nof_elements{nof_elements},
      ip{new int[nof_elements] {}} {
}
```

- Mit dem **new**-Operator kann Speicher dynamisch belegt werden. Neben einem Typnamen kann auch in Array-Notation eine Dimensionierung mit angegeben werden.
- Hier wird ein **int**-Array mit *nof_elements* Elementen angelegt.
- Der **new**-Operator lässt hier auch sogleich die Initialisierung der neuen Speicherfläche hinzu. Da {} hier angegeben ist, wird das Array mit Nullen initialisiert. (Ohne diesen expliziten Hinweis würden die **int** uninitialisiert bleiben.)

array.hpp

```
~Array() {
   delete[] ip;
}
```

- Mit dem Operator **delete**[] kann ein Array wieder freigegeben werden.
- Wenn der Zeiger ein **nullptr** sein sollte, stört das nicht.
- Anders als in Java wird diese Funktion garantiert aufgerufen, wenn die Lebenszeit des Objekts beendet ist.

Der Begriff *Resource Acquisition Is Initialization* (RAII) geht auf Bjarne Stroustrup und Andrew Koenig zurück. Folgende Prinzipien sind damit verknüpft:

- Ressourcen werden mit Objekten fest verknüpft.
- Bei der Initialisierung des Objekts wird die Ressource akquiriert.
- Beim Dekonstruieren des Objekts wird die Ressource freigegeben.

Diese Technik vermeidet Fehler und stellt insbesondere sicher, dass auch im Falle einer Ausnahmenbehandlung (*exception handling*) alles sauber abgebaut und damit freigegeben wird.

Da bei C++ Objekte kopiert und nicht ohne weiteres nur Zeiger einander zugewiesen werden, ist bei Klassen, die mit Ressourcen in Verbindung stehen, Vorsicht geboten:

Wenn immer eine Klasse eine Ressource verwaltet, sind folgende spezielle Methoden immer explizit zu definieren bzw. zu deaktivieren, um die unerwünschte implizite Definition zu unterbinden:

- Kopier-Konstruktor
- Zuweisungs-Operator
- Dekonstruktor

array.hpp

```
Array(const Array& other) : nof_elements{other.nof_elements},
      ip{new int[nof_elements]} {
   for (unsigned int i = 0; i < nof_elements; ++i) {
      ip[i] = other.ip[i];
   }
}
```

- Der Kopierkonstruktor hat die Aufgabe, die gesamte Datenstruktur zu duplizieren.
- Hierfür ist das Kopieren nur des Zeigers unzureichend. Denn dann würde die Klon-Semantik verlorengehen und wir hätten das Problem, dass das Array beim Abbau mehrfach freigegeben würde. Die vom Übersetzer zur Verfügung stehende voreingestellte Implementierung dieses Konstruktors wäre somit fatal.
- Hier wird wiederum dynamisch Speicher von der gegebenen Größe angelegt und dann mit Hilfe der **for**-Schleife die Elemente einzeln kopiert. (Wie das effizienter geht, kommt später.)

Verschiebekonstruktor für das Array

array.hpp

```
Array(Array&& other) : nof_elements{other.nof_elements},
      ip{other.ip} {
   other.nof_elements = 0;
   other.ip = nullptr;
}
```

- Der Verschiebekonstruktor (*move constructor*) wird dann verwendet, wenn das Quellobjekt unmittelbar nach dem Aufruf abgebaut wird. Das ist insbesondere bei temporären Objekten der Fall.
- In diesem Fall können wir die dynamische Datenstruktur einfach übernehmen und müssen dann nur sicherstellen, dass beim Abbau des Quellobjekts keine versehentliche Freigabe der Datenstruktur erfolgt.

array.hpp

```
Array& operator=(const Array& other) = delete;
Array& operator=(Array&& other) = delete;
```

- Der Übersetzer unterstützt auch implizit Zuweisungen. Bei Objekten mit Zeigern ist hier ebenfalls die voreingestellte Implementierung unzureichend.
- Hier wird gezeigt, wie die voreingestellte Implementierung mit Hilfe von = **delete** unterdrückt werden kann, ohne sie durch eine eigene Implementierung zu ersetzen.

Zugriff auf ein Array-Element

array.hpp

```
int& operator()(unsigned int i) {
   assert(i < nof_elements); return ip[i];
}
const int& operator()(unsigned int i) const {
   assert(i < nof_elements); return ip[i];
}
```

- Statt traditioneller *set*- und *get*-Methoden kann auch der direkte Zugriff auf ein ansonsten privates Element gegeben werden, indem eine Referenz zurückgegeben wird.
- Das Resultat kann dann sowohl als *lvalue* (links von einer Zuweisung) als auch als *rvalue* (rechts der Zuweisung) verwendet werden.
- Methoden können auch nach einem Operator benannt werden mit Hilfe des Schlüsselworts **operator**. (Hier ist es der Funktionsoperator ().)

```
Array a{10};
a(1) = 77;
a(2) = a(1) + 10;
```

array.hpp

```
friend void swap(Array& a1, Array& a2) {
   std::swap(a1.nof_elements, a2.nof_elements);
   std::swap(a1.ip, a2.ip);
}
Array(const Array& other) :
      nof_elements{other.nof_elements},
      ip{new int[nof_elements]} {
   for (unsigned int i = 0; i < nof_elements; ++i) {
      ip[i] = other.ip[i];
   }
}
Array(Array&& other) : Array() {
   swap(*this, other);
}
Array& operator=(Array other) {
   swap(*this, other);
   return *this;
}
```

- Wenn die Zuweisung unterstützt werden soll, dann dient ein *swap*-Operator der Vereinfachung.

- Das Lambda-Kalkül geht auf Alonzo Church und Stephen Kleene zurück, die in den 30er-Jahren damit ein formales System für berechenbare Funktionen entwickelten.
- Zu den wichtigsten Arbeiten aus dieser Zeit gehört der Aufsatz von Alonzo Church: *An Unsolvable Problem of Elementary Number Theory*, *Americal Journal of Mathematics*, Band 58, Nr. 2 (April 1936), S. 345–363.
- Diese Arbeit zeigt, dass es keine berechenbare Funktion gibt, die die Äquivalenz zweier Ausdrücke des Lambda-Kalküls feststellen kann.
- Die Turing-Maschine und das Lambda-Kalkül sind in Bezug auf die Berechenbarkeit äquivalent.
- Das Lambda-Kalkül wurde von funktionalen Programmiersprachen übernommen (etwa von Lisp und Scheme) und wird auch gerne zur formalen Beschreibung der Semantik einer Programmiersprache verwendet (denotationelle Semantik).

Da in C++ Funktionsobjekte wegen der entsprechenden STL-Algorithmen recht beliebt sind, gab es mehrere Ansätze, Lambda-Ausdrücke in C++ einzuführen:

- *boost*::*lambda* von Jaakko Järvi, entwickelt von 1999 bis 2004
- *boost*::*phoenix* von Joel de Guzman und Dan Marsden, entwickelt von 2002 bis 2005
- Integration von Lambda-Ausdrücken in den C++-Standard ISO-14882-2012.

transform2.cpp

```
int main() {
   std::list<int> ints;
   for (int i = 1; i <= 10; ++i) {
      ints.push_back(i);
   }

   std::list<int> squares;
   std::transform(ints.begin(), ints.end(),
      std::back_inserter(squares), [](int val) { return val*val; });

   for (int val: squares) {
      std::cout << val << std::endl;
   }
}
```

- Mit Lambda-Ausdrücken werden implizit unbenannte Klassen erzeugt und temporäre Objekte instantiiert.
- In diesem Beispiel ist [](**int** *val*){ **return** *val*∗*val*; } der Lambda-Ausdruck, der ein temporäres unäres Funktionsobjekt erzeugt, das sein Argument quadriert.

⟨lambda-expression⟩ ⟶ ⟨lambda-introducer⟩ [⟨lambda-declarator⟩]
⟨compound-statement⟩

⟨lambda-introducer⟩ ⟶ „[“ [⟨lambda-capture⟩] „]“

⟨lambda-capture⟩ ⟶ ⟨capture-default⟩

⟶ ⟨capture-list⟩

⟶ ⟨capture-default⟩ „,“ ⟨capture-list⟩

⟨capture-default⟩ ⟶ „&“ | „=“

⟨capture-list⟩ ⟶ ⟨capture⟩ [„...“]

⟶ ⟨capture-list⟩ „,“ ⟨capture⟩ [„...“]

Grammatik für Lambda-Ausdrücke

⟨capture⟩ ⟶ ⟨simple-capture⟩
⟶ ⟨init-capture⟩
⟨simple-capture⟩ ⟶ ⟨identifier⟩
⟶ „&“ ⟨identifier⟩
⟶ **this**
⟨init-capture⟩ ⟶ ⟨identifier⟩ ⟨initializer⟩
⟶ „&“ ⟨identifier⟩ ⟨initializer⟩
⟨lambda-declarator⟩ ⟶ „(“ ⟨parameter-declaration-clause⟩ „)“
[**mutable**] [⟨exception-specification⟩]
[⟨attribute-specifier-seq⟩]
[⟨trailing-return-type⟩]

- Die ⟨init-capture⟩ kommt mit dem C++14-Standard hinzu.

- Lambda-Ausdrücke, die in eine Funktion eingebettet sind, „sehen“ die lokalen Variablen aus den umgebenden Blöcken.
- In vielen funktionalen Programmiersprachen überleben die lokalen Variablen selbst dann, wenn der sie umgebende Block verlassen wird, weil es noch überlebende Funktionsobjekte gibt, die darauf verweisen. Dies benötigt zur Implementierung sogenannte *cactus stacks*.
- Da für C++ der Aufwand für diese Implementierung zu hoch ist und auch die Übersetzung normalen Programmtexts ohne Lamda-Ausdrücke verteuern würde, fiel die Entscheidung, einen alternativen Mechanismus zu entwickeln, der über eine *lambda-capture* spezifiziert wird.

Kopien der Variablen aus der Umgebung


```
template<typename T>
auto create_multiplier(T factor) {
   return [=](T val) { return factor*val; };
}

int main() {
   auto multiplier = create_multiplier(7);
   for (int i = 1; i < 10; ++i) {
      std::cout << multiplier(i) << std::endl;
   }
}
```

- *create_multiplier* ist eine Template-Funktion, die ein mit einem vorgegebenen Faktor multiplizierendes Funktionsobjekt erzeugt und zurückliefert.
- Die Standard-Template-Klasse *function* wird hier genutzt, um das Funktionsobjekt in einen bekannten Typ zu verpacken.
- Die *lambda-capture* [=] legt fest, dass die aus der Umgebung referenzierten Variablen beim Erzeugen des Funktionsobjekts kopiert werden.

Erzeugung einer unbenannten Klasse 73

```
template<typename T>
class Anonymous {
   public:
      Anonymous(const T& factor) : factor(factor) {}
      T operator()(T val) const {
         return factor*val;
      }
   private:
      T factor;
};

template<typename T>
auto create_multiplier(T factor) {
   return Anonymous<T>(factor);
}
```

- Der Lambda-Ausdruck führt implizit zu einer Erzeugung einer unbenannten Klasse (hier einfach *Anonymous* genannt).
- Jeder aus der Umgebung referenzierte Variable, die kopiert wird, findet sich als gleichnamige Variable der Klasse wieder, die bei der Konstruktion übergeben wird.

```
std::vector<int> values(10);
int count = 0;
std::generate(values.begin(), values.end(), [&]() { return ++count; });
```

- Alternativ können Variablen nicht kopiert, sondern per impliziter Referenz benutzt werden.
- Dann darf das Funktionsobjekt aber nicht länger leben bzw. benutzt werden, als die entsprechenden Variablen noch leben. Das liegt in der Verantwortung des Programmierers.
- *generate* steht über **#include** <algorithm> zur Verfügung und weist die von dem Funktionsobjekt erzeugten Werte sukzessiv allen referenzierten Werten zwischen dem ersten Iterator (inklusive) und dem zweiten Iterator (exklusive) zu.

- Generische Klassen und Funktionen, in C++ *templates* genannt, sind unvollständige Deklarationen bzw. Definitionen, die von Parametern abhängen. Überwiegend handelt es sich dabei um Typparameter.
- Sie können nur in instantiierter Form verwendet werden, wenn alle Parameter gegeben und ggf. deklariert sind.
- Unter bestimmten Umständen ist auch eine implizite Festlegung eines Typparameter möglich, wenn sich dieser aus dem Kontext ergibt.
- Generische Module wurden zuerst von CLU und Ada unterstützt (nicht in Kombination mit OO-Techniken) und später in Eiffel, einer statisch getypten OO-Sprache.
- Generische Klassen wurden zunächst primär für Container-Klassen verwendet wie etwa in der STL, der Einsatz von Templates wurde aber zunehmend ausgeweitet, insbesondere nach der Einführung von C++11.

- Templates ähneln teilweise den Makros, da
  - der Übersetzer den Programmtext des generischen Moduls erst bei einer Instantiierung vollständig analysieren und nach allen Fehlern durchsuchen kann und
  - für jede Instantiierung (mit unterschiedlichen Parametern) Code zu generieren ist.
- Anders als bei Makros
  - müssen sich generische Module sich an die üblichen Regeln halten (korrekte Syntax, Sichtbarkeit, Typverträglichkeiten),
  - können Syntaxfehler schon vor einer Instantiierung festgestellt werden und es
  - lässt sich die Code-Duplikation im Falle zweier Instanzen mit identischen Parametern vermeiden.

Deklaration eines Templates


```
template<typename Element>
class List {
   public:
      // ...
      void add(const Element& element);
   private:
      struct Linkable {
         Element element;
         Linkable* next;
      };
      Linkable* list;
};
```

- Wenn der Klassendeklaration eine Template-Parameterliste vorangeht, dann wird daraus insgesamt die Deklaration eines Templates.
- Typparameter bei Templates sind typischerweise von der Form **typename** *T*, aber C++ unterstützt auch andere Parameter, die beispielsweise die Dimensionierung eines Arrays bestimmen.

```
List<int> list; // select int as Element type
list.add(7);
```

- Templates werden instantiiert durch die Angabe des Klassennamens und den Parametern in gewinkelten Klammern.

Deklaration eines Templates

⟨template-declaration⟩	⟶	**template**
		„<“ ⟨template-parameter-list⟩ „>“
		⟨declaration⟩
⟨template-parameter-list⟩	⟶	⟨template-parameter⟩
	⟶	⟨template-parameter-list⟩ „,“
		⟨template-parameter⟩
⟨template-parameter⟩	⟶	⟨type-parameter⟩
	⟶	⟨parameter-declaration⟩

- Eine reguläre ⟨parameter-declaration⟩ ist nur zulässig für ganzzahlige Datentypen wie etwa **int**, Aufzählungstypen, Zeiger und Referenzen.

⟨type-parameter⟩ ⟶ ⟨type-parameter-key⟩ [„...“] [⟨identifier⟩]
⟶ ⟨type-parameter-key⟩ [⟨identifier⟩]
„=“ ⟨type-id⟩
⟶ **template**
„<“ ⟨template-parameter-list⟩ „>“
⟨type-parameter-key⟩ [„...“] [⟨identifier⟩]
⟶ **template**
„<“ ⟨template-parameter-list⟩ „>“
⟨type-parameter-key⟩ [⟨identifier⟩]
„=“ ⟨id-expression⟩

⟨type-parameter-key⟩ ⟶ **class**
⟶ **typename**

- Das ist die Syntax von C++17. Bei älteren Versionen ist bei Template-Template-Parametern bei ⟨type-parameter-key⟩ immer **class** anzugeben.

sample-lines.cpp

```
#include <cstdlib>
#include <iostream>
#include <string>
#include "reservoir-sampler.hpp"

int main(int argc, char** argv) {
   ReservoirSampler<std::string> r(10);
   std::string line;
   while (std::getline(std::cin, line)) {
      r.add(std::move(line));
   }
   for (std::size_t index = 0; index < r.get_size(); ++index) {
      std::cout << r(index) << std::endl;
   }
}
```

- Diese Anwendung wählt aus den Zeilen der Standardeingabe bis zu 10 Zeilen zufällig aus und gibt sie anschließend aus.
- *ReservoirSampler* ist eine Container-Klasse, die sich bis zu n Objekten merkt entsprechend dem Reservoir-Sampling-Algorithmus (hier $n = 10$).

sample-lines.cpp

```
ReservoirSampler<std::string> r(nof_lines);
std::string line;
while (std::getline(std::cin, line)) {
   r.add(std::move(line));
}
for (std::size_t index = 0; index < r.get_size(); ++index) {
   std::cout << r(index) << std::endl;
}
```

- Mit *ReservoirSampler<std::string>* wird die Template-Klasse *ReservoirSampler* mit *std::string* als Typparameter instantiiert. Der Typparameter legt hier den Element-Typ des Containers fest.
- Der Konstruktor erwartet eine ganze Zahl als Parameter, der die Zahl der auszuwählenden Einträge bestimmt.
- Der ()-Operator wurde hier überladen, um einen Zugriff auf die ausgewählten Elemente zu erlauben.

reservoir-sampler.hpp

```
#ifndef RESERVOIR_SAMPLER_HPP
#define RESERVOIR_SAMPLER_HPP

#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <random>
#include <utility>

template<typename T>
class ReservoirSampler {
   public:
      /* ... */
   private:
      std::mt19937 engine;
      std::size_t size;
      std::size_t taken;
      T* data;
};

#endif
```

reservoir-sampler.hpp

```
std::mt19937 engine;
std::size_t size;
std::size_t taken;
T* data;
```

- Um Elemente zufällig aussuchen zu können, verwenden wir ein *std*::*mt19937*-Objekt als Generator für Pseudo-Zufallszahlen.
- *size* gibt die Zahl der gewünschten Elemente an.
- *taken* gibt die Zahl der mit *add* hinzugefügten Elemente an.
- *data* zeigt auf ein Array mit *size* Elementen.
- Bei *data* sind nur $min(size, taken)$ Elemente tatsächlich existent. Wir werden hier noch sehen, wie wir in C++ mit der Situation umgehen können, nur die tatsächlich existierenden Elemente zu konstruieren. Um das zu erreichen, wird das Belegen und Freigeben von Speicher von dem Konstruieren und Abbauen von Objekten völlig getrennt.
- Die Methode *get_size* liefert die Zahl der verfügbaren Elemente, also $min(size, taken)$.

reservoir-sampler.hpp

```
template<typename T>
class ReservoirSampler {
   public:
      ReservoirSampler(std::size_t size) :
         engine(std::random_device()()),
         size(size), taken(0),
         /* allocate raw memory without constructing anything */
         data(static_cast<T*>(operator new[](sizeof(T) * size))) {
      }
      /* ... */

   private:
      /* ... */
};
```

- Das ist der „normale“ Konstruktor, mit dem ein Reservoir einer gegebenen Größe angelegt werden kann.
- Um das Konstruieren noch nicht vorhandener Elemente des Typs T zu vermeiden, legen wir nur den Speicher an, ohne eines der Objekte zu konstruieren. Dies geht mit der expliziten Verwendung von **operator new**[], wobei dann die Größe in Bytes anzugeben ist.

reservoir-sampler.hpp

```
ReservoirSampler(const ReservoirSampler& other) :
      engine(std::random_device()()),
      size(other.size), taken(other.taken),
      /* allocate raw memory without constructing anything */
      data(static_cast<T*>(operator new[](sizeof(T) * size))) {
   for (std::size_t index = 0; index < other.get_size(); ++index) {
      /* copy-construct already constructed elements of other */
      new (data + index) T(other.data[index]);
   }
}
```

- Der Kopierkonstruktor konstruiert nur die n Elemente, die bei *other* bereits existieren.
- Die Methode *get_size* liefert uns n, d.h. die Zahl existierender Elemente.
- Das Konstruieren auf bereits vorhandenem Speicher geht mit **new**, wenn vor dem Datentyp in Klammern die Adresse angegeben wird (*placement*).

reservoir-sampler.hpp

```
~ReservoirSampler() {
   /* as we allocated raw memory we need to deconstruct
      all elements ourselves */
   for (std::size_t index = 0; index < get_size(); ++index) {
      data[index].~T();
   }
   /* release raw memory */
   operator delete[](data);
}
```

- Da das Belegen von Speicher und das Konstruieren getrennt erfolgte, müssen wir beim Abbau auch beides getrennt vornehmen.
- Der *destructor* wird hier für die existierenden Elemente explizit aufgerufen.
- Mit **operator delete**[](*data*) wird dann nur der Speicher freigegeben.

reservoir-sampler.hpp

```
friend void swap(ReservoirSampler& rs1, ReservoirSampler& rs2) {
   /* there is no need to swap the engines */
   std::swap(rs1.size, rs2.size);
   std::swap(rs1.taken, rs2.taken);
   std::swap(rs1.data, rs2.data);
}
```

- Die *swap*-Funktion wird wie gewohnt implementiert – wir verzichten hier aus pragmatischen Gründen auf das Vertauschen der Pseudo-Zufallsgeneratoren.

reservoir-sampler.hpp

```
ReservoirSampler(ReservoirSampler&& other) : ReservoirSampler() {
   swap(*this, other);
}
ReservoirSampler& operator=(ReservoirSampler other) {
   swap(*this, other);
   return *this;
}
```

- Entsprechend dem *copy and swap idiom* lassen sich der *move constructor* und der Zuweisungsoperator wie gewohnt leicht implementieren.

reservoir-sampler.hpp

```
void add(T value) {
   if (taken < size) {
      /* move-construct new element in reservoir */
      new (data + taken) T(std::move(value));
   } else {
      std::size_t select = std::uniform_int_distribution<std::size_t>
         (0, taken)(engine);
      if (select < size) {
         using std::swap;
         /* use argument-dependent lookup (ADL) */
         swap(value, data[select]);
      }
   }
   ++taken;
}
```

- Wenn *taken* $<$ *size*, dann ist ein weiteres Element zu konstruieren. Das erfolgt hier wieder mit **new** unter Verwendung eines *placement* auf *data* + *taken*.
- Da die lokale Variable *value* danach nicht mehr benötigt wird, können wir hier *std*::*move*(*value*) verwenden, so dass ggf. der *move constructor* zum Einsatz kommt.

reservoir-sampler.hpp

```
if (select < size) {
   using std::swap;
   /* use argument-dependent lookup (ADL) */
   swap(value, data[select]);
}
```

- Wenn ein bereits existierendes Element auszutauschen ist, erledigen wir das mit *swap*.
- Wir benutzen hier *swap* ohne Qualifikation, damit ggf. die *swap*-Funktion gefunden wird, die im Namensraum von T liegt.
- Mit **using** *std*::*swap* wird sichergestellt, dass notfalls *std*::*swap* verwendet wird, wenn keine passendere *swap*-Funktion vorliegt.
- Da die Suche nach dem passenden *swap* den Datentyp der Argumente berücksichtigt, wird dies als *argument-dependent lookup* (ADL) bezeichnet.

reservoir-sampler.hpp

```
std::size_t get_taken() const {
   return taken;
}
std::size_t get_size() const {
   return std::min(size, taken);
}
const T& operator()(std::size_t index) const {
   assert(index < get_size());
   return data[index];
}
```

- Die verbleibenden Funktionen sidn trivial zu implementieren.
- Die Minimumfunktion *std*::*min* wird von **#include** <algorithm> geliefert und akzeptiert beliebig viele Argumente.

- Template-Klassen können nicht ohne weiteres mit beliebigen Typparameter instantiiert werden.
- C++ verlangt, dass *nach* der Instantiierung die gesamte Template-Deklaration und alle zugehörigen Methoden zulässig sein müssen in C++.
- Entsprechend führt jede neuartige Instantiierung zur völligen Neuüberprüfung der Template-Deklaration und aller zugehörigen Methoden unter Verwendung der gegebenen Parameter.
- Daraus ergeben sich Abhängigkeiten, die ein Typ, der als Parameter bei der Instantiierung angegeben wird, einzuhalten hat.

- Folgende Abhängigkeiten sind zu erfüllen für den Typ-Parameter T der Template-Klasse *ReservoirSample*:
  - Es wird wird ein Kopierkonstruktor für T benötigt. Dieser ist zwingend für den Kopierkonstruktor von *ReservoirSample* notwendig. In den anderen Fällen kann auch alternativ ein *move constructor* bzw. *swap* zum Zuge kommen, falls vorhanden.
  - Destruktor: Für den Abbau der Objekte (entweder beim Austauschen oder beim Abbau des gesamten Reservoirs) wird dieser benötigt.

template-failure.cpp

```
#include "reservoir-sampler.hpp"

struct Test {
   int i;
   Test(int i) : i(i) {}
   Test(const Test& other) = delete;
};

int main() {
   ReservoirSampler<Test> r(5);
   Test val{1};
   r.add(val);
}
```

- Hier wurde der Kopierkonstruktur explizit unterbunden, womit implizit auch der *move constructor* wegfällt.
- Damit wird eine der Template-Abhängigkeiten von *ReservoirSample* nicht erfüllt.

```
theon$ g++ -o template-failure template-failure.cpp 2>&1 | head -7
template-failure.cpp: In function 'int main()':
template-failure.cpp:12:13: error: use of deleted function 'Test::Test(const Test&)'
    r.add(val);
             ^
template-failure.cpp:6:4: note: declared here
    Test(const Test& other) = delete;
    ^~~~
theon$
```

- Die Fehlermeldungen können bei nicht erfüllten Template-Abhängigkeiten ungemein umfangreich werden.
- Es lohnt sich hier aber ein Blick auf die ersten Meldungen, die das Problem recht treffend beschreiben.

- Bei der Übersetzung von Templates gibt es ein schwerwiegendes Problem:
  - Dort, wo die Methoden einer Template-Klasse implementiert sind, ist nicht bekannt, welche Instanzen benötigt werden.
  - Dort, wo das Template instantiiert wird sind die Methodenimplementierungen der Template-Klasse unbekannt, wenn diese nicht in der entsprechenden Header-Datei stehen.
- Folgende Fragen stellen sich:
  - Wie kann der Übersetzer die benötigten Template-Instanzen generieren?
  - Wie kann vermieden werden, dass die gleiche Template-Instanz mehrfach generiert wird?

- Beim Inclusion-Modell wird mit Hilfe einer **#include**-Anweisung auch die Methoden-Implementierung hereinkopiert, so dass sie beim Übersetzung der instantiierenden Module sichtbar ist.
- Das funktioniert grundsätzlich bei allen C++-Übersetzern, aber es führt im Normalfall zu einer Code-Vermehrung, wenn das gleiche Template in unterschiedlichen Quellen in gleicher Weise instantiiert wird.
- Das Borland-Modell sieht hier eine zusätzliche Verwaltung vor, die die Mehrfach-Generierung unterbindet.
- Der *gcc* unterstützt das Borland-Modell, wenn jeweils die Option *-frepo* gegeben wird, die dann die Verwaltungsinformationen in Dateien mit der Endung *rpo* unterbringt. Dies erfordert die Zusammenarbeit mit dem Linker und funktioniert beim *gcc* somit nur mit dem GNU-Linker.

- Der elegantere Ansatz vermeidet zusätzliche **#include**-Anweisungen. Entsprechend muss der Übersetzer selbst die zugehörige Quelle finden.
- Hierfür gibt es kein standardisiertes Vorgehen. Jeder Übersetzer, der dieses Modell unterstützt, hat dafür eigene Verwaltungsstrukturen.
- *gcc* unterstützt dieses Modell jedoch nicht.
- Der von Sun ausgelieferte C++-Übersetzer (bei uns mit *CC* aufzurufen) folgt diesem Modell.
- Im C++-Standard von 2003 wurde dies explizit über das Schlüsselwort **export** unterstützt.
- Da dies jedoch von kaum jemanden implementiert worden ist, wurde dies bei C++11 gestrichen. Entsprechend ist das Inclusion-Modell das einzige, das sich in der Praxis durchgehend etabliert hat.

```
template class ReservoirSampling<std::string>;
```

- Die Kontrolle darüber, genau wann und wo der Code für eine konkrete Template-Instantiierung zu erzeugen ist, kann mit Hilfe expliziter Instantiierungen kontrolliert werden.
- Eine explizite Instantiierung wiederholt die Template-Deklaration ohne das Innenleben, nennt aber die Template-Parameter.
- Dann wird an dieser Stelle der entsprechende Code erzeugt.
- Das darf dann aber nur einmal im gesamten Programm erfolgen. Sonst gibt es Konflikte beim Zusammenbau.
- Seit C++11 ist es möglich, so eine explizite Instantiierung mit dem Schlüsselwort **extern** zu versehen. Dann wird die Generierung des entsprechenden Codes unterdrückt und stattdessen die anderswo explizit instantiierte Fassung verwendet.

```
extern template class ReservoirSampling<std::string>;
```

- Der Vorteil expliziter Instantiierungen liegt in der Vermeidung redundanten Codes, ohne sich auf entsprechende implementierungsabhängige Unterstützungen des Übersetzers verlassen zu müssen.
- Ein weiterer Vorzug ist die kürzere Übersetzungszeit, da die Template-Implementierung dann nur noch dort benötigt wird, wo explizite Instantiierungen vorgenommen werden.
- Diese Vorgehensweise nötigt den Programmierer jedoch, selbst einen Überblick zu behalten, welche Instantiierungen alle benötigt werden. Das wird sehr schnell sehr unübersichtlich.
- Das liegt an der sogenannten *one-definition-rule* (ODR), d.h. Objekte dürfen beliebig oft deklariert, aber global nur einmal definiert werden. Bei impliziten Instantiierungen ist das ein Problem des Übersetzters, bei expliziten Instantiierungen übernimmt der Programmierer die Verantwortung hierfür.
- Diese Technik wird daher typischerweise nur in isolierten Fällen benutzt.