- 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⟩

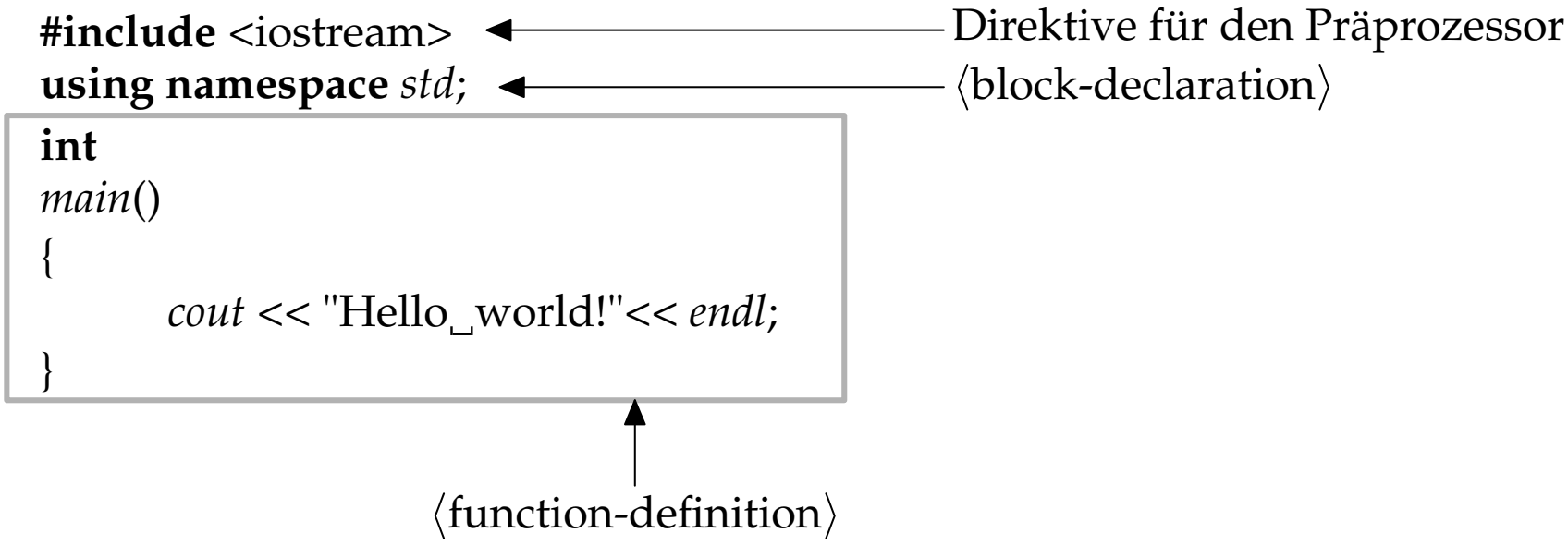
⟶ ⟨explicit-specialization⟩

⟶ ⟨linkage-specification⟩

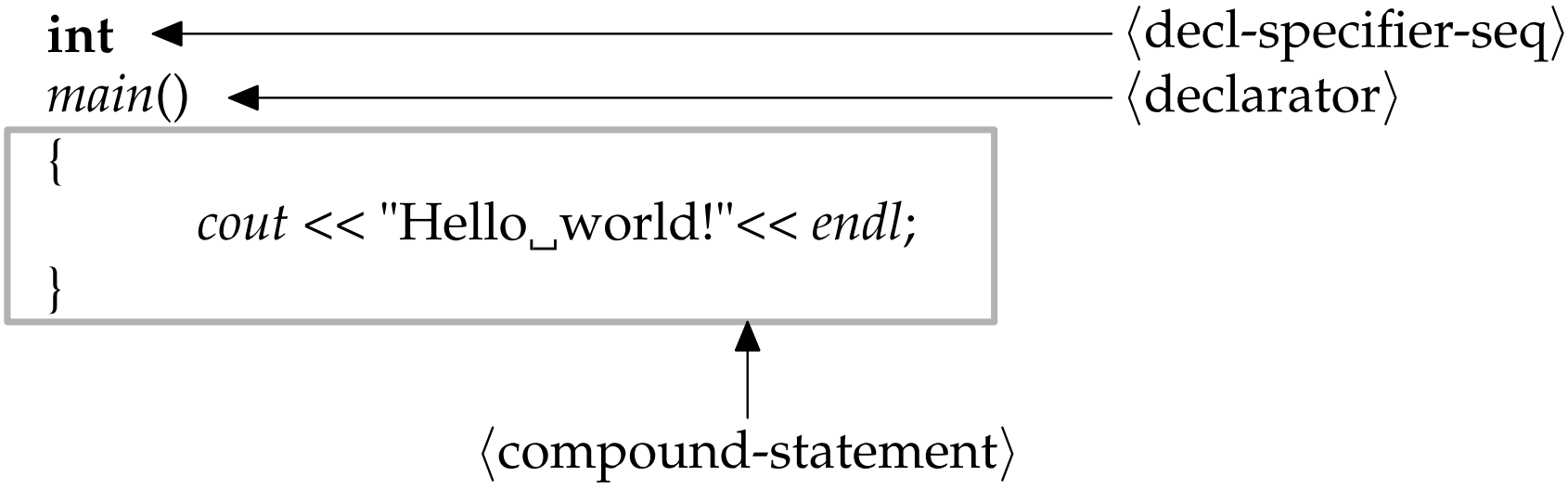
⟶ ⟨namespace-definition⟩

⟶ ⟨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

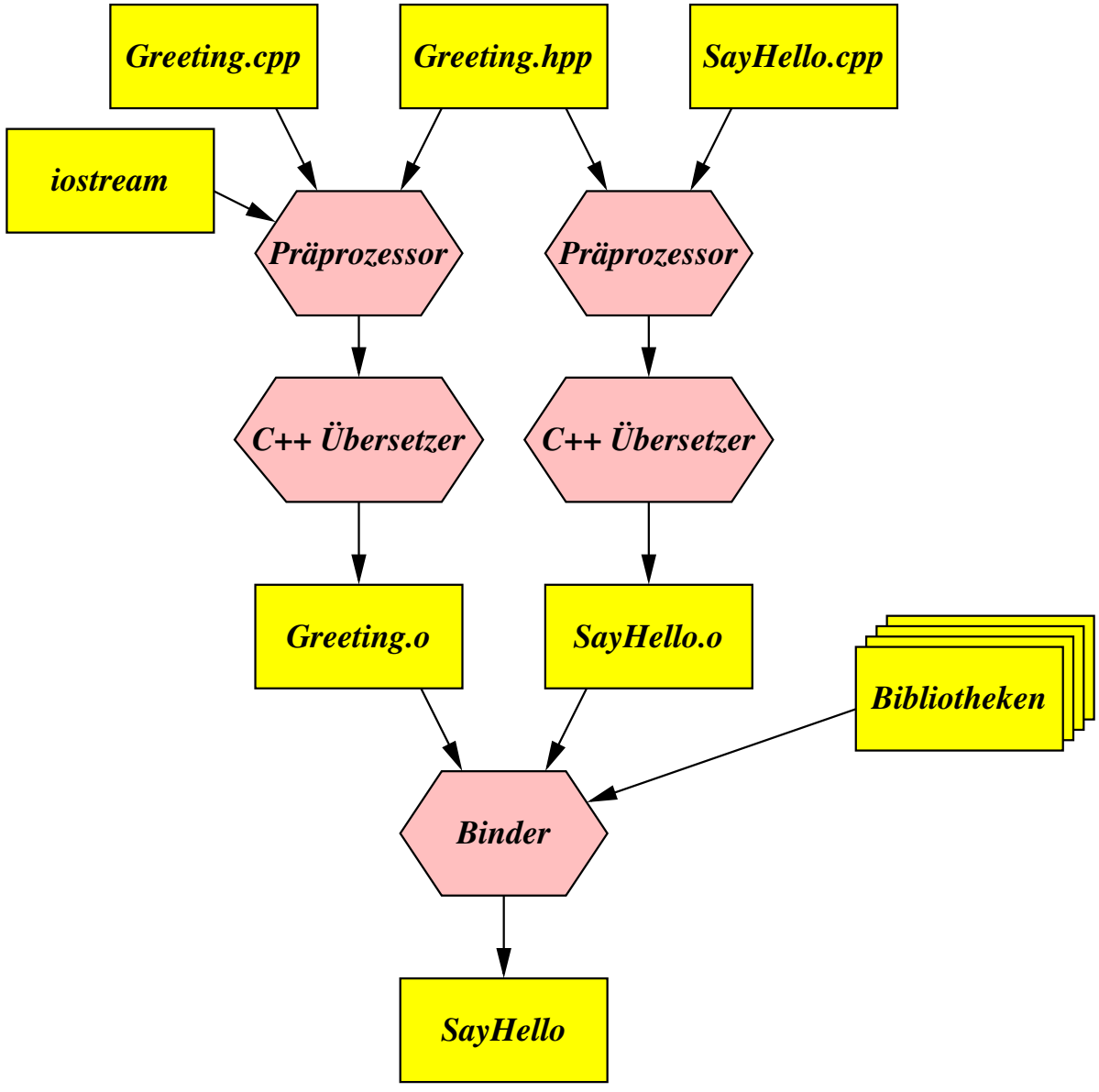
```
#include "Greeting.hpp"

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.


Greeting.cpp
Greeting.hpp
SayHello.cpp
iostream
Präprozessor
Präprozessor
C++ Übersetzer
C++ Übersetzer
Greeting.o
SayHello.o
Bibliotheken
Binder
SayHello

- 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
Greeting.cpp  Greeting.hpp  SayHello.cpp
theon$ wget --quiet \
> http://www.mathematik.uni-ulm.de/numerik/cpp/ss18/Makefile
theon$ make depend
gcc-makedepend  Greeting.cpp SayHello.cpp
theon$ make
g++ -Wall -g -std=gnu++17   -c -o Greeting.o Greeting.cpp
g++ -Wall -g -std=gnu++17   -c -o SayHello.o SayHello.cpp
g++ -o SayHello  Greeting.o SayHello.o
theon$ ./SayHello
Hello, fans of C++!
Hello, fans of C++!
theon$ make realclean
rm -f Greeting.o SayHello.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$ wget --quiet \
> http://www.mathematik.uni-ulm.de/numerik/cpp/ss18/Makefile
```

- Unter der genannten URL steht eine Vorlage für ein für C++ geeignetes *Makefile* zur Verfügung.
- Das Kommando *wget* lädt Inhalte von einer gegebenen URL in das lokale Verzeichnis.

```
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.

Vollständige Implementierung im Header

Greeting.hpp

```
#ifndef GREETING_H
#define GREETING_H

#include <iostream>

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

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).

Lebensort und -dauer eines Objekts

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.