- 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.

Greeting.h

```
#ifndef GREETING_H
#define GREETING_H

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

#endif
```

- Klassendeklarationen (mitsamt allen öffentlichen und auch privaten Datenfeldern und Methoden) sind in Dateien, die mit ».h« oder ».hpp« enden, unterzubringen. Hierbei steht ».h« für Header-Datei bzw. ».hpp« Header-Datei von C++.
- Kommentare starten mit »//« und erstrecken sich bis zum Zeilenende.
- 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.h

```
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.

Greeting.h

```
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.h

```
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.C

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

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 ».C«, ».cc« oder ».cpp« üblich.
- Die erste 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.
- Der Übersetzer erhält als Argumente nur diese Dateien, nicht die Header-Dateien.

Greeting.C

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

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.C

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

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.C

```
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.C

```
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 Operator einen *ostream* und als rechten Operator 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.C

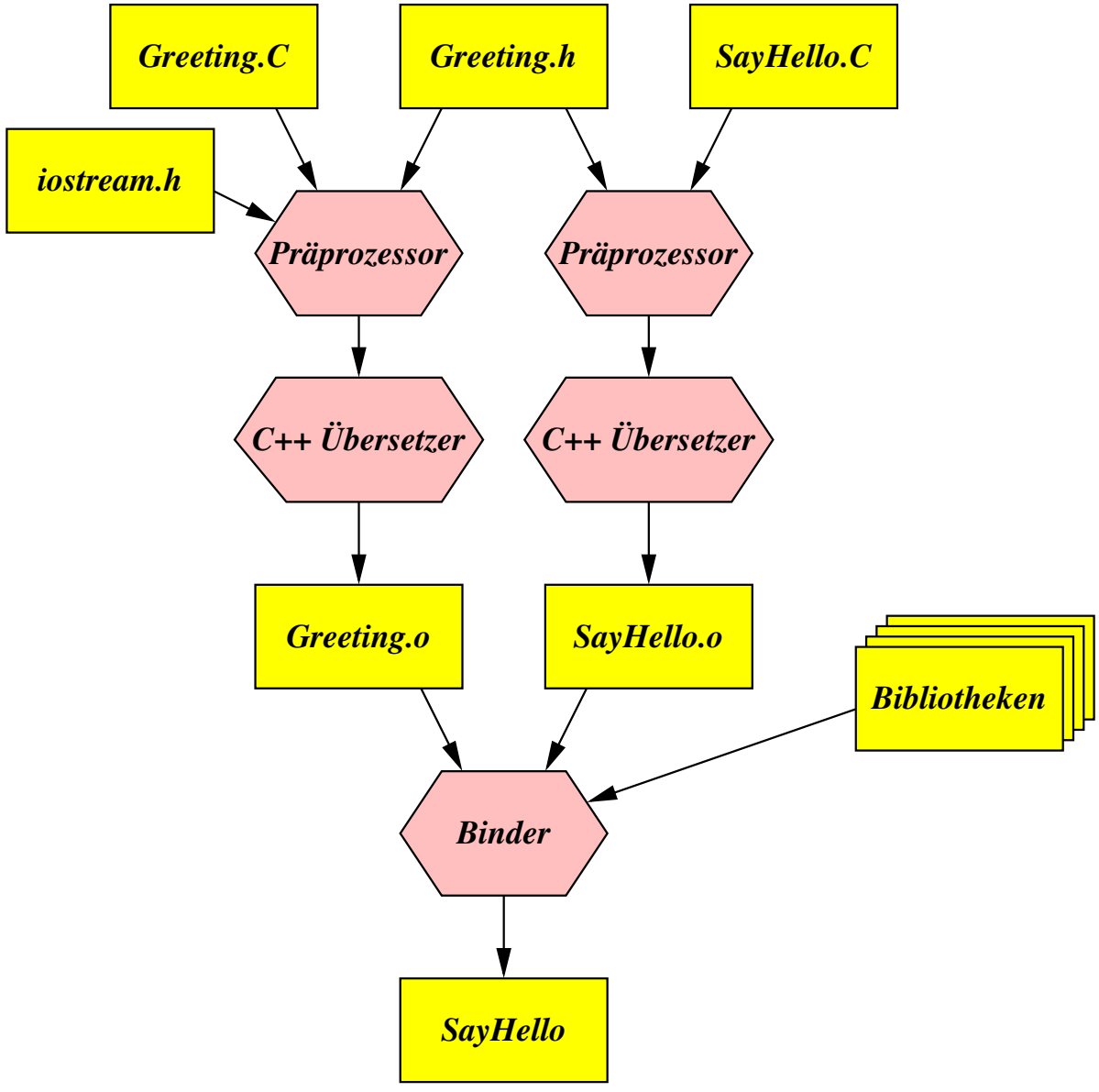
```
#include "Greeting.h"

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.C

```
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.C
Greeting.h
SayHello.C
iostream.h
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.

```
clonard$ ls
Greeting.C  Greeting.h  SayHello.C
clonard$ wget --quiet \
> http://www.mathematik.uni-ulm.de/sai/ss07/cpp/cpp/makefile
clonard$ sed 's/PleaseRenameMe/SayHello/' <makefile >makefile.tmp &&
> mv makefile.tmp makefile
clonard$ make depend
clonard$ make
g++ -Wall -g   -c -o SayHello.o SayHello.C
g++ -Wall -g   -c -o Greeting.o Greeting.C
g++ -R/usr/local/lib  SayHello.o Greeting.o   -o SayHello
clonard$ ./SayHello
Hello, world!
clonard$ make realclean
rm -f Greeting.o SayHello.o
rm -f SayHello
clonard$
```

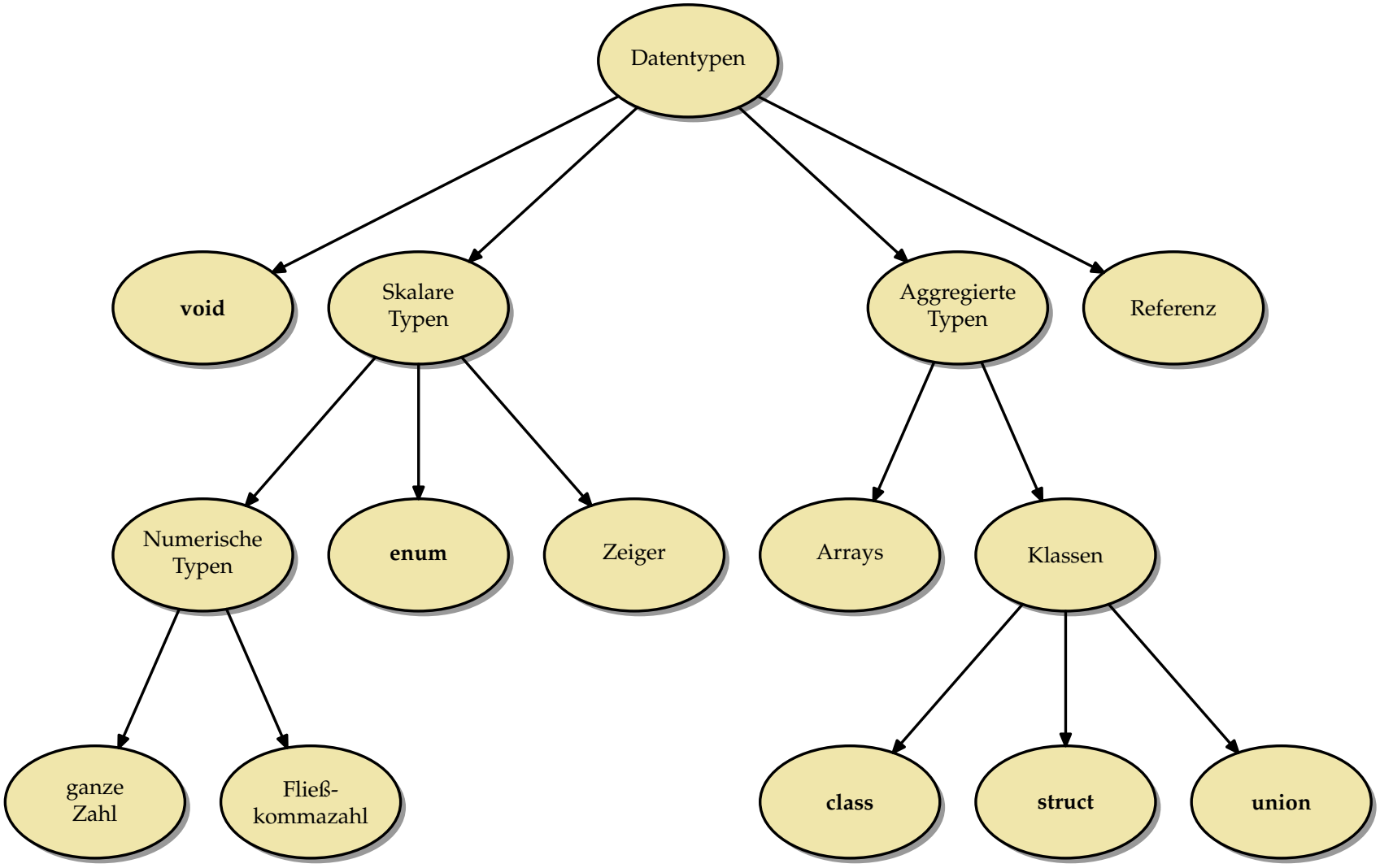
- *make* ist ein Werkzeug, das eine Datei namens *Makefile* (oder *makefile*) im aktuellen Verzeichnis erwartet, in der Methoden zur Generierung bzw. Regenerierung von Dateien beschrieben werden und die zugehörigen Abhängigkeiten.
- *make* ist dann in der Lage festzustellen, welche Zieldateien fehlen 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.

```
clonard$ wget --quiet \
> http://www.mathematik.uni-ulm.de/sai/ss07/cpp/cpp/makefile
clonard$ sed 's/PleaseRenameMe/SayHello/' <makefile >makefile.tmp &&
> mv makefile.tmp 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.
- In der Vorlage fehlt noch die Angabe, wie Ihr Programm heißen soll. Das wird hier mit dem Kommando *sed* nachgeholt, indem der Text »PleaseRenameMe« entsprechend ersetzt wird.

```
clonard$ make depend
```

- Das heruntergeladene *makefile* geht davon aus, dass Sie den g++ verwenden (GNU C++ Compiler) und die regulären C++-Quellen in ».C« enden und die Header-Dateien in ».h«.
- 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 *gcc-makedepend* von uns klauen. Sie finden es auf einem beliebigen unserer Rechner unter »/usr/local/bin/gcc-makedepend«. Es ist in Perl geschrieben und sollte mit jeder üblichen Perl-Installation zurechtkommen.


Datentypen
void
Skalare Typen
Aggregierte Typen
Referenz
Numerische Typen
enum
Zeiger
Arrays
Klassen
ganze Zahl
Fließ-kommazahl
class
struct
union

- Zu den skalaren Datentypen gehören alle elementaren Typen, die entweder numerisch sind oder sich zu einem numerischen Typ konvertieren lassen.
- Ein Wert eines skalaren Datentyps kann beispielsweise ohne weitere Konvertierung in einer Bedingung verwendet werden.
- Entsprechend wird die 0 im entsprechenden Kontext auch als Null-Zeiger interpretiert oder umgekehrt ein Null-Zeiger ist äquivalent zu *false* und ein Nicht-Null-Zeiger entspricht innerhalb einer Bedingung *true*.
- Ferner liegt die Nähe zwischen Zeigern und ganzen Zahlen auch in der von C++ unterstützten Adressarithmetik begründet.

- Die Spezifikation eines ganzzahligen Datentyps besteht aus einem oder mehreren Schlüsselworten, die die Größe festlegen, und dem optionalen Hinweis, ob der Datentyp vorzeichenbehaftet ist oder nicht.
- Fehlt die Angabe von **signed** oder **unsigned**, so wird grundsätzlich **signed** angenommen.
- Die einzigen Ausnahmen hiervon sind **char** und **bool**.
- Bei **char** darf der Übersetzer selbst eine Voreinstellung treffen, die sich am effizientesten auf der Zielarchitektur umsetzen lässt.

Auch wenn Angaben wie **short** oder **long** auf eine gewisse Größe hindeuten, so legt keiner der C++-Standards die damit verbundenen tatsächlichen Größen fest. Stattdessen gelten nur folgende Regeln:

- Der jeweilige „größere“ Datentyp in der Reihe **char**, **short**, **int**, **long**, **long long** umfasst den Wertebereich der kleineren Datentypen, d.h. **char** ist nicht größer als **short**, **short** nicht größer als **int** usw.
- **wchar_t** basiert auf einem der anderen ganzzahligen Datentypen und übernimmt die entsprechenden Eigenschaften.
- Die korrespondierenden Datentypen mit und ohne Vorzeichen (etwa **signed int** und **unsigned int**) belegen exakt den gleichen Speicherplatz und verwenden die gleiche Zahl von Bits. (Entsprechende Konvertierungen erfolgen entsprechend der Semantik des Zweier-Komplements.)

In C++ werden alle ganzzahligen Datentypen durch Bitfolgen fester Länge repräsentiert: $\{a_i\}_{i=1}^{n}$ mit $a_i \in \{0, 1\}$. Bei ganzzahligen Datentypen ohne Vorzeichen ergibt sich der Wert direkt aus der binären Darstellung:

$$a = \sum_{i=1}^{n} a_i 2^{i-1}$$

Daraus folgt, dass der Wertebereich bei n Bits im Bereich von 0 bis $2^n - 1$ liegt.

Bei ganzzahligen Datentypen mit Vorzeichen übernimmt a_n die Rolle des Vorzeichenbits. Für die Repräsentierung gibt es bei C++ nur drei zugelassene Varianten:

- **Zweier-Komplement**:

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n 2^n$$

Wertebereich: $\left[-2^{n-1}, 2^{n-1} - 1\right]$
Diese Darstellung hat sich durchgesetzt und wird von fast allen Prozessor-Architekturen unterstützt.

- **Einer-Komplement**:

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n (2^n - 1)$$

Wertebereich: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$
Vorsicht: Es gibt zwei Repräsentierungen für die Null. Es gilt:
$-a == \tilde{}a$
Diese Darstellung gibt es auf einigen historischen Architekturen wie etwa der PDP-1, der UNIVAC 1100/2200 oder der 6502-Architektur.

- **Trennung zwischen Vorzeichen und Betrag**:

$$a = (-1)^{a_n} \sum_{i=1}^{n-1} a_i 2^{i-1}$$

Wertebereich: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$
Vorsicht: Es gibt zwei Repräsentierungen für die Null.
Diese Darstellung wird ebenfalls nur von historischen Architekturen verwendet wie etwa der IBM 7090.

- Leider verzichtet der C++-Standard (anders als bei C) auf Angaben hierzu.
- Die Header-Dateien <*limits*>, <*climits*> und <*cfloat*> liefern die unterstützten Wertebereiche und weitere Eigenschaften der Basistypen der lokalen C++-Implementierung.

links	Postfix-Operatoren: ++, −−, −>, ., etc
rechts	Unäre Operatoren: ++, −−, ∗, &, +, −, !, ~, **new**, **delete**
links	Multiplikative Operatoren: ∗, /, %
links	Additive Operatoren: +, −
links	Schiebe-Operatoren: <<, >>
links	Vergleichs-Operatoren: <, >, <=, >=
links	Gleichheits-Operatoren: ==, !=
links	Bitweises Und: &
links	Bitweises Exklusiv-Oder: ^
links	Bitweises Inklusiv-Oder: |
links	Logisches Und: &&
links	Logisches Oder: ||
rechts	Bedingungs-Operator: ?:
rechts	Zuweisungs-Operatoren: =, ∗=, /=, %=, +=, −=, >>=, <<=, &=, ^=, |=
links	Komma-Operator: ,

- Die Operatoren sind in der Reihenfolge ihres Vorrangs aufgelistet, beginnend mit den Operatoren höchster Priorität.
- Klammern können verwendet werden, um Operatoren mit Operanden auf andere Weise zu verknüpfen.
- Da nur wenige die gesamte Tabelle auswendig wissen, ist es gelegentlich ratsam, auch dann Klammern aus Gründen der Lesbarkeit einzusetzen, wenn sie nicht strikt notwendig wären.
- Sofern die Operatoren auch von C unterstützt werden, gibt es keine Änderungen der Prioritäten.

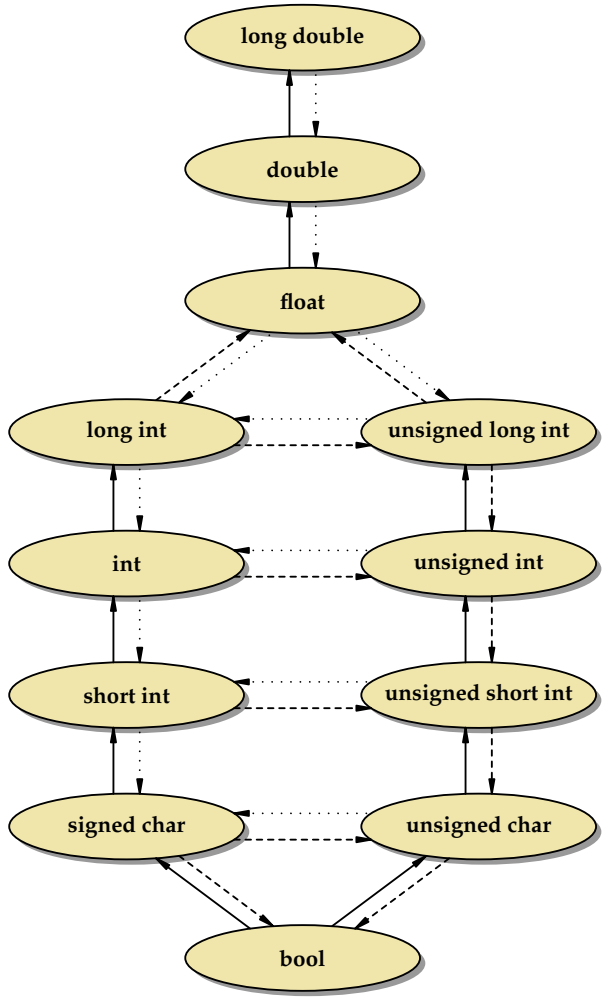
- Typ-Konvertierungen können in C sowohl implizit als auch explizit erfolgen.
- Implizite Konvertierungen werden angewendet bei Zuweisungs-Operatoren, Parameterübergaben und Operatoren. Letzteres schließt auch die unären Operatoren mit ein.
- Explizite Konvertierungen erfolgen durch den sogenannten Cast-Operator.

Bei einer Konvertierung zwischen numerischen Typen gilt der Grundsatz, dass – wenn irgendwie möglich – der Wert zu erhalten ist. Falls das jedoch nicht möglich ist, gelten folgende Regeln:

- Bei einer Konvertierung eines vorzeichenbehafteten ganzzahligen Datentyps zum Datentyp ohne Vorzeichen *gleichen Ranges* (also etwa von **int** zu **unsigned int**) wird eine ganze Zahl $a < 0$ zu b konvertiert, wobei gilt, dass $a \bmod 2^n = b \bmod 2^n$ mit n der Anzahl der verwendeten Bits, wobei hier der mod-Operator entsprechend der F-Definition bzw. Euklid gemeint ist. Dies entspricht der Repräsentierung des Zweier-Komplements.
- Der umgekehrte Weg, d.h. vom ganzzahligen Datentyp ohne Vorzeichen zum vorzeichenbehafteten Datentyp gleichen Ranges (also etwa von **unsigned int** zu **int**) hinterlässt ein *undefiniertes* Resultat, falls der Wert nicht darstellbar ist.

- Bei einer Konvertierung von größeren ganzzahligeren Datentypen zu entsprechenden kleineren Datentypen werden die nicht mehr darstellbaren höherwertigen Bits weggeblendet, d.h. es gilt wiederum $a \bmod 2^n = b \bmod 2^n$, wobei n die Anzahl der Bits im kleineren Datentyp ist. (Das Resultat ist aber nur bei ganzzahligen Datentypen ohne Vorzeichen wohldefiniert.)
- Bei Konvertierungen zu *bool* ist das Resultat 0 (**false**), falls der Ausgangswert 0 ist, ansonsten immer 1 (**true**).
- Bei Konvertierungen von Gleitkommazahlen zu ganzzahligen Datentypen wird der ganzzahlige Anteil verwendet. Ist dieser im Zieltyp nicht darstellbar, so ist das Resultat undefiniert.
- Umgekehrt (beispielsweise auf dem Wege von **long int** zu **float**) ist einer der beiden unmittelbar benachbarten darstellbaren Werte zu nehmen, d.h. es gilt entweder $a = b$ oder $a < b \wedge \nexists x : a < x < b$ oder $a > b \wedge \nexists x : a > x > b$ mit x aus der Menge des Zieltyps.

Konvertierungen bei numerischen Datentypen



Ausdrücke:	Ein Ausdruck, gefolgt von einem terminierenden Semikolon. Der Ausdruck wird bewertet und das Resultat nicht weiter verwendet. Typische Fälle sind Zuweisungen und Funktions- und Methodenaufrufe.
Blöcke:	Erlaubt die Zufammenfassung mehrerer Anweisungen und eröffnet einen lokalen lexikalisch begrenzten Sichtbereich.
Verzweigungen:	**if** (*condition*) *statement* **if** (*condition*) *statement* **else** *statement* **switch** (*condition*) *statement*
Wiederholungen:	**while** (*condition*) *statement* **do** *statement* **while** (*condition*); **for** (*for-init* ; *condition* ; *expression*) *statement*
Sprünge:	**return**; **return** *expression* ; **break**;, **continue**;, **goto** *identifier* ;

- Deklarationen sind überall zulässig. Die so deklarierten Objekte sind innerhalb des umgebenden lexikalischen Blocks sichtbar, jedoch nicht vor der Deklaration.
- Im Rahmen der (später vorzustellenden) Ausnahmenbehandlungen gibt es **try**-Blöcke.
- Anweisungen können Sprungmarken vorausgehen. Da **goto**-Anweisungen eher vermieden werden, sind Sprungmarken typischerweise nur im Kontext von **switch**-Anweisungen zu sehen unter Verwendung der Schlüsselworte **case** und **default**.

MetaChars.C

```
#include <iostream>
using namespace std;

int main() {
   char ch;
   int meta_count(0);
   bool within_range(false);
   bool escape(false);
   while ((ch = cin.get()) != EOF) {
      // counting ...
   }
   if (meta_count == 0) {
      cout << "No";
   } else {
      cout << meta_count;
   }
   cout << " meta characters were found.\n";
} // main()
```

- **int** *meta_count*(0); ist eine Deklaration, die eine Variable namens *meta_count* anlegt, die mit 0 initialisiert wird.
- Ohne Initialisierungen bleibt der Wert einer lokalen Variable solange undefiniert, bis ihr explizit ein Wert zugewiesen wird.
- Zu beachten ist hier der Unterschied zwischen = (Zuweisung) und == (Vergleich).
- Innerhalb der Bedingung der **while**-Schleife wird von *cin.get*() das nächste Zeichen von der Eingabe geholt, an *ch* zugewiesen und dann mit *EOF* verglichen.

MetaChars.C

```
      if (escape) {
         escape = false;
      } else {
         switch (ch) {
         case '*':
         case '?':
         case '\\':
            meta_count += 1; escape = true;
            break;
         case '[':
            meta_count += 1;
            if (!within_range) within_range += 1;
            break;
         case ']':
            if (within_range) meta_count += 1;
            within_range = 0;
            break;
         default:
            if (within_range) meta_count += 1;
            break;
         }
      }
```

MetaChars.C

```
switch (ch) {
case '*':
case '?':
case '\\':
   meta_count += 1; escape = true;
   break;
// cases '[' and ']' ...
default:
   if (within_range) {
      meta_count += 1;
   }
   break;
}
```

- Zu Beginn wird der Ausdruck innerhalb der **switch**-Anweisung ausgewertet.
- Dann erfolgt ein Sprung zu der Sprungmarke, die dem berechneten Wert entspricht.
- Falls keine solche Sprungmarke existiert, wird die Sprungmarke **default** ausgewählt, sofern sie existiert.

MetaChars.C

```
switch (ch) {
case '*':
case '?':
case '\\':
   meta_count += 1; escape = true;
   break;
// cases '[' and ']' ...
default:
   if (within_range) {
      meta_count += 1;
   }
   break;
}
```

- Beginnend von der ausgesuchten Sprungmarke wird die Ausführung bis zur nächsten *break*-Anweisung fortgesetzt oder eben bis zum Ende der **switch**-Anweisung.
- Zu beachten ist hier, dass *break*-Anweisungen jeweils nur die innerste **switch**-, **for**-, **while**- oder **do**-Anweisung verlassen.

Squares.C

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   int limit;
   cout << "Limit: "; cin >> limit;
   for (int n(1); n <= limit; n += 1) {
      cout << setw(4) << n << setw(11) << n*n << endl;
   }
}
```

- **for** (*initialization* ; *condition* ; *expression*) *statement*
  ist nahezu äquivalent zu

```
{
        initialization;
        while (condition){
                statement
                expression;
        }
}
```

- Die Initialisierung, die Bedingung und der Ausdruck dürfen leer sein. Wenn die Bedingung fehlt, wird es zur Endlosschleife.
- Der Sichtbereich von *n* wird lexikalisch auf den Bereich der **for**-Anweisung begrenzt.

Point1.h

```
class Point {
   public: // access
      void set_x(float x_coord);
      void set_y(float y_coord);
      float get_x();
      float get_y();

   private: // data
      float x;
      float y;
}; // Point
```

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

Quelle: Die Beispielserie der *Point*-Klassen ist von Jim Heliotis, RIT, geklaut worden.

- Abfragemethoden (*accessors*) wie etwa *get_x* und *get_y* in diesem Beispiel eröffnen der Außenwelt einen Blick auf den Zustand des Objekts. Ein Aufruf eines Akzessors sollte den von außen einsehbaren Objektzustand nicht verändern.
- Änderungsmethoden (wie etwa *set_x* und *set_y*) ermöglichen eine Veränderung des von außen beobachtbaren Objektzustands.
- Der Verzicht auf den direkten Zugang ermöglicht das Durchsetzen semantischer Bedingungen wie etwa der Klasseninvarianten.

Point1.C

```
void Point::set_x(float x_coord) {
   x = x_coord;
}

void Point::set_y(float y_coord) {
   y = y_coord;
}

float Point::get_x() {
   return x;
}

float Point::get_y() {
   return y;
}
```

- Im einfachsten Falle können Zugriffsmethoden direkt durch entsprechende **return**-Anweisungen und Zuweisungen implementiert werden.
- Dies mag umständlich erscheinen. Es erlaubt jedoch die einfache Änderung der internen Repräsentierung, ohne dass dabei die Schnittstelle angepasst werden muss.

Point1.C

```
#include <iostream>

// class declaration and definition ...

int main() {
   Point p;

   p.set_x(3.0);
   p.set_y(4.0);

   cout << "p=(" << p.get_x() << ','
      << p.get_y() << ')' << endl;

   return 0;
} // main
```

- Zu beachten ist hier, dass p solange undefiniert bleibt, bis beide Änderungsmethoden *set_x* und *set_y* aufgerufen worden sind.

```
dublin$ Point1
p=(3,4)
dublin$
```

Point2.h

```
class Point {
   public: // access
      void set_x(float x_coord);
      void set_y(float y_coord);
      float get_x();
      float get_y();

   private: // data
      float radius;
      float angle;
}; // Point
```

- Da die Datenfelder privat sind, ist ein Wechsel von kartesischen zu Polar-Koordinaten möglich. Diese Änderung ist für die Klienten dieser Klasse nicht zu ersehen, da diese weiterhin mit kartesischen Koordinaten arbeiten können.
- Hier ergibt sich ein Unterschied zwischen dem abstrakten Zustand (von den Klienten beobachtbar) und dem internen Zustand (aus der Sicht der Implementierung).

Point2.C

```
#include <cmath>
// ...
void Point::set_x(float x_coord) {
   float new_radius(sqrt(x_coord * x_coord +
      get_y() * get_y()));
   angle = atan2(get_y(), x_coord);
   radius = new_radius;
} // set_x

void Point::set_y(float y_coord) {
   float new_radius(sqrt(get_x() * get_x() +
      y_coord * y_coord));
   angle = atan2(y_coord, get_x());
   radius = new_radius;
} // set_y

float Point::get_x() {
   return cos(angle) * radius;
} // get_x

float Point::get_y() {
   return sin(angle) * radius;
} // get_y
```

Point2.C

```
#include <iostream>
// ...
int main() {
   Point p;

   p.set_x(3.0);
   p.set_y(4.0);

   cout << "p=(" << p.get_x() << ','
      << p.get_y() << ')' << endl;

   return 0;
}
```

- Leider führt der gleiche benutzende Programmtext zu einem anderen Resultat:

```
dublin$ Point2
p=(-NaN,-NaN)
dublin$
```

- *NaN* steht hier für »not-a-number«, d.h. für eine undefinierte Gleitkommazahl.

Point3.h

```
class Point {
   public: // construction
      Point(float x, float y);

   public: // access
      void set_x(float x_coord);
      void set_y(float y_coord);
      float get_x();
      float get_y();

   private: // data
      float radius;
      float angle;
}; // Point
```

- Konstruktoren erlauben es, von Anfang an einen wohldefinierten Zustand zu haben. Der Name einer Konstruktor-Methode ergibt sich immer aus dem Namen der Klasse.
- Wenn mindestens ein Konstruktor in der Klassendeklaration spezifiziert wird, dann ist es nicht mehr möglich, Objekte dieser Klasse zu deklarieren, ohne einen der Konstruktoren zu verwenden.

Point3.C

```
Point::Point(float x_coord, float y_coord) {
   radius = sqrt(x_coord * x_coord + y_coord * y_coord);
   angle = atan2(y_coord, x_coord);
} // Point::Point

// ...

int main() {
   Point p(18, -84.2);

   p.set_x(3.0);
   p.set_y(4.0);

   cout << "p=(" << p.get_x() << ',' <<
      p.get_y() << ')' << endl;

   return 0;
} // main
```

- Wenn mit einem wohldefinierten Zustand begonnen wird, dann lassen sich die kartesischen Koordinaten problemlos einzeln ändern.

Point4.h

```
class Point {
   public: // creation
      Point(const Point &p); // "const" is TBD
      Point(float x, float y);

   public: // access
      void set_x(float x_coord);
      void set_y(float y_coord);
      float get_x();
      float get_y();

   private: // data
      float radius;
      float angle;
}; // Point
```

- Der gleiche Name darf mehrfach für Methoden der gleichen Klasse vergeben werden, wenn die Signaturen sich bei den Typen der Parameter voneinander unterscheiden.
- Bei einer mehrfachen Verwendung eines Namens wird von einer Überladung gesprochen (*overloading*).

Point4.h

```
class Point {
   public: // creation
      Point(const Point &p); // "const" is TBD
      Point(float x, float y);

   public: // access
      void set_x(float x_coord);
      void set_y(float y_coord);
      float get_x();
      float get_y();

   private: // data
      float radius;
      float angle;
}; // Point
```

- Entsprechend kann ein zweiter Konstruktor definiert werden, der in diesem Beispiel die Koordinaten von einem existierenden Punkt-Objekt bezieht.
- **const** *Point*& *p* vermeidet im Vergleich zu *Point p* das Kopieren des Parameters und lässt (dank dem **const**) Änderungen nicht zu.

```
int i(0);
int& j(i); // j ist eine Referenz auf i
j = 3; // i hat jetzt den Wert 3
```

- Referenzen sind konstante Zeiger, die von Anfang an mit einem anderen Objekt fest verknüpft sind.
- Die Verknüpfung ergibt sich entweder
  - aus der Parameterübergabe,
  - aus der Rückgabe einer Funktion oder Methode oder
  - aus der Initialisierung
- Im Unterschied zu Zeigern entfällt die explizite Dereferenzierung.
- Referenzen werden durch die Verwendung eines & deklariert.

Überladen von Methoden

Point4.C

```
Point::Point(const Point &p) {
   radius = p.radius;
   angle = p.angle;
} // Point::Point
// ...
int main() {
   Point p(18, -84.2);
   Point q(p);

   p.set_x(3.0);
   p.set_y(4.0);

   cout << "p=(" << p.get_x() << ',' <<
      p.get_y() << ')' << endl;
   cout << "q=(" << q.get_x() << ',' <<
      q.get_y() << ')' << endl;

   return 0;
} // main
```

```
dublin$ Point4
p=(3,4)
q=(18,-84.2)
dublin$
```

Point5.C

```
Point::Point(float x_coord, float y_coord):
   radius(sqrt(x_coord * x_coord + y_coord * y_coord)),
   angle(atan2(y_coord, x_coord)) {
}

Point::Point(const Point &p):
   radius(p.radius), angle(p.angle) {
}
```

- Vor dem eigentlichen Block können bei Konstruktoren hinter dem Doppelpunkt Initialisierungssequenzen spezifiziert werden, die abgearbeitet werden, bevor die eigentliche Methode des Konstruktors aufgerufen wird.
- Dabei ist zu beachten, dass die Initialisierungsreihenfolge abgeleitet wird von der Reihenfolge in der Klassendeklaration und nicht der Reihenfolge der Initialisierungen.

Point5.C

```
Point::Point(float x_coord, float y_coord):
   radius(sqrt(x_coord * x_coord + y_coord * y_coord)),
   angle(atan2(y_coord, x_coord)) {
}

Point::Point(const Point &p):
   radius(p.radius), angle(p.angle) {
}
```

- Die Verwendung von Konstruktoren der Basisklasse ist dabei zulässig.
- Da in C++ grundsätzlich keine voreingestellten Initialisierungen stattfinden (anders als in Java oder vielen anderen OO-Sprachen), empfiehlt es sich, diese Initialisierungsmöglichkeit konsequent zur Vermeidung von Überraschungen einzusetzen.

```
const double PI = 3.141592653589793238 46;
```

- Hier wird *PI* als Konstante deklariert, d.h. *PI* kann nicht nachträglich verändert werden.

```
Point(const Point& p);
```

- Hier wird *p* über eine Referenz (d.h. einem nicht veränderbaren Zeiger) übergeben und gleichzeitig sichergestellt, dass *p* nicht vom Aufrufer verändert wird. Dies verbessert die Effizienz (da das Kopieren vermieden wird) ohne dies zu Lasten der Sicherheit zu tun.

```
float get_x() const;
```

- Durch das Schlüsselwort **const** am Ende der Signatur einer Methode wird diese zu einer reinen auslesenden Methode, d.h. sie darf den Zustand des Objekts nicht verändern.

```
const Key& get_key();
```

- Hier wird der Aufrufer dieser Funktion daran gehindert, den Wert hinter der zurückgelieferten Referenz zu verändern.