Dateibezugsrahmen Ein Projekt besteht im Normalfall aus mehreren Dateien. Jedes Modul besteht aus `*.cc` und `*.h`. Bei ungewollten Namenskonflikten besteht die Möglichkeit die entsprechende Funktion `static` zu deklarieren. Ansonsten die Möglichkeit mit `extern` eine "gewollten" Namenskonflikt zu kennzeichnen

Lokaler Bezugsrahmen Hier gilt, daß lokale Namen globale verdecken. Will man explizit auf globale Namen zugreifen, so verwendet man den `scope operator ::`.

```
goto label;
{
  // hier eingefuehrte Variablen sind lokal
  // dieser Block hilft evtl. dem Compiler
}
label:
```

Eine Alternative zu Variablen mit Filebezugsrahmen sind lokale Variablen in Funktionen, die static deklariert sind. Sie haben einen lokalen Bezugsrahmen, aber eine Lebensdauer, die über den Funktionsaufruf hinausgeht.

Klassenbezugsrahmen Bezeichner im Klassenbezugsrahmen werden alternativ durch `.name`, `->name` oder für `static` Variablen `Klassenname::name` angesprochen. Static Variablen existieren unabhängig von der Instanzbildung nur einmal für die ganze Klasse, d.h. alle Objekte/Instanzen einer Klasse teilen sich eine Variable oder Funktion

Namensräume Namensräume sind dazu gedacht, größere Einheiten wie z.B. die gesamte Standardbibliothek zusammenzufassen. Durch Ihre Benutzung wird vermieden, daß es bei Benutzung von mehreren großen Programmpaketen zu Problemen wegen doppelt vergebener Namen kommt. Bei der Benutzung von Teilen dieser Pakete spezifiziert man den Namen mit vorangestelltem `namespace Name{...}` oder importiert diese Teile mit `using` Direktiven.

```
namespace Name{
  ....
}
```

Durch `using` Direktiven kann man einzelne Elemente oder auch einen ganzen Namensraum "importieren", d.h. auf seine Elemente zugreifen ohne explizit anzugeben, in welchem Namensraum sie sich befinden. Allerdings kann man so nur einen Namensraum importieren.

2.8 Ein- und Ausgabe

Die Ein- und Ausgabe von C++ wurde gegenüber C vollständig überarbeitet. Dies war unter anderem deshalb notwendig, weil in C die Ein- und Ausgabefunktionen eine variable

Anzahl von Argumenten hatten und daher eine Überprüfung der Typen zur Compilezeit nicht stattfinden konnte.

Die beiden wichtigsten Objekte, mit denen man ständig zu tun hat, sind die Ein-/Ausgabe Kanal-Typen `istream` und `ostream`. Beide sind im header `iostream` deklariert. In den vorangegangenen Kapiteln sind die Standard Ein-/Ausgabe Kanäle '`cin`' und '`cout`' mit den Ein-/Ausgabe Operatoren '`<<`' oder '`>>`' bereits wiederholt benutzt worden. Im folgenden sollen die Anwendungs- und Erweiterungsmöglichkeiten genauer erklärt werden.

Wie funktioniert die Verkettung der Ein- oder Ausgabe?

```
#include <iostream>
//       Was der user eingibt:
     cout << "Gehalt: "   << Personal.mGehalt;

//       und wie es der Compiler interpretiert :
   ( cout << "Gehalt: " ) << Personal.mGehalt;
```

Zur Ausgabe des Strings `Gehalt:` wird der Operator `<<` vom `ostream cout` aufgerufen, dieser gibt wieder einen `ostream` zurück, mit dem wieder der Operator `<<` aufgerufen wird, diesmal mit `Personal.mGehalt` als zweitem Argument. Aus diesem Grund ist es unter Umständen auch erforderlich, die Argumente zu klammern. Im Zweifel ist es immer besser eine Klammer zuviel als keine Klammer zu setzen. Analog wird nach einer Eingabe wieder ein `cin` zurückgegeben, so daß man eine weitere Eingabe anfügen kann usw..

2.8.1 Elementfunktionen der iostreams

Anstatt die obige, bisher gebrauchte Formulierung der Ein- und Ausgabeoperationen zu verwenden, kann man für besondere Aufgaben auf Funktionen zurückgreifen, die in Tabelle 2.3 gezeigt sind.

```
   char c1='x';
   cout << c1 << '\n';    // Bisher verwendete Schreibweise
   cout.put( c1 );        // Kann ersetzt werden durch Funktionsaufrufe
   cout.put( '\n' );
```

Einige der Elementfunktionen erlauben Zugriffe, die in der bisherigen Formulierung schwieriger zu realisieren sind.

`ostream::put (char c);`	gibt einen character 'c' aus.
`char c='x';` `cout << c;`	Zum Vergleich ... schreibt den character 'c'
`istream::get (char c);`	liest einen character 'c' vom Eingabe-Kanal.
`char c;` `cin >> c;`	Zum Vergleich ... liest einen character 'c' ein, ignoriert aber SPACE, TAB, EOL.
`istream::get (char*pc, int anzahl, char terminator);`	liest maximal 'anzahl' character von `istream` oder endet, wenn der 'terminator'-character eingelesen wurde. Voreinstellungen: `anzahl=1`; `terminator='\n'`. Der 'terminator' wird im Kanal belassen.
`istream::getline (char*pc, int anzahl, char terminator);`	wie `istream::get`, aber der 'terminator'-character wird gelöscht.
`istream::putback (char c);`	legt den vorher mit `istream::get` aus dem Eingabe-Kanal gelesenen character `c` wieder zurück.
`istream::peek (char c);`	liest den character 'c' von `cin`, beläßt ihn aber darin.

Tabelle 2.3: Prototypen einiger Elementfunktionen der Ein-/Ausgabe Kanal-Typen `ostream` und `istream`. Die Kanal-Typen werden bei der Anwendung im Programm z.B. durch `cout` und `cin` ersetzt.

2.8.2 Formatierung

Oft ist es nötig das Format der Ausgabe zu ändern, um z.B. eine schönere, besser lesbare Tabelle zu erhalten, oder um die volle Genauigkeit einer Variablen auf dem Bildschirm auszugeben. Solche Operationen geschehen mit Hilfe der in Tabelle 2.4 zusammengefaßten Funktionen.

Dabei ist zu beachten, daß die Änderung, die durch solche Funktionen herbeigeführt wird, evtl. nur für die nächste Ausgabe gültig ist. Um diese Funktionen verwenden zu können, muß man zuerst `#include<iomanip>` aufrufen.

```
#include<iomanip>
   double x=1234.5;
   cout << x << '\n';   // Ergibt: 1234.5
   cout << setfill ('*') << setw (10) << setprecision (5);
   cout << x << '\n';   // Ergibt: ****1234.5
   cout << x << '\n';   // Ergibt wieder: 1234.5
```

Manipulator	Beschreibung
`int width=12;` `cout << setw(width);`	setzt die minimale Größe der direkt folgenden Ausgabe.
`int prec=5;` `cout << setprecision(prec);`	Gibt die Zahl der Stellen für Festkomma-, oder die Zahl der Stellen hinter dem Komma für Gleitkommadarstellung an; sonst die Maximalzahl der Stellen.
`char c='*';` `cout << setfill(c);`	Definiert ein Füllzeichen wie z.B. `*`.

Tabelle 2.4: Beispiele von Formatierungsfunktionen für die Ausgabe.

Weiterhin gibt es sog. Flags zum Schalten von Formatfunktionen. Dies sind Operationen, die keinen Parameter brauchen. Beispiele sind in Tabelle 2.5 gezeigt.

Flag	Gruppe	Funktion
left	adjustfield	links ausrichten
right	adjustfield	rechts ausrichten
internal	adjustfield	+/- links, Zahl rechts
dec	basefield	Dezimalzahlen
hex	basefield	Hexadezimalzahlen
oct	basefield	Oktalzahlen
showbase		zeigt Dezimalbasis von hex- und oct-Zahlen
showpos		'+' vor Zahl, wenn positiv
uppercase		E, X, A-F statt e, x, a-f
fixed	floatfield	Festkommazahl
scientific	floatfield	wissenschaftliche Darstellung

Tabelle 2.5: Schalter für formatierte Ausgabe.

Will man ein Flag setzen, d.h. einen Ausgabemodus aktivieren, so verwendet man alternativ:

```
cout.setf (ios:: 'flag', ios:: 'group')
cout << setiosflags (ios:: 'flag', ios:: 'group')
```

wobei die Möglichkeiten für `'flag'` und `'group'` Tabelle 2.5 zu entnehmen sind. Der zweite Übergabeparameter ist nicht unbedingt notwendig. Will man die Funktionalität rückgängig machen, bzw. die Voreinstellung wiederherstellen, so kann man dies (wieder alternativ) für jedes einzelne Flag, oder auch für eine ganze Gruppe tun.

```
cout.unsetf (ios:: 'flag')
cout << resetiosflags (ios:: 'flag')

cout.unsetf (ios:: 'group')
cout << resetiosflags (ios:: 'group')
```

2.8.3 Dateien (Files)

Zum Lesen (von) und Schreiben (auf) Dateien gibt es die beiden Kanal-Typen `ifstream` und `ofstream`. Da diese beiden Kanal-Typen von den Standardstreams abgeleitet sind existieren für sie alle Elementfunktionen der Standardstreams.

```
# include <fstream>

   double  z=1.234;
   int     m=5;
   ofstream outs ( "filename", ios::out );
                        // open the file with name
                        // 'filename' for writing
                        // ios::out can be skipped
   outs << x << '\n';   // the use of the self-defined output-stream
   outs << j << '\n';   // 'outs' is analogous to the use of 'cout'
   outs.close();        // close the out-fstream 'outs'

   ifstream ins  ( "filename", ios::in );
                        // open the file with name
                        // 'filename' to read from
                        // ios::in can be skipped
   ins >> z;            // the use of the self-defined input-stream
   ins >> m;            // 'ins' is analogous to the use of 'cin'
   ins.close();         // close the in-fstream 'ins'
```

Spätestens bei der Bearbeitung von Dateien benötigt man Informationen darüber, ob man am Ende des Files (EOF) angelangt ist, oder ob das Öffnen erfolgreich war. Dafür gibt es die sogenannten Statusinformationen, die je nach Zustand der Datei `true` oder `false` zurückgeben:

```
ins.good()   // true, if no error occured in stream 'ins'
ins.eof()    // true, if the end of file/stream is reached
ins.bad()    // true, if an hardware error occured
ins.fail()   // true, if a logical error or a
```

```
             //         hardware error occured

ins.clear()  // set ins.good() to true
ins.clear( ios::failbit | ins.rdstate() )
             // set fail() manually to true
```

Wie liest man nun aus einer Datei bis zu deren Ende? Die folgenden Beispiele sollen die Schwierigkeiten aufzeigen, auf die man bei der Implementierung dieser einfachen Aufgabe stoßen kann.

```
int n;

while (!cin.eof()){
   cin >> n;
   // .... this reads too far
}

while((cin >> n).good){
   // ... this reads not far enough
}

while(cin >> n){
  // ... this is the way it works
}
```

2.9 Dynamische Speicherverwaltung

Zum Anfordern von Speicher dienen in C++ die Operatoren `new` und `new []` für einzelne Datenelemente oder Felder. Das Freigeben des Speichers erfolgt mit `delete` und `delete []`.

```
   int    *p_i  = new int;
   Person *p_p  = new Person("B.Stroustrup",45);
   int        n = 40;
   int    *pa_i = new int[n];
// ...
   delete   p_i;
   delete   p_p;
   delete[] pa_i;
```

Die Anforderung geschieht durch Angabe des Typs oder des Klassennamens nach `new`. In diesem Fall wird ein Datenelement der Klasse oder des Typs im Speicher mit Hilfe des zugehörigen Defaultkonstruktors angelegt und der Operator gibt einen Zeiger auf dieses Element zurück. Um einen anderen als den Defaultkonstruktor zu verwenden, benutzt man nach dem `new` einfach den entsprechenden Konstruktoraufruf, wie oben für die Klasse Person gezeigt.

Für Felder wird `new[]` verwendet. Hier wird Speicherplatz für die Anzahl von Speicherelementen angefordert, die innerhalb der eckigen Klammern angegeben ist. Für jedes Element wird der Defaultkonstruktor aufgerufen. Der Operator `new[]` gibt einen Zeiger auf das erste Datenelement zurück. Zum Zugriff auf die Elemente jenseits des ersten verwendet man pointer-Arithmetik oder subscripting mit `[]`.

```
Person  *p = new Person[100];
p[0] = Person("Matthias",30);
p[1] = Person("Stefan",32);
```

Der Operator `new` wirft die exception `bad_alloc`, falls die Speicheranforderung fehlschlägt. Sollte diese unbehandelt bleiben, schreibt das Programm ein core file und bricht ab.

2.10 Organisation in Implementierungs- und Header-Dateien

Ein großes Programm in einer einzigen Datei zu verwalten ist schwierig! Dies führt zu langen Übersetzungszeiten, unübersichtlicher Anordnung der verschiedenen Teile und zu extremen Problemen, die Datei konsistent zu halten, wenn mehrere Personen an diesem Projekt arbeiten.

Man trennt Programme daher meist in mehrere Dateien auf, die getrennt voneinander compiliert werden. Übersetzbare (Implementierungs-) C++-Dateien erhalten den Suffix `.cc` (aber auch andere Endungen sind üblich). Dateien, die nur Deklarationen enthalten, heissen auch header-Dateien und dienen dazu, Funktionen und Daten aus einem Programmteil im anderen bekanntzumachen. Diese Dateien haben den Suffix `.h`. In der Regel (Ausnahme z.B. `main.cc`) gehört zu jeder Implementierungsdatei auch eine Header-Datei.

Header-Dateien sollten enthalten:

1. Sog. include-guards, die das mehrfache Einschließen von Header-Dateien verhindern (vom Sprachstandard verlangt)

```
#ifndef HEADER_H
#define HEADER_H
  // declarations
#endif
```

2. Deklarationen von Funktionen, die modulübergreifend verwendet werden sollen. Zu Deklaration von Funktionen, die nur jeweiligen Programmteil Verwendung finden sollen, können weitere Header-Dateien verwendet werden;
3. Definition von `inline` Funktionen
4. Deklaration von Klassen und deren Elementfunktionen
5. `extern` Deklarationen globaler Variablen, Deklaration statischer Klassenvariablen.
6. Deklaration und Definition von `template` - Funktionen und Klassen.

Implementierungsdateien sollten enthalten:

1. Implementierung von Elementfunktionen von Nicht-Template-Klassen,
2. Implementierung von regulären Funktionen.
3. Deklaration von file-scope, class-scope und globalen statischen Variablen.

2.11 Weiterführende Literatur

Die Vielzahl von unbekannten Begriffen in den beiden letzten Abschnitten weist darauf hin, das die Programmiersprache C++ noch viele Möglichkeiten für den interessierten Programmierer bereithält. Im begrenzten Rahmen dieser Einführung ist es leider nicht möglich auf höhere Sprachelemente wie Klassen, Templates, Überladung, Scopes etc. einzugehen. Wenigstens soll hier jedoch auf weiterführende Literatur hingewiesen werden.

1. Brian W. Kernighan and Dennis M. Ritchie, *The C-Programming Language*, 2nd edition, Prentice Hall, 1988.
2. B. Stroustrup, *The C++ Programming Language*, 3rd edition, Addison Wesley, 1997.
3. C. S. Horstmann, *Mastering C++*, John Wiley & Sons, New York, 1996.
4. S. B. Lippman, *C++ Einführung und Leitfaden*, 3. Auflage, Addison-Wesley, Bonn, 1998.