Deutsch

Fujitsu Software BS2000

C++-Bibliotheksfunktionen

Benutzerhandbuch

Stand der Beschreibung
C++ V4.0

Ausgabe Juni 2023

Kritik… Anregungen… Korrekturen…

Die Redaktion ist interessiert an Ihren Kommentaren zu diesem Handbuch. Ihre Rückmeldungen helfen uns, die Dokumentation zu optimieren und auf Ihre Wünsche und Bedürfnisse abzustimmen.

Sie können uns Ihre Kommentare per E-Mail an bs2000.info@fujitsu.com senden.

Zertifizierte Dokumentation nach DIN EN ISO 9001:2015

Um eine gleichbleibend hohe Qualität und Anwenderfreundlichkeit zu gewährleisten, wurde diese Dokumentation nach den Vorgaben eines Qualitätsmanagementsystems erstellt, welches die Forderungen der DIN EN ISO 9001: 2015 erfüllt.

Copyright und Handelsmarken

Inhaltsverzeichnis

C++-Bibliotheksfunktionen

1 Einleitung

Dieses Handbuch beschreibt alle Klassen, Funktionen und Operatoren, die die C++-Standardbibliothek (C++ V2.1) im Betriebssystem BS2000 für komplexe Mathematik und stromorientierte Ein-/Ausgabe zur Verfügung stellt. Die C++-Standardbibliothek ist Bestandteil des Common Runtime Environment (CRTE).

Voraussetzung für die Arbeit mit diesem Handbuch sind Kenntnisse der Programmiersprachen C und C++ sowie des Betriebssystems BS2000.

Das Kapitel "Benutzung der C++-Bibliotheksfunktionen" enthält allgemeine Informationen über die C++-Bibliothek SYSLNK.CRTE.CPP und den Zusammenhang zwischen der C++-Bibliothek und dem C-Laufzeitsystem.

Die detaillierte Beschreibung der C++-Bibliotheksfunktionen finden Sie in den Kapiteln "Klassen und Funktionen für komplexe Mathematik" und "Klassen und Funktionen für die Ein-/Ausgabe".
Diese Kapitel sind in der für die SINIX-C++-Bibliotheken bereits üblichen Art aufgebaut. Es werden z.B. die gleichen Abschnitts-Überschriften verwendet (entsprechen in SINIX-C++ den Abschnittsnamen der "Manual Pages").

Literaturhinweise werden im Text in Kurztiteln angegeben. Im Literaturverzeichnis ist der vollständige Titel jeder Druckschrift aufgeführt. Daran anschließend finden Sie Hinweise zur Bestellung von Handbüchern.

Auf folgende wichtige Druckschriften sei an dieser Stelle hingewiesen:

Das "C++-Benutzerhandbuch" beschreibt, wie mit dem C++(BS2000)-Compiler V2.1
und weiteren Komponenten des Betriebssystems BS2000 ein C++-Programm übersetzt,gebunden und gestartet wird. Dazu zählt auch die Benutzung der CRTE-Bibliotheken beim Übersetzen und Binden eines Programms.

Das Handbuch "C-Bibliotheksfunktionen" beschreibt alle C-Funktionen und Makros, died as C-Laufzeitsystem zur Verfügung stellt. Außerdem sind in diesem Handbuch alle Informationen zur Dateiverarbeitung, Pufferung etc. enthalten, die auch für die C++ Ein-/Ausgabe relevant sind (die C++-Ein-/Ausgabe wird intern über das C-Laufzeitsystem realisiert, siehe auch Zusammenhang zwischen den C++-Bibliotheken und dem C-Laufzeitsystem).

Das "CRTE-Benutzerhandbuch" enthält Informationen zum Konzept und zur Handhabung des Common Runtime Environment, das u.a. die Laufzeitsysteme für C, C++, COBOL85 und ILCS beinhaltet.

"Die C++ Programmiersprache", 2. überarbeitete Auflage von Bjarne Stroustrup beschreibt den C++-Sprachumfang des C++(BS2000)-Compilers V2.1.

2 Benutzung der C++-Bibliotheksfunktionen

In diesem Kapitel werden folgende Themen behandelt:

- Die CRTE-Bibliothek SYSLNK.CRTE.CPP
- Zusammenhang zwischen den C++-Bibliotheken und dem C-Laufzeitsystem

2.1 Die CRTE-Bibliothek SYSLNK.CRTE.CPP

Die mit CRTE ab V1.0 ausgelieferte Bibliothek SYSLNK.CRTE.CPP enthält folgende Komponenten:

- Standard-Include-Elemente für die C++-Bibliotheksfunktionen (Typ S)

complex.h	stdiostream.h
iomanip.h	generic.h
fstream.h	new.h
iostream.h	strstream.h

  Diese Include-Elemente werden bei der Übersetzung aufgrund der Präprozessoranweisung #include in das Programm kopiert. Wie dies geschieht, ist ausführlich im C++-Benutzerhandbuch dargestellt.

- Moduln der C++-Bibliotheksfunktionen (LLMs, Typ L)

  Diese Moduln enthalten den Code aller C++-Bibliotheksfunktionen für komplexe Mathematik und Standard-Ein-/Ausgabe.

  Die Moduln können entweder mit dem BINDER fest zum C++-Programm gebunden werden oder dynamisch mit dem DBL. Weitere Einzelheiten hierzu finden Sie ebenfalls im C++-Benutzerhandbuch.

2.2 Zusammenhang zwischen den C++-Bibliotheken und dem C-Laufzeitsystem

Das in der CRTE-Bibliothek SYSLNK.CRTE enthaltene C-Laufzeitsystem ist unbedingte Voraussetzung für die Benutzung der C++-Bibliotheksfunktionen. Sowohl beim Übersetzen als auch beim Binden eines C++-Programms, das die C++-Bibliotheksfunktionen benutzt, muss die Bibliothek mit dem C-Laufzeitsystem angegeben werden (siehe Benutzerhandbuch C/C++-Compiler).

Die eigentliche Realisierung z.B. der C++-Standard-Ein-/Ausgabefunktionen findet durch den internen Aufruf diverser Ein-/Ausgaberoutinen des C-Laufzeitsystems statt.

Bis auf die satzorientierte Ein-/Ausgabe sind mit den C++-Ein-/Ausgabefunktionen alle Arten von Dateizugriffen möglich, die auch mit den C-Bibliotheksfunktionen möglich
sind.
Sofern es im C-Laufzeitsystem Unterschiede gibt zwischen "KR"- oder "ANSI"-Funktionalität, gilt bei der Ausführung der C++-Bibliotheksfunktionen generell die "ANSI"-Funktionalität.

Ausführliche Informationen zur Dateiverarbeitung, ANSI-Funktionalität, Pufferung etc. entnehmen Sie bitte dem Handbuch "C-Bibliotheksfunktionen".

3 Klassen und Funktionen für komplexe Mathematik

In diesem Kapitel werden folgende Themen behandelt:

- cplxintro Einführung in die komplexe Mathematik in C++
- cplxcartpol Kartesische/Polare Funktionen
- cplxerr Fehlerbehandelnde Funktionen
- cplxexp Transzendente Funktionen
- cplxops Operatoren
- cplxtrig Trigonometrische und hyperbolische Funktionen

3.1 cplxintro Einführung in die komplexe Mathematik in C++

Dieser Abschnitt gibt einen Überblick über die Klassen, Funktionen und Operatoren, diedie C++-Bibliothek für die komplexe Mathematik zur Verfügung stellt.

```
#include <complex.h>
class complex;
```

Die Deklarationen aller Operatoren und Funktionen für komplexe Mathematik finden Siein der Include-Datei <complex.h>.

Der Datentyp für komplexe Zahlen ist als Klasse *complex* implementiert.

Für die Bearbeitung von komplexen Zahlen stehen Overloading-Versionen für folgendeOperatoren und mathematische Funktionen zur Verfügung:

- Standardoperatoren zur Ein-/Ausgabe sowie arithmetische, Zuweisungs- und Vergleichsoperatoren; siehe Abschnitt "cplxops Operatoren"
- Standardfunktionen zur Exponentialberechnung, Logarithmusberechnung, Potenzbildung und Quadratwurzel-Berechnung; siehe Abschnitt "cplxexp Transzendente Funktionen"
- trigonometrische Funktionen (Sinus, Kosinus, hyperbolischer Sinus und hyperbolischer Kosinus); siehe Abschnitt "cplxtrig Trigonometrische und hyperbolische Funktionen"

Im Abschnitt "cplxcartpol Kartesische/Polare Funktionen" werden u.a. die Routinen zur Konvertierung zwischen kartesischen und polaren Koordinatensystemen erläutert.
Die Beschreibung der Fehlerbehandlung findet sich im Abschnitt "cplxerr Fehlerbehandelnde Funktionen".

3.2 cplxcartpol Kartesische/Polare Funktionen

In diesem Abschnitt werden u.a. die kartesischen und polaren Funktionen der Klasse*complex* beschrieben.

```
#include <complex.h>

class complex

{

public:

    friend double      abs(complex);

    friend double      arg(complex);

    friend complex     conj(complex);

    friend double      imag(const complex&);

    friend double      norm(complex);

    friend complex     polar(double, double = 0.0);

    friend double      real(const complex&);

/* weitere Deklarationen */

};
```

double d = abs(complex x)

Der Absolutwert oder die Größe von *x* wird geliefert.

double d = norm(complex x)

Das Quadrat der Größe von *x* wird geliefert. Die Funktion dient dem Vergleich vonGrößenwerten. Die Funktion *norm()* ist schneller als *abs()*. Allerdings ist bei *norm()* ein Überlauffehler wahrscheinlicher, da das Quadrat der Größe geliefert wird.

double d = arg(complex x)

Der Winkel von *x* in Radiant wird geliefert. Der Ergebniswert liegt im Bereich -pi bis+pi.

complex y = conj(complex x)

Der konjugierte Wert von *x* wird geliefert. Wenn *x* in der Form *(Realteil, Imaginär- teil)* angegeben wird, dann ist *conj(x)* mit *(Realteil, -Imaginärteil)* identisch.

complex y = polar(double m, double a=0.0)

Es wird ein Wert vom Typ *complex* geliefert. Die Funktion erwartet ein Wertepaar polarer Koordinaten: die Größe *m* und den Winkel *a* in Radiant.

double d = real(complex &x)

Der Realteil von *x* wird geliefert.

double d = imag(complex &x)

Der Imaginärteil von *x* wird geliefert.

BEISPIEL Das folgende Programm konvertiert eine komplexe Zahl ins Polarkoordinatensystem und gibt die konvertierte Zahl aus:

```
#include <iostream.h>
#include <complex.h>
 main ()
 {
   complex d;
   d = polar (10.0, 0.7);
   cout <<real(d)<<" "<<imag(d);
   cout <<"\n";
   return 0;
 }
```

Das Ergebnis der Programmausführung ist:

```
7.64842 6.44218
%  CCM0998 Verbrauchte CPU-Zeit: 0.0006 Sekunden
```

SIEHE AUCH

cplxerr , *cplxexp* , *cplxops* , *cplxtrig*

3.3 cplxerr Fehlerbehandelnde Funktionen

In diesem Abschnitt wird die fehlerbehandelnde Funktion erläutert, die für die komplexe Mathematik in C++ eingesetzt wird.

#include <complex.h>

class c_exception

{

int type;
char *name;
complex arg1;
complex arg2;
complex retval;

public:

c_exception(char *n, const complex& a1, const complex& a2 = complex_zero);

friend int complex_error(c_exception&);

friend complex exp(complex);
friend complex sinh(complex);
friend complex cosh(complex);
friend complex log(complex);

};

---

int i = complex_error(c_exception & x)

Die fehlerbehandelnde Funktion *complex_error* wird aufgerufen, wenn ein Fehler beieiner der vier folgenden Funktionen auftritt:

```
friend complex exp(complex)
friend complex sinh(complex)
friend complex cosh(complex)
friend complex log(complex)
```

Benutzer können eigene Prozeduren zur Fehlerbehandlung definieren, indem eine Funktion mit dem Namen *complex_error* in das Programm aufgenommen wird. *complex_error* muß die oben beschriebene Form aufweisen.

In der Klasse *c_exception* ist das Element *type* eine Ganzzahl, die den Typ des aufgetretenen Fehlers beschreibt. Der Wert muß in der folgenden Liste von Konstanten enthalten sein (die Definition finden Sie in der Include-Datei <*complex.h*>):

SING	Singularität des Arguments
OVERFLOW	Überlauffehler
UNDERFLOW	Unterlauffehler

Das Element *name* zeigt auf eine Zeichenkette, die den Namen der Funktion enthält, inder der Fehler aufgetreten ist. Die Variablen *arg1* und *arg2* sind die Argumente, mitdenen die Funktion aufgerufen wurde. Der Standardrückgabewert der Funktion wird an*retval* zugewiesen, sofern der Wert nicht durch die benutzereigene *complex_error*-Funk-tion verändert wird.

Wenn die benutzerdefinierte Funktion *complex_error* einen Wert ungleich 0 liefert, wirdkeine Fehlermeldung ausgegeben, und *errno* wird nicht gesetzt.

Wenn keine benutzereigene Funktion mit dem Namen *complex_error* bereitsteht, werden beim Auftreten eines Fehlers die Standardprozeduren für die Fehlerbehandlung aufgerufen. Diese werden bei den jeweiligen Funktionen unter "FEHLERERGEBNISSE"beschrieben. Eine Zusammenfassung der Standardfehlerbehandlung finden Sie in der folgendenTabelle. In jedem Fall wird *errno* auf EDOM oder ERANGE gesetzt und das Programmfortgeführt.

In der folgenden Tabelle bedeuten:

M	Meldung wird ausgegeben (EDOM-Fehler)
(H, 0)	(HUGE, 0) wird geliefert
(±H, ±H)	(±HUGE, ±HUGE) wird geliefert
(0, 0)	(0, 0) wird geliefert

STANDARDPROZEDUREN FÜR DIE FEHLERBEHANDLUNG

	Fehlerarten		
type	SING	OVERFLOW	UNDERFLOW
errno	EDOM	ERANGE	ERANGE
EXP: Realteil zu groß oder klein Imaginärteil zu groß		(±H, ±H) (0, 0)	(0,0)
LOG: arg = (0, 0)	M, (H, 0)	-	-
SINH: Realteil zu groß Imaginärteil zu groß		(±H, ±H) (0, 0)	
COSH: Realteil zu groß Imaginärteil zu groß		(±H, ±H) (0, 0)	

BEISPIEL Das folgende Programm legt eine komplexe Zahl unter Verwendung des Standardkonstruktors an, der das Wertepaar (0.0, 0.0) vorgibt, und ruft dann die Funktion *log()* mit (0.0, 0.0) auf. Diese Operation ruft einen Fehler hervor, da *log(0.0, 0.0)* nicht definiert ist. Die Funktion *complex_error()* wird wegen dieses Fehlers aufgerufen.

```
#include <iostream.h>
#include <complex.h>
#include <stdlib.h>
int complex_error(c_exception & p)
{
  cerr << "Fehler bei der Verarbeitung von ";
  cerr << p.name << " ( " << p.arg1 << " )\n";
  exit (1);
  return 0; /* WIRD NICHT ERREICHT */
}
main()
{
  complex c;
  c = log (c);
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
Fehler bei der Verarbeitung von log ( ( 0, 0) )
%  CCM0998 Verbrauchte CPU-Zeit: 0.0005 Sekunden
%  CCM0999 exit 1
```

SIEHE AUCH

cplxcartpol, *cplxexp*, *cplxops*, *cplxtrig*

3.4 cplxexp Transzendente Funktionen

In diesem Abschnitt werden die transzendenten Funktionen der Klasse *complex* beschrieben.

```
#include <complex.h>

class complex
{

public:

   friend complex exp(complex);
   friend complex log(complex);
   friend complex pow(double, complex);
   friend complex pow(complex, int);
   friend complex pow(complex, double);
   friend complex pow(complex, complex);
   friend complex sqrt(complex);

};
```

---

complex z = exp(complex x)

Es wird e^{x} geliefert.

complex z = log(complex x)

Der natürliche Logarithmus von *x* wird geliefert.

complex z = pow(complex x, complex y)

Es wird x^{y} geliefert.

complex z = sqrt(complex x)

Die Quadratwurzel von *x*, die im ersten oder vierten Quadranten der komplexenEbene enthalten ist, wird geliefert.

FEHLERERGEBNISSE

exp liefert (0.0, 0.0), wenn der Realteil von *x* so klein oder der Imaginärteil so groß ist, daß es zu einem Überlauf kommt. Ist der Realteil groß genug, um einen Überlauf hervorzurufen, liefert *exp*:

- (HUGE, HUGE), wenn der Kosinus und Sinus des Imaginärteils von *x* > 0 sind;
- (HUGE, -HUGE), wenn der Kosinus > 0 ist und der Sinus <= 0
- (-HUGE, HUGE), wenn der Sinus > 0 ist und der Kosinus <= 0
- (-HUGE, -HUGE), wenn der Sinus und der Kosinus <= 0 sind.

In allen diesen Fällen wird *errno* auf ERANGE gesetzt.

Die Funktion *log()* liefert (-HUGE, 0.0) und setzt *errno* auf EDOM, wenn *x* (0.0, 0.0) ist. Auf der Standardfehlerausgabe wird eine Meldung angezeigt, die auf einen SING-Fehler hinweist.

Die fehlerbehandelnden Prozeduren können durch die Funktion *complex_error()* (siehe "cplxerr Fehlerbehandelnde Funktionen") verändert werden.

BEISPIEL

Das folgende Programm gibt eine Folge komplexer Zahlen sowie deren Exponentialpotenzen aus.

```
#include <iostream.h>
#include <complex.h>
main()
{
  complex c;
  for (c = complex(1.0,1.0); real(c) < 4.0; c += complex(1.0,1.0))
  {
    cout <<c<<" "<<exp(c)<<"\n";
  }
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
( 1, 1) ( 1.46869, 2.28736)
( 2, 2) ( -3.07493, 6.71885)
( 3, 3) ( -19.8845, 2.83447)
%  CCM0998 Verbrauchte CPU-Zeit: 0.0012 Sekunden
```

Komplexe Zahlen können mit dem Operator << ausgegeben werden. Die Bearbeitung komplexer Zahlen ist genauso einfach wie die der Zahlen vom Typ *float* oder *double*.

SIEHE AUCH

cplxcartpol, *cplxerr*, *cplxops*, *cplxtrig*

3.5 cplxops Operatoren

In diesem Abschnitt werden die grundlegenden Ein-/Ausgabe-, arithmetischen, Vergleichs- und Zuweisungsoperatoren beschrieben.

```
#include <complex.h>

class complex

{

public:

    friend complex    operator+(complex, complex);
    friend complex    operator-(complex);
    friend complex    operator-(complex, complex);
    friend complex    operator*(complex, complex);
    friend complex    operator/(complex, complex);

    friend int        operator==(complex, complex);
    friend int        operator!=(complex, complex);

    void              operator+=(complex);
    void              operator-=(complex);
    void              operator*=(complex);
    void              operator/=(complex);

};

 ostream&   operator<<(ostream&, complex);
 istream&   operator>>(istream&, complex&);
```

Die Operatoren weisen den üblichen Vorrang auf. Für die folgende Beschreibung derOperatoren sind

- *x, y* und *z* Variablen der Klasse *complex*.

Arithmetische Operatoren

z = x + y

Es wird ein *complex*-Objekt geliefert, das die arithmetische Summe der komplexenZahlen *x* und *y* darstellt.

z = -x

Es wird ein *complex*-Objekt geliefert, das die arithmetische Negation der komplexenZahl *x* darstellt.

z = x - y

Es wird ein *complex*-Objekt geliefert, das die arithmetische Differenz der komplexenZahlen *x* und *y* darstellt.

z = x * y

Es wird ein *complex*-Objekt geliefert, das das arithmetische Produkt der komplexenZahlen *x* und *y* darstellt.

z = x / y

Es wird ein *complex*-Objekt geliefert, das den arithmetischen Quotienten der komplexen Zahlen *x* und *y* darstellt.

Vergleichsoperatoren

x == y

Wenn die komplexe Zahl *x* gleich der komplexen Zahl *y* ist, wird eine Ganzzahl ungleich 0 geliefert. Im anderen Fall wird der Wert 0 geliefert.

x != y

Wenn die komplexe Zahl *x* ungleich der komplexen Zahl *y* ist, wird eine Ganzzahlungleich 0 geliefert. Im anderen Fall wird der Wert 0 geliefert.

Zuweisungsoperatoren

x += y

Zur komplexen Zahl *x* wird die komplexe Zahl *y* addiert und die arithmetische Summean *x* zugewiesen.

x -= y

Von der komplexen Zahl x wird die komplexe Zahl *y* abgezogen und die arithmetische Differenz an *x* zugewiesen.

x *= y

Die komplexe Zahl *x* wird mit der komplexen Zahl *y* multipliziert und das arithmetische Produkt an *x* zugewiesen.

x /= y

Von den komplexen Zahlen *x* und *y* wird der arithmetische Quotient gebildet und an *x* zugewiesen.

Achtung

Die Zuweisungsoperatoren erzeugen keine Werte, die in Ausdrücken verwendet werden können. Das folgende Konstrukt ist daher syntaktisch ungültig:

```
complex x, y, z;
x = (y += z);
```

Die Zeilen

```
x = (y + z);
x = (y == z);
```

sind hingegen gültig.

Ein-/Ausgabe-Operatoren

Komplexe Zahlen können mit dem Operator << ausgegeben und mit dem Operator >> eingelesen werden.

Ausgabeformat:

(Realteil, Imaginärteil)

Eingabeformat:

```
Eingabe            entspricht         komplexer Zahl
-----------------------------------------------------
Zahl                    |             (Zahl, 0)
(Zahl)                  |             (Zahl, 0)
(Zahl1, Zahl2)          |             (Zahl1, Zahl2)
```

BEISPIEL Das folgende Programm legt die komplexen Zahlen *c* und *d* an, dividiert *d* durch *c* und gibt dann die Werte von *c* und *d* aus:

```
#include <iostream.h>
#include <complex.h>
main()
{
  complex c,d;
  d = complex(10.0, 11.0);
  c = complex (2.0, 2.0);
  while (norm(c) < norm(d))
  {
    d /= c;
    cout << c << " " <<d << "\n";
  }
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
( 2, 2)  ( 5.25, 0.25)
( 2, 2)  ( 1.375, -1.25)
%  CCM0998 Verbrauchte CPU-Zeit: 0.0009 Sekunden
```

SIEHE AUCH

cplxcartpol, *cplxerr*, *cplxexp*, *cplxtrig*

3.6 cplxtrig Trigonometrische und hyperbolische Funktionen

In diesem Abschnitt werden die trigonometrischen und hyperbolischen Funktionen fürden Datentyp *complex* beschrieben.

```
#include <complex.h>

class complex
{

public:

   friend complex sin(complex);
   friend complex cos(complex);

   friend complex sinh(complex);
   friend complex cosh(complex);

};
```

---

complex y = sin(complex x)

Der Sinus von *x* wird geliefert.

complex y = cos(complex x)

Der Kosinus von *x* wird geliefert.

complex y = sinh(complex x)

Der hyperbolische Sinus von *x* wird geliefert.

complex y = cosh(complex x)

Der hyperbolische Kosinus von *x* wird geliefert.

FEHLERERGEBNISSE

Wenn der Imaginärteil von *x* einen Überlauf verursacht, liefern *sinh* und *cosh* das Ergebnis (0.0, 0.0). Wenn der Realteil groß genug ist, um zu einem Überlauf zu führen, gebendie Funktionen *sinh* und *cosh* folgende Ergebnisse zurück:

- (HUGE, HUGE), wenn der Kosinus und Sinus des Imaginärteils >= 0 sind;
- (HUGE, -HUGE), wenn der Kosinus >= 0 ist und der Sinus < 0;
- (-HUGE, HUGE), wenn der Sinus >= 0 ist und der Kosinus < 0;
- (-HUGE, -HUGE), wenn sowohl der Sinus als auch der Kosinus < 0 sind.

In allen diesen Fällen wird *errno* auf ERANGE gesetzt. Die Fehlerbehandlungsprozeduren können durch die Funktion *complex_error()* verändert werden (siehe *cplxerr*).

BEISPIEL Das folgende Programm gibt einen Bereich komplexer Zahlen und die zugehörigen Werte aus, die durch die Funktion *cosh()* berechnet werden:

```
#include <iostream.h>
#include <complex.h>
main()
{
  complex c;
  while (norm(c) < 10.0)
  {
    cout << c <<" " <<cosh(c) << "\n";
    c += complex(1.0, 1.0);
  }
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
( 0, 0)  ( 1, 0)
( 1, 1)  ( 0.83373, 0.988898)
( 2, 2)  ( -1.56563, 3.29789)
%  CCM0998 Verbrauchte CPU-Zeit: 0.0017 Sekunden
```

Hinweis

Das Ergebnis der Funktion *cosh()* ist ein *complex*-Objekt.

Die Konstanten des Typs *double* (beispielsweise 10.0, 1.0 usw.) werden zur Erstellung komplexer Zahlen verwendet.

SIEHE AUCH

cplxcartpol, *cplxerr*, *cplxexp*, *cplxops*

4 Klassen und Funktionen für die Ein-/Ausgabe

In diesem Kapitel werden folgende Themen behandelt:

- iosintro Einführung in das Puffern, die Formatierung und die Ein-/Ausgabe
- filebuf Puffer für die Datei-Ein-/Ausgabe
- fstream Spezialisierung von iostream und streambuf auf Dateien
- ios Basisklasse für die Ein-/Ausgabe
- istream Formatierte und unformatierte Eingabe
- manip iostream-Manipulation
- ostream Formatierte und unformatierte Ausgabe
- sbufprot Geschützte Schnittstelle der Zeichenpuffer-Klasse
- sbufpub Öffentliche Schnittstelle der Zeichenpuffer-Klasse
- sstreambuf Spezialisierung von streambuf auf Felder
- stdiobuf Spezialisierung von iostream auf stdio-FILE-Objekte
- strstream Spezialisierung von iostream auf Felder

4.1 iosintro Einführung in das Puffern, die Formatierung und die Ein-/Ausgabe

In diesem Abschnitt werden die Mechanismen in C++ beschrieben, die der Benutzer zur Implementierung der Ein-/Ausgabe einsetzen kann. Die Standardbibliothek für Einund Ausgaben wurde in C++ entwickelt und verdeutlicht die Leistungsstärke dieser Programmiersprache.

Das *iostream*-Paket von C++ beinhaltet hauptsächlich eine Zusammenstellung von Klassen, die in folgenden Include-Dateien deklariert sind:
<iostream.h>, <fstream.h>, <strstream.h>, <stdiostream.h>, <iomanip.h>.
Obwohl ursprünglich nur die Ein- und Ausgabe unterstützt werden sollte, umfaßt das Paket nun auch verwandte Leistungen wie die Bearbeitung von Byte-Feldern (Zeichenketten).

```
#include <iostream.h>

typedef long streampos, streamoff;

class streambuf;
class ios;
class istream : virtual public ios;
class ostream : virtual public ios;
class iostream : public istream, public ostream;
class istream_withassign : public istream;
class ostream_withassign : public ostream;
class iostream_withassign : public iostream;

extern istream_withassign cin;
extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;

#include <fstream.h>
class filebuf : public streambuf;
class fstreambase : virtual public ios;
class fstream : public fstreambase, public iostream;
class ifstream : public fstreambase, public istream;
class ofstream : public fstreambase, public ostream;

#include <strstream.h>
class strstreambuf : public streambuf;
class strstreambase : virtual public ios;
class istrstream : public strstreambase, public istream;
class ostrstream : public strstreambase, public ostream;
class strstream : public strstreambase, public iostream;

#include <stdiostream.h>
class stdiobuf : public streambuf;
class stdiostream : public ios;

#include <iomanip.h>
```

Zum *iostream*-Paket gehören einige Funktionen, die Zeichen liefern, aber den Rückgabetyp *int* aufweisen. Der Datentyp *int* wird verwendet, damit alle Zeichen im Zeichensatzdes Rechners einschließlich des Wertes EOF - als Hinweis auf einen Fehler - zurückgegeben werden können. Gewöhnlich wird ein Zeichen in einer Speicherstelle des Typs*char* oder *unsigned char* abgelegt.

Das *iostream*-Paket besteht aus einer Reihe von zentralen Klassen, die die grundlegende Funktionalität der Ein-/Ausgabe-Umwandlung und der Pufferung bereitstellen,sowie aus mehreren spezialisierten Klassen, die aus den zentralen Klassen abgeleitetwurden. Die beiden Klassenarten werden nachstehend aufgelistet.

Die Include-Datei <*iomanip.h*> stellt Makrodefinitionen zur Verfügung, die der Programmierer bei der Definition eigener parametrisierter Manipulatoren (siehe *manip*) verwenden kann.

Zentrale Klassen

Das Kernstück des *iostream*-Paketes setzt sich aus folgenden Klassen zusammen:

streambuf

> Dies ist die Basisklasse für Puffer. Sie unterstützt die *Einfügung* (*Speicherung*) und die *Extraktion* (*Entnahme*) von Zeichen. Aus Effizienzgründen wurden die meisten Elementfunktionen als Inline-Versionen implementiert. Die öffentliche Schnittstelle der Klasse *streambuf* wird bei *sbufpub* beschrieben, während die geschützte Schnittstelle(für abgeleitete Klassen) im Abschnitt zu *sbufprot* erläutert wird.

ios

> Dies ist die Basisklasse für die *stream*-Ein-/Ausgabe in C++. Diese Klasse enthältStatusvariablen, die verschiedenen *stream*-Klassen gemeinsam sind, beispielsweiseFehler- und Formatstatus (siehe *ios*).

istream

> Diese Klasse unterstützt formatierte und unformatierte Umwandlungen von Zeichenfolgen, die aus *streambuf*-Objekten entnommen werden (siehe *istream*).

ostream

> Diese Klasse unterstützt formatierte und unformatierte Umwandlungen in Zeichenfolgen, die in *streambuf*-Objekten gespeichert werden (siehe *ostream*).

iostream

> Diese Klasse kombiniert *istream* und *ostream* . Sie wurde für Fälle entwickelt, in denenbidirektionale Operationen (also das Einfügen in bzw. die Extraktion von einer Zeichenfolge) benötigt werden (siehe *ios*).

istream_withassign
ostream_withassign
iostream_withassign

> Diese Klassen fügen Zuweisungsoperatoren und einen Konstruktor ohne Operandenzur entsprechenden Klasse ohne Zuweisung hinzu. Die vordefinierten Datenströme (siehe unten) *cin*, *cout*, *cerr* und *clog* sind Objekte dieser Klassen (siehe *istream* , *ostream* und *ios*).

Vordefinierte Datenströme

cin

Dies ist die Standardeingabe (Dateideskriptor 0), die *stdin* in der Programmiersprache C entspricht.

cout

Dies ist die Standardausgabe (Dateideskriptor 1), die *stdout* in der Programmiersprache C entspricht.

cerr

cerr ist der Standardfehlerstrom (Dateideskriptor 2). Die Ausgabe über den Datenstrom ist einheitengepuffert; Zeichen werden also nach jeder Insert-Operation ausdem Puffer an das C-Lauzeitsystem übergeben (siehe *ostream::osfx()* bei *ostream* und *ios::unitbuf* bei *ios*). *cerr* entspricht *stderr* in der Programmiersprache C.

clog

Dieser Datenstrom wird ebenfalls an den Dateideskriptor 2 umgeleitet, im Gegensatzzu *cerr* ist die Ausgabe aber gepuffert.

cin, *cerr* und *clog* sind mit *cout* verknüpft. Jede Benutzung dieser Datenströme führtdazu, daß der *cout*-Puffer geleert wird.

Neben den oben aufgelisteten zentralen Klassen enthält das *iostream*-Paket weitere,daraus abgeleitete Klassen, die in anderen Include-Dateien deklariert sind. Sie könnendie abgeleiteten Klassen verwenden oder aus den zentralen Klassen *iostream* eigeneKlassen ableiten.

Aus streambuf abgeleitete Klassen

Die aus *streambuf* abgeleiteten Klassen definieren Einzelheiten über die "Erzeugung" oder das "Verbrauchen" von Zeichen. Die Ableitung einer Klasse aus *streambuf* (die*geschützte Schnittstelle*) wird im Abschnitt zu *sbufprot* erläutert. Die verfügbaren Pufferklassen sind:

filebuf

Diese Pufferklasse unterstützt die Ein-/Ausgabe über Dateideskriptoren. Elementfunktionen unterstützen das Öffnen, Schließen und Positionieren. Üblicherweise muß einProgramm nicht auf die Dateideskriptoren zugreifen (siehe *filebuf*).

stdiobuf

Diese Pufferklasse unterstützt die Ein-/Ausgabe über *stdio*-FILE-Strukturen. Sie sollteverwendet werden, wenn Quellcode in den Sprachen C und C++ nebeneinander verwendet wird. Für neue Programme wird die Verwendung von *filebuf* empfohlen (siehe *stdiobuf*).

strstreambuf

Diese Pufferklasse speichert und entnimmt Zeichen aus Byte-Feldern (z.B. Zeichenketten) im Hauptspeicher (siehe *sstreambuf*).

Aus istream, ostream und iostream abgeleitete Klassen

Klassen, die aus *istream* , *ostream* und *iostream* abgeleitet sind, stellen einen Spezialfall der zentralen Klassen für die Verwendung einer bestimmten Art von *streambuf*-Objektendar. Diese Klassen sind:

ifstream
ofstream
fstream

Diese Klassen unterstützen die formatierte Ein-/Ausgabe in und aus Dateien. Ein *filebuf*-Objekt wird für die Ein-/Ausgabe verwendet. Allgemeine Operationen (wie das Öffnen oder Schließen) können direkt - ohne explizite Nennung des *filebuf*-Objektes -auf die Datenströme angewendet werden (siehe *fstream*).

fstreambase

Die für alle 3 Klassen gemeinsamen Elementfunktionen sind in der Klasse *fstreambase* definiert.

istrstream
ostrstream
strstream

Diese Klassen unterstützen die Bearbeitung von Byte-Feldern (z.B. Zeichenketten) und setzen die Klasse *strstreambuf* ein (siehe *strstream*).

strstreambase

Die für diese Klassen gemeinsamen Elementfunktionen sind in der Klasse *strstre-ambase* definiert.

stdiostream

Diese Klasse ist eine Spezialisierung von *iostream* für *stdio*-FILE-Objekte (siehe *stdiobuf*).

BESONDERHEITEN

Die Performance von Programmen, die von *cin* nach *cout* kopieren, kann teilweise verbessert werden, indem die Verknüpfung zwischen *cin* und *cout* gelöst und der *cout*-Puffer explizit geleert wird. Einige Elementfunktionen von *streambuf* und *ios* (die in diesem Abschnitt nicht erläutert wurden) sind lediglich aus Gründen der Abwärtskompatibilität zum älteren *stream*-Paket verfügbar.

SIEHE AUCH *filebuf*, *fstream*, *ios*, *istream*, *manip*, *ostream*, *sbufprot*, *sbufpub*, *strstream*, *sstreambuf*, *stdiobuf*

4.2 filebuf Puffer für die Datei-Ein-/Ausgabe

In diesem Abschnitt wird die Verwendung der Klasse *filebuf* beschrieben.

```
#include <iostream.h>

typedef long streamoff, streampos;

class ios
{

   public:
   enum  seek_dir {beg, cur, end};
   enum  open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                                bin, tabexp};

   // sowie viele weitere Klassenelemente, siehe ios ...

};

#include <fstream.h>

class filebuf : public streambuf
{
public:

   static const int openprot; /* Standardschutzmodus für Öffnen*/

   filebuf();
   ~filebuf();
   filebuf(int d);
   filebuf(int d, char* p, int len);

   filebuf*          attach(int d);
   filebuf*          close();
   int               fd();
   int               is_open();
   filebuf*          open(const char *name, int omode, int prot=openprot);

   virtual int       overflow(int=EOF);
   virtual streampos seekoff(streamoff, ios::seek_dir, int omode);
   virtual streambuf* setbuf(char* p, int len);
   virtual int       sync();
   virtual int       underflow();

};
```

filebuf ist ein Spezialfall der Klasse *streambuf*, wobei Dateien als Quelle oder Ziel für die Zeichenübertragung eingesetzt werden. Zeichen werden durch Schreiben in dieDatei "aufgebraucht" und durch Lesen aus der Datei "erzeugt". Wenn das Positionierenin einer Datei möglich ist, kann auch in dem *filebuf*-Objekt positioniert werden. Wenndie Datei das Lesen und Schreiben erlaubt, ermöglicht die Klasse *filebuf* sowohl dasSpeichern als auch die Entnahme von Zeichen. Zwischen Schreib- und Lesevorgängenist kein besonderer Aufruf erforderlich (im Gegensatz zu *stdio*). Ein *filebuf*-Objekt, dasmit einem Dateideskriptor verknüpft ist, wird als geöffnet bezeichnet. Dateien werden im BS2000 ohne Angabe eines Schutzmodus verwendet.

Der *Reservierungsbereich* (oder *Puffer*, siehe *sbufpub* und *sbufprot*) wird automatischzugewiesen, wenn mit einem Konstruktor oder beim Aufruf von *setbuf()* keine expliziteAngabe erfolgt. *filebuf*-Objekte können in einen nicht gepufferten Modus versetzt werden, indem an den Konstruktor oder *setbuf()* bestimmte Argumente übergeben werden.In diesem Fall wird bei jedem Lese- oder Schreibvorgang jedes Zeichen gleich an dasC-Laufzeitsystem durchgereicht. Dabei ist die nicht gepufferte Ein-/Ausgabe langsamerals die gepufferte. Die Zeiger *get* und *put* sind - für den Reservierungsbereich - miteinander verknüpft und verhalten sich wie ein einziger Zeiger. Die folgenden Beschreibungenbeziehen sich daher auf den einen kombinierten Zeiger.

Für die folgenden Beschreibungen wird angenommen, daß

- *f* vom Typ *filebuf* ist.
- *mode* eine *int*-Zahl ist, die den Öffnungsmodus (*open_mode*) darstellt.

Konstruktoren

filebuf()

> Ein anfänglich geschlossenes *filebuf*-Objekt wird angelegt.

filebuf(int d)

> Ein *filebuf*-Objekt, das mit dem Dateideskriptor *d* verknüpft ist, wird angelegt.

filebuf(int d, char * p, int len)

> Ein mit dem Dateideskriptor *d* verknüpftes *filebuf*-Objekt wird angelegt. Das Objektist für die Verwendung des Reservierungsbereichs - beginnend bei *p* mit einer Längevon *len* byte - initialisiert. Wenn *p* gleich NULL oder wenn *len* kleiner oder gleich 0 ist, weist das *filebuf*-Objekt keine Pufferung auf.

Elementfunktionen (nicht virtuell)

filebuf * pfb=f.attach(int d)

> *f* wird mit dem geöffneten Dateideskriptor *d* verknüpft. Die Funktion *attach()* liefertüblicherweise *&f*; der Wert 0 wird zurückgegeben, wenn *f* bereits geöffnet ist.

filebuf * pfb=f.close()

> Auf die Ausgabe wartende Daten werden geschrieben, der Puffer gelöscht, der Dateideskriptor geschlossen und die Verknüpfung mit *f* gelöst. Sofern kein Fehler auftritt,wird der Fehlerstatus von *f* gelöscht. Die Funktion *close()* liefert *&f*, wenn keine Fehler auftreten. Anderenfalls wird der Wert 0 zurückgegeben. Auch beim Auftreten vonFehlern hinterläßt die Funktion *close()* den Dateideskriptor und *f* in geschlossenemZustand.

int i=f.fd()

Es wird der Dateideskriptor *i* geliefert, mit dem *f* verknüpft ist. Wenn *f* geschlossenist, liefert *fd()* den Wert EOF.

int i=f.is_open()

Ein Wert ungleich 0 wird geliefert, wenn *f* mit einem Dateideskriptor verknüpft ist.Anderenfalls wird der Wert 0 geliefert.

filebuf * pfb=f.open(char * name, int mode, int prot)

Die Datei *name* wird geöffnet und mit *f* verknüpft. Wenn die Datei noch nicht existiert,wird versucht, sie anzulegen, sofern in *mode* nicht *ios::nocreate* bzw. *ios::in* angegeben ist.
Der Parameter *prot* wird im BS2000 ignoriert.
Ein Fehler tritt auf, wenn *f* bereits geöffnet ist. Die Funktion *open()* liefert bei Erfolg *f*, im Fehlerfall den Wert 0.
Die Elemente von *mode* sind Bits, die durch eine ODER-Operation miteinander verknüpft werden können. (Da eine ODER-Operation einen *int*-Wert liefert, verwendet *open()* statt dem *open_mode*-Argument ein *int*-Argument). Die Bedeutung der einzelnen Bits in *mode* sind detailliert im Abschnitt *fstream* beschrieben.
Für den Dateinamen *name* sind alle Angaben möglich wie bei den C-Bibliotheksfunktionen *open()* bzw. *fopen()*. Dieser Name wird an das C-Laufzeitsystem weitergereicht. Damit sind alle Steuerungsmöglichkeiten (außer satzorientierte Ein-/Ausgabe)beim Öffnen einer Datei auch in C++ gegeben. Eine Auflistung der möglichen Angaben für *name* finden Sie im Abschnitt *fstream* , weitere detaillierte Informationen zur Dateiverarbeitung im Handbuch "C-Bibliotheksfunktionen".

Virtuelle Elementfunktionen

int i=f.overflow(int c)

Die Beschreibung der prinzipiellen Funktionsweise finden Sie im Abschnitt *sbufprot*(streambuf:: overflow()).
Für filebuf-Objekte bedeutet das: Der Pufferinhalt wird in die dazugehörige Datei geschrieben, sofern *f* mit einer geöffneten Datei verbunden ist. Dadurch steht ein neuer *put*-Bereich zu Verfügung. Beim Auftreten eines Fehlers ist der Returnwert EOF.
Bei Erfolg wird 0 zurückgeliefert.

streampos sp=f.seekoff(streamoff off, ios::seek_dir dir, int mode)

Der Zeiger *get/put* wird entsprechend den Angaben in *off* und *dir* verschoben. Die Operation kann fehlschlagen, wenn die mit *f* verknüpfte Datei kein Positionieren unterstützt oder wenn die versuchte Verschiebung aus anderen Gründen ungültig ist (beispielsweise durch die Suche einer Position, die vor dem Dateibeginn liegt). *off* wird als Zählerwert interpretiert, der relativ zu der durch *dir* spezifizierten Position in derDatei liegt. Eine weitergehende Beschreibung finden Sie im Abschnitt zu *sbufpub* .
mode wird ignoriert. Die Funktion *seekoff()* liefert *sp*, die neue Position, oder EOF beim Auftreten eines Fehlers. Die Position in der Datei ist nach dem Auftreten eines Fehlers nicht definiert. Im BS2000 ist das relative Positionieren in Textdateien nur mit*off* = 0 erlaubt.

streambuf * psb=f.setbuf(char * ptr, int len)

Der Reservierungsbereich wird, beginnend bei *ptr*, auf *len* byte gesetzt. *f* ist nichtgepuffert, wenn *ptr* gleich NULL oder *len* kleiner oder gleich 0 ist. Die Funktion *setbuf()* liefert üblicherweise *&f*. Wenn *f* bereits geöffnet ist und ein Puffer zugewiesen wurde, wird am Reservierungsbereich oder im Pufferungsstatus keine Änderungvorgenommen, und *setbuf()* liefert den Wert 0.

int i=f.sync()

Es wird versucht, den Status des Zeigers *get/put* von *f* mit dem Status der Datei *f.fd()* zu synchronisieren. Dies bedeutet, daß Zeichen in die Datei geschrieben werden, wenn diese in einem Puffer für die Ausgabe vorliegen, oder eine Repositionierung (seek) in der Datei versucht wird, wenn Zeichen gelesen wurden und die Eingabe gepuffert ist. Die Funktion *sync()* liefert üblicherweise den Wert 0. Wenn eineSynchronisation nicht möglich ist, wird der Wert EOF geliefert. *sync()* gewährleistet nicht, daß das "Leeren des Puffers" auf Diskette/Platte erfolgt. Im BS2000 übergibt *sync()* den Pufferinhalt an das C-Laufzeitsystem. Die Synchronisation verwendet relatives Positionieren des C-Laufzeitsystem. Dies ist im BS2000 beiTextdateien nicht möglich. Die *sync*-Funktion kann deshalb nur auf Binärdateien angewendet werden.

Hinweis

Manchmal müssen Zeichen in einem Vorgang geschrieben werden. In diesem Fall sollte das Programm die Funktion *setbuf()* (oder einen Konstruktor) verwenden, damit der Reservierungsbereich groß genug ist, um alle zu schreibenden Zeichen aufzunehmen. Dann kann anschließend *sync()* aufgerufen werden, die Zeichen werden gespeichert, und es erfolgt ein erneuter Aufruf von *sync()*.

int i=f.underflow();

Die Beschreibung der prinzipiellen Funktionsweise finden Sie im Abschnitt *sbufprot* (streambuf:: underflow()).
Für filebuf-Objekte bedeutet das:
Wenn der *get*-Bereich leer ist, wird aus der dazugehörigen Datei gelesen, sofern *f* mit einer geöffneten Datei verbunden ist (auffüllen des *get*-Bereiches). Bei Erfolg wird das nächste Zeichen zurückgeliefert.
Beim Auftreten eines Fehlers ist der Returnwert EOF.

BEISPIEL Das folgende Programm versucht, dem Dateideskriptor 1 (entspricht *cout*) eine Variable des Typs *filebuf* zuzuordnen. Anschließend wird eine Meldung ausgegeben, die den Erfolg oder Mißerfolg von *attach()* anzeigt:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main()
{
  filebuf b;     /* Konstruktor ohne Parameter wird aufgerufen */
  if (b.attach(1))
  {
    static char str[] = "filebuf b wurde mit Dateideskriptor 1 verknuepft\n";
    b.sputn(str, sizeof(str)-1);
  }
  else
  {
    cerr << "filebuf kann nicht mit Dateideskriptor 1 verknuepft werden\n";
    exit(1);      /* Rückgabe bei Fehler */
  }
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
filebuf b wurde mit Dateideskriptor 1 verknuepft
%  CCM0998 Verbrauchte CPU-Zeit: 0.0003 Sekunden
```

BESONDERHEITEN

Die Funktion *attach()* und die Konstruktoren sollten prüfen, ob der übergebene Dateideskriptor geöffnet ist.

Unteilbare Lesevorgänge (*atomic*) können nicht erzwungen werden.

SIEHE AUCH

fstream , *sbufprot* , *sbufpub*

lseek() im C-Laufzeitsystem

4.3 fstream Spezialisierung von iostream und streambuf auf Dateien

In diesem Abschnitt werden die Klassen *ifstream*, *ofstream* und *fstream* beschrieben,die Operationen unterster Ebene auf Dateien und Datenströme ermöglichen.

```
#include <iostream.h>

class ios
{
public:

    enum  seek_dir {beg, cur, end};
    enum  open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                                bin, tabexp};

    enum  io_state {goodbit=0, eofbit, failbit, badbit};
    // siehe ios, dort finden Sie weitere Klassenelemente ...

};

#include <fstream.h>

class fstreambase : virtual public ios
{
public:

    fstreambase();
    ~fstreambase();
    fstreambase(const char* name, int mode, int prot=filebuf::openprot);
    fstreambase(int fd);
    fstreambase(int fd, char * p, int l);

    void        attach(int fd);
    void        close();
    void        open(const char* name, int mode, int prot=filebuf::openprot);

    filebuf*    ();
    void        setbuf(char* p, int l);

};

class ifstream : public fstreambase, public istream
{
public:

    ifstream();
    ~ifstream();
    ifstream(const char* name, int mode=ios::in, int prot=filebuf::openprot);
    ifstream(int fd);
    ifstream(int fd, char* p, int l);
```

```
    void        open(const char* name, int mode=ios::in, int prot=filebuf::openprot);

    filebuf*    rdbuf();

};

class ofstream : public fstreambase, public ostream
{
public:

    ofstream();
    ~ofstream();
    ofstream(const char* name, int mode=ios::out, int prot =filebuf::openprot);ofstream(int fd);
    ofstream(int fd, char* p, int l);

    void        open(const char* name, int mode=ios::out, int prot=filebuf::openprot);

    filebuf*    rdbuf();

};

class fstream : public fstreambase, public iostream
{
public:

    fstream();
    ~fstream();
    fstream(const char* name, int mode, int prot =filebuf::openprot);
    fstream(int fd);
    fstream(int fd, char* p, int l);

    void        open(const char* name, int mode, int prot=filebuf::openprot);

    filebuf*    rdbuf();

};
```

Die Basisklasse *fstreambase* enthält die Standard-Definitionen der Konstruktoren undElementfunktionen für die abgeleiteten Klassen.

ifstream, *ofstream* und *fstream* sind spezialisierte Versionen von *istream*, *ostream* und*iostream*, die sich auf Dateien beziehen. Das in diesen Klassen eingesetzte *streambuf*-Objekt ist also immer vom Typ *filebuf*.

Für die folgenden Beschreibungen wird vorausgesetzt, daß

- *f* vom Typ *ifstream*, *ofstream* oder *fstream* ist.
- *mode* ein *int*-Wert ist, der *open_mode* darstellt.

Konstruktoren

In *xstream* ist *x* entweder:

- *if*
- *of* oder
- *f*.

xstream steht daher für:

- *ifstream*
- *ofstream* oder
- *fstream*.

Die Konstruktoren für *xstream* sind:

*x*stream()

Hierdurch wird ein nicht geöffnetes *xstream*-Objekt angelegt.

*x*stream(char * name, int mode, int prot)

Ein *xstream*-Objekt wird angelegt, und die Datei *name* wird unter Verwendung von *mode* (als Öffnungsmodus) geöffnet. Die Parameter *name* und *mode* sind weiter unten bei *open()* beschrieben.
Der Parameter *prot* wird im BS2000 ignoriert.
Im Fehlerstatus (*io_state*) des angelegten *xstream*-Objektes wird ein Fehler angezeigt,wenn die Datei nicht erfolgreich geöffnet werden konnte.

*x*stream(int fd)

Ein *xstream*-Objekt wird angelegt und mit dem Dateideskriptor *fd* verknüpft, derbereits geöffnet sein muß.

*x*stream(int fd, char * ptr, int len)

Ein *xstream*-Objekt wird angelegt und mit dem Dateideskriptor *fd* verknüpft. Zusätzlich wird das entsprechende *filebuf*-Objekt zur Nutzung eines *len* byte großen Reservierungsbereiches, beginnend bei *ptr*, initialisiert. Wenn *ptr* gleich NULL oder *len*gleich 0 ist, sind die *filebuf*-Operationen nicht gepuffert.

Elementfunktionen

void f.attach(int fd)

f wird mit dem Dateideskriptor *fd* verknüpft. Ein Fehler tritt auf, wenn *f* bereits einerDatei zugeordnet ist. Bei einem Fehler wird *ios::failbit* im Fehlerstatus von *f* gesetzt.

void f.close()

Das entsprechende *filebuf*-Objekt wird geschlossen und dadurch die Verknüpfung von *f* mit der Datei gelöst. Der Fehlerstatus von *f* wird gelöscht, sofern kein Fehleraufgetreten ist. Als Fehler gilt hierbei das Auftreten eines Fehlers im C-Laufzeitsystem-Aufruf *close()*.

void f.open(char * name, int mode, int prot)

Die Datei *name* wird geöffnet und mit *f* verknüpft. Wenn die Datei noch nicht existiert,wird versucht, sie anzulegen, sofern *ios::nocreate* bzw. *ios::in* nicht gesetzt ist. Fehlertreten auf, wenn *f* bereits geöffnet ist oder der C-Laufzeitsystem-Aufruf *open()* einenFehler signalisiert. *ios::failbit* wird beim Auftreten eines Fehlers im Fehlerstatus von *f*gesetzt.
Der Parameter *prot* wird im BS2000 ignoriert.

Für den Dateinamen *name* sind folgende Angaben möglich:

- jeder gültige BS2000-Dateiname
- "link=*linkname*" *linkname* bezeichnet einen BS2000-Linknamen
- "(SYSDTA)", "(SYSOUT)", "(SYSLST)" die entsprechende Systemdatei
- "(SYSTERM)" Terminal-Ein-/Ausgabe
- "(INCORE)" temporäre Binärdatei, die nur im virtuellen Speicher angelegt wird

Detaillierte Informationen finden Sie im Handbuch "C-Bibliotheksfunktionen".

Die Elemente von *open_mode* sind Bits, die durch eine ODER-Operation miteinanderverknüpft werden können. (Da eine ODER-Operation einen *int*-Wert liefert, übernimmt*open()* statt des *open_mode*-Argumentes einen *int*-Wert.) Die Bedeutung dieser Bits in *mode* sind:

ios::app

> Es wird auf das Dateiende positioniert. Alle Daten, die anschließend in die Dateigeschrieben werden, werden immer an das Ende der Datei angehängt. *ios::app* impliziert die Verwendung von *ios::out*.

ios::ate

> Bei der Ausführung von *open()* wird auf das Dateiende positioniert. *ios::ate* impliziert nicht die Verwendung von *ios::out*.

ios::in

> Die Datei wird für die Eingabe geöffnet. *ios::in* wird durch die Konstruktion und dasÖffnen von *ifstream*-Objekten implizit gesetzt. Für *fstream*-Objekte spezifiziert *ios::in*, daß Eingabeoperationen - sofern möglich - erlaubt sind. *ios::in* kann auchin die Modusangabe eines *ostream*-Objektes eingefügt werden. Das impliziert, daßdie Originaldatei (sofern existent) nicht abgeschnitten werden soll.
> Wenn die zu öffnende Datei nicht existiert, schlägt der Aufruf von *open()* fehl.

ios::out

> Die Datei wird für die Ausgabe geöffnet. *ios::out* wird durch die Konstruktion unddas Öffnen von *ofstream*-Objekten implizit gesetzt. Für *fstream*-Objekte bedeutetdie Angabe von *ios::out*, daß Ausgabeoperationen erlaubt sein sollen.

ios::trunc

> Wenn die Datei bereits existiert, wird deren Inhalt abgeschnitten (verworfen). DieserModus wird implizit gesetzt, wenn *ios::out* (einschließlich der impliziten Spezifikation für *ofstream*) angegeben ist und weder *ios::ate* noch *ios::app* angegeben werden.

ios::nocreate

> Wenn die zu öffnende Datei nicht existiert, schlägt der Aufruf von *open()* fehl.

ios::noreplace

Wenn die Datei bereits existiert, kann die Funktion *open()* nicht erfolgreich ausgeführt werden.

ios::bin

Die Datei wird als Binärdatei geöffnet. Wenn dieser Parameter fehlt, wird die Dateials Textdatei geöffnet.

ios::tabexp

Für Binärdateien und Eingabedateien wird diese Angabe ignoriert. Bei Textdateienwird das Tabulatorzeichen (\t) in die entsprechende Anzahl Leerzeichen umgesetzt.Die Tabulatorpositionen haben einen Acht-Spalten-Abstand (1, 9, 17, ...). OhneAngabe des Parameters wird das Tabulatorzeichen als entsprechender EBCDIC-Wert in die Textdatei geschrieben. (Siehe Handbuch "C-Bibliotheksfunktionen")

Hinweis

Wird *open()* der Klasse *istream* verwendet, so ist das Bit *ios::in* immer gesetzt.
Wird *open()* der Klasse *ostream* verwendet, so ist das Bit *ios::out* immer gesetzt.

filebuf * pfb=f.rdbuf()

Ein Zeiger auf das *filebuf*-Objekt, das mit *f* verknüpft ist, wird geliefert.

fstream::rdbuf() ist bedeutungsgleich mit *iostream::rdbuf()*.

void f.setbuf(char * p, int len)

Diese Zeile hat den üblichen Effekt eines Aufrufs von *setbuf()* (siehe *filebuf()*), wobei ein Speicherbereich für den Reservierungsbereich angeboten oder die nicht gepufferte Ein-/Ausgabe angefordert wird. Ein Fehler tritt auf, wenn *f* geöffnet oder der Aufruf *f.rdbuf()->setbuf* erfolglos ist.

BEISPIEL Das folgende Programm öffnet die Datei *#TEMP*. Bei Erfolg wird ein Text in die Datei geschrieben.

```
#include <fstream.h>
#include <iostream.h>
int main()
{
  static char * name = "#TEMP";
  ofstream q(name, ios::out);
  cout << " Datei " << name;
  if (q.ios::failbit)
  {
     cout << " ist geoeffnet.\n";
     q << "Das ist die erste Zeile in der Datei " << name << ".\n";
  }
  else
  {
     cout << " konnte nicht geoeffnet werden.\n";
     exit (1);
  }
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
Datei #TEMP ist geoeffnet.
%  CCM0998 Verbrauchte CPU-Zeit: 0.0395 Sekunden
```

SIEHE AUCH

filebuf, *ios*, *istream*, *ostream*, *sbufprot*, *sbufpub*

4.4 ios Basisklasse für die Ein-/Ausgabe

In diesem Abschnitt werden die Operatoren beschrieben, die sowohl für die Ein- als auch für die Ausgabe definiert sind.

```
#include <iostream.h>

class ios
{
public

   enum  io_state {goodbit=0, eofbit, failbit, badbit};

   enum  open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                                    tabexp};

   enum  seek_dir {beg, cur, end};

   /* Flags zur Formatsteuerung */
   enum
   {

      skipws=01,
      left=02, right=04, internal=010,
      dec=020, oct=040, hex=0100,
      showbase=0200, showpoint=0400,
      uppercase=01000, showpos=02000,
      scientific=04000, fixed=010000,
      unitbuf=020000, stdio=040000

   };

   static const long basefield;

         /* dec | oct | hex */

   static const long adjustfield;

         /* left | right | internal */

   static const long floatfield;

         /* scientific | fixed */

public:

   ios(streambuf*);
   virtual ~ios();
```

```
    int             bad() const;
    static long     bitalloc();
    void            clear(int state=0);
    int             eof() const;
    int             fail() const;
    char            fill() const;

    char            fill(char);
    long            flags() const;
    long            flags(long);
    int             good() const;
    long&           iword(int);
    int             operator!() const;
                    operator void*();
                    operator const void*() const;

    int             precision() const;
    int             precision(int);
    void* &         pword(int);
    streambuf*      rdbuf();
    int             rdstate() const;
    long            setf(long setbits, long field);
    long            setf(long);
    static void     sync_with_stdio();
    ostream*        tie();
    ostream*        tie(ostream*);
    long            unsetf(long);
    int             width() const;
    int             width(int);
    static int      xalloc();

protected:

    ios();

    init(streambuf*);

private:

    ios(ios&);void  operator=(ios&);

};

/* Manipulatoren */

  ios&          dec(ios&);
  ios&          hex(ios&);
  ios&          oct(ios&);
  ostream&      endl(ostream& i);
  ostream&      ends(ostream& i);
  ostream&      flush(ostream&);
  istream&      ws(istream&);
```

Die aus *ios* abgeleiteten *stream*-Klassen stellen eine Schnittstelle auf sehr hoher Ebene zur Verfügung, über die formatierte wie auch unformatierte Informationen in und aus *streambuf*-Objekten übertragen werden können.

In der Klasse *ios* sind Aufzählungen (*open_mode*, *io_state* und *seek_dir*) und Formatflags deklariert, um den globalen Namensbereich durch diese Informationen nicht zu belasten. *io_state* wird in diesem Abschnitt unter der Überschrift *Fehlerstatus* beschrieben. Die Formatfelder werden ebenfalls in diesem Abschnitt - im Absatz über die *Formatierung* - erläutert. Erklärungen zu *open_mode* finden Sie im Abschnitt zu *fstream*bei der Funktion *open()*, und zusätzliche Informationen zu *seek_dir* werden im Abschnitt zu *sbufpub* bei der Funktion *seekoff()* gegeben.

Bei den folgenden Erläuterungen wird vorausgesetzt, dass

- *s* und *s2 ios*-Objekte sind.
- *sr* vom Typ *ios&* ist.
- *mode* ein *int*-Wert ist, der *open_mode* repräsentiert.

Konstruktoren und Zuweisung

ios(streambuf * sb)

> Das durch *sb* bezeichnete *streambuf*-Objekt wird zu dem *streambuf*-Objekt, das mit dem angelegten *ios*-Objekt verknüpft ist. Das Programmverhalten ist für *sb* gleich 0 nicht definiert.

ios(ios& sr)
s2=s

> Das Kopieren von *ios*-Objekten ist allgemein nicht genau definiert, deshalb sind die Konstruktoren und Zuweisungsoperatoren als *private* deklariert. Der Compiler gibt bei einem Versuch, *ios*-Objekte zu kopieren, Meldungen aus. Gewöhnlich werden Zeiger auf *iostream*-Objekte kopiert.

ios()
init(streambuf * sb)

> Da die Klasse *ios* als virtuelle Basisklasse vererbt wird, muß ein Konstruktor ohne Argumente eingesetzt werden. Der Konstruktor wird als *protected* deklariert. Aus diesem Grund ist *ios::init (streambuf*)* als *protected* deklariert und muß für die Initialisierung abgeleiteter Klassen eingesetzt werden.

Fehlerstatus

Ein *ios*-Objekt besitzt einen internen Fehlerstatus, der aus mehreren Bits besteht, die als *io_state* deklariert sind. Die Elementfunktionen, die sich mit der Fehlerstatusangabe beschäftigen, sind:

int i=s.rdstate()

> Der aktuelle Fehlerstatus wird geliefert.

s.clear(int i)

> Der Wert *i* wird als Fehlerstatus gespeichert. Wenn der Wert von *i* gleich 0 ist, werden alle Bits gelöscht. Um ein Bit ohne gleichzeitiges Löschen zuvor gesetzter Bits zusetzen, wird eine Programmzeile in der Form *s.clear(ios::badbit|s.rdstate())* benötigt.

int i=s.good()

Es wird ein Wert ungleich 0 geliefert, wenn im Fehlerstatus keine Bits gesetzt sind.Anderenfalls wird der Wert 0 geliefert.

int i=s.eof()

Es wird ein Wert ungleich 0 geliefert, wenn *eofbit* im Fehlerstatus gesetzt ist. Ansonsten wird der Wert 0 geliefert. Das Bit wird üblicherweise gesetzt, wenn das Dateiende bei der Entnahme von Zeichen erreicht wurde.

int i=s.fail()

Es wird ein Wert ungleich 0 geliefert, wenn entweder *badbit* oder *failbit* im Fehlerstatus gesetzt sind. Im anderen Fall wird der Wert 0 zurückgegeben. Dies deutet üblicherweise darauf hin, dass eine Extraktion oder Konvertierung erfolglos verlaufen ist,der Datenstrom aber weiterhin eingesetzt werden kann. Nach dem Löschen von *failbit* können im allgemeinen weitere Ein- und Ausgaben für *s* ausgeführt werden.

int i=s.bad()

Es wird ein Wert ungleich 0 geliefert, wenn *badbit* im Fehlerstatus gesetzt ist. Im anderen Fall wird der Wert 0 zurückgegeben. Dies deutet üblicherweise darauf hin, dass eine Operation auf *s.rdbuf()* erfolglos verlaufen ist - ein schwerwiegender Fehler,der häufig nicht behoben werden kann. Weitere Ein- und Ausgabeoperationen für *s* können wahrscheinlich nicht ausgeführt werden.

Operatoren

Es sind zwei Operatoren definiert, die eine komfortable Prüfung des Fehlerstatus eine s*ios*-Objektes ermöglichen: *operator!()* und *operator void*()* bzw. *operator const void*() const*. Der zweite Operator wandelt ein *ios*-Objekt in einen Zeiger um, so dass dieser mit 0 verglichen werden kann. Die Umwandlung liefert den Wert 0, wenn *failbit* oder *badbit* im Fehlerstatus gesetzt sind. Im anderen Fall wird ein Zeigerwert zurückgegeben, der aber nicht verwendet werden sollte. Hierdurch können Ausdrücke in folgender Form eingesetzt werden:

```
if (cin) ...
```

```
if (cin >> x) ...
```

Der Operator ! liefert - bei gesetztem *badbit* oder *failbit* im Fehlerstatus - einen Wertungleich 0, wodurch Ausdrücke wie der folgende verwendet werden können:

```
if (!cout) ...
```

Formatierung

Ein *ios*-Objekt weist einen Formatstatus auf, der von Ein- und Ausgabeoperationen zur Steuerung der Details für die Formatierungsoperationen verwendet wird. Die Komponenten des Status können vom Programmcode des Benutzers gesetzt und nach Belieben untersucht werden. Die meisten Formatierungsdetails werden durch die Funktionen *flags()*, *setf()* und *unsetf()* mittels Setzen der folgenden Flags gesteuert, die in einer Aufzählung der Klasse *ios* deklariert sind. Drei weitere Komponenten des Formatstatus werden durch die Funktionen *fill()*, *width()* und *precision()* festgelegt.

skipws

Wenn *skipws* gesetzt ist, wird Zwischenraum in der Eingabe übersprungen. Dies bezieht sich auf skalare Extraktionen.
Bei nicht gesetztem *skipws* wird kein Zwischenraum übersprungen, bevor der Extraktor mit der Umwandlung beginnt. In diesem Fall sollten Felder mit der Länge 0 nicht verwendet werden, um einer Schleifenbildung vorzubeugen. Wenn also das nächste Zeichen ein Zwischenraumzeichen ist und die Variable *skipws* nicht gesetzt ist, melden arithmetische Extraktoren einen Fehler.
Wird bei nicht gesetztem *skipws* eine numerische Eingabe versucht und bei dem ersten Zeichen der Eingabe handelt es sich um ein Zwischenraumzeichen, führt die Extraktion zu einem Fehler. Im Fall der Stringeingabe wird die Extraktion beim ersten auftretenden Zwischenraumzeichen abgebrochen. Für den speziellen Fall, dass das erste Zeichen des Eingabestroms ein Zwischenraumzeichen ist, findet keine Extraktion statt. In beiden Fällen wird der Eingabestrom gelesen, bis das erste Zwischenraumzeichen auftritt. Ein weiterer Lesevorgang findet dann nicht mehr statt.

left
right
internal

Durch diese Flags wird das Auffüllen eines Wertes festgelegt. Wenn *left* gesetzt ist,wird der Wert linksbündig ausgerichtet, so dass Füllzeichen hinter dem Wert angefügtwerden. Bei gesetztem *right* wird der Wert rechtsbündig ausgerichtet, und Füllzeichenwerden vor dem Wert eingefügt. Die Verwendung von *internal* führt dazu, dass Füllzeichen zwar hinter einem führenden Vorzeichen oder einer Basisindikation, aber nochvor dem Wert eingefügt werden. Die rechtsbündige Ausrichtung ist die Standardvorgabe, wenn kein Flag gesetzt ist. Die Felder werden gemeinsam durch das *static*-Element *ios::adjustfield* identifiziert. Die Funktion *fill()* legt das Füllzeichen fest. Die Breite, bis zu der aufgefüllt wird, wird durch die Funktion *width()* definiert.

dec
oct
hex

Diese Flags legen die Konvertierungsbasis eines Ganzzahlwertes fest. Die Konvertierungsbasis ist 10 (dezimal), wenn *dec* gesetzt ist. Sind hingegen *oct* oder *hex* gesetzt,so werden Umwandlungen im oktalen bzw. hexadezimalen System vorgenommen. Ist keines der drei Flags gesetzt, erfolgen Einfügungen dezimal, während Extraktionen entsprechend den lexikalischen Konventionen für Ganzzahlkonstanten in C++ interpretiert werden. Diese Felder werden gemeinsam durch das static-Element *ios::basefield* identifiziert. Die Manipulatoren *hex*, *dec* und *oct* können auch zum Setzen der Konvertierungsbasis verwendet werden (siehe "Vordefinierte Manipulatoren" in diesem Abschnitt).

showbase

Wenn *showbase* gesetzt ist, werden Einfügungen in eine externe Form umgewandelt,die entsprechend den lexikalischen Konventionen für Ganzzahlkonstanten in C++gelesen werden kann. Dies bedeutet, daß vor Oktalzahlen das Zeichen ’0’ erscheint,während Hexadezimalzahlen die Zeichenfolge ’0x’ (vgl. *uppercase*) vorangeht. Standardmäßig ist *showbase* nicht gesetzt.

showpos

Bei gesetztem *showpos* wird bei der dezimalen Umwandlung eines positiven Ganzzahlwertes ein Pluszeichen '+' eingefügt.

uppercase

Wenn *uppercase* gesetzt ist, wird der Großbuchstabe *X* für hexadezimale Ausgaben bei gesetztem *showbase* verwendet. Der Großbuchstabe *E* wird eingesetzt, um Gleitkommazahlen in der wissenschaftlichen Notation auszugeben.

showpoint

Im Ergebnis einer Gleitkomma-Umwandlung erscheinen Nullen und Dezimalpunkte am Ende der Zahlenangabe, wenn *showpoint* gesetzt ist.

scientific
fixed

Diese Flags legen das Format fest, in das ein Gleitkommawert zur Einfügung in einen Datenstrom umgewandelt wird.

- Bei gesetztem *scientific* wird der Wert in die wissenschaftliche Notation umgewandelt. Hierbei befindet sich eine Ziffer vor dem Dezimalpunkt, und die Anzahl der Ziffern hinter dem Dezimalpunkt entspricht der Angabe bei *precision* (siehe unten).Der Standardwert für *precision* ist 6.
- Bei gesetztem *uppercase* wird der Großbuchstabe *E* vor den Exponenten gesetzt,andernfalls erscheint hier der Kleinbuchstabe *e*.
- Bei gesetztem *fixed* wird der Wert in die dezimale Notation mit *precision* (standardmäßig 6) Ziffern hinter dem Dezimalpunkt umgewandelt.
- Sind weder *scientific* noch *fixed* gesetzt, wird der Wert in Abhängigkeit von seinem Inhalt wie folgt umgewandelt: Die wissenschaftliche Notation wird verwendet,wenn der aus der Umwandlung resultierende Exponent kleiner als -4 oder größer/gleich der Genauigkeitsangabe (*precision*) ist. Ansonsten wird die dezimale Notation verwendet.
- Bei nicht gesetztem *showpoint* werden Nullen am Ende des Ergebniswertes gelöscht, und ein Dezimalpunkt erscheint nur, wenn ihm eine Ziffer folgt.

scientific und *fixed* werden gemeinsam durch das *static*-Element *ios::floatfield* ausgedrückt.

unitbuf

Bei gesetztem *unitbuf* wird der Puffer durch *ostream::osfx()* nach jeder Einfügung"geleert". Die Einheitenpufferung stellt einen Kompromiss zwischen der gepufferten und der nicht gepufferten Ausgabe dar. Die Performance ist bei der Einheitenpufferung besser als ohne jede Pufferung, da im letztgenannten Fall für jedes ausgegebene Zeichen ein C-Laufzeitsystem-Aufruf ausgeführt wird. Die Einheitenpufferung führt nur bei jeder Einfügeoperation zu einem C-Laufzeitsystem-Aufruf, und der Benutzer muss *ostream::flush()* nicht aufrufen. Im BS2000 wird durch den Aufruf von *ostream::flush()* der Satz abgeschlossen und ein neuer Satz begonnen.

stdio

Ist *stdio* gesetzt, so werden *stdout* und *stderr* nach jedem Einfügen durch *ostream::osfx()* "geleert". Das bedeutet im BS2000, dass die aktuelle Zeile (Satz) beendet wird und die nächste Ausgabe in eine neue Zeile (Satz) erfolgt.

Die folgenden Funktionen setzen und verwenden die Formatflags und Variablen.

char oc=s.fill(char c)

Die Formatstatusvariable für das Füllzeichen wird auf *c* gesetzt und der vorherige Variablenwert geliefert. *c* wird als Füllzeichen verwendet, wenn dies notwendig ist(siehe *width()*). Das Standardfüllzeichen ist das Leerzeichen. Die Positionierung der Füllzeichen wird durch die Flags *left*, *right* und *internal* (siehe oben) festgelegt. Ein parametrisierter Manipulator, *setfill*, ist ebenfalls verfügbar, um das Füllzeichen zusetzen. Weitere Informationen finden Sie bei *manip* .

Hinweis

Das "Füllzeichen" hat keine Auswirkungen auf die Eingabe.

char c=s.fill()

Es wird die Formatstatusvariable für das Füllzeichen geliefert.

long l=s.flags()

Die aktuelle Formatflagangabe wird geliefert.

long l=s.flags(long f)

Alle Formatflags werden auf die in *f* spezifizierte Angabe zurückgesetzt, und die vorherigen Einstellungen werden geliefert.

int oi=s.precision(int i)

Die Formatstatusvariable für die Genauigkeit wird auf *i* gesetzt, und der zuvor darin enthaltene Werte wird geliefert. Diese Variable steuert die Anzahl signifikanter Stellen,die durch den Gleitkomma-Inserter eingefügt werden. Der Standardwert ist 6. Ein parametrisierter Manipulator, *setprecision*, ist ebenfalls zur Einstellung der Genauigkeit verfügbar. Weitere Informationen finden Sie bei *manip* .

int i=s.precision()

Es wird die Formatstatusvariable für die Genauigkeit geliefert.

long l=s.setf(long b)

In *s* werden die durch *b* markierten Formatflags gesetzt, und die Einstellungen vor der Änderung werden geliefert. Alle anderen Flags bleiben unverändert. Ein parametrisierter Manipulator, *setiosflags*, hat dieselbe Funktion. Weitere Informationen finden Sie bei *manip* .

long l=s.setf(long b, long f)

In *s* werden nur die Formatflags, die durch *f* spezifiziert werden, auf die in *b* angegebenen Werte zurückgesetzt. Die Einstellungen vor der Wertänderung werden geliefert.Die in *f* angegebenen Formatflags werden hierbei in *s* gelöscht und dann auf die in *b spezi*fizierten Werte gesetzt. Um die Konvertierungsbasis von *s* auf *hex* zu setzen, kann folgende Zeile verwendet werden:

```
s.setf(ios::hex, ios::basefield)
```

Hierdurch werden alle vorherigen Einstellungen auf oct oder dec gelöscht.

ios::basefield gibt an, dass die Konvertierungsbasis-Bits verändert werden sollen, während *ios::hex* den neuen Wert spezifiziert. *s.setf(0, f)* löscht alle durch *f* spezifizierten Bits. Der parametrisierte Manipulator *resetiosflags* hat dieselbe Aufgabe (siehe *manip*).

long l=s.unsetf(long b)

Die in *b* gesetzten Bits werden in *s* gelöscht, und es wird die Einstellung vor der Änderung geliefert.

int oi=s.width(int i)

Die Formatvariable für die Feldbreite wird auf *i* gesetzt und der Wert vor der Änderung geliefert.

Dies hat für Aus- und Eingabeströme unterschiedliche Bedeutungen:

- *Ausgabe*: Wenn die Feldbreite 0 ist (Standardeinstellung), werden nur so viele Zeichen eingefügt, wie zur Repräsentation des eingefügten Wertes notwendig sind.Bei einer Feldbreite ungleich 0 wird mindestens die angegebene Anzahl von Zeichen eingefügt. Wenn der eingefügte Wert weniger Zeichen als die Feldbreite zur Darstellung benötigt, werden Füllzeichen zur Auffüllung verwendet. Numerische Inserter schneiden Werte niemals ab.

  Wenn der einzufügende Wert also in *Feldbreite* Zeichen nicht vollständig darzustellen ist, werden mehr als *Feldbreite* Zeichen ausgegeben.

  Die Feldbreite wird immer als minimale Zeichenanzahl interpretiert. Eine maximale Zeichenanzahl kann nicht direkt angegeben werden.

  Die Formatvariable für die Feldbreite wird nach jeder Einfügung oder Extraktionauf den Standardwert 0 gesetzt.
- *Eingabe*: Eine Einstellung der Feldbreite bezieht sich nur auf die Extraktion von*char** und *unsigned char**. Weitere Informationen finden Sie bei *istream*. Wenn die Feldbreite ungleich 0 ist, wird sie als Größe des Feldes verwendet,und es werden nicht mehr als *Feldbreite-1* Zeichen extrahiert.

  Die Formatvariable für die Feldbreite wird nach jeder Extraktion auf den Standardwert (0) gesetzt.

Ein parametrisierter Manipulator (*setw*) ist verfügbar, der die Feldbreite setzt (siehe *manip*).

int i=s.width()

Die Formatvariable für die Feldbreite wird geliefert.

Benutzerdefinierte Formatflags

Bei der Ableitung von Klassen aus der Basisklasse *ios* stehen dem Benutzer mehrere Funktionen für zusätzlich benötigte Formatflags und Variablen zur Verfügung. Die beiden *static*-Elementfunktionen *ios::xalloc* und *ios::bitalloc* erlauben es, mehrere solcher Klassen ohne gegenseitige Interferenzen zu verwenden.

long b=ios::bitalloc()

Es wird ein *long*-Wert geliefert, in dem ein einziges Bit gesetzt ist, das zuvor nicht belegt war. Dies erlaubt dem Benutzer bei Bedarf die Anforderung zusätzlicher Flags.Eine solche Angabe kann als Argument an *ios:setf()* übergeben werden.

int i=ios::xalloc()

Ein zuvor nicht verwendeter Index in einem Feld aus Worten wird geliefert und kann als Formatstatusvariable für abgeleitete Klassen eingesetzt werden.

long & l=s.iword(int i)

Wenn *i* ein durch *ios::xalloc* belegter Index ist, liefert die Funktion *iword()* eine Referenz auf das *i* -te benutzerdefinierte Wort.

void*& vp=s.pword(int i)

Ist *i* ein durch *ios::xalloc* belegter Index, liefert *pword()* eine Referenz auf das *i*-te benutzerdefinierte Wort. *pword()* und *iword()* sind, abgesehen vom unterschiedlichen Typ des Rückgabewerts, gleich.

Weitere Elementfunktionen

streambuf* sb=s.rdbuf()

Es wird ein Zeiger auf das *streambuf*-Objekt geliefert, das beim Anlegen von *s* mit *s* verknüpft war.

static void ios::sync_with_stdio()

Hierdurch werden Probleme gelöst, die bei der gemeinsamen Verwendung von *stdio*und *iostream* auftreten. Beim ersten Aufruf werden die Standard-*iostream*-Objekte (*cin*, *cout*, *cerr* und *clog*; siehe *iosintro*) auf Datenströme zurückgesetzt, die *stdiobuf*-Objekte einsetzen. Hiernach können Ein-/Ausgaben über diese Ströme mit der Ein- und Ausgabe unter Verwendung der entsprechenden FILE-Objekte (*stdin*, *stdout* und *stderr*) vermischt werden, wobei eine korrekte Synchronisierung automatisch erfolgt. Die Funktion *sync_with_stdio()* setzt die Einheitenpufferung für *cout* und *cerr* (siehe *ios::unitbuf* und *ios::stdio* weiter oben im Text). Der Aufruf von *sync_with_stdio()* vermindert die Performance.
Da im BS2000 eine Ausgabe auf stdout (SYSOUT) auch einen Zeilenwechsel nach der Ausgabe der Daten bedeutet, ist das Verhalten bei C++-Ein-/Ausgaben im Falle einer Synchronisation mit der C-Ein-/Ausgabe nicht so wie erwartet: Jede C++-Ausgabe wird in eine eigene Zeile geschrieben. Bei nicht synchronisiertem Betrieb ist die Reihenfolge der Ausgaben undefiniert.

Hinweis

Die Einheitenpufferung für Standard-Ein-/Ausgabe-Dateien im BS2000 bewirkt, dass jede gelesene-/geschriebene Einheit den aktuellen Satz schließt und das Lesen/Schreiben des nächsten Satzes gestartet wird.

ostream * oosp=s.tie(ostream * osp)

Die Verknüpfungsvariable wird auf *osp* gesetzt, und der Variablenwert vor der Änderung wird zurückgegeben. Die Variable unterstützt das automatische "Leeren" von *ios*-Objekten. Wenn die Verknüpfungsvariable ungleich 0 ist und ein *ios*-Objekt weitereZeichen erfordert oder Zeichen enthält, die "verbraucht" werden können, wird das *ios*-Objekt geleert, auf das die Verknüpfungsvariable zeigt. Standardmäßig ist *cin* anfänglich mit *cout* verknüpft, so dass der Versuch einer Entnahme weiterer Zeichen aus der Standardeingabe zum Leeren der Standardausgabe führt. Zusätzlich sind auch *cerr*und *clog* standardmäßig mit *cout* verknüpft. Für alle anderen *ios*-Objekte ist die Verknüpfungsvariable standardmäßig auf 0 gesetzt.

ostream * osp=s.tie()

Der Wert der Verknüpfungsvariable wird geliefert.

Hinweis

Im C-Laufzeitsystem werden vor dem Lesen von stdin (SYSDTA) Textausgabedateien entleert. *iostream::tie* beeinflußt die Übergabe des C++-Pufferinhaltes an das C-Laufzeitsystem. Die Übertragung aller Informationen aus dem C-Laufzeitsystem-Puffer indie Datei erfolgt unabhängig von dem Wert der *tie*-Variablen.

Vordefinierte Manipulatoren

Einige recht komfortable Manipulatoren (das sind Funktionen, die ein Objekt in der Form *ios&*, *istream&* oder *ostream&* übernehmen und ihr Argument zurückgeben, siehe *manip*) sind im folgenden aufgelistet:

sr<<dec
sr>>dec

Hierdurch wird das Konvertierungsbasisflag in der Formatangabe auf 10 gesetzt.

sr<<hex
sr>>hex

Hierdurch wird das Konvertierungsbasisflag in der Formatangabe auf 16 gesetzt.

sr<<oct
sr>>oct

Hierdurch wird das Konvertierungsbasisflag in der Formatangabe auf 8 gesetzt.

sr>>ws

Zwischenraumzeichen werden extrahiert (siehe *istream*).

sr<<endl

Eine Zeile wird beendet, indem ein Neue-Zeile-Zeichen eingefügt und der Puffer geleert wird (siehe *ostream*).
Im BS2000 bedeutet das Schreiben eines Neue-Zeile-Zeichens in einer Textdateieinen Satzwechsel.

sr<<ends

Eine Zeichenkette wird beendet, indem ein Nullzeichen (0) angehängt wird (siehe *ostream*).

sr<<flush

sr wird geleert (siehe *ostream*).

In *manip* werden parametrisierte Manipulatoren beschrieben, die auf *ios*-Objekte angewendet werden: *setbase*, *setw*, *setfill*, *setprecision*, *setiosflags* und *resetiosflags*.

Das mit einem *ios*-Objekt assoziierte *streambuf*-Objekt kann auch durch andere Methoden als über das *ios*-Objekt manipuliert werden. So können Zeichen in einem Warteschlangen-ähnlichen *streambuf*-Objekt durch ein *ostream*-Objekt gespeichert und durch ein *istream*-Objekt entnommen werden. Aus Effizienzgründen kann ein Teil des Programmes auch direkt die *streambuf*-Operationen ansprechen, statt den "Umweg" über *ios* zuwählen. In den meisten Fällen muss sich ein Programm aber nicht mit dieser Möglichkeit beschäftigen, da ein *ios*-Objekt keine Informationen zum internen Status des *streambuf*-Objektes enthält. Wenn das *streambuf*-Objekt zwischen zwei Extraktionsoperationen neu positioniert wird, kann die Extraktion (Eingabe) anschließend normal fortgeführt werden.

BEISPIEL Das folgende Programm setzt einige Datenelemente der Klasse *ios* ein, um das Ausgabeformat von Ganz- und *double*-Zahlen in *cout* zu ändern:

```
#include <iostream.h>
#include <math.h>
void someoutput()
{
  int i;
  const int N = 12;
  for (i = 1; i < N; i += 2)
  {
    cout << "\t" << i << " " << pow( (double) i, (double) i) << endl;
  }
  cout << "\n";
}
int main()
{
  cout << "Standardformat :\n";
  someoutput();
  /* Standardformate für Ganzzahlen und double-Wert anzeigen    */
  cout.setf( ios::fixed, ios::floatfield);
  /* Ausgabeformate für float- und double-Werte auf fixed setzen*/
  cout << "float- bzw. double-Werte werden nun mit fixed ausgegeben :\n";
  someoutput();
  cout.setf( ios::oct, ios::basefield);
  /* Ausgabeformat für Ganzzahlen auf octal setzen              */
  cout << "Ganzzahlen werden nun oktal ausgegeben :\n";
  someoutput();
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
 Standardformat :
         1 1
         3 27
         5 3125
         7 823543
         9 3.8742e+08
         11 2.85312e+11
 float- bzw. double-Werte werden nun mit fixed ausgegeben :
         1 1.000000
         3 27.000000
         5 3125.000000
         7 823543.000000
         9 387420488.999998
         11 285311670610.995117
 Ganzzahlen werden nun oktal ausgegeben :
         1 1.000000
         3 27.000000
         5 3125.000000
         7 823543.000000
         11 387420488.999998
         13 285311670610.995117
%  CCM0998 Verbrauchte CPU-Zeit: 0.0066 Sekunden
```

Hinweis

Die Genauigkeit dieser Resultate hängt vom verwendeten Rechner ab.

BESONDERHEITEN

Das Kopieren von Datenströmen ist im *iostream*-Paket nicht möglich. An Objekte der Typen *istream_withassign*, *ostream_withassign* und *iostream_withassign* können aber Zuweisungen erfolgen. (Die Standardströme *cin*, *cout*, *cerr* und *clog* sind Elemente der"*withassign*"-Klassen, so daß Zuweisungen - wie bei *cin=inputfstream* - ausgeführt werden können.)

SIEHE AUCH

iosintro, *istream*, *manip*, *ostream*, *sbufprot*, *sbufpub*

4.5 istream Formatierte und unformatierte Eingabe

In diesem Abschnitt werden die *istream*-Elementfunktionen und verwandte Funktionen zur formatierten und unformatierten Eingabe beschrieben.

```
#include <iostream.h>

typedef long streamoff, streampos;

class ios
{
public:

   enum  seek_dir {beg, cur, end};
   enum  open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                              bin, tabexp};

   /* Flags zur Formatsteuerung */
   enum
   {

      skipws=01,
      left=02, right=04, internal=010,
      dec=020, oct=040, hex=0100,
      showbase=0200, showpoint=0400,
      uppercase=01000, showpos=02000,
      scientific=04000, fixed=010000,
      unitbuf=020000, stdio=040000

   };

   // siehe ios, dort finden Sie weitere Klassenelemente ...

};

class istream : virtual public ios

{

public:

   istream(streambuf*);
```

```
    virtual     ~istream();
    int         gcount();
    istream&    get(char* ptr, int len, char delim='\n');
    istream&    get(unsigned char* ptr,int len, char delim='\n');
    istream&    get(unsigned char&);
    istream&    get(char&);
    istream&    get(streambuf& sb, char delim ='\n');
    int         get();
    istream&    getline(char* ptr, int len, char delim='\n');
    istream&    getline(unsigned char* ptr, int len, char delim='\n');
    istream&    ignore(int len=1,int delim=EOF);
    int         ipfx(int need=0);
    int         peek();
    istream&    putback(char);
    istream&    read(char* s, int n);
    istream&    read(unsigned char* s, int n);
    istream&    seekg(streampos);
    istream&    seekg(streamoff, ios::seek_dir);
    int         sync();
    streampos   tellg();
    istream&    operator>>(char*);
    istream&    operator>>(char&);
    istream&    operator>>(short&);
    istream&    operator>>(int&);
    istream&    operator>>(long&);
    istream&    operator>>(float&);
    istream&    operator>>(double&);
    istream&    operator>>(unsigned char*);
    istream&    operator>>(unsigned char&);
    istream&    operator>>(unsigned short&);
    istream&    operator>>(unsigned int&);
    istream&    operator>>(unsigned long&);
    istream&    operator>>(streambuf*);
    istream&    operator>>(istream& (*)(istream&));
    istream&    operator>>(ios& (*)(ios&));

};

class istream_withassign : public istream

{

public:
```

```
                        istream_withassign();
    virtual             ~istream_withassign();
    istream_withassign& operator=(istream&);
    istream_withassign& operator=(streambuf*);
};
extern istream_withassign cin;

 istream&   ws(istream&);
 ios&       dec(ios&);
 ios&       hex(ios&);
 ios&       oct(ios&);
```

istream-Objekte unterstützen die Interpretation von Zeichen, die aus einem assoziierten *streambuf* ~~*Objekt entnommen werden. Dieser Vorgang wird üblicherweise als Eingabe*~~ oder Extraktionsoperation bezeichnet.

Für die folgenden Beschreibungen wird angenommen, dass

- *ins* ein *istream*-Objekt ist.
- *sb* vom Typ *streambuf** ist.

Konstruktoren und Zuweisung

istream(streambuf* sb)

> Die *ios*-Statusvariablen werden initialisiert, und der Puffer *sb* wird mit *istream* verknüpft.

istream_withassign()

> Es erfolgt keine Initialisierung. Das istream_withassign-Objekt ist nicht verwendbar, solange es nicht durch eine Zuweisung initialisiert wurde.

```
istream_withassign inswa;
streambuf * sb;
inswa=sb
```

> *sb* wird mit *inswa* verknüpft, und der gesamte Status von *inswa* wird initialisiert.

```
istream_withassign inswa;
inswa=ins
```

> *ins.rdbuf()* wird mit *inswa* verknüpft und der gesamte Status von *inswa* initialisiert.

Eingabepräfixfunktion

int i = ins.ipfx(int need)

Wenn der Fehlerstatus von *ins* ungleich 0 ist, wird direkt der Wert 0 geliefert. Falls notwendig (auch dann wenn der Fehlerstatus ungleich 0 ist), wird jedes mit *ins* verknüpfte *ios*-Objekt geleert (siehe Beschreibung von *ios::tie()* im Abschnitt zu *ios*).Das "Leeren" wird als notwendig erachtet, wenn entweder *need==0* ist oder wenn weniger als *need* Zeichen direkt verfügbar sind. Wenn *ios::skipws* in *ins.flags()* gesetzt und *need* gleich 0 ist, wird führender Zwischenraum aus *ins* extrahiert.

ipfx() liefert den Wert 0, wenn ein Fehler beim Überspringen des Zwischenraumes auftritt. Im anderen Fall wird ein Wert ungleich 0 zurückgegeben.

Die Funktionen für die formatierte Eingabe rufen *ipfx(0)* auf, während die Funktionen fürdie unformatierte Eingabe den Aufruf *ipfx(1)* verwenden (siehe unten).

Funktionen für die formatierte Eingabe (Extraktoren)

istream ins; ins>>x

ipfx(0) wird aufgerufen. Wenn das Ergebnis ungleich 0 ist, werden Zeichen aus *ins* extrahiert und entsprechend dem Typ von *x* umgewandelt. Der umgewandelte Wert wird in *x* gespeichert. Ein Fehler wird gemeldet, indem der Fehlerstatus von *ins*gesetzt wird. Bei gesetztem *ios::failbit* entsprachen die in *ins* enthaltenen Zeichen nicht dem geforderten Typ. Ein gesetztes *ios::badbit* zeigt an, daß die Zeichenextraktion fehlgeschlagen ist. Es wird immer *ins* geliefert.

Die Einzelheiten der Umwandlung hängen von den Werten der Formatstatusflags und Variablen (siehe *ios*) von *ins* und vom Typ von *x* ab. Extraktoren sind für die folgenden Typen definiert; die Umwandlungsregeln werden hier ebenfalls beschrieben.

x kann einer der folgenden Datentypen sein:

char*, unsigned char*

Die Zeichen werden in dem Feld abgelegt, auf das *x* zeigt, bis in *ins* ein Zwischenraumzeichen auftritt. Das beendende Zwischenraumzeichen verbleibt in *ins*. Wenn*ins.width()* ungleich 0 ist, wird der resultierende Wert als Größenangabe für das Feld verwendet, und es werden nicht mehr als *ins.width()-1* Zeichen extrahiert. Ein beendendes Nullzeichen (0) wird immer gespeichert (auch wenn aufgrund des Fehlerstatus von *ins* keine anderen Operationen vorgenommen werden). *ins.width()* wird auf 0 zurückgesetzt.

char&, unsigned char&

Ein Zeichen wird extrahiert und in *x* gespeichert.

short&, unsigned short&,
int&, unsigned int&,
long&, unsigned long&

Die Zeichen werden extrahiert und in einen Ganzzahlwert umgewandelt. Die Umwandlung erfolgt entsprechend den Formatflags von *ins*. Die umgewandelten Zeichen werden in *x* gespeichert. Das erste Zeichen kann ein Vorzeichen (+ oder -) sein. Hiernach wird die Umwandlung entweder oktal, dezimal oder hexadezimal ausgeführt, abhängig davon ob *ios::oct*, *ios::dec* oder *ios::hex* in *ins.flags()* gesetzt ist. Die Umwandlung wird durch das erste Zeichen beendet, das keine Ziffer ist.Oktale Ziffern sind die Zeichen 0 bis 7, dezimale Ziffern setzen sich aus den oktalen Ziffern plus den Zeichen 8 und 9 zusammen, und hexadezimale Ziffern bestehen aus den dezimalen Ziffern zuzüglich der Buchstaben *a* bis *f* (entweder in Klein- oder Großschreibung). Wenn keine Konvertierungsbasis in den Formatflags gesetzt ist, wird die Zahl entsprechend den lexikalischen Konventionen von C++ interpretiert: Es wird eine hexadezimale Umwandlung vorgenommen, wenn die ersten Zeichen (nach einem optionalen Vorzeichen) *0x* oder *0X* sind. Ist das erste Zeichen *0*,wird eine oktale Umwandlung ausgeführt. In allen anderen Fällen kommt es zu einer dezimalen Umwandlung. *ios::failbit* wird gesetzt, wenn keine Ziffern verfügbar

sind (die Ziffer *0* in *0x* oder *0X* bei einer Hexadezimalumwandlung zählt hierbei nicht als Ziffer).

float&, double&

Die Zeichen werden entsprechend der Syntax von C++ für *float*- oder *double*-Werte umgewandelt und das Ergebnis in *x* gespeichert. *ios::failbit* wird gesetzt,wenn in *ins* keine Ziffern verfügbar sind oder wenn die Angabe nicht mit einer korrekt gebildeten Gleitkommazahl beginnt.

Hinweis

skipws sollte während der Extraktion numerischer Werte gesetzt bleiben. Anderenfalls kann ein Fehler auftreten.

Typ und Name der Extraktionsoperationen wurde gewählt, um eine komfortable Syntax für Folgen von Eingabeoperationen bereitzustellen. Das Operator-Overloading in C++ermöglicht die Deklaration von Extraktionsfunktionen für benutzerdefinierte Klassen. Die Operationen können die gleiche Syntax wie die hier beschriebenen Elementfunktionen aufweisen.

ins>>sb

Wenn *ios.ipfx(0)* einen Wert ungleich 0 liefert, werden Zeichen aus *ios* extrahiert undin *sb* eingefügt. Die Extraktion wird beendet, wenn das Dateiende EOF erreicht ist. Es wird immer *ins* zurückgegeben.

Funktionen für die unformatierte Eingabe

Diese Funktionen rufen *ipfx(1)* auf und werden nur weiter abgearbeitet, wenn das Ergebnis des Aufrufs ungleich 0 ist:

istream * insp=&ins.get(char * ptr, int len, char delim)

Es werden Zeichen extrahiert und in einem Byte-Feld, das bei *ptr* beginnt und *len*byte lang ist, gespeichert. Die Extraktion wird beendet, wenn das Zeichen *delim* gefunden wird (*delim* wird in *ins* belassen und nicht gespeichert), wenn in *ins* keine weiteren Zeichen vorliegen oder wenn im Feld nur noch ein freies Byte verbleibt. *get()* speichert immer ein abschließendes Nullzeichen, auch wenn aufgrund des Fehlerstatus keine Zeichen aus *ins* entnommen wurden. *ios::failbit* wird nur gesetzt, wenn *get()* das Ende der Datei erreicht, bevor ein Zeichen gespeichert wurde.

istream * insp=&ins.get(char & c)

Es wird ein einzelnes Zeichen extrahiert und in *c* gespeichert.

istream * insp=&ins.get(streambuf & sb, char delim)

Zeichen werden aus *ins.rdbuf()* extrahiert und in *sb* gespeichert. Der Vorgang wirdbeendet, wenn das Dateiende erreicht ist, ein Speicherversuch in *sb* fehlschlägt oderdas Zeichen *delim* (verbleibt in *ins*) gefunden wird. *ios::failbit* wird gesetzt, wenn die Funktion wegen eines fehlerhaften Speicherversuches in *sb* abgebrochen wird.

int i=ins.get()

Ein Zeichen wird extrahiert und geliefert. *i* ist EOF, wenn bei der Zeichenextraktion das Dateiende erreicht wird. *ios::failbit* wird nie gesetzt.

istream * insp=&ins.getline(char * ptr, int len, int delim)

Diese Funktion entspricht der Funktion *ins.get(char * ptr*, *int len*, *char delim)* mit der Ausnahme, dass auch das beendende *delim*-Zeichen aus *ins* entnommen wird.Wenn *delim* nach der Extraktion von genau *len* Zeichen auftritt, wird die Beendigung für das vollständig gefüllte Feld ausgeführt, und *delim* verbleibt in *ins*.

istream * insp=&ins.ignore(int n, char d)

Es werden bis zu *n* Zeichen extrahiert und verworfen. Die Extraktion wird vorzeitig beendet, wenn das Zeichen *d* entnommen wird oder das Ende der Datei erreicht ist.Wenn *d* gleich EOF ist, kann hierdurch niemals eine Beendigung hervorgerufen werden.

istream * insp=&ins.read(char * ptr, int n)

Es werden *n* Zeichen extrahiert und in dem bei *ptr* beginnenden Feld abgelegt. Beim Erreichen des Dateiendes vor der Extraktion von *n* Zeichen speichert *read* den bislang extrahierten Teil und setzt *ios::failbit*. Die Anzahl der extrahierten Zeichen kann über *ins.gcount()* bestimmt werden.

Weitere Elementfunktionen

int i=ins.gcount()

Es wird die Anzahl der Zeichen, die von der letzten Funktion für unformatierte Eingabe extrahiert wurden, geliefert. Funktionen für die formatierte Eingabe können Funktionen für die unformatierte Eingabe aufrufen und dadurch den gelieferten Wert ändern.

int i=ins.peek()

Zu Beginn der Funktionsausführung wird *ins.ipfx(1)* aufgerufen. Wenn der Aufruf den Wert 0 liefert oder das Dateiende von *ins* erreicht ist, wird EOF zurückgegeben. Im anderen Fall wird das nächste Zeichen geliefert, ohne es dabei zu extrahieren.

istream * insp=ins.putback(char c)

Es wird versucht, den Zeiger *get* von *ins.rdbuf()* zurückzuschieben, um das Zeichen*c* erst zu einem späteren Zeitpunkt zu lesen. *c* muß das Zeichen vor dem Zeiger sein.(Sofern keine andere Operation auf *ins.rdbuf()* angewendet wurde, ist *c* das letzte aus *ins* entnommene Zeichen.) Ist dies nicht der Fall, so ist das Verhalten der Funktion nicht definiert. *putback()* kann fehlschlagen (in diesem Fall wird der Fehlerstatus gesetzt). Obwohl es sich um eine Elementfunktion von *istream* handelt, entnimmt *putback()* niemals Zeichen, so daß auch kein Aufruf von *ipfx()* ausgeführt wird. Wenn der Fehlerstatus ungleich 0 ist, gibt die Funktion die Kontrolle ohne weitere Operationen zurück.

int i=ins.sync()

Die internen Datenstrukturen und die externe Zeichenquelle werden synchronisiert. Die virtuelle Funktion *ins.rdbuf()->sync()* wird aufgerufen, so daß die Details des Funktionsaufrufs von der abgeleiteten Klasse abhängig sind. Beim Auftreten eines Fehlers wird EOF geliefert.

ins>>manip

Diese Zeile entspricht *manip(ins)*. Syntaktisch weist die Zeile zwar das Erscheinungsbild einer Extraktoroperation auf, in semantischer Hinsicht wird aber eine beliebige Operation (statt der Umwandlung einer Zeichenfolge und Speicherung des Ergebnisses in *manip*) ausgeführt. Der vordefinierte Manipulator *ws* wird im folgenden noch beschrieben.

Elementfunktionen für die Positionierung

istream& insp=ins.seekg(streamoff off, ios::seek_dir dir)

Der Zeiger *get* von *ins.rdbuf()* wird neu positioniert. Weitere Erläuterungen zur Positionierung finden Sie im Abschnitt zu *sbufpub* .

istream& insp=ins.seekg(streampos pos)

Der Zeiger *get* von *ins.rdbuf()* wird neu positioniert. Im Abschnitt zu *sbufpub* finden Sie eine Erläuterung der Positionierung.

streampos pos=ins.tellg()

Die aktuelle Position des Zeigers *get* von *ios.rdbuf()* wird geliefert. Eine Erläuterung der Positionierung finden Sie im Abschnitt zu *sbufpub* .

Manipulatoren

ins>>ws

Zwischenraumzeichen werden extrahiert.

ins>>dec

Das Formatflag für die Konvertierungsbasis wird auf 10 gesetzt. Weitere Informationen finden Sie im Abschnitt zu *ios* .

ins>>hex

Das Formatflag für die Konvertierungsbasis wird auf 16 gesetzt.Weitere Informationen finden Sie im Abschnitt zu *ios* .

ins>>oct

Das Formatflag für die Konvertierungsbasis wird auf 8 gesetzt. Weitere Informationen finden Sie im Abschnitt zu *ios* .

BEISPIEL Das folgende Programm liest eine Textzeile ein, die daraufhin in umgekehrter Reihenfolge ausgegeben wird.

```
#include <iostream.h>
const int N = 80;
char a[ N];                /* Textpuffer */
int main()
{
  int i;
  cout << " Bitte eine Textzeile eingeben :\n";
  cin.unsetf(ios::skipws);
  cin.getline(text, N);      /* holt höchstens N Zeichen */
  i = cin.gcount() - 1;
  while (i)
  {
    cout << text [--i];      /* Gibt Textzeile in umgekehrter */
                             /* Reihenfolge aus */
  }
  return 0;                  /* Rückgabe bei erfolgreicher Ausführung */
}
```

Das Ergebnis der Programmausführung ist:

```
Bitte eine Textzeile eingeben :
EIN
NIE
%  CCM0998 Verbrauchte CPU-Zeit: 0.0024 Sekunden
```

BESONDERHEITEN

Bei der Umwandlung von Ganzzahlen erfolgt keine Überlaufprüfung.

SIEHE AUCH

ios, *manip*, *sbufpub*

4.6 manip iostream-Manipulation

In diesem Abschnitt wird die Verwendung von Manipulatoren in Verbindung mit *iostream*-Objekten beschrieben.

```
#include <iostream.h>
#include <iomanip.h>

IOMANIPdeclare(T);

class SMANIP(T)
{
public:

   SMANIP(T)(ios& (*)(ios&,T), T);
   friend istream& operator>>(istream&, const SMANIP(T)&);
   friend ostream& operator<<(ostream&, const SMANIP(T)&);

};

class SAPP(T)
{
public:

   SAPP(T)(ios& (*)(ios&,T));
   SMANIP(T) operator()(T);

};

class IMANIP(T)
{
public:

   IMANIP(T)(istream& (*)(istream&,T),T);
   friend istream& operator>>(istream&, const IMANIP(T)&);

};

class IAPP(T)
{
public:

   IAPP(T)(istream& (*)(istream&,T));
   IMANIP(T) operator()(T);

};

class OMANIP(T)
{
public:

   OMANIP(T)(ostream& (*)(ostream&,T),T);
   friend  ostream& operator<<(ostream&, const OMANIP(T)&);

};
```

```
class OAPP(T)
{
public:

   OAPP(T)(ostream& (*)(ostream&,T));
   OMANIP(T) operator()(T);

};

class IOMANIP(T)
{
public:

   IOMANIP(T)(iostream& (*)(iostream&,T),T);
   friend istream& operator>>(iostream&, const IOMANIP(T)&);
   friend ostream& operator<<(iostream&, const IOMANIP(T)&);

};

class IOAPP(T)
{
public:

   IOAPP(T)(iostream& (*)(iostream&,T));
   IOMANIP(T) operator()(T);

};

IOMANIPdeclare(int);
IOMANIPdeclare(long);

SMANIP(int)      setbase(int);
SMANIP(long)   resetiosflags(long);
SMANIP(int)      setfill(int);
SMANIP(long)   setiosflags(long);
SMANIP(int)      setprecision(int);
SMANIP(int)      setw(int w);
```

Manipulatoren sind Werte, die in Datenströme eingefügt oder daraus entnommen werden können, um einen bestimmten Effekt (nicht nur den des Einfügens oder Extrahierens von Werten) zu erzielen. Zu diesem Zweck ist eine komfortable Syntax verfügbar.Manipulatoren ermöglichen es, einen Funktionsaufruf in einen Ausdruck einzubetten, der mehrere Einfügungen oder Extraktionen umfasst. Der vordefinierte Manipulator *flush* für *ostream*-Objekte kann beispielsweise wie folgt eingesetzt werden, um *cout* zu leeren:

```
cout << flush
```

Einige *iostream*-Klassen stellen Manipulatoren bereit (siehe *ios*, *istream* und *ostream*). *flush* ist ein recht einfacher Manipulator. Einige Manipulatoren übernehmen Argumente (wie die vordefinierten *ios*-Manipulatoren *setfill* und *setw*, die weiter unten beschrieben werden). Die Include-Datei <*iomanip.h*> stellt Makrodefinitionen zur Verfügung, die der Programmierer für die Definition neuer parametrisierter Manipulatoren verwenden kann.

Im Idealfall werden die mit Manipulatoren verknüpften Typen als "Schablonen" parametrisiert. Die in *<iomanip.h>* definierten Makros simulieren Schablonen. *IOMANIPdeclare(T)* deklariert die verschiedenen Klassen und Operatoren. (Der gesamte Code wird in Inline-Form deklariert, so dass keine eigenständigen Definitionen erforderlich sind.) Jeder der anderen Typnamen wird zur Konstruktion von Namen verwendet und muß daher ein einzelner Bezeichner sein. Jedes der anderen Makros erfordert ebenfalls einen Bezeichner und wird zu einem Namen erweitert.

Für die folgenden Beschreibungen wird angenommen, dass

- *t* ein Typname (*T*) ist.
- *s* ein *ios*-Objekt ist.
- *i* ein *istream*-Objekt ist.
- *o* ein *ostream*-Objekt ist.
- *io* ein *iostream*-Objekt ist.
- *f* vom Typ *ios& (*)(ios&, T)* ist.
- *isf* vom Typ *istream& (*)(istream&, T)* ist.
- *osf* vom Typ *ostream& (*)(ostream&, T)* ist.
- *iof* vom Typ *iostream& (*)(iostream&, T)* ist.
- *n* ein *int*-Wert ist.
- *l* ein *long*-Wert ist.

```
s<<SMANIP(T)( (ios& (*)(ios&, T)) f, T t)
s>>SMANIP(T)( (ios& (*)(ios&, T)) f, T t)
s<<SAPP(T)( (ios& (*)(ios&, T)) f)(T t)
s>>SAPP(T)( (ios& (*)(ios&, T)) f)(T t)
```

> Es wird *f(s,t)* geliefert, wobei *s* der linke Operand des Einfüge- oder Extraktoroperators (z.B. *s*, *i*, *o* oder *io*) ist.

```
i>>IMANIP(T)( (istream& (*)(istream&, T)) isf, T t)
i>>IAPP(T)( (istream& (*)(istream&, T)) isf) (T t)
```

> Es wird *isf(i,t)* geliefert.

```
o<<OMANIP(T)( (ostream& (*)(ostream&, T)) osf,T t)
o<<OAPP(T)( (ostream& (*)(ostream&, T)) osf) (T t)
```

> *osf(o,t)* wird geliefert.

```
io<<IOMANIP(T)( (iostream& (*)(iostream&, T)) iof,T t)
io>>IOMANIP(T)( (iostream& (*)(iostream&, T)) iof,T t)
io<<IOAPP(T)( (iostream& (*)(iostream&, T)) iof) (T t)
io>>IOAPP(T)( (iostream& (*)(iostream&, T)) iof) (T t)
```

> *iof(io,t)* wird geliefert.

<iomanip.h> enthält zusätzlich einige Manipulatoren, die ein *int*- oder *long*-Argument übernehmen. Diese Manipulatoren beschäftigen sich mit der Veränderung des Formatstatus eines Datenstroms; weitere Informationen finden Sie bei *ios*.

o<<setbase(int n)
i>>setbase(int n)

Das Konvertierungsbasisflag wird auf *n* gesetzt.

o<<resetiosflags(long l)
i>>resetiosflags(long l)

Die durch *l* spezifizierten Formatbit werden im Datenstrom (*o* oder *i*) gelöscht (hierdurch wird *o.setf(0, l)* oder *i.setf(0, l)* aufgerufen).

o<<setfill(int n)
i>>setfill(int n)

Setzt die Füllzeichen des Datenstroms (*o* oder *i*) auf *n*.

o<<setiosflags(long l)
i>>setiosflags(long l)

Die durch *l* markierten Formatflags werden im Datenstrom (*o* oder *i*) gesetzt (dies führt zu einem Aufruf von *o.setf(l)* oder *i.setf(l)*).

o<<setprecision(int n)
i>>setprecision(int n)

Die Genauigkeit des Datenstroms (*o* oder *i*) wird auf *n* gesetzt.

o<<setw(int n)
i>>setw(int n)

Die Feldbreite eines Datenstroms (linksseitiger Operand: *o* oder *i*) wird auf *n* gesetzt.

BEISPIEL Das folgende Programm veranschaulicht die Verwendung von Manipulatoren (wie *setw*), die eine Ausgabe global ändern, indem *private*-Datenelemente in *cout* modifiziert werden:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
void testline(const char * const p)
{
  /* Parameterzeichenkette in cout einfügen und Feldbreite von      */
  /* cout auf die doppelte Länge der Zeichenkette setzen            */
  int N = 2 * strlen(p);
  cout << setw(N) ;
  cout << p;
}
void someoutput(const char * const p, const char * const q)
{
  /* Gegeben sind eine Zeichenkette p und eine Zeichenkette q, die  */
  /* eine Liste der Füllzeichen enthält; die Zeichenkette p soll    */
  /* in verschiedenen Füllzeichen-Kontexten angezeigt werden         */
  int i;
  int M = strlen(q);
  for (i = 0; i < M; ++i)
  {
    cout << setfill(q[i]);
    testline(p);
  }
}
int main()
{
  someoutput( "Eine Testzeichenkette\n", ".,!$%&*()");
  /* Beachten Sie, daß die Ausgabe von Zeichenkette rechtsbündig    */
  /* erfolgt */
  return 0;
}
```

<*string.h*> muss zur Bereitstellung des Funktionsprototyps von *strlen()* und <*iomanip.h*> muss zur Bereitstellung der Prototypen von *setw()* und *setfill()* eingebunden werden.

Das Ergebnis der Programmausführung ist:

```
.....................Eine Testzeichenkette
,,,,,,,,,,,,,,,,,,,,Eine Testzeichenkette
!!!!!!!!!!!!!!!!!!!!Eine Testzeichenkette
$$$$$$$$$$$$$$$$$$$$Eine Testzeichenkette
%%%%%%%%%%%%%%%%%%%%Eine Testzeichenkette
&&&&&&&&&&&&&&&&&&&&Eine Testzeichenkette
**********************Eine Testzeichenkette
((((((((((((((((((((Eine Testzeichenkette
))))))))))))))))))))Eine Testzeichenkette
%  CCM0998 Verbrauchte CPU-Zeit: 0.0006 Sekunden
```

SIEHE AUCH

ios, *istream*, *ostream*

4.7 ostream Formatierte und unformatierte Ausgabe

In diesem Abschnitt werden die *ostream*-Funktionen zur formatierten und unformatierten Ausgabe beschrieben.

```
#include <iostream.h>

typedef long streamoff, streampos;

class ios
{
public:

   enum seek_dir {beg, cur, end};
   enum open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                              bin, tabexp};

   enum

   {

      skipws=01,
      left=02, right=04, internal=010,
      dec=020, oct=040, hex=0100,
      showbase=0200, showpoint=0400,
      uppercase=01000, showpos=02000,
      scientific=04000, fixed=010000,
      unitbuf=020000, stdio=040000

   };

   // siehe ios ; dort werden weitere Klassenelemente beschrieben

} ;

class ostream : virtual public ios

{

public:
```

```
                ostream(streambuf*);

    virtual     ~ostream();

    ostream&    flush();
    int         opfx();
    void        osfx();
    ostream&    put(char);
    ostream&    seekp(streampos);
    ostream&    seekp(streamoff, ios::seek_dir);
    streampos   tellp();
    ostream&    write(const char* ptr, int n);
    ostream&    write(const unsigned char* ptr, int n);
    ostream&    operator<<(const char*);
    ostream&    operator<<(char);
    ostream&    operator<<(short);
    ostream&    operator<<(int);
    ostream&    operator<<(long);
    ostream&    operator<<(float);
    ostream&    operator<<(double);
    ostream&    operator<<(unsigned char);
    ostream&    operator<<(unsigned short);
    ostream&    operator<<(unsigned int);
    ostream&    operator<<(unsigned long);
    ostream&    operator<<(void*);
    ostream&    operator<<(streambuf*);
    ostream&    operator<<(ostream& (*)(ostream&));
    ostream&    operator<<(ios& (*)(ios&));

};

class ostream_withassign : public ostream

{

public:

                            ostream_withassign();

    virtual                 ~ostream_withassign();

    ostream_withassign&     operator=(ostream&);
    ostream_withassign&     operator=(streambuf*);

};

 extern ostream_withassign cout;
 extern ostream_withassign cerr;
 extern ostream_withassign clog;
```

```
ostream&    endl(ostream&);
ostream&    ends(ostream&);
ostream&    flush(ostream&);
ios&        dec(ios&);
ios&        hex(ios&);
ios&        oct(ios&);
```

---

ostream-Objekte unterstützen das Einfügen (Speichern) in einem *streambuf*-Objekt. Man spricht in diesem Zusammenhang auch von *Ausgabeoperationen*. Die *ostream*-Elementfunktionen und die damit verknüpften Funktionen werden nachstehend beschrieben.

Bei den anschließenden Erläuterungen wird vorausgesetzt, dass

- *outs* ein *ostream*-Objekt ist.
- *outswa* ein *ostream_withassign*-Objekt ist.
- *outsp* vom Typ *ostream** ist.
- *c* vom Typ *char* ist.
- *ptr* vom Typ *char** oder *unsigned char**ist.
- *sb* vom Typ *streambuf** ist.
- *i* und *n int*-Werte sind.
- *pos* vom Typ *streampos* ist.
- *off* vom Typ *streamoff* ist.
- *dir* vom Typ *seek_dir* ist.
- *manip* eine Funktion vom Typ *ostream& (*) (ostream&)* ist.

Konstruktoren und Zuweisung

ostream(streambuf * sb)

> Die *ios*- und *ostream*-Statusvariablen werden initialisiert und der Puffer *sb* mit dem *ostream*-Objekt verknüpft.

ostream_withassign()

> Es wird keine Initialisierung vorgenommen. Hierdurch kann eine *static*-Variable dieses Typs (z. B. *cout*) vor ihrer Konstruktion eingesetzt werden. Dabei wird vorausgesetzt, dass der Variable zuvor ein Wert zugewiesen wurde.

outswa=sb

> *sb* wird mit *outswa* verknüpft und die gesamte Statusangabe von *outswa* initialisiert.

outswa=outs

> *outs.rdbuf()* wird mit *outswa* verknüpft und die gesamte Statusangabe von *outswa* initialisiert.

Ausgabepräfixfunktion

int outs.opfx()

Wenn der Fehlerstatus von *outs* ungleich 0 ist, wird direkt der Wert 0 zurückgegeben.Wenn *outs.tie()* einen Wert ungleich 0 liefert, werden die Puffer der mit outs verknüpften ios-Objekte geleert. In allen anderen Fällen wird ein Wert ungleich 0 geliefert.

Ausgabesuffixfunktion

void osfx()

Vor dem Rücksprung von der Inserter-Abarbeitung werden "Suffix"-Aktionen ausgeführt. Wenn *ios::unitbuf* gesetzt ist, leert die Funktion *osfx()* das *ostream*-Objekt. Beigesetztem *ios::stdio* leert die Funktion *osfx()* sowohl *stdout* als auch *stderr*. Im BS2000 bedeutet das Leeren (*flush*) von stdio und stderr unter anderem, dass die aktuelle Zeile (Satz) beendet wird. Die nächste Ausgabe erfolgt in die nächste Zeile.

osfx() wird von allen vordefinierten Insertern aufgerufen und sollte auch von benutzerdefinierten Insertern nach der direkten Manipulation von *streambuf*-Objekten eingesetzt werden. Binäre Ausgabefunktionen rufen *osfx()* nicht auf.

Funktionen für die formatierte Ausgabe (Inserter)

outs<<x

Es wird zunächst *outs.opfx()* aufgerufen; wenn diese Funktion den Wert 0 liefert, wird keine weitere Operation ausgeführt. Im anderen Fall wird eine Zeichenfolge, die *x* darstellt, in *outs.rdbuf()* eingefügt. Fehler werden durch Setzen des Fehlerstatus von *outs* angezeigt. Es wird immer *outs* zurückgegeben.
x wird in eine Folge von Zeichen (deren Darstellung oder Repräsentation) umgewandelt. Die hierbei geltenden Regeln sind vom Typ von *x* und den Formatstatusflags und Variablen in *outs* abhängig (siehe *ios*). Inserter werden für die folgenden Datentypen (mit den angegebenen Umwandlungsregeln) definiert:

char*

Die Repräsentation ist eine Folge von Zeichen bis zu (aber nicht einschließlich) der beendenden 0 der Zeichenkette, auf die *x* gerichtet ist.

alle Ganzzahltypen
(außer char und unsigned char)

- Wenn *x* positiv ist, enthält die Repräsentation eine Folge von dezimalen, oktalen oder hexadezimalen Ziffern ohne führende Nullen. Welche Notation verwendet wird, hängt von den gesetzten Formatflags in *ios* (*ios::dec*, *ios::oct* oder *ios::hex*) ab. Wenn keines der Flags gesetzt ist, wird standardmäßig die Dezimalnotation verwendet.
- Wenn *x* gleich 0 ist, besteht die Repräsentation aus einem einzelnen Nullzeichen(0).
- Wenn *x* negativ ist, führt die Dezimalumwandlung zu einer Darstellung, bei der vor den dezimalen Stellen ein Minuszeichen (-) erscheint.
- Wenn *x* positiv und *ios::showpos* gesetzt ist, führt die dezimale Umwandlung zur Anzeige eines Pluszeichens (+), dem dezimale Stellen folgen. Die anderen Umwandlungen behandeln alle Werte als vorzeichenlos. Wenn *ios::showbase* in den Formatflags von *ios* gesetzt ist, enthält die hexadezimale Repräsentation vor den hexadezimalen Stellen die Zeichenfolge 0x oder 0X (letzteres bei gesetztem *ios::uppercase*). Bei gesetztem *ios::showbase* beginnt die oktale Darstellung mit einer führenden 0.

void*

Zeiger werden in Ganzzahlwerte und anschließend in hexadezimale Werte umgewandelt, so als ob *ios::showbase* gesetzt wäre.

float, double

Die Argumente werden entsprechend den aktuellen Werten von *outs.precision()*, *outs.width()* und den Formatflags *ios::scientific*, *ios::fixed* und *ios::uppercase* (siehe *ios*) von *out* umgewandelt. Der Standardwert für *outs.precision()* ist 6. Wenn weder *ios::scientific* noch *ios::fixed* gesetzt sind, wird für die Darstellung die Festpunktnotation oder die wissenschaftliche Notation in Abhängigkeit vom Wert *x* gewählt.

char, unsigned char

Es ist keine spezielle Umwandlung notwendig.

Nachdem die Darstellung bestimmt ist, wird das Auffüllen behandelt. Wenn *outs.width()* größer als 0 ist und die Repräsentation weniger als *outs.width()* Zeichen umfaßt, werden genügend *outs.fill()*-Zeichen hinzugefügt, um die Gesamtzahl derZeichen auf *ios.width()* aufzufüllen. Wenn *ios::left* in den Formatflags von *ios* gesetzt ist, wird die Zeichenfolge linksbündig ausgerichtet, und die Füllzeichen werden am Ende der Zeichenkette angehängt. Bei gesetztem *ios::right* werden die notwendigenFüllzeichen vor der Zeichenfolge eingefügt. Ist *ios::internal* gesetzt, erfolgt das Auffüllen zwar hinter einem vorangestellten Vorzeichen oder einer Basisindikation, aber noch vor den Zeichen, die den Wert darstellen. *ios.width()* wird auf 0 zurückgesetzt, während alle anderen Formatvariablen unverändert bleiben. Die resultierende Folge (Füllzeichen plus Wertrepräsentation) wird in *outs.rdbuf()* eingefügt.

outs<<sb

Wenn *outs.opfx()* einen Wert ungleich 0 liefert, wird eine Zeichenfolge in *outs.rdbuf()*eingefügt, die aus *sb* entnommen werden kann. Die Einfügung wird abgebrochen,wenn in *sb* keine weiteren Zeichen zur Verfügung stehen. Das Auffüllen unterbleibt,und es wird immer *outs* zurückgegeben.

Funktionen für die unformatierte Ausgabe

ostream * outsp=&outs.put(char c)

c wird in *outs.rdbuf()* eingefügt. Der Fehlerstatus wird gesetzt, wenn die Einfügung misslingt.

ostream * outsp=&outs.write(char * s, int n)

Es werden *n* Zeichen, beginnend bei *s*, in *outs.rdbuf()* eingefügt. In diesen Zeichen können auch Nullbytes enthalten sein, so dass *s* keine mit \0 beendete Zeichenkette sein muss.

Weitere Elementfunktionen

ostream * outsp=&outs.flush()

Zeichen werden beim Speichern in einem *streambuf*-Objekt nicht immer sofort "verbraucht" (indem Sie beispielsweise in eine externe Datei geschrieben werden). Die Verwendung von *flush()* führt dazu, dass gespeicherte, aber noch nicht "verbrauchte"Zeichen durch den Aufruf von *outs.rdbuf()->sync* verbraucht werden.
Im BS2000 bedeutet dies, dass die Zeichen an das C-Laufzeitsystem durchgereicht werden.

outs<<manip

Diese Zeile entspricht *manip(outs)*. Syntaktisch erscheint dies zwar wie eine Einfügeoperation, in semantischer Hinsicht handelt es sich aber um eine beliebige Operation(statt der Umwandlung von *manip* in eine Folge von Zeichen, wie dies durch die Einfügeoperatoren erfolgt).

Positionierungsfunktionen

ostream * outsp=&outs.seekp(streamoff off, ios::seek_dir dir)

Der Zeiger *put* von *outs.rdbuf()* wird neu positioniert. Weitere Informationen zum Positionieren finden Sie im Abschnitt zu *sbufpub* .

ostream * outsp=&outs.seekp(streampos pos)

Der Zeiger *put* von *outs.rdbuf()* wird neu positioniert. Weitere Informationen zum Positionieren finden Sie bei *sbufpub* .

streampos pos=outs.tellp()

Es wird die aktuelle Position des Zeigers *put* von *outs.rdbuf()* geliefert. Weitere Informationen zur Positionierung finden Sie im Abschnitt zu *sbufpub* .

Manipulatoren

outs<<endl

Eine Zeile wird beendet, indem ein Neue-Zeile-Zeichen eingefügt und der Puffer geleert wird. Das Neue-Zeile-Zeichen wird im BS2000 in einen Satzwechsel umgewandelt.

outs<<ends

Eine Zeichenkette wird beendet, indem ein Nullzeichen (0) angehängt wird.

outs<<flush

outs wird geleert.

outs<<dec

Das Formatflag für die Konvertierungsbasis wird auf 10 gesetzt (siehe *ios*).

outs<<hex

Das Formatflag für die Konvertierungsbasis wird auf 16 gesetzt (siehe *ios*).

outs<<oct

Das Formatflag für die Konvertierungsbasis wird auf 8 gesetzt (siehe *ios*).

BEISPIEL Das folgende Programm gibt verschiedene Datentypen in unterschiedlichen Formaten aus.

```
#include <iostream.h>
#include <iomanip.h>  /* für setw */
int main()
{
  int i = 50;
  char c = 'd';
  double d = 1.2;
  float f = 3.1232;
  const char * const p = "abcdefghijklmnopqrstuvwxyz";
  /* zunächst Standarddarstellung für die verschiedenen  */
  /* Datentypen anzeigen                                 */
  cout << i << endl;
  cout << c << endl;
  cout << d << endl;
  cout << f << endl;
  cout << p << endl;
  cout << endl;
  cout.setf( ios::oct, ios::basefield);
  cout << i << endl;  /* gleiche Zahl in Oktalnotation */
  cout << c << endl;
  cout.setf( ios::fixed, ios::floatfield);
  /* fixed-Format für float- und double-Werte */
  cout << d << endl;
  cout << f << endl;  /* obiges Format wird weiterhin eingesetzt */
  cout.setf( ios::right, ios::basefield);
  cout << setw( 50) << flush;
  cout << p << endl;  /* Zeichenkette wird in Feld mit der Breite 50 */
                      /* übertragen */
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
50
d
1.2
3.1232
abcdefghijklmnopqrstuvwxyz
62
d
1.200000
3.123199
                         abcdefghijklmnopqrstuvwxyz
%  CCM0998 Verbrauchte CPU-Zeit: 0.0009 Sekunden
```

Beachten Sie, wie der Ganzzahlwert *i* in zwei verschiedenen Formaten ausgegeben wird. Weiterhin ist bemerkenswert, wie leicht das Format von *double*- und *float*-Werten beeinflusst werden kann. Aus dem ersten Teil der Funktion *main()* wird deutlich, dass die Ausgabebibliothek sinnvolle Standardwerte zur Verfügung stellt, ohne dass der Programmierer diese explizit setzen muss.

SIEHE AUCH

ios, *manip*, *sbufpub*

4.8 sbufprot Geschützte Schnittstelle der Zeichenpuffer-Klasse

In diesem Abschnitt werden die geschützten und virtuellen Teile der Klasse *streambuf* beschrieben. Diese Teile sind insbesondere bei der Nutzung abgeleiteter Klassen von Interesse.

```
#include <iostream.h>

typedef long streamoff, streampos;

class ios
{
   public:
   enum  seek_dir {beg, cur, end};
   enum  open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                              bin, tabexp};

   // und viele weitere Deklarationen, siehe ios ...

} ;

class streambuf

{

public:

                streambuf();
                streambuf(char* p, int len);

   virtual   ~streambuf();
   void      dbp();

protected:
```

```
    int           allocate();
    char*         base();
    int           blen() const;
    char*         eback();
    char*         ebuf();
    char*         egptr();
    char*         epptr();
    void          gbump(int n);
    char*         gptr();
    char*         pbase();
    void          pbump(int n);
    char*         pptr();
    void          setb(char* b, char* eb, int a=0);
    void          setg(char* eb, char* g, char* eg);
    void          setp(char* p, char* ep);
    int           unbuffered() const;
    void          unbuffered(int);
    virtual int   doallocate();

public:

  virtual int   overflow(int c=EOF);
  virtual int   pbackfail(int c);

  virtual streampos

                seekoff(streamoff, ios::seek_dir, int =ios::in|ios:out);

  virtual streampos

                seekpos(streampos, int =ios::in|ios:out);

  virtual streambuf*

                setbuf(char* p, int len);

  virtual int   sync();
  virtual int   underflow();

};
```

---

streambuf-Objekte implementieren die Pufferabstraktion, die im Abschnitt zu *sbufpub* beschrieben wird. Die Klasse *streambuf* selbst enthält nur die grundlegenden Elemente zur Zeichenmanipulation; üblicherweise wird im Programm eine aus *streambuf* abgeleitete Klasse eingesetzt. In diesem Abschnitt wird die Schnittstelle beschrieben, die der Programmierer zum Kodieren einer abgeleiteten Klasse benötigt.

Die Elementfunktionen von *streambuf* können - vereinfacht - in zwei Gruppen unterteilt werden. Die nicht-virtuellen Funktionen manipulieren *streambuf*-Objekte so, wie dies in abgeleiteten Klassen erforderlich ist. Die Beschreibungen zeigen Implementationsdetails,die in einer öffentlichen Schnittstelle nicht erscheinen sollten. Durch die virtuellen Funktionen können abgeleitete Klassen als Spezialfälle der Klasse *streambuf* entwickelt werden, die auf die entsprechenden Quellen und Ziele (von Zeichenübertragungen) ausgelegt sind.

Die Beschreibung der virtuellen Funktionen enthält auch die "Pflichtaufgaben" der virtuellen Funktionen in den abgeleiteten Klassen. Wenn sich die virtuellen Funktionen erwartungsgemäß verhalten, ist auch das korrekte Verhalten der öffentlichen Schnittstelle von *streambuf* sichergestellt. Anderenfalls kann das Verhalten von *streambuf* allerdings Unregelmäßigkeiten aufweisen. In diesem Fall verhält sich auch ein *iostream*-Objekt (oder anderer Programmcode), das vom korrekten *streambuf*-Verhalten abhängt, anders als erwartet.

Bei den folgenden Beschreibungen wird angenommen, dass:

- *sb* vom Typ *streambuf** ist.
- *ptr, b, eb, p, ep, g* und *eg* vom Typ *char** sind.
- *c* ein *int*-Zeichen (positiv oder EOF) ist.
- *pos* vom Typ *streampos* (siehe *sbufpub*) ist.
- *off* vom Typ *streamoff* ist.
- *dir* vom Typ *seekdir* ist.
- *mode* eine *int*-Zahl ist, die *open_mode* darstellt.

Konstruktoren

streambuf()

> Ein leerer Puffer wird angelegt, der einer leeren Folge entspricht.

streambuf(char * b, int len)

> Ein leerer Puffer wird angelegt und der Reservierungsbereich auf *len* Bytes, beginnend bei *b*, gesetzt.

Die Bereiche get, put und der Reservierungsbereich

Die *protected*-Elemente von *streambuf* stellen die Schnittstelle zu den abgeleiteten Klassen dar, die in drei Bereichen (Byte-Felder) strukturiert sind. Die Bereiche werden gemeinsam von Basis- und abgeleiteten Klassen verwaltet und werden *get*, *put* und *Reservierungsbereich* (oder Puffer) genannt. Die Bereiche *get* und *put* sind üblicherweise nicht identisch, können aber den Reservierungsbereich überlappen. Der Reservierungsbereich ist in erster Linie eine Ressource, aus der Kapazität für die Bereiche *put* und *get* belegt werden kann. Die Bereiche *get* und *put* werden beim Einfügen und Entnehmen von Zeichen in den/aus dem Puffer verändert, der Reservierungsbereich ist aber im allgemeinen festgelegt. Die Bereiche werden durch eine Reihe von *char**-Werten definiert. Die Pufferabstraktion wird in Form von Zeigern beschrieben, die zwischen die Zeichenzeigen. Die *char**-Werte zeigen aber auf *char*-Objekte (Zeichen). Das Erstellen einer Beziehung mit den *char**-Werten kann man sich so vorstellen, dass der Zeiger vor das Byte gerichtet ist, auf das er eigentlich zeigt.

Funktionen zur Untersuchung von Zeigern

char * ptr=sbbase()

Ein Zeiger auf das erste Byte des Reservierungsbereiches wird geliefert. Der Bereich zwischen *sb->base()* und *sb->ebuf()* ist der Reservierungsbereich.

char * ptr=sb->eback()

Ein Zeiger auf die untere Grenze von *sb->gptr()* wird geliefert. Der Bereich zwischen *sb->eback()* und *sb->gptr()* ist zum "Zurückschieben" von Zeichen verfügbar.

char * ptr=sb->ebuf()

Ein Zeiger auf das erste Byte hinter dem Reservierungsbereich wird geliefert.

char * ptr=sb->egptr()

Ein Zeiger auf das erste Byte hinter dem Bereich *get* wird geliefert.

char * ptr=sb->epptr()

Ein Zeiger auf das erste Byte hinter dem Bereich *put* wird geliefert.

char * ptr=sb->gptr()

Ein Zeiger auf das erste Byte des Bereiches *get* wird geliefert. Die verfügbaren Zeichen befinden sich zwischen *sb->gptr()* und *sb->egptr()*. Das nächste zu entnehmende Zeichen ist *(*sb->gptr()*), sofern *sb->egptr()* nicht kleiner als oder gleichsb->gptr()* ist.

char * ptr=sb->pbase()

Es wird ein Zeiger auf die Basis des *put*-Bereichs geliefert. Die Zeichen zwischen *sb->pbase()* und *sb->pptr()* sind im Puffer gespeichert und wurden noch nicht verbraucht.

char * ptr=sb->pptr()

Es wird ein Zeiger auf das erste Byte des Bereiches *put* geliefert. Der Bereich zwischen *sb->pptr()* und *sb->epptr()* ist der *put*-Bereich. Hier sind Zeichen gespeichert.

Funktionen zum Setzen der Zeiger

Hinweis

Als Hinweis darauf, daß ein Bereich (*get*, *put* oder Reservierungsbereich) nicht existiert, sollten alle damit verknüpften Zeiger auf Null gesetzt werden.

void sb->setb(char * b, char * eb, int i)

base() und *ebuf()* werden auf *b* bzw. *eb* gesetzt. Die Angabe *i* legt fest, ob der Bereich für die automatische Löschung vorgesehen ist. Wenn *i* ungleich 0 ist, wird *b*bei einer Änderung von *base* durch einen Aufruf von *setb()* oder beim Aufruf des Destruktors für **sb* gelöscht. Wenn sowohl *b* als auch *eb* 0 sind, ist kein Reservierungsbereich verfügbar. Ist *b* ungleich 0, so ist ein Reservierungsbereich auch dann vorhanden, wenn *eb* kleiner als *b* ist und der Reservierungsbereich daher die Länge 0aufweist.

void sb->setp(char * p, char * ep)

pptr() wird auf *p*, *pbase()* auf *p* und *epptr()* auf *ep* gesetzt.

void sb->setg(char * eb, char * g, char * eg)

eback() wird auf *eb*, *gptr()* auf *g* und *egptr()* auf *eg* gesetzt.

Weitere nicht-virtuelle Elementfunktionen

int i=sb->allocate()

Es wird versucht, einen Reservierungsbereich anzulegen. Wenn bereits ein Reservierungsbereich existiert oder wenn *sb->unbuffered()* ungleich 0 ist, liefert die Funktion *allocate()* den Wert 0, ohne dass eine weitere Operation vorgenommen wurde. Beim erfolglosen Versuch der Speicherzuweisung liefert *allocate()* den Wert EOF; anderenfalls (also bei erfolgreicher Zuweisung) wird der Wert 1 zurückgegeben. Die Funktion *allocate()* wird von keiner der nicht-virtuellen Elementfunktionen von *streambuf* aufgerufen.

int i=sb->blen()

Die Größe (in *char*-Einheiten) des aktuellen Reservierungsbereichs wird geliefert.

void dbp()

Die Funktion schreibt Pufferstatus-Informationen im EBCDIC-Format direkt in die Einheit, die mit dem Dateideskriptor 1 verknüpft ist. Diese Funktion wurde zu Testzwecken eingefügt; es sind keine Spezifikationen zum Format der Ausgabe vorgegeben.
Die Funktion wird als Teil der geschützten Schnittstelle angesehen, da die ausgegebenen Informationen nur im Zusammenhang mit der Schnittstelle sinnvoll sind. Die Funktion ist als *public* deklariert, damit sie beim Testen des Programms an beliebiger
Stelle aufgerufen werden kann.

void sb->gbump(int n)

gptr() wird um den Wert *n* inkrementiert, wobei *n* positiv oder negativ sein kann. Es wird nicht geprüft, ob der neue Wert von *gptr()* in den erlaubten Grenzen liegt.

void sb->pbump(int n)

pptr() wird um den Wert *n* inkrementiert; *n* kann positiv oder negativ sein. Es wird nicht geprüft, ob der neue Wert von *pptr()* in den erlaubten Grenzen liegt.

void sb->unbuffered(int i)
int i=sbunbuffered()

Es gibt eine *private*-Variable, die den Status der Pufferung von *sb* beschreibt. *sb->unbuffered(i)* setzt den Wert dieser Variable auf *i*, während *sb->unbuffered()*
den aktuellen Variablenwert liefert. Diese Statusangabe erfolgt unabhängig von der tatsächlichen Belegung des Reservierungsbereichs. Die Variable dient primär dazu, festzustellen, ob der Reservierungsbereich automatisch durch *allocate()* reserviert wurde.

Virtuelle Elementfunktionen

Virtuelle Funktionen können in abgeleiteten Klassen erneut definiert werden, um das Verhalten von *streambuf*-Objekten genauer festzulegen. In diesem Abschnitt wird das Soll-Verhalten der virtuellen Funktionen in den abgeleiteten Klassen beschrieben. Im nächsten Abschnitt finden Sie eine Beschreibung des Verhaltens dieser Funktionen in der Basisklasse *streambuf*.

int i=sb->doallocate()

Die Funktion wird aufgerufen, wenn *allocate()* bestimmt, dass Speicherbereich benötigt wird. *doallocate()* wird zum Aufruf von *setb()* verwendet, damit ein Reservierungsbereich bereitgestellt oder - wenn dies nicht möglich ist - der Wert EOF geliefert wird.Der Aufruf erfolgt nur, wenn *sb->unbuffered()* und *sb->base()* gleich 0 sind.

int i=overflow(int c)

Diese Funktion "verbraucht" Zeichen. Wenn *c* nicht das Zeichen EOF ist, muß die Funktion *overflow() c* entweder sichern oder verbrauchen. Die Funktion wird üblicherweise aufgerufen, wenn der Bereich *put* gefüllt ist und das Speichern eines weiteren Zeichens versucht wurde. Der Aufruf ist aber auch in anderen Fällen möglich. Üblicherweise werden durch den Aufruf die Zeichen zwischen *pbase()* und *pptr()* verbraucht, *setp()* wird aufgerufen, um einen neuen *put*-Bereich anzulegen, und *c* wird über die Funktion *sputc()* gespeichert, wenn *c!=EOF* ist. *sb->overflow()* sollte zur Anzeige eines Fehlers den Wert EOF liefern, anderenfalls sollte eine andere Rückgabe erfolgen.

int i=sb->pbackfail(int c)

Die Funktion wird aufgerufen, wenn *eback()* gleich *gptr()* ist und versucht wurde, das Zeichen *c* "zurückzuschieben". Wenn diese Situation korrekt gehandhabt werden kann(beispielsweise durch Repositionieren in einer externen Datei), sollte *pbackfail()* das Zeichen *c* liefern; anderenfalls sollte EOF zurückgegeben werden.

streampos pos=sb->seekoff(streamoff off, seekdir dir, int mode)

seekoff() ist eine öffentliche virtuelle Elementfunktion. Eine detaillierte Beschreibung finden Sie im Abschnitt Abschnitt *sbufpub*. Die *get*- und/oder *put*-Zeiger werden neu positioniert. Diese Positionierung wird nicht von allen abgeleiteten Klassen unterstützt.

streampos pos=sb->seekpos(streampos pos, int mode)

seekpos() ist eine öffentliche virtuelle Elementfunktion. Eine detaillierte Beschreibung finden Sie im Abschnitt Abschnitt *sbufpub*. Die *get*- und/oder *put*-Zeiger werden neu positioniert. Diese Positionierung wird nicht von allen abgeleiteten Klassen unterstützt.

streambuf * sb=sb->setbuf(char * ptr, int len)

Das Feld, das bei *ptr* beginnt und *len* byte lang ist, wird als Reservierungsbereich angeboten. Üblicherweise wird ein Aufruf als Anforderung interpretiert, *sb* ohne Pufferung anzulegen, wenn *ptr* oder *len* gleich 0 sind. Die abgeleitete Klasse kann - muss aber nicht - diesen Bereich nutzen. Auch die Anforderung des nicht gepufferten Status kann akzeptiert oder ignoriert werden. *setbuf()* sollte *sb* liefern, wenn die Anforderung erfüllt wird; anderenfalls sollte der Wert 0 zurückgegeben werden.

int i=sbsync()

sync() ist eine öffentliche virtuelle Elementfunktion. Eine detaillierte Beschreibung finden Sie im Abschnitt *sbufpub*.

int i=sbunderflow()

Die Funktion wird zur Bereitstellung von Zeichen eingesetzt. Hierdurch kann beispielsweise eine Situation geschaffen werden, in der ein Bereich *get* vorliegt, der nicht leer ist. Erfolgt der Aufruf, wenn Zeichen im Bereich *get* vorliegen, sollte die Funktion das erste verfügbare Zeichen liefern. Im Fall eines leeren Bereichs *get* sollte zunächst einnicht leerer Bereich *get* angelegt und das nächste Zeichen (das im Bereich *get* verbleiben sollte) geliefert werden. Wenn keine weiteren Zeichen verfügbar sind, sollte die Funktion *underflow()* den Wert EOF liefern und einen leeren Bereich *get* hinterlassen.

Die Standarddefinitionen der virtuellen Funktionen

int i=sbstreambuf::doallocate()

Die Zuweisung von Speicherkapazität an einen Reservierungsbereich wird über den Operator *new* versucht.

int i=sb->streambuf::overflow(int c)

streambuf::overflow() sollte behandelt werden, als wäre das Funktionsverhalten nicht definiert. Das bedeutet, dass abgeleitete Klassen die Funktion immer definieren sollten.

int i=sb->streambuf::pbackfail(int c)

Beim Auftreten eines Fehlers wird EOF und bei erfolgreicher Ausführung wird *c* geliefert.

streampos pos=sb->streambuf::seekpos(streampos pos, int mode)

Es wird *sb->seekoff(streamoff(pos),ios::beg,mode)* geliefert. Zur Definition der Repositionierung in einer abgeleiteten Klasse ist es häufig nur nötig, *seekoff()* zu definieren und die ererbte Funktion *streambuf::seekpos()* einzusetzen.

streampos pos=sb->streambuf::seekoff(streamoff off, seekdir dir, int mode)

EOF wird geliefert.

streambuf * sb=sb->streambuf::setbuf(char* ptr, int len)

Die Anforderung wird erfüllt, wenn kein Reservierungsbereich vorhanden ist.

int i=sb->streambuf::sync()

Der Wert 0 wird geliefert, wenn der Bereich *get* leer ist und keine nicht verbrauchten Zeichen vorliegen. Andernfalls wird EOF geliefert.

int i=sb->streambuf::underflow()

streambuf::underflow() sollte behandelt werden, als wäre das Funktionsverhalten nicht definiert. Das bedeutet, daß die Funktion in abgeleiteten Klassen immer definiert werden sollte.

BEISPIEL Das Programm gibt die Adresse des Basisbereiches einer aus *streambuf* abgeleiteten Klasse aus.

Das Programm ist ein Beispiel zur Darstellung von Speicherinhalten. Es hätten andere *trivial*-Elementfunktionen, wie *get_base*, verwendet werden können, die die Adressen der Bereiche *get* und *put* liefern.

```
#include <iostream.h>
const int N = 20;
class trivial : public streambuf
{
  int a;                /* Einige Beispieldaten in einer Klasse */
  public:
  trivial() : streambuf(new char[ N], N)
  {
    /* trivial-Konstruktor wird durch streambuf-Konstruktor     */
    /*definiert*/
    a = 0;
  };
   trivial() {};
  /* Annahme, daß der streambuf-Destruktor den N Byte großen    */
  /* Reservierungsbereich löscht                                */
  char * get_base()
  {
    /* Diese Funktion wird benötigt, da die Elementfunktion     */
    /* streambuf::base() geschützt ist.                         */
    /* Qualifikation streambuf:: wird nicht benötigt, da        */
    /* Gültigkeitsbereich ok ist.                               */
    return base();
  };
};
int main()
{
  trivial test_var;
  cout << (void *) test_var.get_base() << endl;
  /* Umwandlung nach void *, damit cout nicht den Inhalt des     */
  /* ersten Byte des Reservierungsbereiches anzeigt.             */
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
0xc6008
%  CCM0998 Verbrauchte CPU-Zeit: 0.0005 Sekunden
```

Beachten Sie, dass der im Zeiger gespeicherte Wert variieren kann und dass *cout* ein Standardformat für Zeigerwerte aufweist.

BESONDERHEITEN

Die Konstruktoren wurden als *public* deklariert, um die Kompatibilität zum älteren *stream*-Paket zu erhalten. Sie sollten als *protected* deklariert sein.

Die Schnittstelle für nicht gepufferte Operationen ist unhandlich. Die Entwicklung der virtuellen Funktionen *underflow()* und *overflow()* mit korrektem Verhalten für ungepufferte *streambuf*-Objekte ist ohne besondere Vorkehrungen kompliziert. Zudem gibt es für virtuelle Funktionen keine Möglichkeit, gesondert auf *get*- und *put*-Operationen zureagieren, die mehrere Zeichen umfassen.

Obwohl die öffentliche Schnittstelle der Klasse *streambuf* in Zeichen und Bytes beschrieben wird, arbeitet die Schnittstelle zu abgeleiteten Klassen mit *char*-Objekten. Da eine Entscheidung bezüglich des Typs der tatsächlichen Datenzeiger getroffen werden musste, erschien es einfacher, dies durch die Datentypen der *protected*-Elemente wiederzugeben, als alle Elemente doppelt zu implementieren (in einer *char*- und *unsigned char*-Version). Vielleicht hätten auch alle Einsatzbereiche von *char** über eine *typedef*-Anweisung realisiert werden sollen.

SIEHE AUCH

istream, *sbufpub*

4.9 sbufpub Öffentliche Schnittstelle der Zeichenpuffer-Klasse

In diesem Abschnitt werden die *public*-Elementfunktionen von *streambuf* beschrieben. Es sollten keine reinen Objekte vom Typ *streambuf* in einem Programm verwendet werden, sondern nur von *streambuf* abgeleitete Objekte (z.B. *filebuf*, *strstreambuf*, *stdio- buf*)!

```
#include <iostream.h>

typedef long streamoff, streampos;

class ios
{
public:

   enum  seek_dir {beg, cur, end};
   enum  open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                              bin, tabexp};

   // und viele andere Klassen, siehe ios.

};

class streambuf

{

public:

    int      in_avail();
    int      out_waiting();
    int      sbumpc();
    int      sgetc();
    int      sgetn(char* ptr, int n);
    int      snextc();
    int      sputbackc(char);
    int      sputc(int c);
    int      sputn(const char* s, int n);
    void     stossc();

    virtual streampos

             seekoff(streamoff, ios::seek_dir, int =ios::in|ios:out);

    virtual streampos

             seekpos(streampos, int =ios::in|ios:out);

    virtual int sync();

};
```

Die Klasse *streambuf* unterstützt Puffer, in die Zeichen eingefügt (*put*) oder aus denen Zeichen entnommen (*get*) werden können. Ein solcher Puffer ist eine Folge von Zeichen mit einem oder zwei Zeigern (einem *get*- und/oder einem *put*-Zeiger), die die Position beschreiben, an der Zeichen eingefügt oder entnommen werden. Man sollte sich vorstellen, dass die Zeiger zwischen die Zeichen und nicht auf die Zeichen zeigen. Das vereinfacht das Verständnis der Grenzbedingungen (Zeiger vor dem ersten oder hinter dem letzten Zeichen). Einige Auswirkungen des Entnehmens und Einfügens von Zeichen werden durch diese Klasse definiert, die meisten Details bleiben aber den spezialisierten -aus *streambuf* abgeleiteten -Klassen vorbehalten, siehe *filebuf*, *sstreambuf* und *stdiobuf*.

Die aus *streambuf* abgeleiteten Klassen variieren in der Behandlung der Zeiger *get* und *put*. Im einfachsten Fall erlauben die Puffer entweder nur das Einfügen oder nur das Entnehmen von Zeichen. Solche Klassen dienen als reine Quellen (Erzeuger) oder Ziele(Verbraucher) von Zeichen. Warteschlangen-ähnliche Puffer (siehe *strstream* und *sstreambuf*) besitzen einen *put*- und einen *get*-Zeiger, die unabhängig voneinander verschoben werden können. In solchen Puffern werden gespeicherte Zeichen abgelegt, bis sie zu einem späteren Zeitpunkt wieder entnommen werden. Dateiähnliche Puffer (beispielsweise *filebuf*; siehe *filebuf*) erlauben sowohl Einfügungen als auch Entnahmen, weisen aber nur einen Zeiger auf. (In anderen Worten: Die Zeiger *get* und *put* sind hierbei zu einem Zeiger kombiniert. Wird ein Zeiger verschoben, so führt dies auch zu einer Verschiebung des anderen Zeigers.)

Die meisten Elementfunktionen von *streambuf* sind in zwei Phasen unterteilt. Soweit als möglich werden die Operationen auf das Speichern in oder Entnehmen aus Feldern (den Bereichen *get* und *put*, die gemeinsam den Reservierungsbereich oder Puffer bilden) beschränkt. Von Zeit zu Zeit werden virtuelle Funktionen zur Bearbeitung einer Reihe von Zeichen in den Bereichen *get* und *put* aufgerufen. Die virtuellen Funktionen holen weitere Zeichen von einem "Zeichenlieferanten" oder übergeben eine Zeichenfolge an einen "Endverbraucher" von Zeichen. Im allgemeinen benötigt der Benutzer von *streambuf* hierfür kein Detailwissen, einige der *public*-Elemente geben aber Informationen zum Status der Bereiche zurück. Weitere Informationen zu diesen Bereichen finden Sie im Abschnitt zu *sbufprot*, wo die geschützte Schnittstelle beschrieben wird.

Die *public*-Elementfunktionen der Klasse *streambuf* werden im folgenden beschrieben. Dabei wird angenommen, dass

- *i*, *n* und *len* *int*-Werte sind.
- *c* ein *int*-Wert ist. *c* enthält einen "Zeichenwert" oder EOF. Ein Zeichenwert ist immer positiv, auch wenn ein *char*-Objekt üblicherweise um das Vorzeichen erweitert wird.
- *sb* und *sb1* vom Typ *streambuf** sind.
- *ptr* vom Typ *char** ist.
- *off* vom Typ *streamoff* ist.
- *pos* vom Typ *streampos* ist.
- *dir* vom Typ *seek_dir* ist.
- *mode* ein *int*-Wert ist, der *open_mode* darstellt.

public-Elementfunktionen

int i=sbin_avail()

> Die Anzahl der Zeichen, die im Bereich *get* sofort zur Entnahme bereitstehen, wird geliefert. Es können garantiert *i* Zeichen entnommen werden, ohne daß ein Fehler auftritt.

int i=sbout_waiting()

Die Anzahl der Zeichen im Bereich *put*, die noch nicht (durch einen Endverbraucher) aufgebraucht wurde, wird geliefert.

int c=sbsbumpc()

Der Zeiger *get* wird um ein Zeichen vorwärts verschoben; das auf diese Weise übersprungene Zeichen wird geliefert. Wenn sich der Zeiger am Ende der Zeichenfolge befindet, wird der Wert EOF geliefert.

int c=sbsgetc()

Das Zeichen hinter dem Zeiger *get* wird geliefert. Der Zeiger *get* wird hierbei nicht verschoben! Wenn kein Zeichen verfügbar ist, wird EOF geliefert.

streambuf* sb1=sb->setbuf(char * ptr, int len, int i)

Es werden *len* Bytes, beginnend bei *ptr*, als Reservierungsbereich angeboten. Falls *ptr* NULL oder *len* kleiner oder gleich 0 ist, wird der Status ohne Pufferung angefordert. Ob der angebotene Bereich eingesetzt oder einer Anforderung des ungepufferten Status nachgegeben wird, hängt von den Details in der abgeleiteten Klasse ab. *setbuf()* liefert üblicherweise *sb*; falls das Angebot aber nicht angenommen oder die Anforderung abgelehnt wird, wird der Wert 0 geliefert.

int i=sb->sgetn(char * ptr, int n)

Es werden *n* Zeichen hinter dem Zeiger *get* entnommen und in den Bereich kopiert,der bei *ptr* beginnt. Wenn weniger als *n* Zeichen vor dem Ende der Zeichenfolge verbleiben, entnimmt *sgetn()* alle verbleibenden Zeichen. Die Funktion *sgetn()* repositioniert den Zeiger *get* hinter die entnommenen Zeichen und liefert die Anzahl der entnommenen Zeichen.

int c=sbsnextc()

Der Zeiger *get* wird um ein Zeichen vorwärts verschoben; das auf die neue Position folgende Zeichen wird geliefert. Wenn sich der Zeiger - vor oder nach der Verschiebung - am Ende der Zeichenfolge befindet, wird EOF geliefert.

int i=sb->sputbackc(int c)

Der Zeiger *get* wird um ein Zeichen zurückgesetzt. *c* muss das aktuelle Zeichen sein,das sich direkt vor dem Zeiger befindet. Der zugrunde liegende Mechanismus kann einfach den Zeiger *get* zurücksetzen oder die internen Datenstrukturen so verändern, dass *c* gespeichert wird. Die Auswirkung der Funktion *sputbackc()* ist folglich nicht definiert, wenn *c* nicht das vor dem Zeiger *get* befindliche Zeichen ist. Die Funktion *sputbackc()* liefert bei erfolgloser Ausführung EOF. Die Bedingungen, unter denen ein Aufruf erfolglos ist, hängen von Details der abgeleiteten Klasse ab.

int i=sb->sputc(int c)

c wird hinter dem Zeiger *put* gespeichert und der Zeiger hinter das gespeicherte Zeichen verschoben. Üblicherweise führt dies zu einer Verlängerung der Zeichenfolge.Im Fehlerfall wird EOF geliefert. Die Bedingungen, unter denen Fehler auftreten, sind von der abgeleiteten Klasse abhängig.

int i=sb->sputn(const char * ptr, int n)

Hinter dem Zeiger *put* werden die *n* Zeichen, die bei *ptr* beginnend abgelegt sind,gespeichert; der Zeiger *put* wird hinter die Zeichenfolge positioniert. Die Funktion *sputn()* liefert *i*, die Anzahl der erfolgreich gespeicherten Zeichen. Im Normalfall ist *i* mit *n* identisch; *i* kann beim Auftreten eines Fehlers aber auch niedriger als *n* sein.

void sb stossc()

Der Zeiger *get* wird um ein Zeichen nach vorn verschoben. Falls sich der Zeiger beim Funktionsaufruf am Ende der Zeichenfolge befindet, hat der Funktionsaufruf keine Auswirkung.

streampos pos=sb->seekoff(streamoff off, ios::seek_dir dir, int mode)

Die Zeiger *get* und/oder *put* (die abstrakten *get*- und *put*-Zeiger und nicht *pptr()* und *gptr()*) werden neu positioniert. *mode* gibt dabei an, ob der Zeiger *put* (das Bit *ios::out* ist gesetzt) oder *get* (das Bit *ios::in* ist gesetzt) modifiziert werden soll. Eskönnen beide Bits gesetzt sein; in diesem Fall sollten auch beide Zeiger angesprochen werden.

off wird als Byte-Offset interpretiert (beachten Sie, dass es sich hierbei um einen vorzeichenbehafteten Wert handelt). Die Bedeutung der Werte, die *dir* annehmen kann,sind

ios::beg

Der Beginn des Datenstroms.

ios::cur

Die aktuelle Position.

ios::end

Das Ende des Datenstroms (Ende der Datei).

Nicht alle aus *streambuf* abgeleiteten Klassen unterstützen die Repositionierung. Die Funktion *seekoff()* liefert den Wert EOF, falls das Repositionieren von der Klasse nicht unterstützt wird. Unterstützt die Klasse das Repositionieren, liefert *seekoff()* die neue Position oder - im Fehlerfall - EOF.

streampos pos=sb->seekpos(streampos pos, int mode)

Die *streambuf*-Zeiger *get* und/oder *put* werden auf *pos* gesetzt. *mode* spezifiziert, welche Zeiger - wie bei *seekoff()* - verändert werden. Es wird *pos* (das Argument) oder - wenn die Klasse keine Repositionierung unterstützt oder ein Fehler auftritt - EOF geliefert. Auf eine Variable des Typs *streampos* sollte im allgemeinen keine arithmetische Operation angewendet werden. Zwei Werte besitzen aber eine besondere Bedeutung:

streampos(0)

Stellt den Beginn der Datei dar.

streampos(EOF)

Wird als Hinweis auf einen Fehler genutzt.

int i=sbsync()

Die internen Datenstrukturen werden mit der externen Quelle oder dem externen Ziel synchronisiert. Die Details dieser Funktion hängen von der abgeleiteten Klasse ab.Die Funktion *sync()* wird aufgerufen, damit die abgeleitete Klasse die Statusangaben der Bereiche einsehen und diese mit der externen Repräsentation synchronisieren kann. Üblicherweise sollte *sync()* alle im Bereich *put* gespeicherten Zeichen verbrauchen und - wenn möglich - die Zeichen im Bereich *get*, die noch nicht entnommen wurden, an die Quelle zurückgeben. Nach der Rückkehr von *sync()* sollten keine weiteren, nicht verbrauchten Zeichen vorliegen, und der Bereich *get* sollte leer sein. *sync()* sollte den Wert EOF liefern, wenn ein Fehler aufgetreten ist. *sync()* "leert" also alle gespeicherten Zeichen, die noch nicht verbraucht wurden, und es werden alle Zeichen zurückgegeben, die zwar erzeugt, aber noch nicht entnommen wurden.

BEISPIEL Im folgenden Programm wird eine Variable vom Typ *filebuf* definiert, die mit *cin* verknüpft ist. Aus diesem *filebuf*-Objekt werden Zeichen blockweise eingelesen, bis das Dateiende erreicht ist. Daraufhin wird die Anzahl der tatsächlich eingelesenen Zeichen ermittelt, wobei jedes Zeilenende-Zeichen '\n' ein Zeichen darstellt:

```
#include <iostream.h>
#include <fstream.h>
int main()
{
  filebuf in_file(0);
  /* in_file ist mit cin verknüpft */
  const int N = 10;
  int k;
  char text_b[N+1];
  /* Textpuffer */
  cout << "Bitte " << N << " Zeichen eingeben :\n";
  cout.flush();
  k = in_file.sgetn(&text_b[0],N);
  cout << "Es wurden " << (k+in_file.in_avail()) ;
  cout << " Zeichen eingegeben.\n";
       /* jedes \n ist auch ein Zeichen. */
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
Bitte 10 Zeichen eingeben :
0123456789
Es wurden 11 Zeichen eingegeben.
%  CCM0998 Verbrauchte CPU-Zeit: 0.0040 Sekunden
```

Der Benutzer kann die Puffergröße durch den Aufruf der Elementfunktion *setbuf()* ändern. Durch einen Aufruf von *setbuf()* kann jede Puffergröße gesetzt werden.

BESONDERHEITEN

setbuf gehört nicht im eigentlichen Sinne zur öffentlichen Schnittstelle und wurde nur aus Kompatibilitätsgründen im *stream*-Paket aufgenommen.

SIEHE AUCH

istream, *sbufprot*

4.10 sstreambuf Spezialisierung von streambuf auf Felder

In diesem Abschnitt wird beschrieben, wie eine Zeichenkette als Puffer für einen Datenstrom eingesetzt werden kann.

```
#include <iostream.h>
#include <strstream.h>

class strstreambuf : public streambuf

{

public:

                strstreambuf() ;
                strstreambuf(char*, int, char* pstart=0);
                strstreambuf(int);
                strstreambuf(unsigned char*, int, unsigned char* pstart=0);
                strstreambuf(void* (*a)(long), void(*f)(void*));
                 ~strstreambuf();

   void         freeze(int n=1) ;
   char*        str();

   virtual int  doallocate();
   virtual int  overflow(int);

   virtual streampos

                seekoff(streamoff, ios::seek_dir, int);

   virtual streambuf*

                setbuf(char* p, int n);

   virtual int  underflow();

};
```

Ein *strstreambuf*-Objekt ist ein *streambuf*-Objekt, das ein Feld von mehreren Bytes (Zeichenkette) zur Aufnahme einer Folge von Zeichen einsetzt. Unter der Maßgabe, dass *char** vor das *char*-Objekt zeigen soll (auf das es eigentlich gerichtet ist), ergibt sicheine direkte Zuordnung zwischen den abstrakten *get/put*-Zeigern (siehe *sbufpub*) und den *char**-Zeigern. Das Verschieben der Zeiger entspricht genau der Inkrementierung und Dekrementierung der *char**-Werte.

strstreambuf unterstützt einen dynamischen Modus, um die Forderung nach Zeichenketten beliebiger Länge erfüllen zu können. Einem *strstreambuf*-Objekt im dynamischen Modus wird die Speicherkapazität für die Zeichenkette nach Bedarf zugewiesen. Wird die Zeichenfolge für das derzeit eingesetzte Feld zu lang, wird sie in ein neues Feld kopiert.

Für die folgenden Beschreibungen wird vorausgesetzt, dass

- *ssb* vom Typ *strstreambuf** ist.

- *sb* vom Typ *streambuf** ist.
- *ptr* und *pstart* vom Typ *char** oder *unsigned char** sind.

Konstruktoren

strstreambuf()

> Ein leeres *strstreambuf*-Objekt wird im dynamischen Modus angelegt. Das bedeutet, dass - durch die Operatoren *new* und *delete* - automatisch Speicherkapazität zugewiesen wird, um alle Zeichen aufnehmen zu können, die in das *strstreambuf*-Objekt eingefügt werden. Da hierzu unter Umständen das Kopieren der Originalzeichen notwendig wird, sollte das Programm beim Einfügen vieler Zeichen die Funktion *setbuf()* (Beschreibung folgt) einsetzen, um das *strstreambuf*-Objekt zu informieren.

strstreambuf(void * (*a)(long), void * (*f)(void*))

> Ein leeres *strstreambuf*-Objekt im dynamischen Modus wird angelegt. *a* wird als Belegungsfunktion im dynamischen Modus eingesetzt. Das an *a* übergebene Argument ist ein *long*-Wert, der die Anzahl der zuzuweisenden Bytes darstellt. Wenn *a* den Wert 0 aufweist, wird der Operator *new* eingesetzt. *f* wird eingesetzt, um die von *a* gelieferten Bereiche freizugeben (oder zu löschen). Das Argument zu *f* ist ein Zeiger auf das von *a* belegte Feld. Bei einem Wert von 0 für *f* wird der Operator *delete* eingesetzt.

strstreambuf(int n)

> Ein leeres *strstreambuf*-Objekt wird im dynamischen Modus angelegt. Die anfängliche Speicherplatzzuweisung umfasst mindestens *n* Bytes.

strstreambuf(char * ptr, int n, char * pstart)
strstreambuf(unsigned char * ptr, int n, unsigned char * pstart)

> Es wird ein *strstreambuf*-Objekt angelegt, das die bei *ptr* beginnenden Bytes als Puffer nutzt. Das Objekt befindet sich im statischen Modus und kann nicht dynamisch erweitert werden. Wenn *n* positiv ist, werden die bei *ptr* beginnenden *n* Bytes als *strstreambuf*-Objekt verwendet. Ist *n* gleich 0, wird angenommen, daß *ptr* auf den Beginn einer mit 0 beendeten Zeichenkette gerichtet ist; die Bytes dieser Zeichenkette(ohne das beendende Nullzeichen) stellen das *strstreambuf*-Objekt dar. Bei negativem *n* wird angenommen, dass das *strstreambuf*-Objekt eine unendliche Länge aufweist. Der Zeiger *get* wird auf *ptr* und der Zeiger *put* wird auf den Wert von *pstart* initialisiert. Wenn *pstart* den Wert 0 besitzt, werden Speicherversuche als Fehler behandelt.Ist *pstart* hingegen ungleich 0, besteht die Zeichenfolge zur Entnahme (der Bereich *get*) - zu Anfang - aus den Bytes zwischen *ptr* und *pstart*. Für *pstart* gleich 0 umfasst der Bereich *get* anfangs das gesamte Feld.

Elementfunktionen

ssb->freeze(int n)

> Unterbindet (für *n* ungleich 0) oder erlaubt (für *n* gleich 0) die automatische Löschung des aktuellen Feldes. Eine Löschung erfolgt gewöhnlich, wenn mehr Speicherbereich benötigt oder *ssb* zerstört wird. Nur dynamisch zugewiesener Speicherbereich wird freigegeben. Das Speichern von Zeichen in einem *strstreambuf*-Objekt, das im dynamischen Belegungsmodus vorlag und nun "eingefroren" ist, ist ein Fehler (die Folgen einer solchen Operation sind nicht definiert). Es ist aber möglich, ein solches *strstreambuf*-Objekt "aufzutauen" und dann Zeichen zu speichern.

char* ptr=ssb->str()

Ein Zeiger auf das erste *char*-Objekt des aktuellen Feldes wird geliefert und *ssb* "eingefroren". Wenn *ssb* mit einer expliziten Feldangabe angelegt wurde, zeigt *ptr* auf dieses Feld. Befindet sich *ssb* hingegen im dynamischen Belegungsmodus und hat noch keine Speicherung stattgefunden, kann *ptr* auch den Wert 0 aufweisen.

streambuf * sb=ssb->setbuf(char *, int n)

Der Wert *n* wird von *ssb* zwischengespeichert, und bei der nächsten dynamischen Speicherplatzbelegung werden wieder mindestens *n* Bytes zugewiesen.

BEISPIEL

Das folgende Programm deklariert eine Variable vom Typ *strstreambuf* und initialisiert sie mit der Zeichenkette *p*. Die Elementfunktion *str()* wird aufgerufen, um zu gewährleisten, dass die Zeichenkette *p* vom *strstreambuf*-Konstruktor erfolgreich verarbeitet wird.

```
#include <strstream.h>
#include <iostream.h>
#include <string.h>
char * const p = "Wirklich eine lange Zeichenkette\
                  abcdefghijlkmnopqrstuvwxyz\n";
int main()
{
  strstreambuf s(p, 0, (char *) NULL);
  /* Zeichenkette p liegt in strstreambuf vor.  */
  /* get-Zeiger ist auf Beginn von p gerichtet. */
  char *tp = s.str();
  cout << "Laenge des Original-Strings " << strlen(p) << endl;
  cout << "Laenge des strstreambuf-Strings " << strlen(tp) << endl;
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
Laenge des Original-Strings 77
Laenge des strstreambuf-Strings 77
%  CCM0998 Verbrauchte CPU-Zeit: 0.0018 Sekunden
```

Beachten Sie, dass sich die Länge der Originalzeichenkette nicht verändert hat.

SIEHE AUCH

sbufpub, *strstream*

4.11 stdiobuf Spezialisierung von iostream auf stdio-FILE-Objekte

In diesem Abschnitt wird die Klasse *stdiobuf* beschrieben, eine spezialisierte *streambuf*-Klasse, die sich mit der Ein-/Ausgabestruktur FILE (Operationen auf hoher Ebene) beschäftigt.

stdiobuf sollte immer dann eingesetzt werden, wenn Quellcode in C und C++ in einem Programm gemeinsam verwendet wird. Neuerer Quellcode in C++ sollte *filebuf*-Objekte einsetzen.

```
#include <iostream.h>
#include <stdiostream.h>
#include <stdio.h>

class stdiobuf : public streambuf

{

public:

                stdiobuf(FILE* f);

   FILE *       stdiofile();

   virtual      ~stdiobuf();

   virtual int  overflow(int c=EOF);
   virtual int  pbackfail(int c);

   virtual streampos

                seekoff(streamoff, ios::seek_dir, int);

   virtual      int sync();
   virtual int  underflow();

};

class stdiostream : public ios

{

public:

                stdiostream(FILE*);

                ~stdiostream();

   stdiobuf *   rdbuf();

};
```

Die auf *stdiobuf* angewendeten Operationen werden in der zugehörigen FILE-Struktur reflektiert. Ein *stdiobuf*-Objekt wird im ungepufferten Modus angelegt, wodurch alle Operationen sich sofort auch in der zugehörigen FILE-Struktur auswirken. Aufrufe der Funktionen *seekg()* und *seekp()* werden in Aufrufe von *fseek()* umgewandelt. Die Funktion *setbuf()* besitzt ihre übliche Bedeutung; wenn hierdurch ein Reservierungsbereich bereitgestellt wird, wird auch die Pufferung wieder aktiviert.

Für die folgende Beschreibung wird angenommen, dass

- *std* vom Typ *stdiobuf* ist.
- *sts* vom Typ *stdiostream* ist.
- *fp* vom Typ *FILE* * ist.

Konstruktoren

stdiobuf(FILE * fp)

Ein *stdiobuf*-Objekt wird im ungepufferten Modus angelegt und mit *fp* verknüpft.

stdiostream(FILE * fp)

Es wird ein *stdiostream*-Objekt konstruiert und mit *fp* verknüpft.

stdiobuf-Elemente

FILE * fp = std.stdiofile()

Der mit *stdiobuf* verknüpfte Dateizeiger wird geliefert.

int l = std.overflow(int c)

Wenn die mit dem *stdiobuf* verknüpfte Datei geschlossen ist oder *c*=EOF, wird EOF geliefert. Ansonsten wird *putc()* aufgerufen und sein Returnwert zurückgeliefert.

int l = std.pbackfail(int c)

Der Wert von *ungetc()* wird geliefert.

streampos sp = std.seekoff(streamoff p, ios::seek_dir d, int l)

Der Parameter *l* wird ignoriert. Der Returnwert von dem entsprechenden *fseek()*-Aufruf wird zurückgeliefert.

int l = std.sync()

Wenn die letzte Operation ein Schreibzugriff war, wird *fflush()* aufgerufen. Der Returnwert von *fseek()* auf die aktuelle Position wird zurückgeliefert.

int l = std.underflow()

Wenn die mit dem *stdiobuf* verknüpfte Datei geschlossen ist oder das Dateiende erreicht ist, wird EOF geliefert. Ansonsten wird das nächste Zeichen zurückgeliefert.

stdiostream-Element

stdiobuf * std = sts.rdbuf()

Es wird ein Zeiger auf das *stdiobuf*-Objekt geliefert, das mit *sts* verknüpft ist.

BEISPIEL Im folgenden Programm wird die Datei *#TEMP* geöffnet. Daraufhin wird eine Variable vom Typ *stdiobuf* mit der Datei verknüpft und eine Meldung ausgegeben, ob die Verknüpfung des *stdiobuf*-Objektes mit der Datei gelungen ist.

```
#include <stdiostream.h>
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
int main()
{
  FILE *qw;
  const char * const name = "#TEMP";
  if (!(qw = fopen(name, "w")))
  {
    cerr << "kann Datei " << name << " nicht oeffnen.\n";
    exit(1);
  }
  stdiobuf s(qw);
  FILE *rt = s.stdiofile();
  if (rt != qw)
  {
    cerr << "Fehler in stdiofile().\n";
  }
  else
  {
    cerr << "stdiofile() funktioniert korrekt.\n";
  }
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
stdiofile() funktioniert korrekt.
%  CCM0998 Verbrauchte CPU-Zeit: 0.0086 Sekunden
```

Das Programm zeigt, dass die Elementfunktion *stdiofile()* von *stdiobuf* in diesem Fall das korrekte Ergebnis liefert.

SIEHE AUCH

filebuf, *istream*, *ostream*, *sbufpub*

4.12 strstream Spezialisierung von iostream auf Felder

In diesem Abschnitt wird die Klasse *strstream* beschrieben, die eine Spezialisierung von *iostream* ist. *strstream* stellt die Ein- und Ausgabeoperationen für Byte-Felder zur Verfügung.

```
#include <iostream.h>

class ios

{
public:enum open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                                        bin, tabexp};

// sowie viele weitere, siehe ios ...

};

#include <strstream.h>

class strstreambase : public virtual ios

{

public:

   strstreambuf*  rdbuf();

};

class istrstream : public strstreambase, public istream

{

public:

                istrstream(char*);
                istrstream(char*, int);
                istrstream(const char*);
                istrstream(const char*, int);
                 ~istrstream();

};

class ostrstream : public strstreambase, public ostream

{

public:

              ostrstream();
              ostrstream(char*, int, int=ios::out);
               ~ostrstream();

   int        pcount();
   char*      str();

};
```

```
class strstream : public strstreambase, public iostream

{

public

            strstream();
            strstream(char*, int, int mode);
             ~strstream();

   char*    str();

};
```

---

strstreambase stellt die Elementfunktion *rdbuf()* zur Verfügung. Es ist nicht beabsichtigt,Objekte vom Typ *strstreambase* zu definieren.

strstream ist eine Spezialisierung von *iostream* für die Speicherung und die Entnahme von Werten aus Byte-Feldern. Der einem *strstream*-Objekt zugeordnete *streambuf*-Teilist vom Typ *strstreambuf* (siehe *sstreambuf*).

Für die folgende Beschreibung wird angenommen, dass

- *ss* vom Typ *strstream* ist.
- *iss* vom Typ *istrstream* ist.
- *oss* vom Typ *ostrstream* ist.
- *mode* eine *int*-Zahl ist, die *open_mode* darstellt.

Konstruktoren

istrstream(char * cp)

> Zeichen werden aus der (mit 0 beendeten) Zeichenkette *cp* entnommen. Das abschließende Nullzeichen ist nicht Teil der Zeichenfolge. Suchvorgänge (*istream::seekg()*) sind in diesem Feld erlaubt.

istrstream(char * cp, int len)

> Zeichen werden aus dem bei *cp* beginnenden Feld entnommen, das *len* byte lang ist. Suchvorgänge (*istream::seekg()*) sind an beliebiger Stelle im Feld erlaubt.

ostrstream()

> Speicherbereich wird zur Aufnahme gespeicherter Zeichen dynamisch zugeordnet.

ostrstream(char * cp, int n, int mode)

> Zeichen werden in dem bei *cp* beginnenden Feld, das *n* byte lang ist, gespeichert.Wenn *ios::ate* oder *ios::app* in *mode* gesetzt sind, wird angenommen, dass *cp* eine mit 0 endende Zeichenkette ist; das Speichern beginnt beim Nullzeichen. Anderenfalls wird bei *cp* mit der Speicherung begonnen. Suchvorgänge sind an beliebiger Stelle im Feld erlaubt.

strstream()

> Speicherkapazität wird dynamisch zur Aufnahme der gespeicherten Zeichen zugewiesen.

strstream(char * cp, int n, int mode)

Zeichen werden in dem bei *cp* beginnenden Feld - mit der Länge von *n* Bytes -gespeichert. Wenn *ios::ate* oder *ios::app* in *mode* gesetzt sind, wird *cp* als mit 0 beendete Zeichenkette behandelt, und die Speicherung beginnt beim Nullzeichen. Im anderen Fall beginnt das Speichern bei *cp*. Suchvorgänge sind an beliebiger Stelle imFeld erlaubt.

strstreambase-Elemente

strstreambuf * ssb = iss.rdbuf()
strstreambuf * ssb = oss.rdbuf()
strstreambuf * ssb = ss.rdbuf()

rdbuf() darf nur in abgeleiteten Klassen verwendet werden. Die Funktion liefert dasmit *iss*/*oss*/*ss* verknüpfte *strstreambuf*-Objekt.

ostrstream-Elemente

char * cp=oss.str()

Ein Zeiger auf das verwendete Feld wird geliefert und das Feld "eingefroren". Nachdem *str* aufgerufen wurde, ist die Auswirkung eines versuchten Speicherns weiterer Zeichen in *oss* nicht definiert. Wenn *oss* mit einem explizit bereitgestellten Feld konstruiert wurde, ist *cp* ein Zeiger auf dieses Feld. Anderenfalls zeigt *cp* auf einen dynamisch zugewiesenen Bereich. Bis zum Aufruf von *str()* ist *oss* für das Löschen des dynamisch zugewiesenen Bereichs verantwortlich. Nach der Rückkehr von der Funktion *str()* gehört das Feld zum Zuständigkeitsbereich des Benutzerprogramms an.

int i=oss.pcount()

Die Anzahl der Bytes, die im Puffer gespeichert sind, wird geliefert. Diese Funktion ist hauptsächlich von Nutzen, wenn binäre Daten gespeichert wurden und *oss.str()* nicht auf eine mit 0 beendete Zeichenkette gerichtet ist.

strstream-Element

char * cp=ss.str()

Ein Zeiger auf das verwendete Feld wird geliefert und das Feld "eingefroren". Nachdem Aufruf von *str()* ist die Wirkung eines Versuchs, weitere Zeichen in *ss* zu speichern, nicht definiert. Wenn *ss* mit einer expliziten Feldangabe angelegt wurde, ist *cp*ein Zeiger auf das Feld. Im anderen Fall zeigt *cp* auf den dynamisch zugewiesenen Bereich. Bis zum Aufruf von *str* obliegt *ss* das Löschen des dynamisch zugewiesenen Bereichs. Nach der Rückkehr vom Funktionsaufruf *str()* gehört das Feld aber dem Verantwortlichkeitsbereich des Benutzerprogramms an.

BEISPIEL Das folgende Programm definiert eine Zeichenkette *str1* und liest diese mit dem Operator >> wie einen Eingabestrom. Jedes aus der Zeichenkette gelesene Zeichen wird nach *cout* ausgegeben.

```
#include <iostream.h>
#include <strstream.h>
const char * const str1 = "Test-String zur Pruefung von strstream\n";
/* const wird verwendet, um sicherzustellen, dass die Zeichenkette */
/* und der Zeiger darauf nicht verändert werden können             */
int main()
{
  istrstream is((char*) str1);
  /* Variable is wird zur Verwendung der Zeichenkette str1 deklariert */
  is.unsetf(ios::skipws);
  /* Standardmäßig überspringt istrstream Zwischenraum bei der        */
  /* Eingabe. Das Standardverhalten wird durch Löschen des Flags      */
  /* skipws verändert; Zwischenraum wird bei der Eingabe dann         */
  /* nicht mehr übersprungen                                          */
  while (EOF != is.peek())
  {
    char c;
    is >> c;
    /* Beachten Sie: Auf die Zeichenkette wird wie auf eine           */
    /* Eingabezeichenkette zugegriffen.                               */
    cout << c;
  }
  return 0;
}
```

Das Ergebnis der Programmausführung ist:

```
Test-String zur Pruefung von strstream
%  CCM0998 Verbrauchte CPU-Zeit: 0.0007 Sekunden
```

SIEHE AUCH

istream, *sstreambuf*

5 Literatur

Die Handbücher sind online unter https://bs2manuals.ts.fujitsu.com zu finden.

[1] **CRTE** (BS2000)
Common RunTime Environment
Benutzerhandbuch

Zielgruppe
Programmierer und Systemverwalter im BS2000
Inhalt
Beschreibung der gemeinsamen Laufzeitumgebung für COBOL85-, C- und C++-Objekte sowie für "Fremdsprachmix":

- Komponenten des CRTE
- Programmkommunikationsschnittstelle ILCS
- Bindebeispiele

[2] **C** (BS2000)
C-Bibliotheksfunktionen
Beschreibung

Zielgruppe
C- und C++-Anwender im BS2000
Inhalt

- Beschreibung aller C-Funktionen und Makros, die das C-Laufzeitsystem zur Verfügung stellt;
- Grundlegende Informationen, Programmierhinweise und Beispiele zu:Dateiverarbeitung, STXIT- und Contingency-Routinen, Lokalität

[3] **C/C++** (BS2000)
C/C++-Compiler
Benutzerhandbuch

Zielgruppe
C- und C++-Anwender im BS2000
Inhalt

- Beschreibung aller Tätigkeiten zum Erzeugen von ablauffähigen C- und C++-Programmen: Übersetzen, Binden, Laden, Testen;
- Programmierhinweise und weitergehende Informationen zu: Programmablaufsteuerung, Sprachverknüpfung, C- und C++-Sprachumfang des C++-Compilers

[4] **Die C++ Programmiersprache**
von Bjarne Stroustrup

Zielgruppe
Das Buch richtet sich an C++-Programmierer und an Programmierer, die C++ erlernen wollen.
Inhalt
Dieses Standardwerk des C++-Erfinders Bjarne Stroustrup enthält eine Einführung in C und C++ mit vielen Beispielen, drei Kapitel zur Softwareentwicklung mit C++ und ein vollständiges Referenzhandbuch.