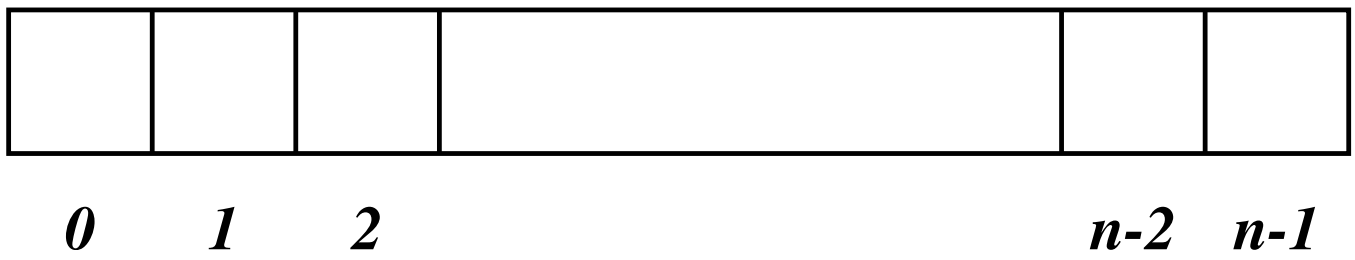
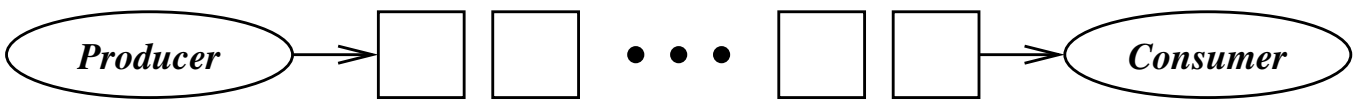
- Die erste Fassung wurde 1984 von Bjarne Stroustrup entwickelt, um im Vergleich zur C-stdio-Bibliothek eine höhere statische Typsicherheit zu erhalten. Stroustrup führte auch die Operatoren $<<$ und $>>$ ein in Anlehnung der Umleitungssyntax der Shell unter UNIX.
- 1989 wurde die Bibliothek von Jerry Schwarz neu entworfen. Zu den Verbesserungen gehörten eine höhere Effizienz. Neu waren ebenso die Manipulatoren von Andrew Koenig.
- Während des ISO-Standardisierungsprozesses wurde die Unterstützung für beliebige Zeichensätze und Locales aufgenommen. Dies führte zu den Template-Klassen *basic_istream*, *basic_ostream* usw. die etwa mit **char** instantiiert werden können (was dann *istream* und *ostream* entspricht) oder auch mit anderen Datentypen für Zeichen wie etwa **wchar_t**.

Zwei Abstraktionen sind üblich für das Ein- und Ausgaben von Bytes:

- Eine Datei, die aus n Bytes besteht an den Positionen 0 bis $n-1$, vergleichbar mit einem Array:

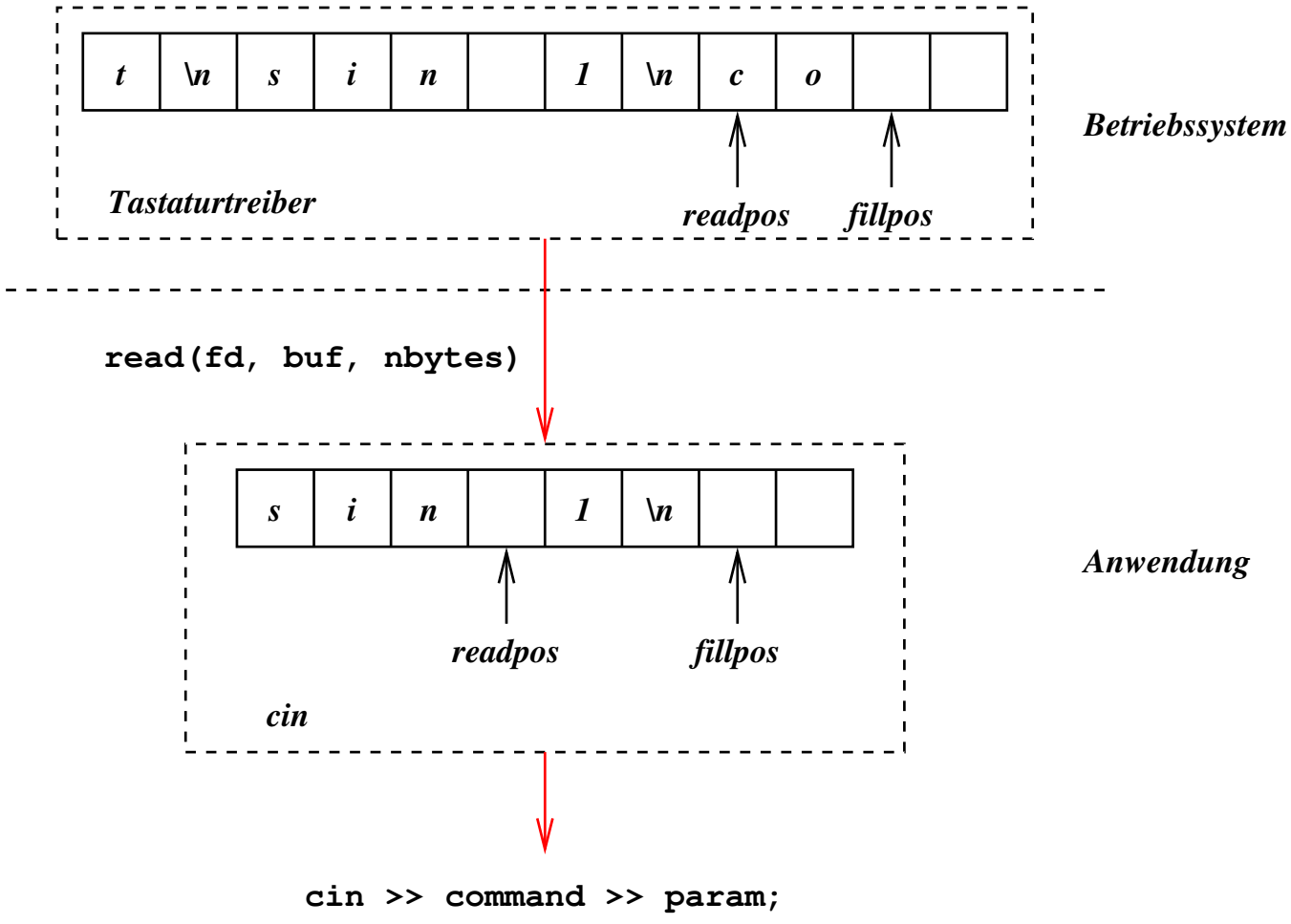


- Kommunikation zwischen zwei aktiven Partnern:



- UNIX bietet eine vereinheitlichte Schnittstelle an für beide Abstraktionen auf Grundlage einer kleinen Menge einfach gehaltener Systemaufrufe.
- Analog dazu ermöglicht die Ein- und Ausgabe-Bibliothek von C++ ebenfalls eine vereinheitlichte Schnittstelle. Entsprechend kann etwa ein *istream* mit einer Tastatur, einer Datei auf der Festplatte, einem Prozess auf einem anderen Rechner oder, in Erweiterung zu der reinen UNIX-Schnittstelle, mit einer Zeichenkette im lokalen Speicher verbunden sein.

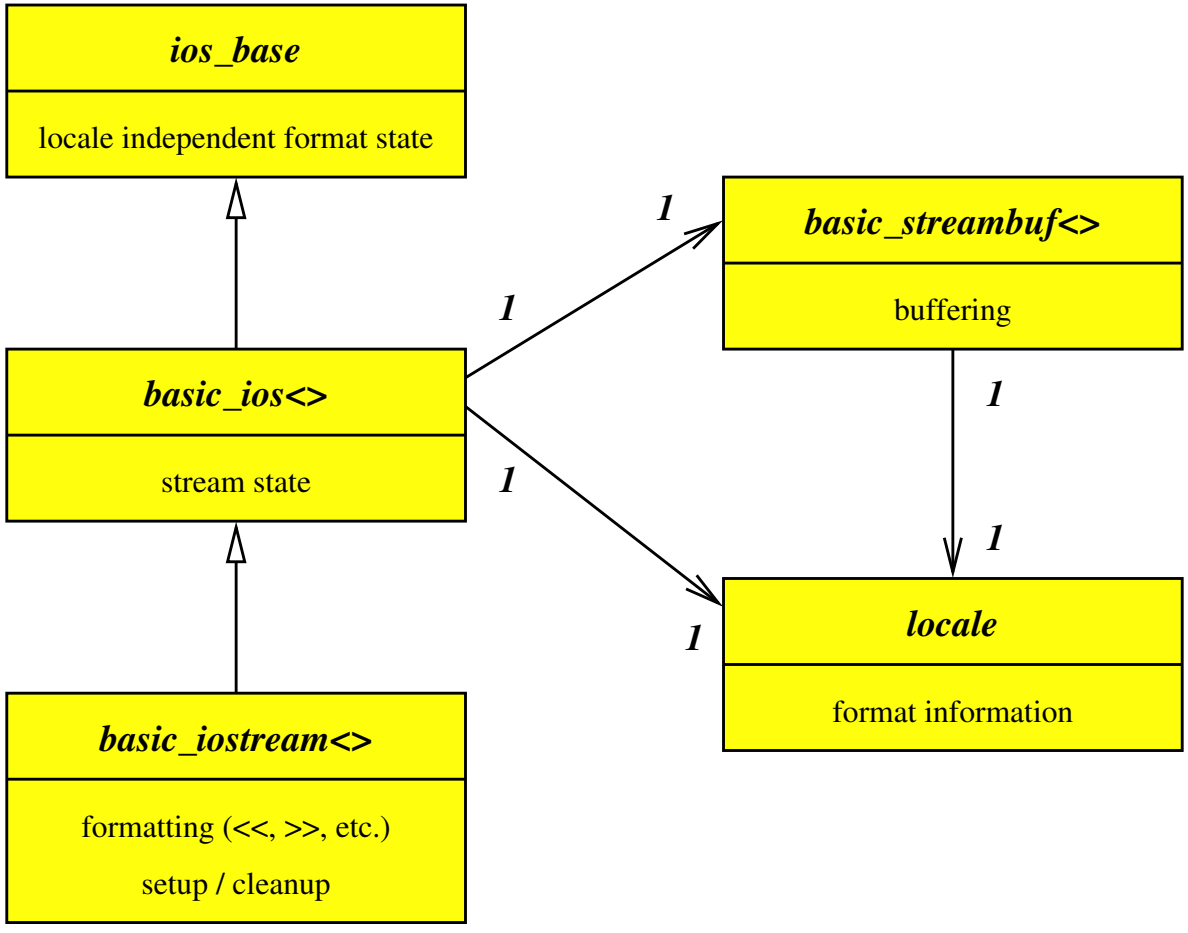
Streams als Puffer



- Eingaben von einer Tastatur werden auf zwei Ebenen gepuffert:
  - Das Betriebssystem speichert die eingetippten Zeichen, bis sie von der lesenden Anwendung abgeholt werden. Dies erlaubt das Voraustippen von Eingabezeichen und befreit die Anwendung davon, sich mit Ereignissen wie dem Drücken einer Taste zu beschäftigen.
  - Wenn die Anwendung dann etwas liest, versucht die Bibliothek, die gesamte verfügbare Eingabe (bis zu einem gewissen Limit) zu übernehmen und in einem Puffer zu speichern. Dies dient der Minimierung der Systemaufrufe, da dann folgende Lese-Operationen u.U. aus dem Puffer versorgt werden können.
- Puffer-Mechanismen gibt es auch für andere Eingabearten (wie etwa von Dateien) und auch bei Ausgaben.

- Bjarne Stroustrup: “An I/O facility should be easy, convenient, and safe to use; efficient and flexible; and above all, complete.”
- Effizienz: Die C++-Bibliothek sollte auch in dieser Hinsicht besser sein als die stdio-Bibliothek von C.
- Typsicherheit: *printf* und *scanf* bieten in C keinen Schutz gegen Übereinstimmungsfehler der Formatzeichenkette und den zugehörigen Parametern. In C++ wird das Problem durch das Auswählen der korrekten Funktionen mit Hilfe des Überladens gelöst.

- Erweiterbarkeit: Neben den elementaren Typen können auch selbst definierte Klassen mit einer Unterstützung für Ein- und Ausgabe-Operatoren versehen werden.
- Flexibilität: Streams müssen nicht zwangsweise mit Ein- und Ausgabeverbindungen des Betriebssystems zusammenfallen, sie können mit anderen Implementierungen verbunden werden wie etwa Zeichenketten.
- Internationalisierung: Andere Zeichensätze (Unicode!) und Lokalisierungen sollten unterstützt werden.



- Dieses Diagramm wurde dem Werk *The C++ Programming Language*, 3. Auflage von Bjarne Stroustrup, Seite 606 entnommen und geringfügig an UML angepasst.

Folgende **bool**-wertige Statusvariablen werden u.U. *nach* einer Stream-Operation gesetzt:

Statusvariable	Beschreibung
ios_base::*eofbit*	Das Ende der Eingabe wurde erkannt.
ios_base::*failbit*	Die letzte Operation war nicht erfolgreich.
ios_base::*badbit*	Der Stream ist in einem undefinierten Zustand; eine weitere Nutzung ist nicht sinnvoll.

Folgende Test-Operationen werden von *basic_ios*<> angeboten:

Methode	Beschreibung
good()	liefert **true** genau dann, wenn keiner dieses Status-Variablen gesetzt ist
eof()	liefert **true** genau dann, wenn *ios_base*::*eofbit* gesetzt ist
fail()	liefert **true** genau dann, wenn *ios_base*::*failbit oder ios_base*::*badbit* gesetzt ist
bad()	liefert **true** genau dann, wenn *ios_base*::*badbit* gesetzt ist

- Wenn einer der Status-Variablen gesetzt ist, scheitern alle folgenden Operationen. Mit Hilfe der Methode *clear()* ist es möglich, die Status-Variablen zurückzusetzen und weiterzumachen.
- Ein Stream-Objekt, das zu **bool** konvertiert wird, liefert den Wert von !*fail*() zurück.
- Selbst-definierte Eingabe-Operatoren sollten *ios_base*::*failbit* setzen, wenn sie eine Eingabe vorfinden, die syntaktisch nicht zulässig ist.

IntSum1.C

```
#include <iostream>
int main() {
   int sum(0); int i;
   while (std::cin >> i) sum += i;
   std::cout << sum << std::endl;
} // main
```

IntSum2.C

```
#include <iostream>
int main() {
   int sum(0); int i;
   std::cin >> i;
   while (!std::cin.eof()) {
      sum += i; std::cin >> i;
   }
   std::cout << sum << std::endl;
} // main
```

```
dublin$ echo "1 2\c" | IntSum1
3
dublin$ echo "1 2\c" | IntSum2
1
dublin$
```

TestContact.C

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

using namespace std;

int main() {
   Contact contact;
   while (cin >> contact) {
      cout << contact << endl;
   }
} // main
```

- Eigene Klassen (wie etwa *Contact* in diesem Beispiel) können mit einer Unterstützung für die Operatoren $>>$ und $<<$ versehen werden.

```
dublin$ cat testinput
("George W. Bush", "White House")
(  "George W. Bush","White House" )
("George W. Bush",)
("George W. Bush", "White House")
dublin$ TestContact <testinput
("George W. Bush", "White House")
("George W. Bush", "White House")
dublin$
```

- In diesem Beispiel bestehen Kontakte aus einem Namen und einer Adresse. Für die Ein- und Ausgabe werden beide Zeichenketten in Anführungszeichen gesetzt und als Tupel repräsentiert.

- Im Falle sich wiederholender lexikalischer Elemente (wie etwa Zeichenketten in Anführungszeichen) lohnt es sich, diese jeweils durch eigene Klassen zu repräsentieren.
- Entsprechend gibt es in diesem Beispiel neben der Klasse *Contact* auch noch die Klasse *QuotedString*.

QuotedString.h

```
class QuotedString {
   public:
      typedef std::string string;

      // constructors
      QuotedString();
      QuotedString(const std::string& s);

      // accessors
      const std::string& get() const;

      // mutators
      void set(const std::string& s);

   private:
      std::string s; // without quotes
}; // class QuotedString

std::istream& operator>>(std::istream& in,
   QuotedString& qs);
std::ostream& operator<<(std::ostream& out,
   const QuotedString& qs);
```

QuotedString.C

```
istream& operator>>(istream& in, QuotedString& qs) {
   char ch;
   bool failure(true);

   if (in >> ch && ch == '"') {
      string s("");

      while (in.get(ch) && ch != '"') {
         s += ch;
      }
      if (in) {
         qs.set(s); failure = false;
      }
   }
   if (failure) {
      in.setstate(ios::failbit);
   }
   return in;
} // operator>>
```

- Die Eingabe-Variable darf solange nicht verändert werden, bis klar ist, dass die gesamte Eingabe-Operation erfolgreich war.
- Es kann nicht nur an den Eingabe-Funktionen scheitern, die hier aufgerufen werden. Wenn beispielsweise ein 'x' anstelle eines beginnenden Anführungszeichen gefunden wird, dann muss *ios*::*failbit* explizit gesetzt werden.
- Zu beachten ist der Unterschied zwischen *in* >> *ch* und *in*.*get*(*ch*). Während *in* >> *ch* führende Leerzeichen überspringt, liefert *in*.*get*(*ch*) das nächste Eingabezeichen selbst dann, wenn es sich dabei um ein Leerzeichen handelt.

Contact.C

```
istream& operator>>(istream& in, Contact& contact) {
   char ch;
   QuotedString name, address;

   if (in >> ch && ch == '(' &&
         in >> name &&
         in >> ch && ch == ',' &&
         in >> address &&
         in >> ch && ch == ')') {
      contact.set_name(name.get());
      contact.set_address(address.get());
   } else {
      in.setstate(ios::failbit);
   }
   return in;
} // operator>>
```

QuotedString.C

```
ostream& operator<<(ostream& out, const QuotedString& qs) {
   return out << '"' << qs.get() << '"';
} // operator<<
```

Contact.C

```
ostream& operator<<(ostream& out, const Contact& contact) {
   return out << '(' <<
      QuotedString(contact.get_name()) <<
      ", " <<
      QuotedString(contact.get_address()) <<
      ')';
} // operator<<
```

- Ausgabeformate sollten normalerweise den Eingabeformaten entsprechen.
- Dies ist hier nicht vollständig erfüllt, falls der Name oder die Adresse eines Kontakts ebenfalls Anführungszeichen enthalten sollte.

- Im Falle von Klassenhierarchien ist es sinnvoll, die Ein- und Ausgabe-Operatoren nur einmal zu definieren und diese auf entsprechende virtuelle reguläre Methoden verweisen zu lassen.
- Die zugehörigen virtuellen Methoden sollten dann privat deklariert werden, damit sie nicht direkt verwendet werden können, sondern nur indirekt über die Operatoren.
- Damit die privaten Methoden noch zugänglich bleiben für die Ein- und Ausgabe-Operatoren, müssen sie den Zugang explizit über **friend**-Deklarationen erhalten.

PrintableObject.h

```
class PrintableObject {
   friend std::ostream& operator<<(std::ostream& out,
      const PrintableObject& object);
   public:
      virtual ~PrintableObject() { cleanup_printable(); };
   private:
      virtual void print(std::ostream& out) const = 0;
      virtual void cleanup_printable() {};
};
inline std::ostream& operator<<(std::ostream& out,
      const PrintableObject& object) {
   object.print(out); return out;
};
```

- In abstrakten Klassen sollte es immer einen virtuellen Destruktor geben. Im Falle drohender Mehrfachvererbung (wie hier) sollte der auf eine ebenfalls virtuellen Methode basieren. Der Destruktor müsste dann nicht mehr virtuell sein, aber in dieser Form geht es durch den gcc ohne Warnungen.

ScannableObject.h

```
class ScannableObject {
   friend std::istream& operator>>(std::istream& in,
      ScannableObject& object);
   public:
      virtual ~ScannableObject() { cleanup_scannable(); };
   private:
      virtual void scan(std::istream& in) = 0;
      virtual void cleanup_scannable() {};
};

inline std::istream& operator>>(std::istream& in,
      ScannableObject& object) {
   object.scan(in); return in;
};
```

- Analog kann dies für den Eingabe-Operator geschehen.

IPv4Address.h

```
class IPv4Address: public PrintableObject, public ScannableObject {
   private:
      unsigned char octets[4];
      virtual void print(std::ostream& out) const;
      virtual void scan(std::istream& in);
};
```

- Eine mehrfache Erweiterung ist möglich durch die mehrfache Angabe von Basisklassen.
- Da sich Namenskonflikte hier nicht mehr elegant auflösen lassen, sollte dies entweder bereits zuvor geschehen (wie etwa durch *cleanup_printable* vs. *cleanup_scannable*). Wenn die Basisklassen dafür noch nicht vorbereitet sind, sollten Hilfsklassen eingesetzt werden.

IPv4Address.C

```
void IPv4Address::print(ostream& out) const {
   out << (int) octets[0] << "." << (int) octets[1] << "."
      << (int) octets[2] << "." << (int) octets[3];
} // IPv4Address::print
```

- Es ist zu beachten, dass der Datentyp **char** sowohl als kleiner numerischer Datentyp verwendet wird als auch eben als Datentyp für Zeichen. Der entsprechende überladene Operator geht von einem Zeichen aus, so dass explizite Casts notwendig sind, um Werte des Typs **char** als Zahlenwerte auszugeben.