Themen

- Bitweise-Operatoren
- Formatierte und unformatierte Ausgabe (Streams)
  - Bildschirm
  - Datei

Operatoren zur Bitmanipulation

- Zählen zu den binären Operatoren
- Verknüpfen zwei Integer Werte (char, short, int, long und enum)
- Das Ergebnis hat denselben Typ wie die Operanden
- Operatoren:
  - & (bitweises Und)
  - | (bitweises Oder)
  - ^ (bitweises exclusives Oder)
  - ~ (bitwieses Nicht)
  - << , >> (bitweise links und rechts Verschiebung)

Beispiel & (Und)

```
char a=32;      // 00100000
char b=8;       // 00001000
char c=a & b;
```

Beispiel & (Und)

```
char a=32;      // 00100000
char b=8;       // 00001000
char c=a & b; // 00000000
```

Beispiel 2 & (Und)

```
char a=41;          // 00101001
char b=25;          // 00011001
char c=a & b;
```

Beispiel 2 & (Und)

```
char a=41;          // 00101001
char b=25;          // 00011001
char c=a & b;       // 00001001
```

Beispiel | (Oder)

```
char a=32;      // 00100000
char b=8;       // 00001000
char c=a | b;
```

Beispiel | (Oder)

```
char a=32;      // 00100000
char b=8;       // 00001000
char c=a | b; // 00101000
```

Beispiel 2 | (Oder)

```
char a=41;          // 00101001
char b=25;          // 00011001
char c=a | b;
```

Beispiel 2 | (Oder)

```
char a=41;          // 00101001
char b=25;          // 00011001
char c=a | b;       // 00111001
```

Beispiel ^ (Exklusives Oder)

```
char a=32;      // 00100000
char b=8;       // 00001000
char c=a ^ b;
```

Beispiel ^ (Exklusives Oder)

```
char a=32;      // 00100000
char b=8;       // 00001000
char c=a ^ b; // 00101000
```

Beispiel 2 ^ (Eklusives Oder)

```
char a=41;          // 00101001
char b=25;          // 00011001
char c=a ^ b;
```

Beispiel 2 ^ (Exklusives Oder)

```
char a=41;          // 00101001
char b=25;          // 00011001
char c=a ^ b;       // 00110000
```

Beispiel ~ (Nicht)

```
char a = 45;     // 00101101
char b = ~a;
```

Beispiel ~ (Nicht)

```
char a = 45;      // 00101101
char b = ~a;      // 11010010
```

Beispiel << , >> (Verschiebung)

```
int a=45;                  // 00101101
int b = a << 2;
int c = a >> 1;
int d = a << 8;
```

Beispiel << , >> (Verschiebung)

```
char a=45;              // 00101101
char b = a << 2;        // 10110100
char c = a >> 1;        // 00010110
char d = a << 8;        // 00000000
```

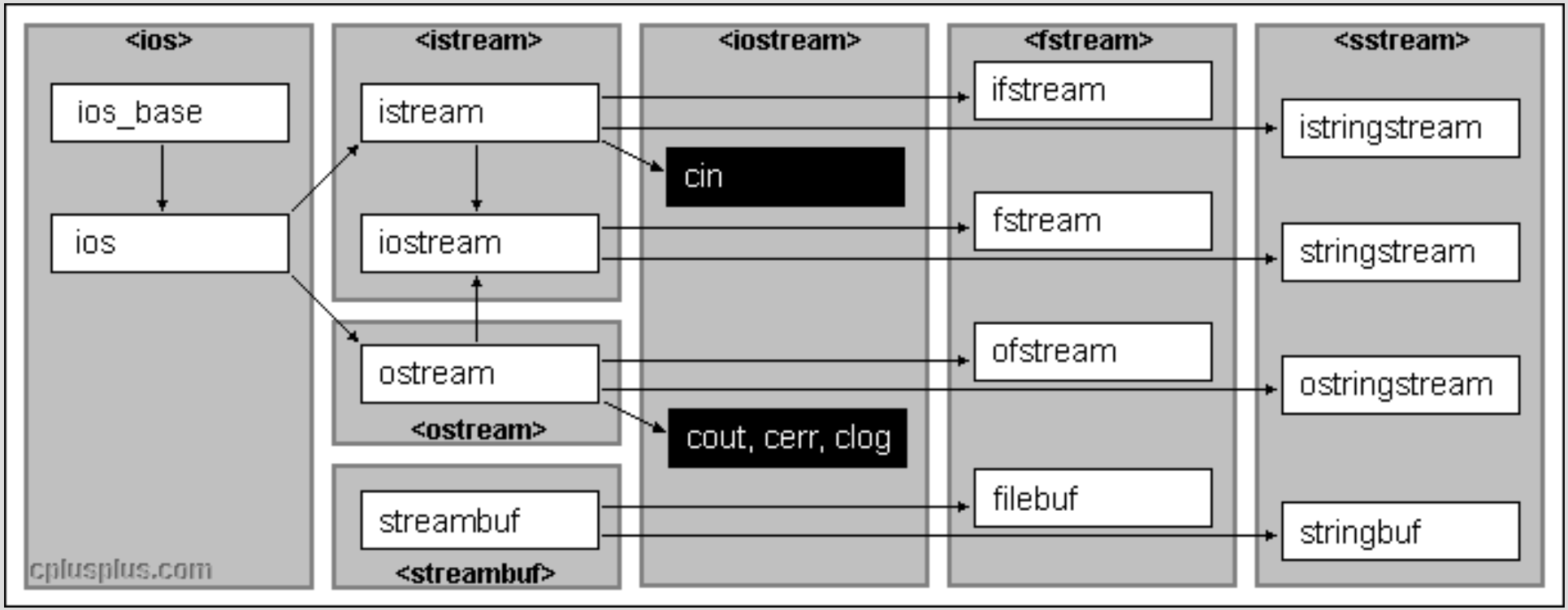
Je nach Compiler bleibt das Vorzeichen bei >> mit vorzeichenbehafteten Zahlentypen erhalten!

```
char a = -12;
char b = a >> 2;  // -3
```

Beispiel 2 << , >>

```
char a = 64;       // 01000000
char b = a << 1; // 10000000 = -128 !
char c = b >> 1; // 11000000 = -64 !
```

Stream-Klassen



Quelle: http://www.cplusplus.com/reference/iostream/

<iostream>

- beinhaltet Deklarationen von Klassen für die Bildschirmausgabe und die Eingabe von der Tastatur
  - **`cin`** : liest Daten vom Standardeingabegerät
  - **`cout`** : schreibt Daten auf das Standardausgabegerät
  - **`cerr`** : zum Ausgeben von Fehlermeldungen auf das Standardausgabegerät
  - **`clog`** : zum Ausgeben von Protokolldaten auf das Standardausgabegerät

Standardausgabe- und Eingabe

- Das Standardausgabegerät ist auf neuen Rechnern meist der Bildschirm
- Das Standardeingabegerät ist auf neuem Rechnern meist die Tastatur
- Die meisten Betriebsysteme erlauben es, die Eingabe und Ausgabe auf andere Geräte (z.B. Dateien) umzuleiten

z.B. **`./fakultaet 12 > ergebnis.txt`**

Stream-Objekte

- Die Ausgabe auf ein Stream-Objekt erfolgt mit dem “Ausgabeoperator” (<<)
- Die Eingabe von einem Stream-Objekt erfolgt mit dem “Eingabeoperator” (>>)
- Manche Stream-Objekte dienen lediglich zur Ausgabe (**`cout`**) von Daten, andere nur zur Eingabe (**`cin`**) und wieder andere können für beides verwendet werden (z.B. **`fstream`**).

Unformatierte Ausgabe mit `cout`

- cout ist eine Instanz der Klasse **`ostream`**
- besitzt die Methoden **`put`** und **`write`** zur unformatierten Ausgabe

```
ostream& ostream::put(char c);

ostream& ostream::write(
const char* s, streamsize n);
```

- die Ausgabe wird geschrieben wie angegeben, d.h. ohne weitere Formatierung

Beispiel – unformatierte Ausgabe mit cout

```
#include <iostream>

int main()
{
  const char* vorname=”Hans”;
  const char* nachname=”Meier”;
  std::cout.write(vorname,5);
  std::cout.put(' ');
  std::cout.write(nachname,6);
  std::cout.out('\n');
}
```

Formatierte Ausgabe mit `cout`

- Die formatierte Ausgabe erfolgt über den Ausgabeoperator <<
- Ausgabe wird über zusätzliche Formatierungsanweisungen formatiert

Formatierungsanweisungen

- Deklarationen für Formatierungs-anweisungen findet man z.B. in den Klassen *basic_ios* und *ios_base (basic_ios* ist von *ios_base* abgeleitet)
- *istream* (*cin*) und *ostream* (*cout*) sind von *basic_ios* abgeleitet, d.h. sie erben alle Eigenschaften für die formatierte Ein- und Ausgabe

Formatierungsanweisung

- häufige Formatierungen:
  - Änderung der Basis bei der Ausgabe von Zahlen
  - Festlegen der Breite einer Ausgabe
  - Festlegen der Ausrichtung bei der Ausgabe
  - Ausgabe von **`true`** und **`false`** als Wort und nicht als 0 und 1
  - Festlegen der Genauigkeit von Gleitkommazahlen
  - Festlegen der Reprräsentation von Gleitkommazahlen

getf / setf / flags

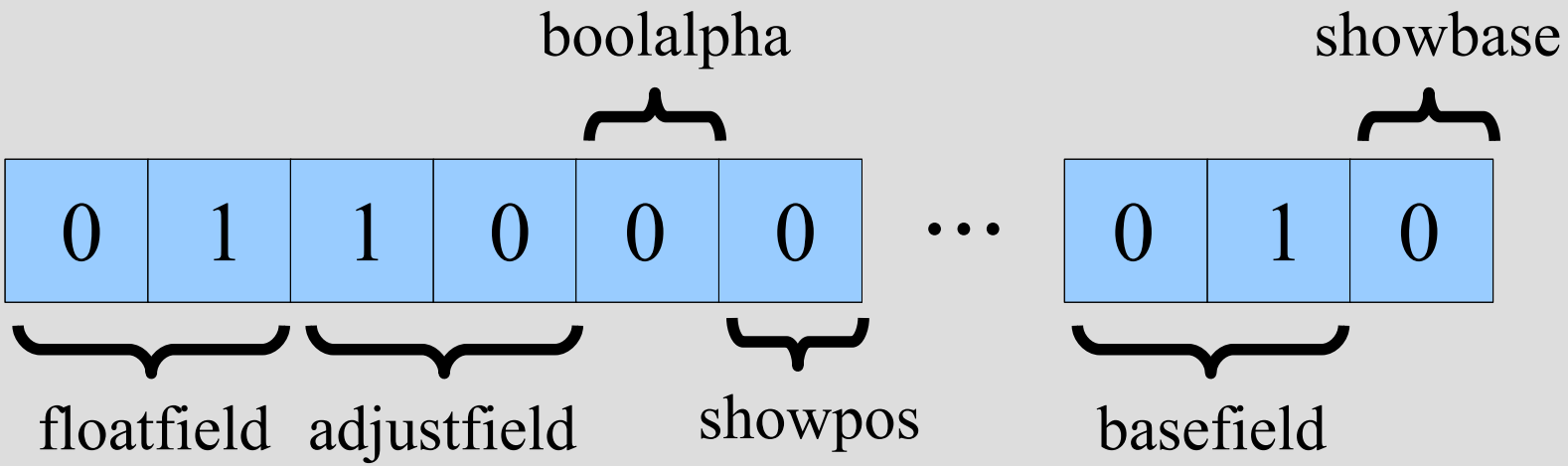
- Formatierungen bestehen meist aus einzelnen Bit-Werten, welche mit der Methode *setf* (set flags) gesetzt und mit der Methode *flags* gelesen werden.
- Alternativ kann man die Werte auch mit der *flags* Methode setzen und abfragen
- *setf* setzt einzelne Werte und *flags* setzt alle Werte auf einmal
- Formatierung bleibt so lange gesetzt, bis sie explizit gelöscht wird

Signatur von setf / unsetf / flags

```
class ios_base{
      ...
   fmtflags setf(fmtflags f);
   fmtflags setf(fmtflags f,
                 fmtflags mask);
   void unsetf(fmtflags mask);
   fmtflags flags() const;
   fmtflags flags(fmtflags f);
      ...
};
```

Formatierungsflags

Das Format der Flags ist abhängig von der Implementierung und kann auf jedem Compiler anders aussehen, die Namen der Flags sind jedoch standardisiert.



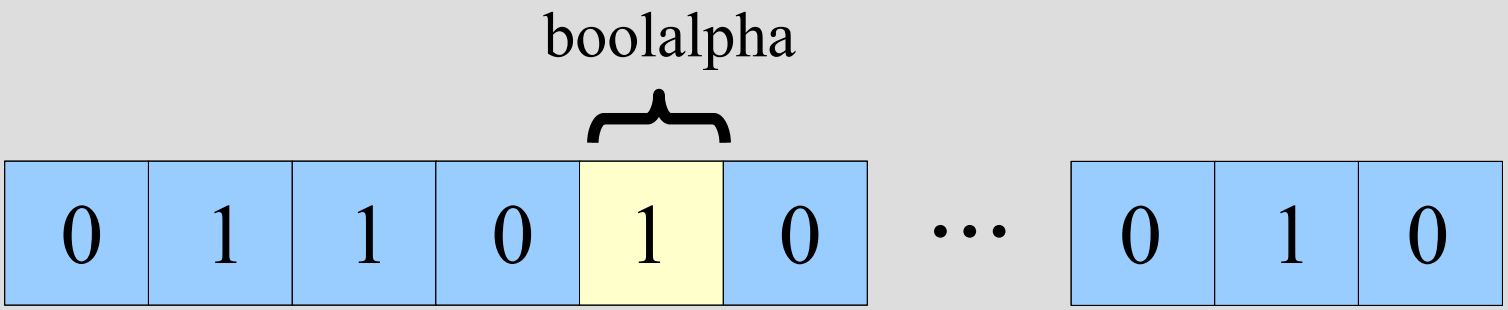
Beispiel – formatierte Ausgabe von boolean Werten

```
std::cout << true << " ";
std::cout << false << " ";
std::cout.setf(
     std::ios_base::boolalpha);
std::cout << true << " ";
std::cout << false << "\n";
```

1 0 true false

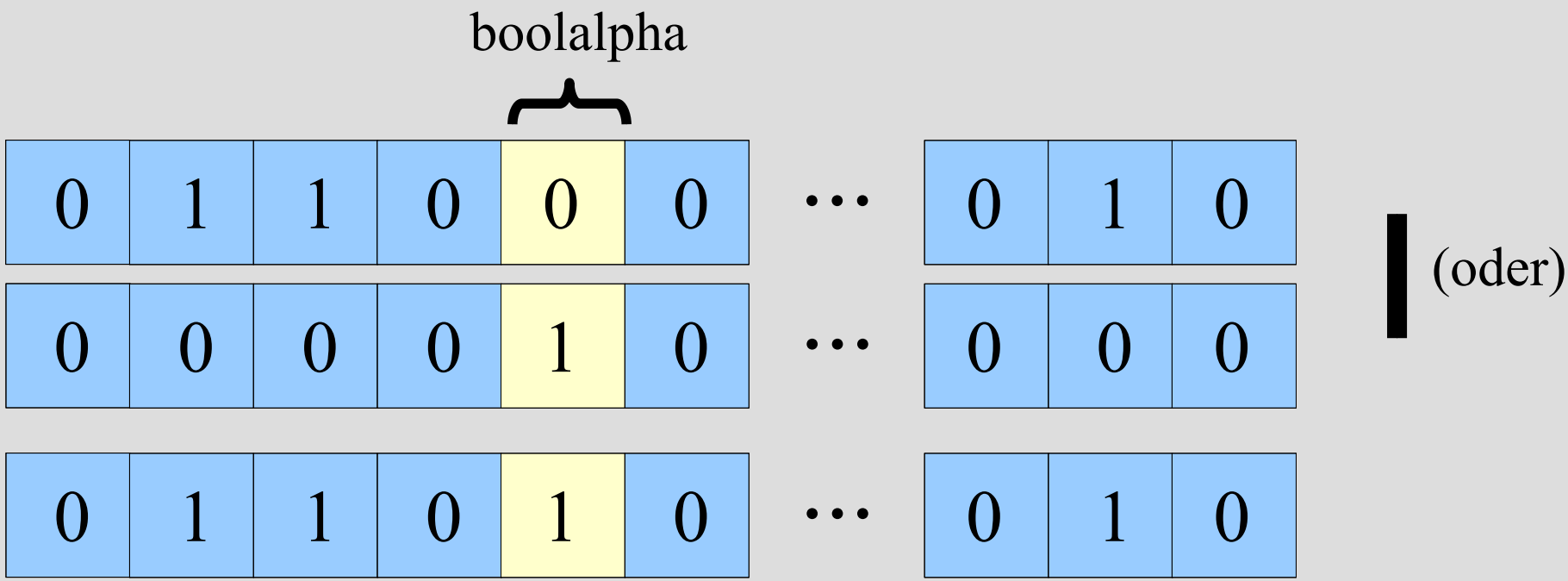
Setzen eines Flags

```
std::cout.setf(std::ios_base::boolalpha);
```



Setzen eines Flags 2/2

```
std::cout.flags(std::cout.flags() | std::ios_base::boolalpha);
```



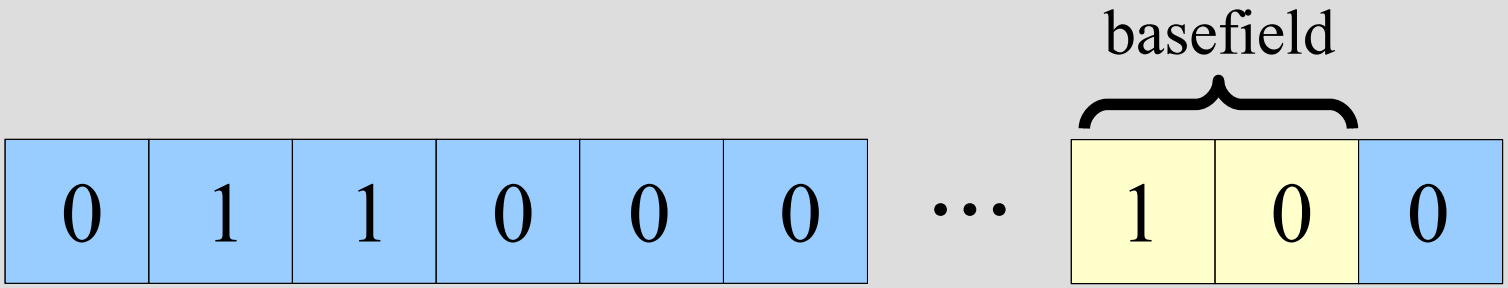
Beispiel – formatierte Ausgabe von Integer-Werten

```
std::cout << 16 << " ";
std::cout.setf(
     std::ios_base::hex,
     std::ios_base::basefield);
std::cout << 16 << " ";
std::cout.setf(
     std::ios_base::oct,
     std::ios_base::basefield);
std::cout << 16 << "\n";
```

16 10 20

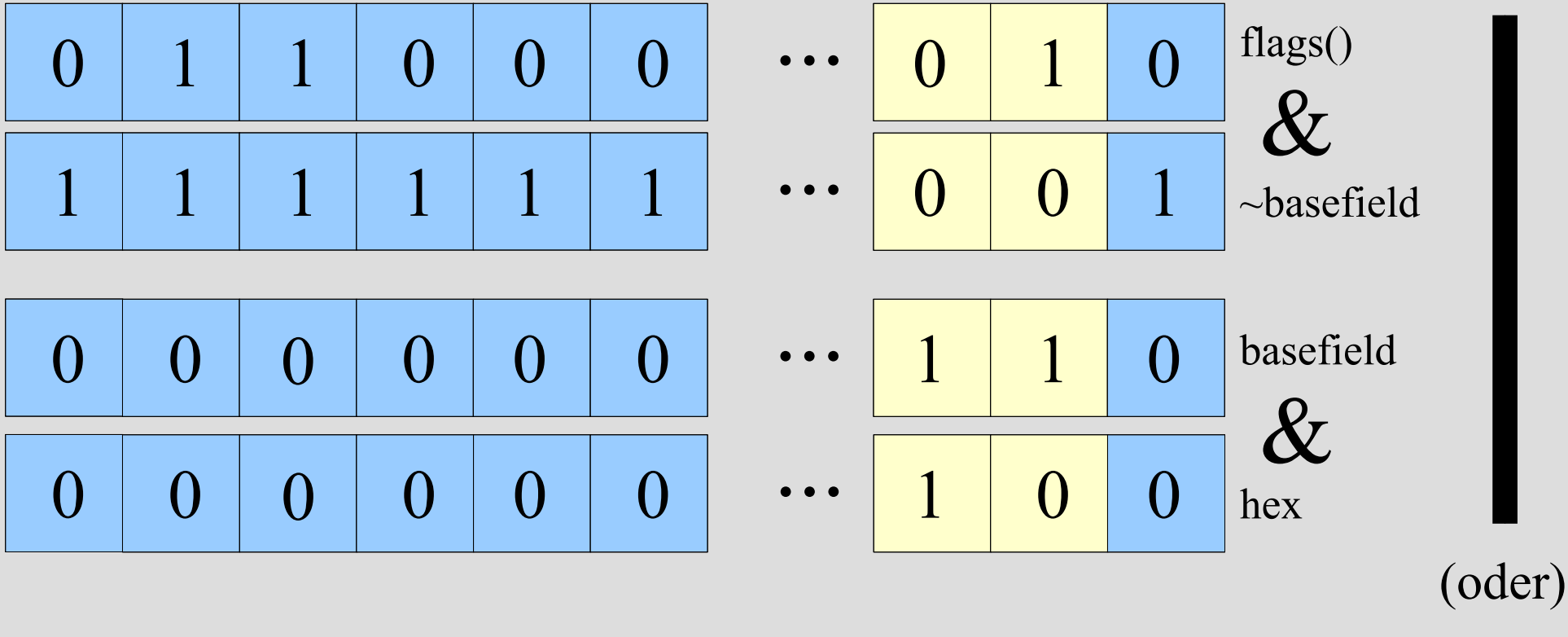
Formatierungsflags

```
std::cout.setf(std::ios_base::hex,std::ios_base::basefield);
```



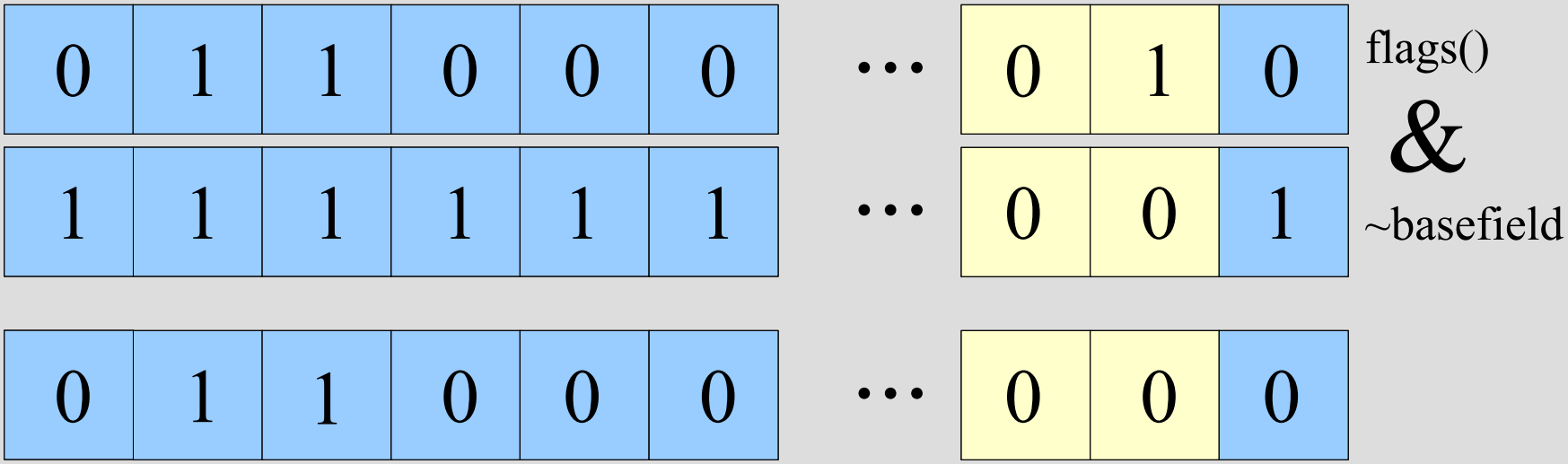
Formatierungsflags

```
cout.flags( (cout.flags() & ~ios_base::basefield) |
            (ios_base::basefield & ios_base::hex) );
```



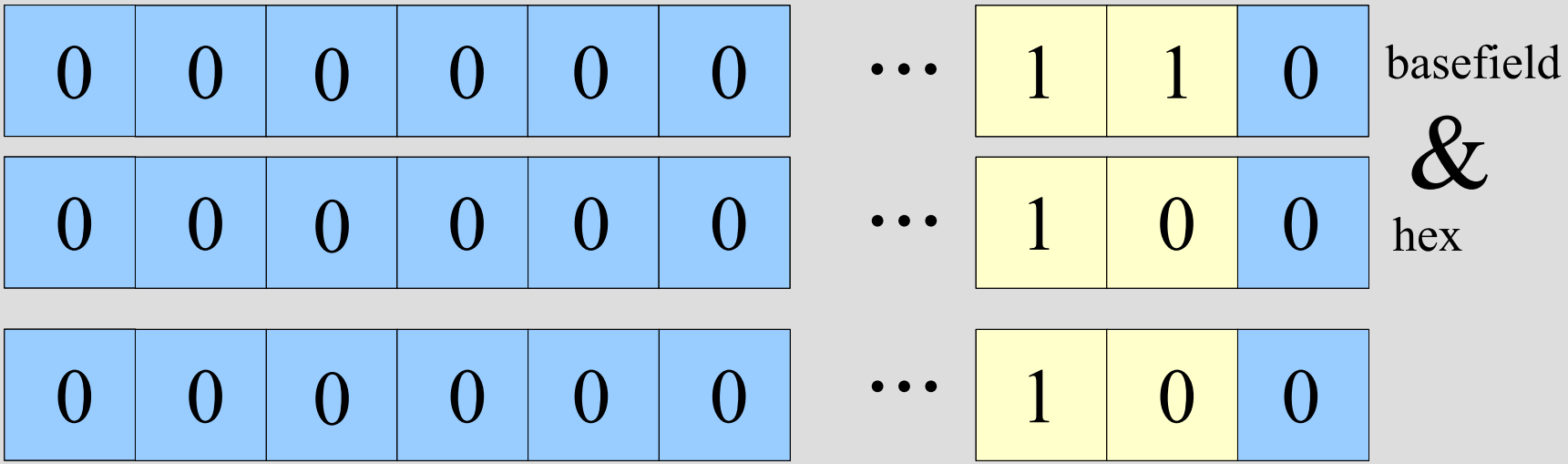
Formatierungsflags

```
cout.flags( (cout.flags() & ~ios_base::basefield) |
            (ios_base::basefield & ios_base::hex) );
```



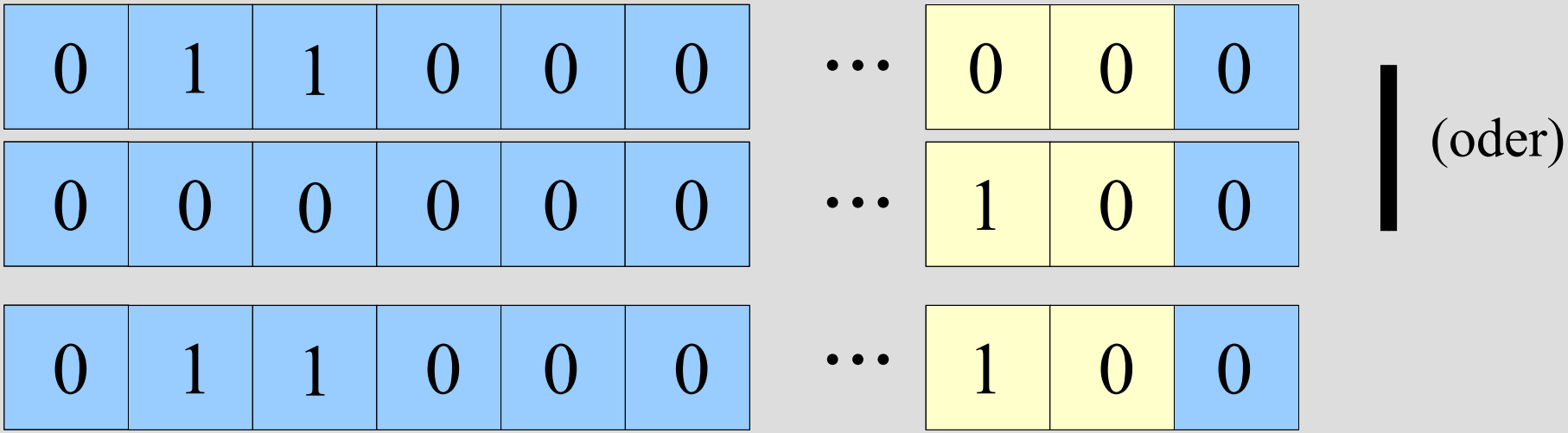
Formatierungsflags

```
cout.flags( (cout.flags() & ~ios_base::basefield) |
            (ios_base::basefield & ios_base::hex) );
```



Formatierungsflags

```
cout.flags( (cout.flags() & ~ios_base::basefield) |
            (ios_base::basefield & ios_base::hex) );
```



Beispiel – formatierte Ausgabe von Gleitkommazahlen 1/3

```
std::cout << 3.1 << " ";
std::cout.setf(
     std::ios_base::scientific,
     std::ios_base::floatfield);
std::cout << 3.1 << " ";
std::cout.setf(
     std::ios_base::fixed,
     std::ios_base::floatfield);
std::cout << 3.1 << "\n";
```

3.1 3.100000e+00 3.100000

Beispiel – formatierte Ausgabe von Gleitkommazahlen 2/3

```
std::cout << 3.1 << " ";
std::cout.precision(3);
std::cout << 3.1 << " ";
std::cout.setf(
     std::ios_base::fixed,
     std::ios_base::floatfield);
std::cout << 3.1 << "\n";
```

3.1 3.1 3.100

Beispiel – formatierte Ausgabe von Gleitkommazahlen 3/3

```
std::cout << 3125.13 << " ";
std::cout.precision(3);
std::cout << 3125.13 << " ";
std::cout.precision(5);
std::cout << 3125.13 << "\n";
```

3125.13 3.13e+03 3125.1

Beispiel – formatierte Ausgabe von Zeichenkette 1/3

```
const char* str=”Ausgabe”;
std::cout << “-” << str << “-\n-”;
std::cout.width(3);
std::cout << str << “-\n-“;
std::cout.width(11);
std::cout << str << “-\n”;
```

-Ausgabe-
-Ausgabe-
- Ausgabe-

Beispiel – formatierte Ausgabe von Zeichenkette 2/3

```
const char* str=”Ausgabe”;
std::cout << “-” << str << “-\n-”;
std::cout.width(3);
std::cout << str << “-\n-“;
std::cout.width(11);
std::.cout.fill('#');
std::cout << str << “-\n”;
```

-Ausgabe-
-Ausgabe-
-####Ausgabe-

Beispiel – formatierte Ausgabe von Zeichenkette 3/3

```
const char* str=”Ausgabe”;
std::cout << “-” << str << “-\n-”;
std::cout.width(3);
std::cout << str << “-\n-“;
std::cout.width(11);
std::.cout.fill('#');
std::cout.setf(std::base_ios::left,
    std::base_ios::adjustfield);
std::cout << str << “-\n”;
```

-Ausgabe-
-Ausgabe-
-Ausgabe####-

Beispiel – formatierte Ausgabe von Zahlen

```
std::cout << 16 << "\n";
std::cout.width(8);
std::cout << 16 << "\n";
std::cout.width(8);
std::cout.fill('0');
std::cout << 16 << "\n";
```

```
16
      16
00000016
```

Formatierung durch Manipulatoren

- Manipulatoren sind Funktionen, die direkt mit dem Ausgabeoperator verwendet werden, können um nachfolgende Ausgaben zu formatieren
- Manipulatoren heissen gleich oder zumindest ähnlich wie die Flags mit derselben Funktion

Manipulator Beispiele

```
std::cout << std::boolalpha << true;
std::cout << std::setwidth(4) <<  24;
std::cout << std::hex << 16;

std::cout << "A" << std::endl;
std::cout << "A" << "\n" << std::flush;
```

Fehlerbehandlung bei Streams

- tritt bei einer Operation mit einem Stream ein Fehler auf, schlagen alle nachfolgenden Operationen auf den Stream auch fehl.
- Ob ein Fehler aufgetreten ist, kann man über die Methode **`good()`** bzw. **`fail()`** herausfinden
- Um den Stream wieder benutzen zu können, muss man mit der Methode **`clear()`** den Status zurücksetzen

fstream

- Stream zum Lesen aus bzw. Schreiben in Dateien
  - *ifstream* nur zum Lesen
  - *ofstream* nur zum Schreiben
  - *fstream* für gemischte Lese- und Schreibzugriffe
- Deklarationen sind in <fstream> Header Datei
- Ein- und Ausgabe funktioniert gleich, wie bei der Standardeingabe und -ausgabe, da die Klassen ebenfalls von *ostream* und *istream* abgeleitet sind

Schreiben in eine Datei 1/3

- Um eine Datei zum Schreiben zu öffnen erzeugt man eine Instanz der Klasse std::ofstream und übergibt dem Konstruktor den Namen der Datei

```
std::ofstream ofs("test.txt");
```

- Anschließend sollte man immer mit **`good()`** prüfen, ob die Datei wirklich geöffnet werden konnte.

```
if(ofs.good()) ...
```

Schreiben in eine Datei 2/3

- Alternativ kann man eine Datei auch über die **`open`** Methode der *ofstream* Klasse öffnen.

```
std::ofstream ofs;
ofs.open("test.txt");
```

- Die **`open`** Methode verfügt über dieselben Argumente wie der zuvor benutzte Konstruktor.

Schreiben in eine Datei 3/3

- In die Datei schreibt man mit dem Ausgabeoperator

```
ofs << "Test" << (char)(10);
```

- Sind alle Daten geschrieben, wird der Stream mit `close()` geschlossen.

```
ofs.close();
```

Textmodus <-> Binärmodus

- Standardmäßig werden Dateien im Textmodus geöffnet
- Binärdateien im Textmodus zu schreiben ist ineffizient
- Um eine Datei im Binärmodus zu öffnen, muss man dem Konstruktor einen zusätzlichen Parameter übergeben

Binärmodus

- Um eine Datei im Binärmodus zu öffnen übergibt man zusätzlich zum Namen das Flag **`ios_base::binary`**

```
ofstream
ofs("test.bin",ios_base::binary);
```

- Im Binärmodus schreibt man Daten als Speicherblöcke mit der Methode

```
write(const char* d,streamsize s);
```

Weitere Datei Flags

- **`ios_base::nocreate`** : Datei wird bei Bedarf nicht erzeugt
- **`ios_base::app`** : Daten werden an das Ende der Datei angehängt
- einzelne Flags können über oder (|) verknüpft werden

```
ofstream o("test.bin",
        ios_base::binary |
        ios_base::app);
```

Stringstream

- Stream zum Lesen aus bzw. Schreiben in C++ Strings
  - *istringstream* nur zum Lesen
  - *ostringstream* nur zum Schreiben
  - *stringstream* für gemischte Lese- und Schreibzugriffe
- Deklarationen sind in <sstream> Header Datei
- Ein- und Ausgabe funktioniert wie bei anderen Streams

`ostringstream`

- nimmt formatierte und unformatierte Ausgaben entgegen und speichert diese in einem String.
- Der gespeicherte String kann mit der **`str`** Methode ausgelesen und gesetzt werden

Beispiel ostringstream

```
unsigned int zahl=0;
...
std::ostringstream ostr;
ostr << “Sie haben die Zahl “;
ostr << zahl;
ostr << “ eingegeben.”;
std::string s=ostr.str();
ostr.str(“”);
```

Ausgabeoperator 1/2

- Der Ausgabeoperator kann, wie die meisten Operatoren, überladen werden
- Man kann für eigene Klassen einen Ausgabeoperator definieren, um eine Instanz der Klasse über << ausgeben zu können
- Der Ausgabeoperator wird meist als globale Funktion implementiert

Ausgabeoperator 2/2

- Der Ausgabeoperator hat zwei Argumente:
  - Referenz auf den Ausgabe-Stream
  - const Referenz auf das auszugebende Objekt
- Der Ausgabeoperator gibt eine Referenz auf den Ausgabestrom zurück.

```
std::ostream& operator<<
( std::ostream& os,
  const TYPE& data);
```

Beispiel - Ausgabeoperator

```
std::ostream operator<<
 (std::ostream& os, const Bruch& b)
{
  os << b.getZaehler() << "/";
  os << b.getNenner();
  return os;
}

Bruch b(4,5);
std::cout << b << std::endl;
```