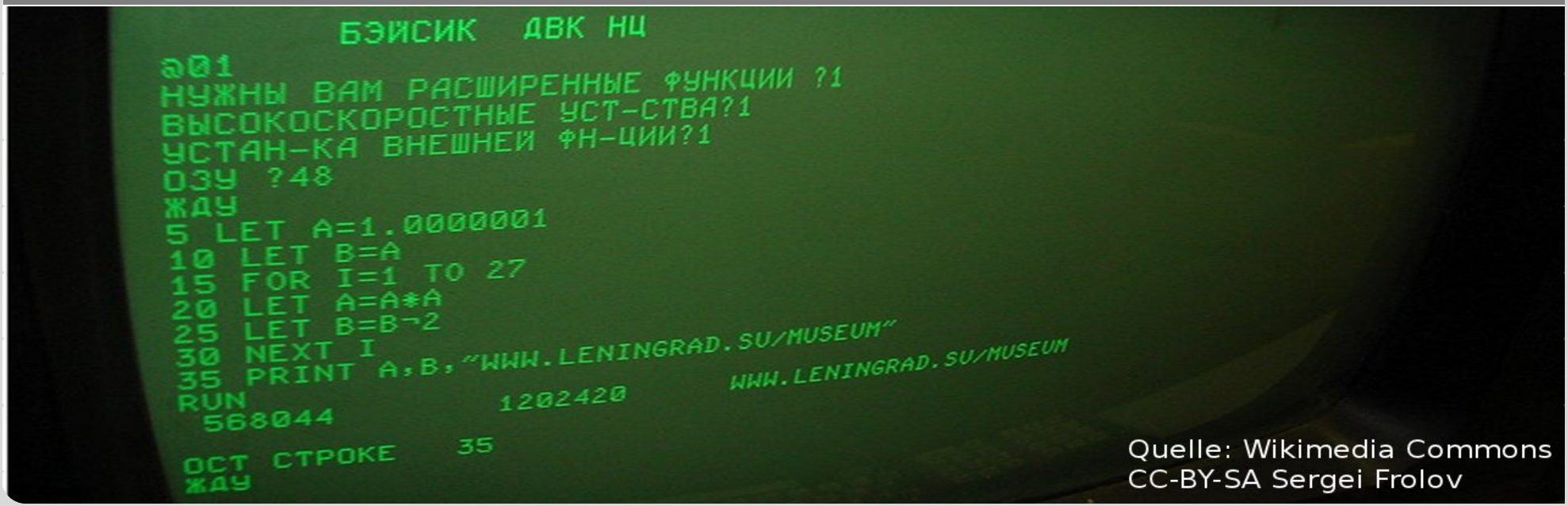
Karlsruher Institut für Technologie

C++ Workshop

10. Block, 06.07.2012

Markus Jung, Oliver Schneider, Robert Schneider | 6. Juli 2012



Quelle: Wikimedia Commons
CC-BY-SA Sergei Frolov

Gliederung

1. STL-Algorithmen
   - Konzept
   - Die Wichtigsten
   - Nutzung
2. Exceptions
   - Konzept
   - Was ist eine Exception
   - Wie werfe ich eine Exception
3. Streams
   - Grundlegendes
   - output stream
   - input stream
   - stream manipulators
   - konkrete Stream-Klassen
4. Datenstrukturen
5. Praxis

1. STL-Algorithmen
   - Konzept
   - Die Wichtigsten
   - Nutzung

Konzept

- Auf viele Container anwendbar
- Gleiches interface

Die Wichtigsten

- copy
- find
- fill
- for_each
- remove
- max
- max_element
- min
- min_element
- sort
- swap

Nutzung

```
1   std::vector<int> liste = {1, 42, 83, 7, 12};
2   int c_liste[] = {1, 42, 83, 12, 7};
3
4   std::sort(std::begin(liste), std::end(liste));
5   std::sort(std::begin(c_liste), std::end(c_liste));
6
7   if(std::equal(std::begin(liste), std::end(liste), std::begin(c_liste))) {
8       std::cout << "liste == c_liste" << std::endl;
9   } else {
10      std::cout << "liste != c_liste" << std::endl;
11  }
12
13  std::vector<int>::iterator it = std::find(std::begin(liste), std::end(liste), 3);
14
15  if(it == std::end(liste)) {
16      std::cout << "zahl nicht gefunden" << std::endl;
17  } else {
18      *it = 99;
19  }
```

2. Exceptions
   - Konzept
   - Was ist eine Exception
   - Wie werfe ich eine Exception

Konzept

- Ersetzen viele Fehlercodes bei Funktionen (nicht alle)
- In zeitkritischen Bereichen nur in Ausnahmefällen
- In nicht zeitkritischen Bereichen oefter nutzbar
- Setzen „sauberen“ C++ code vorraus
- Inkorrekte Nutzung kann sehr viel Overhead verursachen

Typ

Jede Exception sollte eine eigene Klasse sein, die von std::exception (oder einer ihrer Unterklassen) erbt

```
1  #include <exception>
2
3  class ATinyKittenDied : std::runtime_error
4  {
5  public:
6      ATinyKittenDied()
7      :std::runtime_error("Armes Kaetzchen")
8      {}
9  };
```

So gehts (nicht richtig)

```
1  int groesserNull = -32;
2
3  if(groesserNull <= 0) {
4      throw ATinyKittenDied();
5  }
```

Name der Exception hat nix mit dem Fehler zu tun, beheben wir das

So gehts richtig

```
1  #include <exception>
2
3  class ZahlIstNichtGroesserNull : std::runtime_error
4  {
5  public:
6      ZahlIstNichtGroesserNull()
7      :std::runtime_error("Die Zahl muss groesser Null sein")
8      {}
9  };
10
11 void testeZahl(int groesserNull)
12 {
13     if(groesserNull <= 0) {
14         throw ZahlIstNichtGroesserNull();
15     }
16     // ... do something
17 }
```

Hier ist eine exception unpraktisch

```
1  #include <exception>
2
3  class KeineNeueDaten : std::runtime_error
4  {
5  public:
6      KeineNeueDaten()
7      :std::runtime_error("Es liegen keine neue Daten an")
8      {}
9  };
10
11 struct Daten{};
12
13 Daten getDataFromTCPSocket()
14 {
15     if(datenMenge == 0) throw KeineNeueDaten();
16
17     return neueDaten;
18 }
```

Hier ist eine exception nützlich

```
1  Socket openSocket(uint16_t port)
2  {
3      if(portAlreadyOpen(port)) throw PortIsAlreadyUsed();
4      if(thereIsNoSpoon()) throw YouAreInTheMatrix();
5
6      return Socket(port);
7  }
8
9  ErrCode openSocket(Sock* sock, uint16_t port)
10 {
11     if(!sock) return ERR_SOCK_IS_NULL;
12     if(portAlreadyOpen(port)) return ERR_PORT_IN_USE;
13     if(thereIsNoSpoon()) return ERR_MATRIX_HAS_YOU;
14     sock->port = port;
15     sock->stream = newStream(port);
16
17     return SUCCESS;
18 }
```

Konstruktor schlägt fehl

```
1   class Socket
2   {
3   private:
4       SomePortInfo * info;
5   public:
6       Socket(uint16_t port)
7       {
8           if(portAlreadyOpen(port)) throw PortIsAlreadyUsed();
9           if(thereIsNoSpoon()) throw YouAreInTheMatrix();
10          info = new SomePortInfo(port);
11      }
12  };
13
14
15  Socket * sock = nullptr;
16
17  try{
18      sock = new Socket(99);
19  }
20  catch(PortIsAlreadyUsed &) {
21      std::cout << "port already in use" << std::endl;
22  }
23  catch(std::exception &e) {
24      std::cout << "an unkown error has ocurred: " << e.what() << std::endl;
25  }
```

3 Streams

- Grundlegendes
- output stream
- input stream
- stream manipulators
- konkrete Stream-Klassen

Datenfluss

- Container dienen zur Speicherung von Daten
- Streams dienen zum Versenden oder Empfangen von Daten
  $\Longrightarrow$ Datenfluss
- Meist entweder nur source (read-only) oder nur sink (write-only)
- Puffern von Datensequenzen vor/nach der Übertragung
- InputIterator (read-only, forward-only) für sources
- OutputIterator (write-only, forward-only) für sinks

Grundlegendes

Ein STL-Stream

- ist in erster Linie ein character-Streams
- nutzt im Hintergrund einen Puffer (`basic_streambuf`-Derivat)
  $\rightarrow$ Der Puffer stellt die eigentliche Übertragungs-Funktionalität bereit!
- dient quasi nur der vereinfachten Nutzung des Puffers
- ist Zustands-behaftet z.B. für Eingabe mit fester Breite (`setw`)

Locales

Unterschiedliche Konventionen in verschiedenen Ländern:

USA	France	Deutschland
3, 042.12	3 042, 12	3 042, 12
2:00 pm	14 h 05	14:00
June 22, 1996	22 juin 1996	22. Juni 1996

Locales

Unterschiedliche Konventionen in verschiedenen Ländern:

USA	France	Deutschland
3, 042.12	3 042, 12	3 042, 12
2:00 pm	14 h 05	14:00
June 22, 1996	22 juin 1996	22. Juni 1996

Locale

Beschreibt in Facetten Regions-spezifische Aspekte des UI, z.B.

- Zahlenformatierung (facets: `num_get`, `num_put`)
- Minuskel/Majuskel-Konversion (facet: `ctype`)
- Datum/Zeit (facets: `time_get`, `time_put`, ...)
- lexikographische Sortierung (facets: `collate`, `collate_byname`)
- ...

Locales in C++

An sich eigenes Thema!

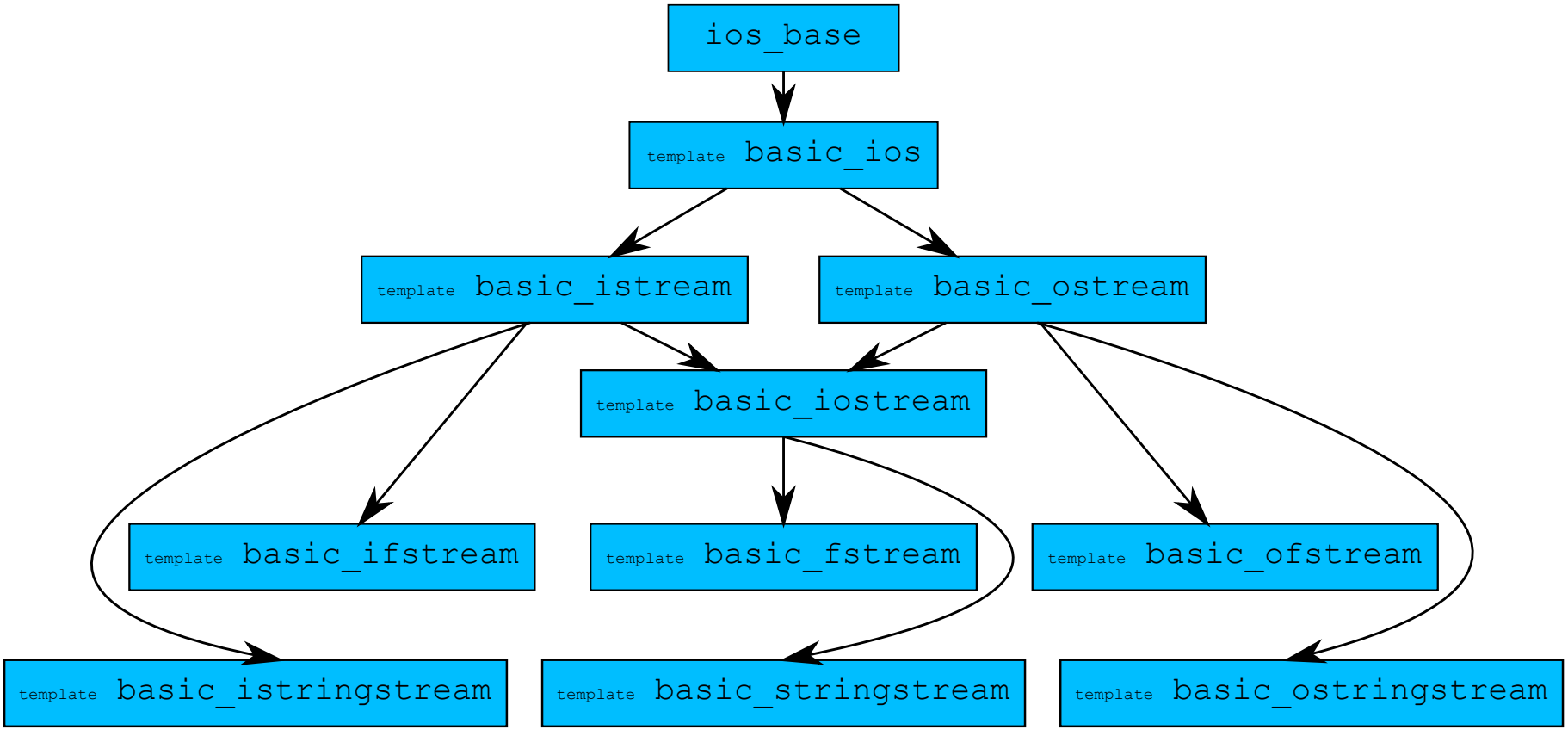
Locales in C++

An sich eigenes Thema!

```
1  #include <locale>
2
3  char const* const locale_name = "..."; // implementation-defined!
4  std::locale loc(locale_name);
5
6  char Ue = std::toupper('ü', loc); // Ue == 'Ü'
7  char foo = std::toupper('ß', loc); // foo == 'ß' [sic!]
```

Meist aber indirekte Verwendung, z.B. über Streams.

class hierarchy



template parameters of streams

Introducing: basic_ios, Standard 27.4.4

```
1  template
2  <
3      typename T_Char,
4      typename T_CharTraits = std::char_traits<T_Char>
5  >
6  class basic_ios;
```

`T_Char` der zugrunde liegende Block-/Character-Datentyp

`T_CharTraits` „String“-Operationen und Eigenschaften von `T_Char`

`ios_base` und `basic_ios`

Hauptsächlicher Inhalt:

- Formattierungs-Status (z.B. `std::ios_base::width` ← `setw`) (üblicherweise mittels stream manipulators)
- Fehler-Zustand
- locale-Management
- Puffer-Ownership; Puffer lässt sich zur Laufzeit austauschen: `basic_ios::rdbuf`
- Einführung von Konstanten und Typen
- (private storage)
- (callback registration)

Fehlerzustände von Streams

Konstanten des Fehler-Zustands

`0 ==`	`ios_base::goodbit`	kein Fehler
	`ios_base::badbit`	„unheilbarer“ Fehler
	`ios_base::failbit`	I/O Fehler (Formattierung oder Extraktion)
	`ios_base::eofbit`	Input hat das Dateiende erreicht (end-of-file)

Fehlerzustände von Streams

Konstanten des Fehler-Zustands

`0 == ios_base::goodbit`	kein Fehler
`ios_base::badbit`	„unheilbarer“ Fehler
`ios_base::failbit`	I/O Fehler (Formattierung oder Extraktion)
`ios_base::eofbit`	Input hat das Dateiende erreicht (end-of-file)

EOF bzw. eofbit ist kein Fehlerzustand des Streams selbt, jedoch kann EOF zu einem Fehlerzustand führen (z.B. wenn extrahiert werden soll).

member function	returns true if	meaning
`good()`	`rdstate() == goodbit`	stream ready (for extraction/insertion)
`fail()`	`failbit or badbit set`	something went wrong
`bad()`	`badbit set`	stream is irrecoverably „damaged“
`eof()`	`eofbit set`	stream is at the EOF (not ready)
`operator bool()`	`!fail()`	everything went fine (EOF still possible!)
`operator! ()`	`fail()`	something went wrong

Fehlerzustände von Streams

Konstanten des Fehler-Zustands

`0 == ios_base::goodbit`	kein Fehler
`ios_base::badbit`	„unheilbarer“ Fehler
`ios_base::failbit`	I/O Fehler (Formattierung oder Extraktion)
`ios_base::eofbit`	Input hat das Dateiende erreicht (end-of-file)

EOF bzw. eofbit ist kein Fehlerzustand des Streams selbt, jedoch kann EOF zu einem Fehlerzustand führen (z.B. wenn extrahiert werden soll).

member function	returns true if	meaning
`good()`	`rdstate() == goodbit`	stream ready (for extraction/insertion)
`fail()`	`failbit or badbit set`	something went wrong
`bad()`	`badbit set`	stream is irrecoverably „damaged“
`eof()`	`eofbit set`	stream is at the EOF (not ready)
`operator bool()`	`!fail()`	everything went fine (EOF still possible!)
`operator! ()`	`fail()`	something went wrong

Merke: `fail` $\subset$ `bad`: `fail` allein kann man nicht direkt erfragen

`basic_ostream` template

Hauptsächlicher Inhalt:

- formatted output, stream manipulator „interface“: `operator<<`
- unformatted output: `put`, `write`
- `flush`
- `sentry`-Klasse

`basic_ostream` template

Hauptsächlicher Inhalt:

- formatted output, stream manipulator „interface“: `operator<<`
- unformatted output: `put`, `write`
- `flush`
- `sentry`-Klasse

sentry-Klasse

Kümmer Dich genau dann darum, wenn Du selbst eine *formatted-output*-Funktion für einen Stream schreibt, die *unformatted-output*-Funktionen nutzt (also nicht ausschließlich die standardmäßig vorhandenen `operator<<`)!

`basic_ostream`: formatted output

`operator<<` standardmäßig schon definiert für:

- built-in data types (`short`, `int`, `bool`, `float` usw.)
- pointer (als `void const*`)
- `std::basic_streambuf`
- `std::string` (im Header `<string>`) (globale Funktion)
- `char`, `char const*` und Abarten (globale Funktion)

`basic_ostream`: formatted output

`operator<<` standardmäßig schon definiert für:

- built-in data types (`short`, `int`, `bool`, `float` usw.)
- pointer (als `void const*`)
- `std::basic_streambuf`
- `std::string` (im Header `<string>`) (globale Funktion)
- `char`, `char const*` und Abarten (globale Funktion)

Verkettung von `operator<<`

Gibt `basic_ostream < .... > &` zurück, daher verketten möglich!

```
1  /* 0. */  std::cout << 5  << "hallo";
2  /* 1. */ (std::cout << 5) << "hallo";
3  /* 2. */ std::ostream& rcout = std::cout << 5;
4            rcout << "hallo";
```

`basic_ostream`: formatted output & errors

- jede formatted-ouput-Funktion prüft zunächst den Fehlerzustand des Streams (`good()`)
  - falls `!good()`: setze `failbit`
  - falls `good()`: führe den output durch

`basic_ostream`: formatted output & errors

- jede formatted-ouput-Funktion prüft zunächst den Fehlerzustand des Streams (`good()`)
  - falls `!good()`: setze `failbit`
  - falls `good()`: führe den output durch

Heißt: wenn exceptions nicht explizit aktiviert, geht jeglicher formatted output *lautlos* schief bis zum Aufheben des Fehlerzustandes!

```
1 std::cout << make_error;
2 // angenommen, cout waere nun in einem Fehlerzustand
3 std::cout << "hello world" << 5 << 42.2 << std::endl;
4 // Programm kommt hier an (falls !std::cin.exceptions()),
5 // aber nichts wurde ausgegeben!
```

`basic_istream` template

Hauptsächlicher Inhalt:

- formatted input, stream manipulator „interface“: `operator>>`
- unformatted input
- `sentry`-Klasse

`basic_istream`: formatted input

`operator>>` standardmäßig schon definiert für:

- built-in data types (`short`, `int`, `bool`, `float` usw.)
- pointer (als `void const*`)
- `std::basic_streambuf`
- `std::string` (im Header `<string>`)
- `char` und Abarten (globale Funktion)
- `char const*` und Abarten (globale Funktion), UNSAFE!

Gibt `basic_istream < .... > &` zurück, daher verketten möglich!

`(std::cin >> myInt >> myString;)`

`basic_istream`: formatted input

`operator>>` standardmäßig schon definiert für:

- built-in data types (`short`, `int`, `bool`, `float` usw.)
- pointer (als `void const*`)
- `std::basic_streambuf`
- `std::string` (im Header `<string>`)
- `char` und Abarten (globale Funktion)
- `char const*` und Abarten (globale Funktion), UNSAFE!

Gibt `basic_istream < .... > &` zurück, daher verketten möglich!
`(std::cin >> myInt >> myString;)`

Wichtig für formatted input

`operator>>` hört auf zu Lesen bei space-characters. space-characters sind locale-dependent! Zumeist aber mindestens:
`' ', '\t', '\n', '\v', '\f', '\r'`

`basic_istream`: formatted input & errors

Vorgehen jeder formatted-input-Funktion

- prüfe den Fehlerzustand des Streams (`good()`)
- falls `!good()`: setze `failbit`
- falls `good()`:
  - falls `flags() & skipws`: entferne white-spaces vom Beginn
  - falls jetzt EOF: setze `failbit | eofbit` (da nichts gelesen)
  - versuche zu lesen; bei exception: setze `badbit`
  - lesen schlägt fehl: setze `failbit`

`basic_istream`: formatted input & errors

Vorgehen jeder formatted-input-Funktion

- prüfe den Fehlerzustand des Streams (`good()`)
- falls `!good()`: setze `failbit`
- falls `good()`:
  - falls `flags() & skipws`: entferne white-spaces vom Beginn
  - falls jetzt EOF: setze `failbit | eofbit` (da nichts gelesen)
  - versuche zu lesen; bei exception: setze `badbit`
  - lesen schlägt fehl: setze `failbit`

Heißt: wenn exceptions nicht explizit aktiviert, geht jeglicher formatted output *lautlos* schief bis zum Aufheben des Fehlerzustandes!

```
1  std::cin >> make_error;
2  // angenommen, cin waere nun in einem Fehlerzustand
3  int i; int j;
4  std::cin >> i >> j;
5  // Programm kommt hier an (falls !std::cin.exceptions())
6  //- Wert von i und j undefiniert!
```

formatted input: lesen in Schleife

```
1  int i; int j;
2
3  // schlechte Idee:
4  while( !cin.eof() )
5  {
6      cin >> i >> j;
7      // use i and j
8  }
```

formatted input: lesen in Schleife

```
1  int i; int j;
2
3  // schlechte Idee:
4  while( !cin.eof() )
5  {
6      cin >> i >> j;
7      // use i and j
8  }
9
10 // nicht viel besser
11 while(cin)
12 {
13     cin >> i >> j;
14     // use i and j
15 }
```

formatted input: lesen in Schleife

```
1  int i; int j;
2
3  // schlechte Idee:
4  while( !cin.eof() )
5  {
6      cin >> i >> j;
7      // use i and j
8  }
9
10 // nicht viel besser
11 while(cin)
12 {
13     cin >> i >> j;
14     // use i and j
15 }
```

```
17 // gute Idee
18 while( cin >> i >> j )
19 {
20     // use i and j
21 }
```

formatted input: lesen in Schleife

```
1  int i; int j;
2
3  // schlechte Idee:
4  while( !cin.eof() )
5  {
6      cin >> i >> j;
7      // use i and j
8  }
9
10 // nicht viel besser
11 while(cin)
12 {
13     cin >> i >> j;
14     // use i and j
15 }
```

```
17 // gute Idee
18 while( cin >> i >> j )
19 {
20     // use i and j
21 }
22
23
24 // gute, flexiblere Idee
25 while(true)
26 {
27     cin >> i >> j;
28     if(!cin) { break; }
29     // use i and j
30 }
```

Konsole: Warten auf ENTER

- C++ kennt weder eine Konsole noch die Enter-Taste
  $\Longrightarrow$ keine feste Beziehung zwischen z.B. `cin` und Anfrage nach Eingabe
- Was ist mit stream redirections?
- es gibt afaik keine robuste portable Variante
- nutze libs ($\Longrightarrow$ platform-dependent calls), z.B. ncurses

stream manipulators

Schon bekannt:

- `std::endl`, `std::flush`
- `std::setw`, `std::setprecision`

Finden sich in `<ios>`, `<ostream>` bzw. `<istream>` und in `<iomanip>`!

Weitere Beispiele:

- `std::scientific` (floating-point IO)
- `std::setfill`, `std::left`, `std::right`
- `std::ws`, `std::skipws`, `std::noskipws`

stream manips: Funktionsweise

die Manipulatoren

- beschreiben Funktionen (sind Funktionen $\Longrightarrow$ function ptrs)
- werden mit `operator<<` bzw. `operator>>` an den Stream „übergeben“
- der Stream ruft die Funktion auf und übergibt sich selbst als Parameter
- die Manipulator-Funktion arbeitet auf dem Stream

stream manips: Funktionsweise

die Manipulatoren

- beschreiben Funktionen (sind Funktionen $\Longrightarrow$ function ptrs)
- werden mit `operator<<` bzw. `operator>>` an den Stream „übergeben“
- der Stream ruft die Funktion auf und übergibt sich selbst als Parameter
- die Manipulator-Funktion arbeitet auf dem Stream

Erläuterndes Beispiel

```
1   std::ostream& endl( std::ostream& p ) {
2       p.put( p.widen('\n') ); // insert a new-line char
3       p.flush();
4       return p;
5   }
6
7   std::ostream& operator<< (std::ostream& p, Manipulator m) {
8       m(p); // apply manipulator function
9       return p;
10  }
11
12  std::cout << endl;
```

fstream

Zusätzliche member functions:

Signatur	Beschreibung
`void open(const char*, ios_base::openmode)`	öffnet eine Datei
`void open(string const& s, ios_base::openmode m)`	öffnet eine Datei
`void close()`	flushed den Puffer und schließt die Datei
`bool is_open() const`	`true` genau dann, wenn die Datei offen ist

Varianten des file-based streams:

`basic_ifstream`, `basic_ofstream`, `basic_fstream`

stringstream

Zusätzliche member functions:

Signatur	Beschreibung
`basic_string str() const`	gibt einen string mit einer Kopie des Inhalts des Streams zurück
`void str(basic_string const&)`	ersetzt den Inhalt des Streams durch eine Kopie des Inhalt des Strings

Varianten des string-based streams:

`basic_istringstream`, `basic_ostringstream`, `basic_stringstream`

Aliase für streams

```
using alias = name < T_Char, T_CharTraits >;
```

alias	name	T_Char	T_CharTraits
ios	basic_ios	char	char_traits < char >
wios	basic_ios	wchar_t	char_traits < wchar_t >
istream	basic_istream	char	char_traits < char >
wfstream	basic_fstream	wchar_t	char_traits < wchar_t >

usw.

Standard-Streams: Wohin gehen diese?

- implementation-defined!
- meist: auf eine Konsole
- möglich: stream redirection / capture, z.B. Umleitung in Datei
  `myprog > myLog.log`
- C++ weiß noch nicht mal was von einer Tastatur!

`cin` und Konsorten

Typ	Name	Beschreibung
`extern istream&`	`cin`	standard character input stream
`extern ostream&`	`cout`	standard character output stream
`extern ostream&`	`cerr`	standard character error stream
`extern ostream&`	`clog`	standard character log stream

Das ganze gibt es noch als wide-character-Variante (anderes, weiteres Thema).

4 Datenstrukturen

Hashtable

Hashing

- (Schnelles) Finden durch Unordnung
- „to hash“ = zerhacken

Hashtable

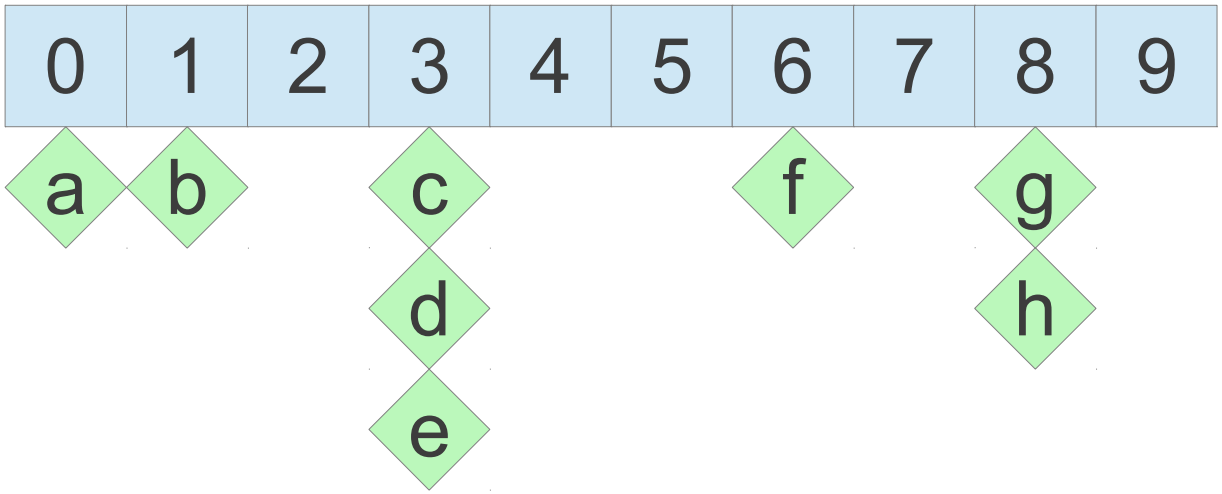
Hashing

- (Schnelles) Finden durch Unordnung
- „to hash“ = zerhacken
- Trick: Hashfunktionen bilden (komplexe) Objekte (durcheinander) auf einfache Zahlen (Hashwerte) ab
- Hashwerte dienen als Indizes in der Hashtabelle
- Folge: (Fast) direkte Abbildung Schlüsselobjekt $\rightarrow$ Element

Hashtable

Hashing

- (Schnelles) Finden durch Unordnung
- „to hash“ = zerhacken
- Trick: Hashfunktionen bilden (komplexe) Objekte (durcheinander) auf einfache Zahlen (Hashwerte) ab
- Hashwerte dienen als Indizes in der Hashtabelle
- Folge: (Fast) direkte Abbildung Schlüsselobjekt $\rightarrow$ Element



50, a 73, e
21, b 26, f
93, c 18, g
43, d 68, h

Hashtables in C++

`std::unordered_map` (ab C++11)

- `find(key)`
- `operator[key]`
- `insert(pair<key, value>)`
- `erase(key)`

Hashtables in C++

`std::unordered_map` (ab C++11)

- `find(key)`
- `operator[key]`
- `insert(pair<key, value>)`
- `erase(key)`

Alles (amortisiert) in $\mathcal{O}(1)$

- `unordered_map` vs. `map`
  - Wichtige Operationen in $\mathcal{O}(1)$ statt $\mathcal{O}(log(n))$
  - `unordered_map` ist gut bei Einzel-Operationen
  - `map` erlaubt effiziente Bereichsoperationen (Iteratoren!)

5 Praxis

Praxis!

- Bearbeitung der Aufgaben aus den letzten Workshops

`https://github.com/kit-cpp-workshop/workshop-ss12-10`

Aufgabenbeschreibungen und Hinweise: Siehe `README.md`