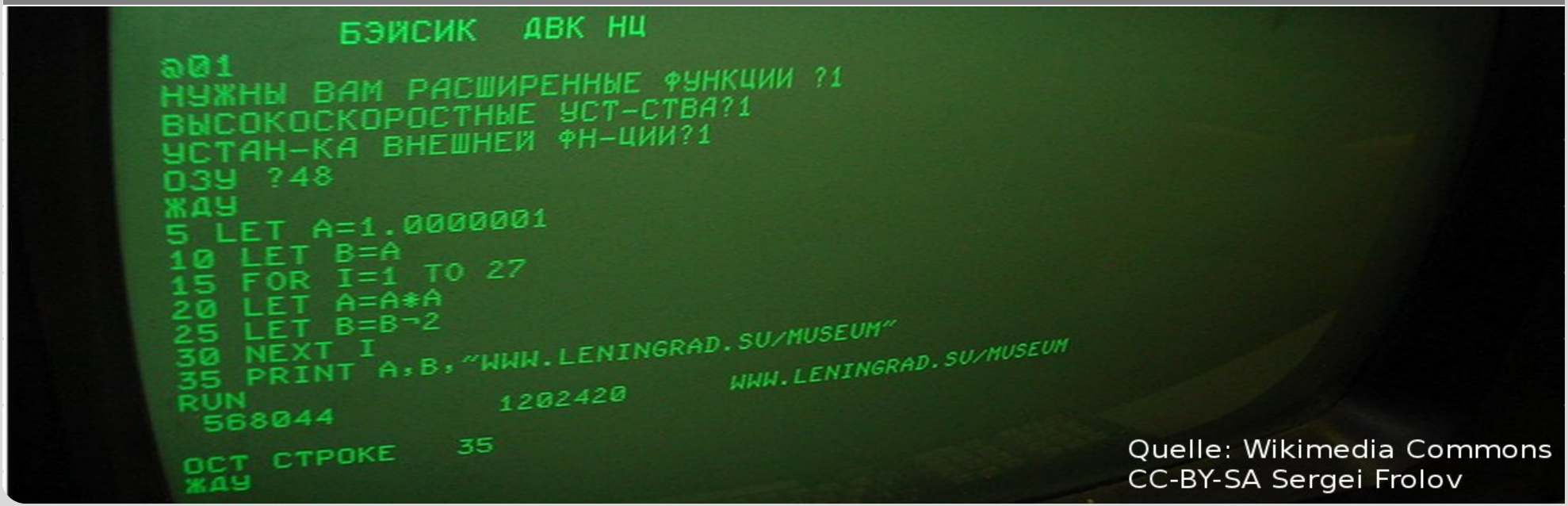
Karlsruher Institut für Technologie

C++ Workshop

9. Block, 29.06.2012

Sven Brauch, Markus Jung, Oliver Schneider, Robert Schneider | 30. Juni 2012



Quelle: Wikimedia Commons
CC-BY-SA Sergei Frolov

Gliederung

1 Literale

Literale

Was ist ein Literal?

Literale sind Bestandteil der Syntax der meisten Sprachen, die dazu dienen, Daten direkt in den Quellcode zu schreiben.

Literale

Was ist ein Literal?

Literale sind Bestandteil der Syntax der meisten Sprachen, die dazu dienen, Daten direkt in den Quellcode zu schreiben.

Beispiele für Literale in C++

Typ	Beispiel
char	'A'
const char[] (char-Array, Null-terminiert!)	"Hello World!"
int	42
double	1.6e-19

Tabelle: Literale in C++

Escape-Sequenzen

Escape-Sequenzen in String-Literalen in C++

Sequenz	Wirkung
\n	Neue Zeile
\t	Tabulator
\\	Backslash (\)
\"	Doppeltes Anführungszeichen (")
\'	Einfaches Anführungszeichen (')

Weniger häufig genutzte Literale

Weitere Beispiele für Literale

Typ	Beispiel
Oktalzahl (int)	042
Hexadezimalzahl (int)	0x732

Weniger häufig genutzte Literale

Weitere Beispiele für Literale

Typ	Beispiel
Oktalzahl (int)	042
Hexadezimalzahl (int)	0x732
unsigned int	42u
unsigned int, Oktal	042u

Weniger häufig genutzte Literale

Weitere Beispiele für Literale

Typ	Beispiel
Oktalzahl (int)	042
Hexadezimalzahl (int)	0x732
unsigned int	42u
unsigned int, Oktal	042u
C++11: Benutzerdefiniert	42.7_meter

Tabelle: Weitere Literale in C++

2 Strings

Probleme mit C-Strings

Probleme mit C-Strings

- Schwierige Handhabung, z. B. wegen Null-Terminierung
- Keine einfache Möglichkeit, mehr als die Ascii-Zeichen zu benutzen (zum Beispiel Unicode)

String-Klassen

std::string

`std::string` ist die String-Klasse der Standardbibliothek. Praktische Methoden:

- `size()`
- `append(str)`, `insert(position, str)`
- `operator==(str)`

Anders als mit `char`-Arrays kann man also zum Beispiel Strings direkt vergleichen: `if ( str1 == str2 ) ...`

String-Klassen

QString

`QString` ist die String-Klasse der Qt-Bibliothek[a]. Im Gegensatz zu `std::string` bietet `QString` echte Unicode-Unterstützung, und viele weitere nützliche Methoden, zum Beispiel:

- `replace(QRegExp, QString)` – suchen und Ersetzen mit regulären Ausdrücken
- `repeated(int)` – wiederholt den String einige Male
- `trimmed()` – entfernt Leerzeichen vorne und hinten im String

[a] `http://qt-project.org/`

String-Klassen

enemy slide injection :P

STL und Unicode

Die Unicode-Unterstützung der STL steckt in den Algorithmen, nicht in `std::string`:

- `std::regex_replace(string, regex, string)` – suchen und Ersetzen mit regulären Ausdrücken
- `string+=string` – wiederholt den String einmal (einige Male == einmal 5-zeilige Funktion)
- `stream>>skipws>>string` – entfernt Leerzeichen vorne und hinten im String
  - `string.substr( find_if(string.begin(), string.end(), bind( isprint, _1, loc), reverse_iterator(find_if(string.rbegin(), string.rend(), bind( isprint, _1, loc) )` – entfernt Leerzeichen vorne und hinten im String

3 „Abkürzungen“ in C++

„Abkürzungen“ für `namespace`s

```
1  namespace Gebaeck
2  {
3      namespace Kuchen
4      {
5          struct A { int a; }
6          struct B { A b; }
7      }
8  }
9  Gebaeck::Kuchen::A blub;
10 blub.a = 99;
11
12 using namespace Gebaeck::Kuchen;
13
14 B bla;
15 bla.b.a = 42;
```

„Abkürzungen“ für Typennamen

Problem: 5 namespaces mit struct A... welchen nutzen?

```
1  namespace Gebaeck
2  {
3      namespace Kuchen
4      {
5          struct A { int a; }
6          struct B { A b; }
7      }
8  }
9   using C = Gebaeck::Kuchen::B;        // C++11
10 // typedef Gebaeck::Kuchen::B C;      // C++03
11
12 C meinC;
13 C.b.a = 273;
```

4. Etwas mehr zu templates
   - template argument deduction
   - non-type template parameters

template argument deduction

Bei einem „Aufruf eines function templates“ kann der Compiler (manche) template-Argumente anhand der Typen der function-call-Argumente bestimmen:

```
1  template < typename T >
2  void foo( T p )
3  { std::cout << p; }
4
5  foo < int > ( 42 );
```

template argument deduction

Bei einem „Aufruf eines function templates“ kann der Compiler (manche) template-Argumente anhand der Typen der function-call-Argumente bestimmen:

```
1  template < typename T >
2  void foo( T p )
3  { std::cout << p; }
4
5  foo < int > ( 42 );
6
7  foo(42);
```

non-type template parameters

Bisher: Typen als Parameter der Vorlage

```
1  template < typename T_SomeType, int t_someInt >
2  struct MyStruct
3  {
4      T_SomeType arr[t_someInt];
5  };
```

non-type template parameters

Bisher: Typen als Parameter der Vorlage

Jetzt neu: z.B. `int` als Parameter der Vorlage

```
1  template < typename T_SomeType, int t_someInt >
2  struct MyStruct
3  {
4      T_SomeType arr[t_someInt];
5  };
6
7
8  MyStruct < double, 42 >     myArr;
9
10 int myInt;
11 std::cin >> myInt;
12 MyStruct < bool, myInt >    myArr2;   // FEHLER!
```

non-type template parameters: constraints

Der Wert muss eine constant-expression sein, also dem Compiler bekannt!
Denn sonst kann dieser das Template nicht Instantiieren.

Erlaubte Typen:

- integral
- enumeration
- exotisch: Referenz oder Pointer zu ≫Ding≪ *mit external linkage*
- kein String!

5 Iteratoren

- Grundlegendes
- Iteratoren und Generizität
- Iterator-Kategorien
- Hinweise

Grundlegendes

- Einheitlicher Zugriff auf alle STL-Container
- Dadurch Algorithmen leicht auf unterschiedliche Container anzuwenden
- Notwendig für Container ohne Random Access (list)
- Praktisch für Assoziative container (set, map)
- Verallgemeinerung von Pointern

Beispiel anhand `list`

```
1  #include <list>
2  using intList = std::list<int>;
3  using intIt = intList::iterator;
4  intList meineListe;
5  meineListe.push_back(42);
6  meineListe.push_back(1024);
7  meineListe.push_front(1);
8
9  // C++03
10 for(intIt it  = meineListe.begin();
11           it != meineListe.end();
12         ++it) {
13     std::cout << *it << std::endl;
14 }
15 // C++11
16 for(int i : meineListe) {
17     std::cout << i << std::endl;
18 }
```

`begin` und `end`

- jeder STL-Container hat member functions `begin` und `end`
- Adapter nicht unbedingt (wo es eben keinen Sinn ergibt...)
- `std::begin` in `<iterator>` für alle Container und plain-C-Arrays
- `begin` liefert einen Iterator, der auf das erste Element zeigt
- `end` liefert einen speziellen Iterator, der sozusagen auf das (letztes + 1)ste Element zeigt
- man iteriert über $[$`begin`, `end`$)$, ebenso arbeiten Algorithmen alle auf einem $[x, y)$-Bereich

`begin` und `end`

- jeder STL-Container hat member functions `begin` und `end`
- Adapter nicht unbedingt (wo es eben keinen Sinn ergibt...)
- `std::begin` in `<iterator>` für alle Container und plain-C-Arrays
- `begin` liefert einen Iterator, der auf das erste Element zeigt
- `end` liefert einen speziellen Iterator, der sozusagen auf das (letztes + 1)ste Element zeigt
- man iteriert über $[$`begin`, `end`$)$, ebenso arbeiten Algorithmen alle auf einem $[x, y)$-Bereich

Nützlich: backwards-compatible zu Pointern:

```
1  char const myString[] = "foobar";
2  char const* myStringEnd = (myString+5) + 1;
3  using It = char const*;
4
5  for(It i = myString; i != myStringEnd; ++i)
6  { std::cout << *i; }
```

generischer Ansatz

```
1  template < typename T_Iterator >
2  void myPrint(T_Iterator p_beg, T_Iterator p_end)
3  {
4      for(T_Iterator i  = p_beg; i != p_end; ++i)
5      { std:: cout << *i << " : "; }
6  }
```

generischer Ansatz

```
1  template < typename T_Iterator >
2  void myPrint(T_Iterator p_beg, T_Iterator p_end)
3  {
4      for(T_Iterator i  = p_beg; i != p_end; ++i)
5      { std:: cout << *i << " : "; }
6  }
7
8  int main() {
9      std::vector < int >  myVector = { 1, 2, 42, 3 };
10     std::list < double > myList = { 42, 3, 1, 22 };
11
12     myPrint( myVector.begin(), myVector.end() );
13     myPrint( myList.begin(), myList.end() );
14
15     myPrint( myVector.begin() + 3, myVector.end() );
16     myPrint( myList.begin()   + 3, myList.end() );   // FEHLER!
17     myPrint( std::next(myList.begin(), 3), myList.end() ); // OK
18 }
```

STL und Iteratoren

Iteratoren bilden das Bindeglied zwischen Containern und Algorithmen, z.B.:

```
1  std::vector<int> myVec = { 4, 3, 1, 2 };
2  std::sort( myVec.begin(), myVec.end() );
```

STL und Iteratoren

Iteratoren bilden das Bindeglied zwischen Containern und Algorithmen, z.B.:

```
1  std::vector < int > myVec = { 4, 3, 1, 2 };
2  std::sort( myVec.begin(), myVec.end() );
```

Mittels Iteratoren lassen sich auch Daten überführen:

```
1  std::vector < int > myVec = { 4, 4, 4, 3 };
2  std::list < int > myList = {myVec.begin(), myVec.end()};
```

Iterator-Kategorien

Inspiration: `http://en.cppreference.com/w/cpp/iterator`

<table>
<tr><th colspan="4">iterator category</th><th colspan="2">capabilities</th></tr>
<tr><td rowspan="4">random-access</td><td rowspan="3">bi-directional</td><td rowspan="2">forward</td><td>input</td><td>read</td><td rowspan="2">iterate single-pass</td></tr>
<tr><td>output</td><td>write</td></tr>
<tr><td colspan="2"></td><td></td><td>iterate multi-pass</td></tr>
<tr><td colspan="3"></td><td></td><td>reverse-iterate multi-pass</td></tr>
<tr><td colspan="3"></td><td></td><td>random access</td></tr>
</table>

Iterator-Kategorien

Inspiration: `http://en.cppreference.com/w/cpp/iterator`

<table>
<tr><th colspan="4">iterator category</th><th colspan="2">capabilities</th></tr>
<tr><td rowspan="4">random-access</td><td rowspan="3">bi-directional</td><td rowspan="2">forward</td><td>input</td><td>read</td><td rowspan="2">iterate single-pass</td></tr>
<tr><td>output</td><td>write</td></tr>
<tr><td></td><td></td><td>iterate multi-pass</td></tr>
<tr><td colspan="2"></td><td></td><td></td><td>reverse-iterate multi-pass</td></tr>
<tr><td colspan="3"></td><td></td><td>random access</td></tr>
</table>

Mit:

```
iterator i, j;
value_t value;
int n;
```

capability	syntax
read	`value = *i`
write	`*i = value`
iterate	`++i`, `i++`
reverse-iterate	`--i`, `i--`
random access	`j = i + n`

Iterator-Kategorien und Datenstrukturen

Datenstruktur	Iterator
`vector`	random-acccess
`list`	bi-directional
`map`	bi-directional
input stream	input
output stream	output
io stream	forward

Hinweise

- man iteriert über $[$`begin`, `end`$)$, ebenso arbeiten Algorithmen alle auf einem $[x, y)$-Bereich
- `*( myContainer.end() )` ist eine GANZ schlechte Idee (wie bei Pointern)
- Iteratoren sind oftmals schneller als member acccess (z.B. `vector`)
- und (fast) immer generischer
- man vergleicht `i != end` und nicht `i < end`
- Iteratoren werden bei bestimmen Operationen (möglicherweise) ungültig (z.B. `vector::push_back`)

Infamous vector loop erase

```
1  std::vector< int > vec = { 0, 1, 2, 3 };
2
3  for(It i = vec.begin(); i != vec.end(); ++i) {
4      if( 1 == *i ) {
5          vec.erase(i);    // schlechte Idee!
6
7      }
8  }
```

Infamous vector loop erase

```
1  std::vector< int > vec = { 0, 1, 2, 3 };
2
3  for(It i = vec.begin(); i != vec.end(); ++i) {
4      if( 1 == *i ) {
5
6          i = vec.erase(i);    // genauso schlecht!
7      }
8  }
```

Infamous vector loop erase

```
1  std::vector < int > vec = { 0, 1, 2, 3 };
2
3  for(It i = vec.begin(); i != vec.end(); ++i) {
4      if( 1 == *i ) {
5
6          i = vec.erase(i);    // genauso schlecht!
7      }
8  }
9
10 It i = vec.begin();
11 while(i != vec.end()) {
12     if( 1 == *i ) { i = vec.erase(); continue; }
13     ++it;
14 }
```

Infamous vector loop erase

```
1  std::vector< int > vec = { 0, 1, 2, 3 };
2
3  for(It i = vec.begin(); i != vec.end(); ++i) {
4      if( 1 == *i ) {
5
6          i = vec.erase(i);    // genauso schlecht!
7      }
8  }
9
10 It i = vec.begin();
11 while(i != vec.end()) {
12     if( 1 == *i ) { i = vec.erase(); continue; }
13     ++it;
14 }
15
16 #include <algorithm>
17 std::remove_if( vec.begin(), vec.end(), 1 );
```

6 STL-Datenstrukturen

- container adaptors
- associative containers
- bitset

Überblick - container adaptors

Adapter (Entwurfsmuster)

Überblick - container adaptors

Adapter (Entwurfsmuster)

Ausgangssituation: Eine Klasse bietet den erforderlichen Funktionsumfang, aber nicht (genau) die gewünschte Schnittstelle.

Überblick - container adaptors

Adapter (Entwurfsmuster)

Ausgangssituation: Eine Klasse bietet den erforderlichen Funktionsumfang, aber nicht (genau) die gewünschte Schnittstelle.

Ein Adapter:

- Übersetzt eine inkompatible Schnittstelle
- Wandelt ggf. die verwendeten Datenformate um
- Ist nach außen hin nicht als ein solcher erkennbar

Überblick - container adaptors

Adapter (Entwurfsmuster)

Ausgangssituation: Eine Klasse bietet den erforderlichen Funktionsumfang, aber nicht (genau) die gewünschte Schnittstelle.

Ein Adapter:

- Übersetzt eine inkompatible Schnittstelle
- Wandelt ggf. die verwendeten Datenformate um
- Ist nach außen hin nicht als ein solcher erkennbar

- `stack`
- `queue`
- `priority_queue`

Überblick - container adaptors

Adapter (Entwurfsmuster)

Ausgangssituation: Eine Klasse bietet den erforderlichen Funktionsumfang, aber nicht (genau) die gewünschte Schnittstelle.
Ein Adapter:

- Übersetzt eine inkompatible Schnittstelle
- Wandelt ggf. die verwendeten Datenformate um
- Ist nach außen hin nicht als ein solcher erkennbar

- `stack`
- `queue`
- `priority_queue`

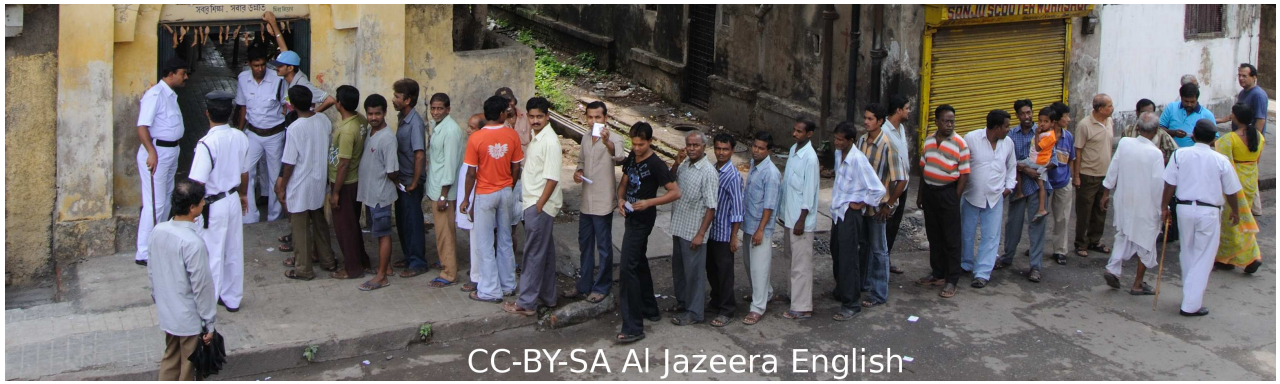
`stack` und `queue` sind schon fast mehr Interface als eine vollwertige Klasse. Reicht deren Funktionsumfang aus, darf daher gerne deren Schnittstelle genutzt werden, was mehr Wahlfreiheit bei der Auswahl der zugrundeliegenden Datenstruktur erlaubt.

stack

- Kann mit `deque`, `list` und `vector` arbeiten
- Wichtig: `top()`, `push()` und `pop()`
- Laufzeit: $\mathcal{O}(1)$ für alle Einzel-Operationen

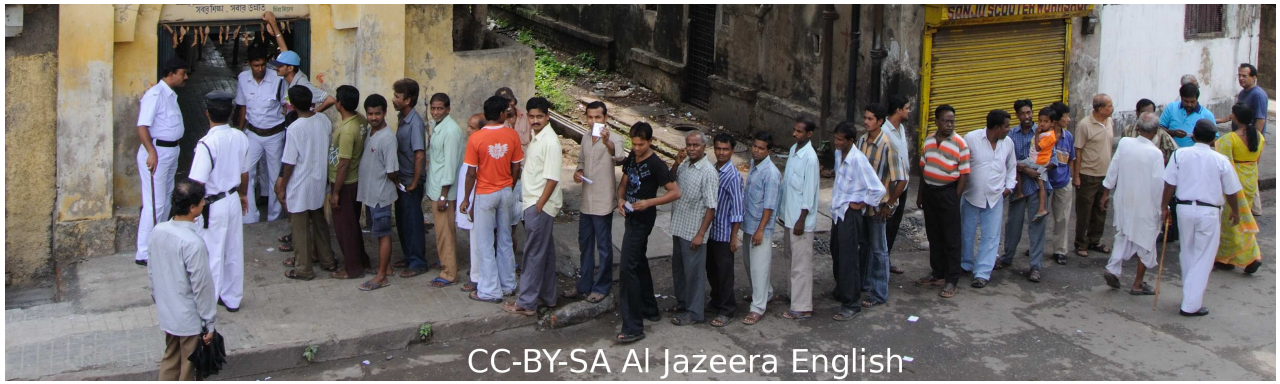
- Einsatz: Bei Bedarf

queue



- Kann mit `deque` und `list` arbeiten, aber nicht mit `vector`
  - Warum?

queue



- Kann mit `deque` und `list` arbeiten, aber nicht mit `vector`
  - Warum?
  - Weil entfernen am Anfang eines `vector` „teuer“ ($\mathcal{O}(n)$) ist
- Wichtig: `front()`, `back()`, `push()` und `pop()`
- Laufzeit: $\mathcal{O}(1)$ für alle Einzel-Operationen
- Einsatz: Bei Bedarf

Prioritätslisten (Priority Queues)

Was?

- Menge von Elementen mit einem Schlüssel
- Geordnet nach Schlüssel (Ordnungsrelation: $\leq$)
- Wichtigste Operationen:
  - `insert(key, value)`
  - `deleteMin()` (Alternativ: `deleteMax()`) $\rightarrow$ MinHeap/MaxHeap

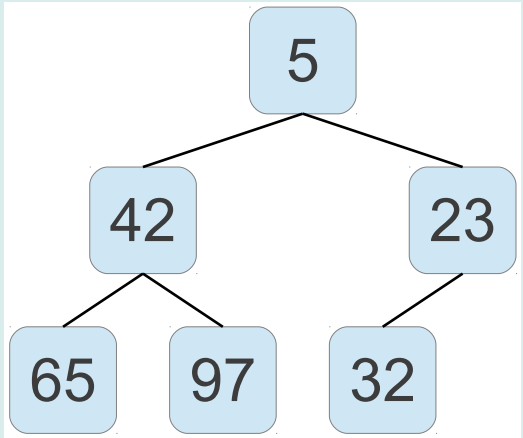
Prioritätslisten (Priority Queues)

Was?

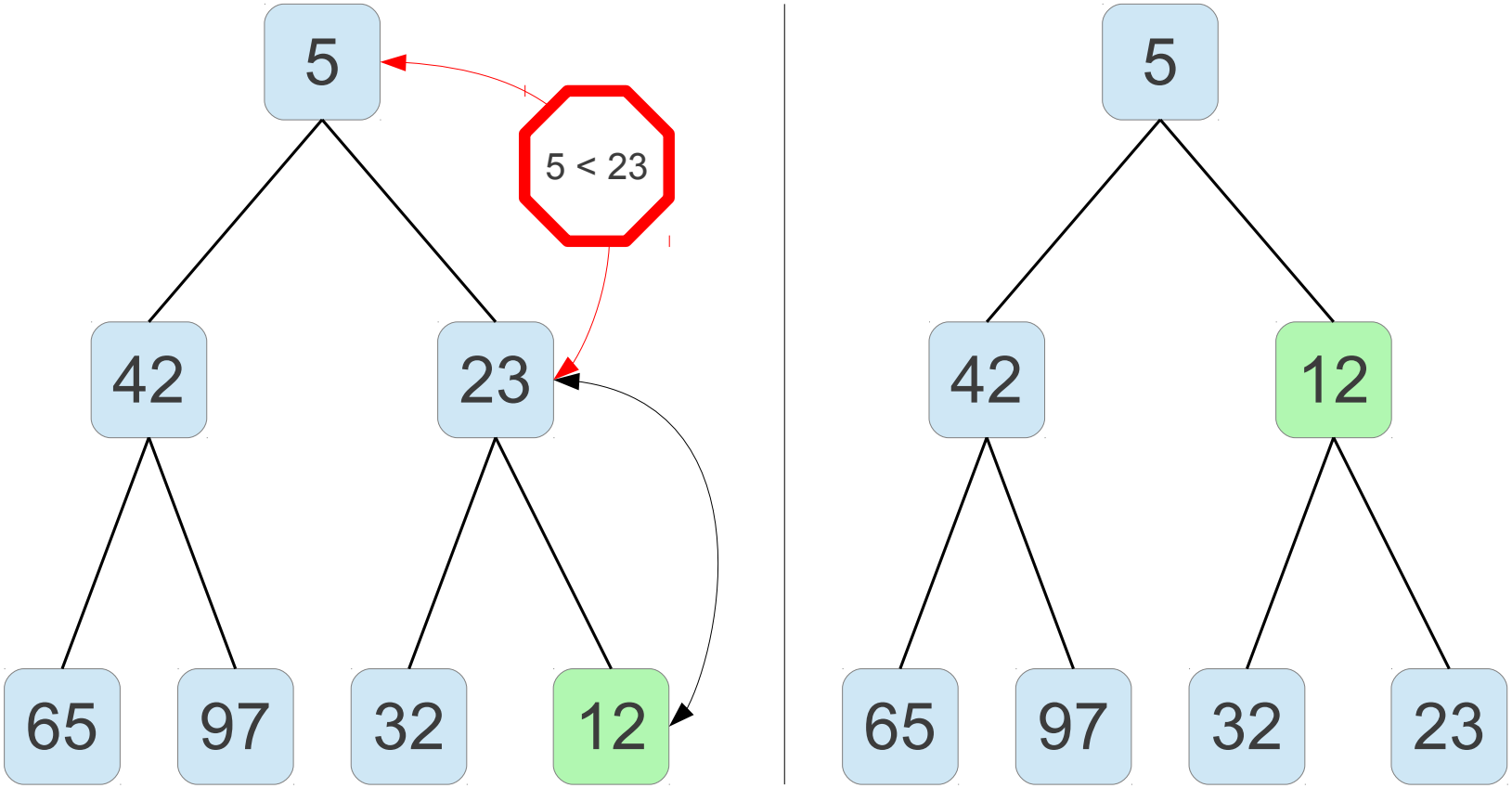
- Menge von Elementen mit einem Schlüssel
- Geordnet nach Schlüssel (Ordnungsrelation: $\leq$)
- Wichtigste Operationen:
  - `insert(key, value)`
  - `deleteMin()` (Alternativ: `deleteMax()`) $\rightarrow$ MinHeap/MaxHeap

Wie?

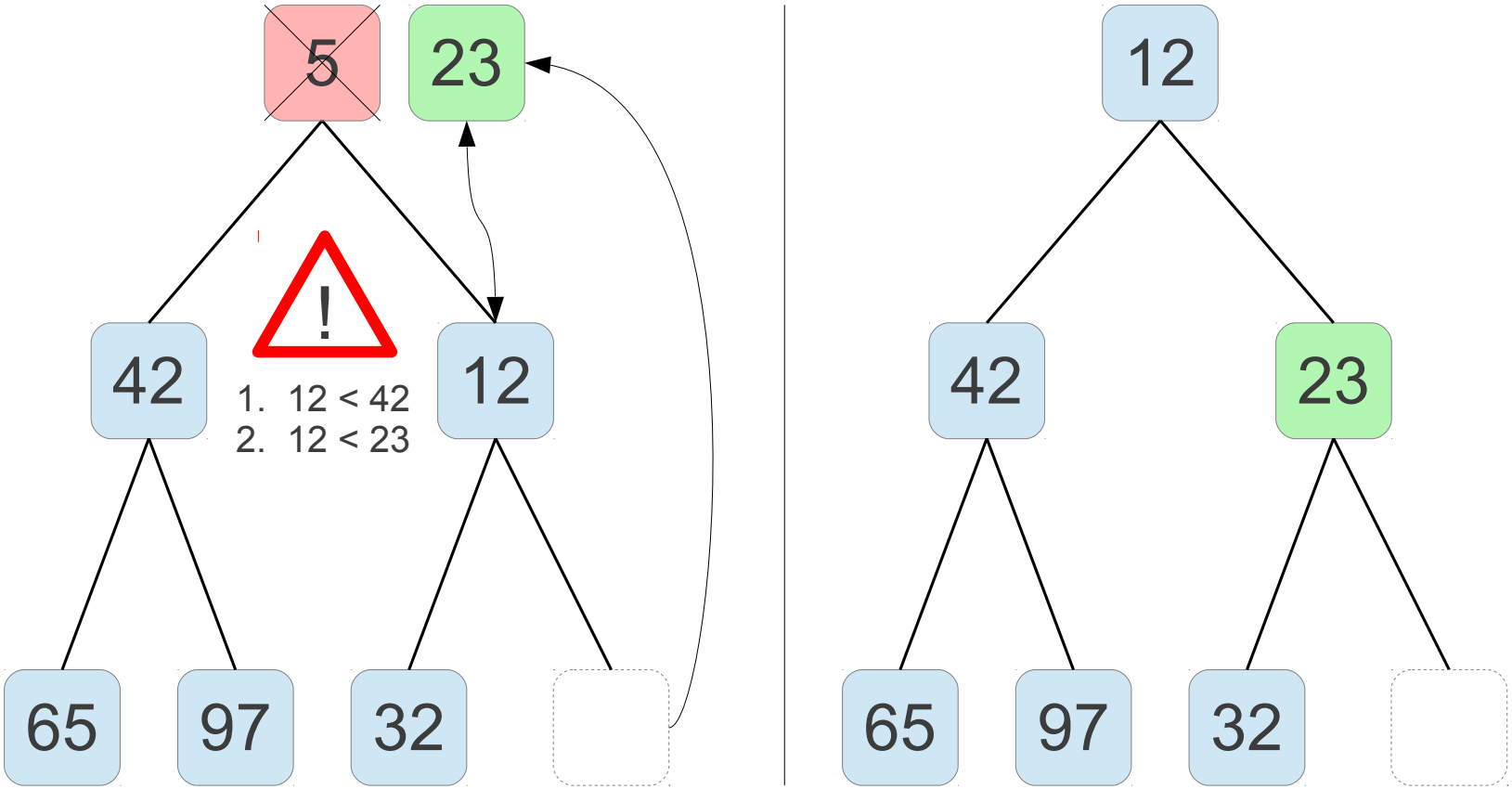
- Binäre Heaps
  - Baumstruktur
  - Jeder Knoten hat 0..2 Kindknoten
  - Keine Löcher, Fehlstellen nur rechtsseitig in der untersten Schicht
  - Tiefe: $\lfloor log_2(n) \rfloor$
- Heap-Eigenschaft: Elternknoten $<$ Kindknoten



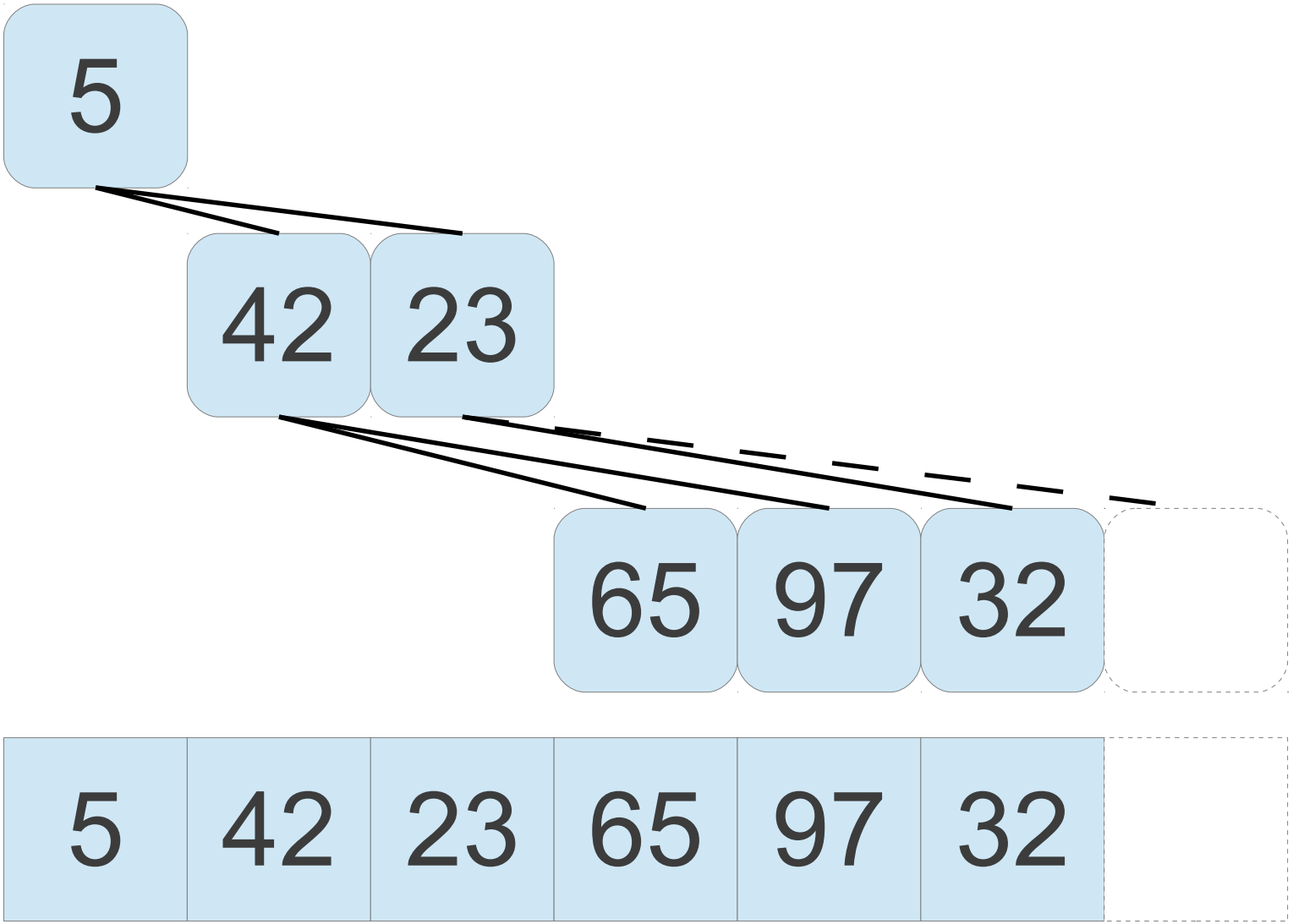
Binäre Heaps: `insert()`



Binäre Heaps: `deleteMin()`



Binäre Heaps: Einbettung



priority_queue

- Kann mit `deque` und `vector` arbeiten, aber nicht mit `list`
  - Warum?

priority_queue

- Kann mit `deque` und `vector` arbeiten, aber nicht mit `list`
  - Warum?
  - Einbettung: Größere Sprünge beim Traversieren $\rightarrow$ „teuer“ ($\mathcal{O}(n)$)
- Wichtig: `top()`, `push()` und `pop()`

priority_queue

- Kann mit `deque` und `vector` arbeiten, aber nicht mit `list`
  - Warum?
  - Einbettung: Größere Sprünge beim Traversieren $\rightarrow$ „teuer“ ($\mathcal{O}(n)$)
- Wichtig: `top()`, `push()` und `pop()`
- Laufzeit: $\mathcal{O}(log(n))$ für Modifikationen
- Heap-Eigenschaft hier: child @ parent (@ : Vergleichsoperator)

priority_queue

- Kann mit `deque` und `vector` arbeiten, aber nicht mit `list`
  - Warum?
  - Einbettung: Größere Sprünge beim Traversieren $\rightarrow$ „teuer“ ($\mathcal{O}(n)$)
- Wichtig: `top()`, `push()` und `pop()`
- Laufzeit: $\mathcal{O}(log(n))$ für Modifikationen
- Heap-Eigenschaft hier: child @ parent (@ : Vergleichsoperator)

```
1  // #include <priority_queue> #include <functional> // for greater<T>
2
3  struct Bar_t { int a, b; };
4  struct BartComp {
5      bool operator() (const Bar_t &lhs, const Bar_t &rhs) const {
6          return (lhs.a != rhs.a) ? (lhs.a < rhs.a) : (lhs.b < rhs.b);
7      }
8  };
9
10 std::priority_queue<int> pq1; // PQ of ints, ordering default == less<T>
11 std::priority_queue<int, greater<int>> pq2; // Inv. ordering -> MinHeap
12 std::priority_queue<Bar_t, BartComp> pq3; // Bar_t's ordered by BartComp
```

Überblick

Assoziative Container?!

- Erweiterung der `priority_queue`: Zugriff über Schlüssel
- Universell einsetzbar

Überblick

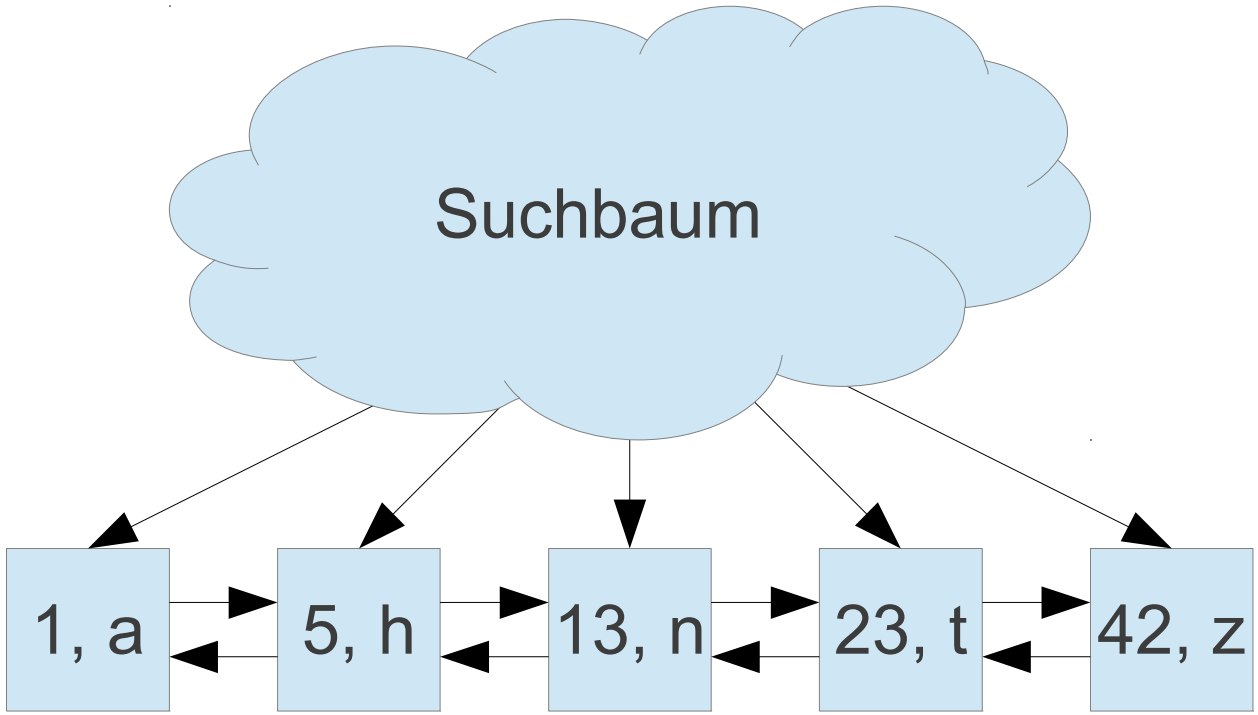
Assoziative Container?!

- Erweiterung der `priority_queue`: Zugriff über Schlüssel
- Universell einsetzbar

- `(multi)map`
- `(multi)set`

Sortierte Folgen

- Zuordnung Schlüssel $\rightarrow$ Element
- Speicherung: Geordnet nach Schlüssel



(multi)map

- `multimap`: Erlaubt mehr als ein Element je Schlüsselwert
- Schlüssel + Element: `pair<key_t, value_t>`

(multi)map

- `multimap`: Erlaubt mehr als ein Element je Schlüsselwert
- Schlüssel + Element: `pair<key_t, value_t>`
- Wichtige Operationen:
  - `find(key_t)`
  - `operator[key_t]`
  - `insert(pair<...>)`, `erase(key_t)`
  - `lower_bound(key_t)`, `upper_bound(key_t)`

(multi)map

- `multimap`: Erlaubt mehr als ein Element je Schlüsselwert
- Schlüssel + Element: `pair<key_t, value_t>`
- Wichtige Operationen:
  - `find(key_t)`
  - `operator[key_t]`
  - `insert(pair<...>)`, `erase(key_t)`
  - `lower_bound(key_t)`, `upper_bound(key_t)`
- Zeitkomplexität: Meist $\mathcal{O}(log(n))$

(multi)map

- `multimap`: Erlaubt mehr als ein Element je Schlüsselwert
- Schlüssel + Element: `pair<key_t, value_t>`
- Wichtige Operationen:
  - `find(key_t)`
  - `operator[key_t]`
  - `insert(pair<...>)`, `erase(key_t)`
  - `lower_bound(key_t)`, `upper_bound(key_t)`
- Zeitkomplexität: Meist $\mathcal{O}(log(n))$

```
1  #include <map>
2  typedef std::map< int, bar_t > Barmap_t;
3  Barmap_t barmap;                    // create a map mapping int -> bar_t
4  barmap[23] = bar_t("foo");          // insert a bar_t with key 23
5
6  typedef Barmap_t::const_iterator it_t;
7  it_t found = barmap.find(23);    // search for an element using a key
8  if(barmap.end() == found) { /* no element with this key */ }
```

(multi)map

- `multimap`: Erlaubt mehr als ein Element je Schlüsselwert
- Schlüssel + Element: `pair<key_t, value_t>`
- Wichtige Operationen:
  - `find(key_t)`
  - `operator[key_t]`
  - `insert(pair<...>)`, `erase(key_t)`
  - `lower_bound(key_t)`, `upper_bound(key_t)`
- Zeitkomplexität: Meist $\mathcal{O}(log(n))$

```
1  #include <map>
2  typedef std::map< int, bar_t > Barmap_t;
3  Barmap_t barmap;                    // create a map mapping int -> bar_t
4  barmap[23] = bar_t("foo");          // insert a bar_t with key 23
5
6  typedef Barmap_t::const_iterator it_t;
7  it_t found = barmap.find(23);    // search for an element using a key
8  if(barmap.end() == found) { /* no element with this key */ }
```

Mehr/bessere Beispiele: `http://www.cplusplus.com/reference/stl/map/`

(multi)set

- `set<type>` == `map<type, type>`

  → Element ist gleichzeitig Schlüssel

(multi)set

- `set<type>` == `map<type, type>`

  → Element ist gleichzeitig Schlüssel
- Wichtige Operationen:
  - `find()`
  - `insert()`, `erase()`
  - `lower_bound()`, `upper_bound()`

(multi)set

- `set<type>` == `map<type, type>`

  → Element ist gleichzeitig Schlüssel
- Wichtige Operationen:
  - `find()`
  - `insert()`, `erase()`
  - `lower_bound()`, `upper_bound()`
- Zeitkomplexität: Meist $\mathcal{O}(log(n))$

bitset

- Der Name ist irreführend! Besser wäre `bitvector`
- Kann: Bitfolgen kompakt speichern
- Achtung: Die Größe ist fix (Template-Parameter)
  - Im Gegensatz zu `vector<bool>`

bitset

- Der Name ist irreführend! Besser wäre `bitvector`
- Kann: Bitfolgen kompakt speichern
- Achtung: Die Größe ist fix (Template-Parameter)
  - Im Gegensatz zu `vector<bool>`
- Wichtige Operationen:
  - `set()`, `reset()` und `flip()`
  - `test()`, `any()` und `none()`
- Zeitkomplexität: $\mathcal{O}(1)$

7 Praxis

Praxis!

- Bearbeitung der Aufgaben aus den letzten Workshops

`https://github.com/kit-cpp-workshop/workshop-ss12-09`

Aufgabenbeschreibungen und Hinweise: Siehe `README.md`