Mengen (<set>)

Der Datentyp `set` gehört zu den assoziativen Behältern, er stellt wie dieser 4 *bidirektionale* Iteratoren zur Verfügung. Im Unterschied zum `map` gibt es keinen Indexoperator `[]`, Mengenelemente lassen sich nicht ändern, sondern nur einfügen und löschen.

Vereinbarung und Operationen

Operation	*Wirkung*
`set<I> a  set<I,Cmp> a`	vereinbart leere Menge, Ordnung evtl. durch `Cmp`
`set<I,Cmp> a(cmp)`	bzw. `cmp` induziert
`set<I> a(b)  set<I,Cmp> a(b)`	vereinbart Menge a als Kopie von b
`a=b`	Zuweisung
`a==b a!=b a>b a<b a<=b a>=b`	Vergleiche
`a.size()`	Zahl der Komponenten
`a.max_size()`	maximale Zahl der Komponenten
`a.clear()`	löscht alle Indizes und Komponenten
`a.empty()`	wahr, falls Menge leer
`a.count(i)`	1, falls Menge Element i enthält, 0 sonst
`a.find(i)`	liefert Position von Element i oder `end()`
`a.swap(b)`	vertauscht Indizes und Komp. von a mit denen von b
`pos++ ++pos pos-- --pos`	In-/Dekrementieren von Iterator `pos`
`*pos`	Liefert Element zum Iterator `pos`
`a.lower_bound(i)`	Liefert Iterator zu kleinstem Element $\geq i$
`a.upper_bound(i)`	Liefert Iterator zu kleinstem Element $> i$
`a.insert(i)`	fügt i zur Menge hinzu, liefert $(pos, success)$:
	$success$=`true`, falls Element neu eingefügt
`a.insert(pos0,i)`	fügt i zur Menge hinzu, liefert $(pos, success)$,
	$pos0$ Vorschlag für Einfügepostion
`a.insert(anf,end)`	fügt Elemente zu $[anf, end)$ ein
`a.erase(i)`	löscht Element i, Rückgabe: Zahl der gelöschten Elem.
`a.erase(pos)`	löscht Element an Position pos
`a.erase(anf,end)`	löscht Elemente zu $[anf, end)$
`a.clear()`	löscht alle Elemente (Menge dann leer)

Mengenoperationen lassen sich mit den Algorithmen für vorsortierte Bereiche durchführen

Bitsets (<bitset>)

Vereinbarung und Operationen

Der Datentyp `bitset<n>` dient zur Bearbeitung von Folgen aus n Bits mit festem n. Seien im folgenden a und b Variablen vom Typ `bitset<n>`, `ul` eine vorzeichenlose ganze Zahl vom Typ `unsigned long`, i, k vorzeichenlose Zahlen vom Typ `size_t`, j eine ganze Zahl vom Typ `int` mit Wert 0 oder 1, s eine C++-Zeichenkette, die nur Nullen oder Einsen enthält.

Operationen	*Bedeutung*
`bitset<n> a`	vereinbart a als Folge von n Bits, initialisiert mit 0
`bitset<n> a(b)`	vereinbart a als Folge von n Bits, initialisiert anhand b
`bitset<n> a(ul)`	vereinbart a als Folge von n Bits, initialisiert anhand ul
`bitset<n> a(s)`	vereinbart a als Folge von n Bits, initialisiert anhand s
`bitset<n> a(s,i)`	vereinbart a als Folge von n Bits, initialisiert anhand s ab
`bitset<n> a(s,i,k)`	Index i mit k Zeichen bzw. bis zum Ende von s

Operationen	*Bedeutung*
`a[i]`	liefert Bit zum Index i als bool. Wert bzw. ermögl. Schreibzugriff
`a.at(i)`	wie `a[i]` mit Bereichsüberprüfung (ggf. `out_of_range`)
`a.set() a.set(i) a.set(i,j)`	setzt alle Bits bzw. das Bit zum Index i auf 1 bzw. j
`a.reset() a.reset(i)`	setzt alle Bits bzw. das Bit zum Index i von a auf 0
`a.flip() a.flip(i)`	kehrt alle Bits bzw. das Bit zum Index i von a um
`a=b a=ul`	Zuweisung
`a==b a!=b`	Vergleiche
`a&b a|b a^b`	liefert und/oder/entweder-oder-Verknüpfung von a und b
`~a`	liefert Bitset, in dem alle Bits von a umgekehrt sind
`a<<k a>>k`	liefert Bitset, bei dem die Bits von a um k Positionen nach links bzw. rechts verschoben sind (Nullen werden nachgeschoben)
`a&=b a|=b a^=b`	entspricht `a=a&b a=a|b a=a^b`
`a<<=k a>>=k`	entspricht `a=a<<k a=a>>k`
`a.size()`	liefert n
`a.count()`	Zahl der auf 1 gesetzten Bits
`a.any()`	wahr, falls mindestens ein Bit auf 1 gesetzt ist
`a.none()`	wahr, falls alle Bits auf 0 gesetzt sind
`a.test(i)`	wahr, falls Bit zum Index i auf 1 gesetzt ist
`a.to_ulong()`	liefert die a entsprechende Zahl vom Typ `unsigned long`
`a.to_string()`	liefert die a entsprechende C++-Zeichenkette mit 0 oder 1
`cout << a`	gibt a mit 0 oder 1 aus
`cin >> a`	liest a ein (Eingabe besteht nur aus 0 oder 1)

Bei Umwandlungen zwischen Bitsets und Zeichenketten wird das höchstwertige Bit am Anfang der Zeichenkette (Index 0) gespeichert und daher auch zuerst eingelesen bzw. ausgegeben.

Bei Umwandlungen zwischen Bitsets und vorzeichenlosen ganzen Zahlen vom Typ `unsigned long` entsprechen sich die jeweiligen Bits in ihrer Wertigkeit.

Bsp.: Zahlumwandlungen (g++-9.4, Ubuntu Linux 20.04, amd64)

```
#include <iostream>
#include <bitset>
#include <string>

using namespace std;

int main()
{
  const int n=64;
  unsigned long u=9999999;
  double        x=-4.0/3.0;
  bitset<n> bu,bx;
  bu = u;                                  // Bitmuster von u auf bu
  cout << u << ":   "   << bu << endl;
  bx = *(unsigned long*)&x;                // Bitmuster von x auf bx
  cout << "-4.0/3.0:  " << bx << endl;

  string s("10111111111111111110000000000000000000000000000000000000000000000000");
  bitset<n> by(s);
  unsigned long ul = by.to_ulong();
  cout << s << ":   " << ul << endl;
  double y = *(double*)&ul;
  cout << s << ":   " << y << endl;
```

```
  bitset<n> bz;
  for (int i=62; i>=52; --i) bz[i]=1;    // Bitmuster setzen
  unsigned long zul = bz.to_ulong();
  double z = (double&)zul;               // Bitmuster setzen (per Referenz)
  cout << bz << ":   " << zul << endl;
  cout << bz << ":   " << z << endl;
  return 0;
}
```

Ausgabe:

```
9999999:   00000000000000000000000000000000000000000010011000100101100111111
-4.0/3.0:  10111111111101010101010101010101010101010101010101010101010101010101
10111111111111111111000000000000000000000000000000000000000000000000:   13835040463096119296
10111111111111111111000000000000000000000000000000000000000000000000:   -1.99609
01111111111100000000000000000000000000000000000000000000000000000000:   9218868437227405312
01111111111100000000000000000000000000000000000000000000000000000000:   inf
```

Bsp.: Sieb des Eratosthenes, Zweiquadratesatz von Fermat

```
// Stacksize unter Linux heraufsetzen:  ulimit -s unlimited
// oder n verkleinern

#include <iostream>
#include <bitset>
#include <string>
#include <set>

using namespace std;

const size_t n=10000000;

void ausgabe(string s, bitset<n> a, size_t k)
// Bitindizes kleiner k mit Wert 1 ausgeben
{
  cout << s << ":" << endl;
  for (size_t i=0; i<k; ++i) if (a[i]==1) cout << i << " ";
  cout << endl;
}

int main()
{
  size_t i,j,k;
  bitset<n> primzahlen, zweiquadratzahlen, rest1mod4, ungeradeprimzahlen;
  set<int> primzahlmenge;
  primzahlen.set();
  primzahlen[0]=0; primzahlen[1]=0; // 0 und 1 sind keine Primzahlen

  // Sieb des Eratosthenes
  for (k=1; k*k<n; ++k) {
    if (primzahlen[k]==1)
      // Vielfache von k streichen
      for (i=2; i*k<n; ++i) primzahlen[i*k]=0;
  }
```

```
  ausgabe("Primzahlen unter 100",primzahlen,100);

  // Zweiquadratzahlen
  for (i=0; 2*i*i<n; ++i)
    for (j=i; j<n; ++j) {
      k=i*i+j*j; if (k>=n) break;
      zweiquadratzahlen[k]=1;
    }

  ausgabe("Zweiquadratzahlen unter 50",zweiquadratzahlen,50);

  // Zahlen mit Rest 1 modulo4
  for (i=0; 4*i+1<n; ++i) rest1mod4[4*i+1]=1;

  ungeradeprimzahlen=primzahlen; ungeradeprimzahlen[2]=0; // 2 ist gerade Primzahl

  ausgabe("Primzahlen unter 200 mit Rest 1 mod 4",primzahlen & rest1mod4, 200);
  ausgabe("Ungerade Primzahlen unter 200, die Zweiquadratzahlen sind",
           ungeradeprimzahlen & zweiquadratzahlen, 200);

  if ( (ungeradeprimzahlen & rest1mod4) == (ungeradeprimzahlen & zweiquadratzahlen) )
    cout << "Jede Primzahl mit Rest 1 mod 4 < " << n
         << " ist Summe zweier Quadrate" << endl;

  // Primzahlen als set<int>
  for (k=2; k<n; ++k) if (primzahlen[k]==1) primzahlmenge.insert(k);

  // Primzahlen im Bereich [M,N] ausgegeben
  int M,N; cout << "M N: "; cin  >> M >> N;
  cout << "Primzahlen im Bereich [" << M << "," << N << "]:" << endl;
  for (set<int>::iterator pos =  primzahlmenge.lower_bound(M);
                          pos != primzahlmenge.upper_bound(N); ++pos)
    cout << *pos << " ";
  cout << endl;

  return 0;
}
```

Ausgabe:

```
Primzahlen unter 100:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Zweiquadratzahlen unter 50:
0 1 2 4 5 8 9 10 13 16 17 18 20 25 26 29 32 34 36 37 40 41 45 49
Primzahlen unter 200 mit Rest 1 mod 4:
5 13 17 29 37 41 53 61 73 89 97 101 109 113 137 149 157 173 181 193 197
Ungerade Primzahlen unter 200, die Zweiquadratzahlen sind:
5 13 17 29 37 41 53 61 73 89 97 101 109 113 137 149 157 173 181 193 197
Jede Primzahl mit Rest 1 mod 4 < 10000000 ist Summe zweier Quadrate
M N: 1000000 1000100
Primzahlen im Bereich [1000000,1000100]:
1000003 1000033 1000037 1000039 1000081 1000099
```