- Die Schnittstelle für Threads ist eine Abstraktion des Betriebssystems (oder einer virtuellen Maschine), die es ermöglicht, mehrere Ausführungsfäden, jeweils mit eigenem Stack und PC ausgestattet, in einem gemeinsamen Adressraum arbeiten zu lassen.
- Der Einsatz lohnt sich insbesondere auf Mehrprozessormaschinen mit gemeinsamen Speicher.
- Vielfach wird die Fehleranfälligkeit kritisiert wie etwa von C. A. R. Hoare in *Communicating Sequential Processes*: „In its full generality, multithreading is an incredibly complex and error-prone technique, not to be recommended in any but the smallest programs.“

- Wie die *comp.os.research* FAQ belegt, gab es Threads bereits lange vor der Einführung von Mehrprozessormaschinen:
  „The notion of a thread, as a sequential flow of control, dates back to 1965, at least, with the Berkeley Timesharing System. Only they weren't called threads at that time, but processes. Processes interacted through shared variables, semaphores, and similar means. Max Smith did a prototype threads implementation on Multics around 1970; it used multiple stacks in a single heavyweight process to support background compilations."
  `http://www.serpentine.com/blog/threads-faq/the-history-of-threads/`
- UNIX selbst kannte zunächst nur Prozesse, d.h. jeder Thread hatte seinen eigenen Adressraum.

- Zu den ersten UNIX-Implementierungen, die Threads unterstützten, gehörten der Mach-Microkernel (eingebettet in NeXT, später Mac OS X) und Solaris (zur Unterstützung der ab 1992 hergestellten Multiprozessormaschinen). Heute unterstützen alle UNIX-Varianten einschließlich Linux Threads.
- 1995 wurde von *The Open Group* (einer Standardisierungsgesellschaft für UNIX) mit POSIX 1003.1c-1995 eine standardisierte Threads-Bibliotheksschnittstelle publiziert, die 1996 von der IEEE, dem ANSI und der ISO übernommen wurde.
- Diverse andere Bibliotheken für Threads (u.a. von Microsoft und von Sun) existierten und existieren, sind aber nicht portabel und daher von geringerer Bedeutung.

- Spezifikation der *Open Group*:
  `http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html`
- Unterstützt
  - das Erzeugen von Threads und das Warten auf ihr Ende,
  - den gegenseitigen Ausschluss (notwendig, um auf gemeinsame Datenstrukturen zuzugreifen),
  - Bedingungsvariablen (*condition variables*), die einem Prozess signalisieren können, dass sich eine Bedingung erfüllt hat, auf die gewartet wurde,
  - Lese- und Schreibsperren, um parallele Lese- und Schreibzugriffe auf gemeinsame Datenstrukturen zu synchronisieren.
- Freie Implementierungen der Schnittstelle für C:
  - GNU Portable Threads:
    `http://www.gnu.org/software/pth/`
  - Native POSIX Thread Library:
    `http://people.redhat.com/drepper/nptl-design.pdf`

- Seit dem aktuellen C++-Standard ISO 14882-2012 (C++11) werden POSIX-Threads direkt unterstützt, wobei die POSIX-Schnittstelle in eine für C++ geeignete Weise verpackt ist.
- Ältere C++-Übersetzer unterstützen dies noch nicht, aber die Boost-Schnittstelle für Threads ist recht ähnlich und kann bei älteren Systemen verwendet werden. (Alternativ kann auch die C-Schnittstelle in C++ verwendet werden, was aber recht umständlich ist.)
- Die folgende Einführung bezieht sich auf C++11. Bei g++ sollte also die Option „-std=gnu++11“ verwendet werden.

- Die ausführende Komponente eines Threads wird in C++ immer durch ein sogenanntes Funktionsobjekt repräsentiert.
- In C++ sind alle Objekte Funktionsobjekte, die den parameterlosen Funktionsoperator unterstützen.
- Das könnte im einfachsten Falle eine ganz normale parameterlose Funktion sein:

```
void f() {
   // do something
}
```

- Das ist jedoch nicht sehr hilfreich, da wegen der fehlenden Parametrisierung unklar ist, welche Teilaufgabe die Funktion für einen konkreten Thread erfüllen soll.

```
class Thread {
   public:
      Thread( /* parameters */ );
      void operator()() {
         // do something in dependence of the parameters
      }
   private:
      // parameters of this thread
};
```

- Eine Klasse für Funktionsobjekte muss den parameterlosen Funktionsoperator unterstützen, d.h. **void operator**()().
- Im privaten Bereich der Thread-Klasse können alle Parameter untergebracht werden, die für die Ausführung eines Threads benötigt werden.
- Der Konstruktor erhält die Parameter eines Threads und kann diese dann in den privaten Bereich kopieren.
- Dann kann die parameterlose Funktion problemlos auf ihre Parameter zugreifen.

Repräsentierung eines Threads

fork-and-join.cpp

```
class Thread {
   public:
      Thread(int i) : id(i) {};
      void operator()() {
         cout << "thread " << id << " is operating" << endl;
      }

   private:
      const int id;
};
```

- In diesem einfachen Beispiel wird nur ein einziger Parameter für den einzelnen Thread verwendet: *id*
- (Ein Parameter, der die Identität des Threads festlegt, genügt in vielen Fällen bereits.)
- Für Demonstrationszwecke gibt der Funktionsoperator nur seine eigene *id* aus.
- So ein Funktionsobjekt kann auch ohne Threads erzeugt und benutzt werden:
  Thread t(7); *t*();

fork-and-join.cpp

```
#include <iostream>
#include <thread>

using namespace std;

// class Thread...

int main() {
   // fork off some threads
   thread t1(Thread(1)); thread t2(Thread(2));
   thread t3(Thread(3)); thread t4(Thread(4));
   // and join them
   cout << "Joining..." << endl;
   t1.join(); t2.join(); t3.join(); t4.join();
   cout << "Done!" << endl;
}
```

- Objekte des Typs *std*::*thread* (aus **#include** <thread>) können mit einem Funktionsobjekt initialisiert werden. Die Threads werden sofort aktiv.
- Mit der *join*-Methode wird auf die Beendigung des jeweiligen Threads gewartet.

$$P_i = (fork \rightarrow join \rightarrow SKIP)$$

- Beim Fork-And-Join-Pattern werden beliebig viele einzelne Threads erzeugt, die dann unabhängig voneinander arbeiten.
- Entsprechend bestehen die Alphabete nur aus *fork* und *join*.
- Das Pattern eignet sich für Aufgaben, die sich leicht in unabhängig voneinander zu lösende Teilaufgaben zerlegen lassen.
- Die Umsetzung in C++ sieht etwas anders aus mit $\alpha P_i = \{fork_i, join_i\}$ und $\alpha M = \alpha P = \cup_{i=1}^{n} \alpha P_i$:

$$P = M \,||\, P_1 \,||\, \cdots \,||\, P_n$$
$$M = (fork_1 \rightarrow \cdots \rightarrow fork_n \rightarrow join_1 \rightarrow \cdots \rightarrow join_n \rightarrow SKIP)$$
$$P_i = (fork_i \rightarrow join_i \rightarrow SKIP)$$

fork-and-join2.cpp

```
// fork off some threads
thread threads[10];
for (int i = 0; i < 10; ++i) {
   threads[i] = thread(Thread(i));
}
```

- Wenn Threads in Datenstrukturen unterzubringen sind (etwa Arrays oder beliebigen Containern), dann können sie nicht zeitgleich mit einem Funktionsobjekt initialisiert werden.
- In diesem Falle existieren sie zunächst nur als leere Hülle.
- Wenn Thread-Objekte einander zugewiesen werden, dann wird ein Thread nicht dupliziert, sondern die Referenz auf den eigentlichen Thread wandert von einem Thread-Objekt zu einem anderen (Verlagerungs-Semantik).
- Im Anschluss an die Zuweisung hat die linke Seite den Verweis auf den Thread, während die rechte Seite dann nur noch eine leere Hülle ist.

fork-and-join2.cpp

```
// and join them
cout << "Joining..." << endl;
for (int i = 0; i < 10; ++i) {
   threads[i].join();
}
```

- Das vereinfacht dann auch das Zusammenführen all der Threads mit der *join*-Methode.

simpson.cpp

```
double simpson(double (*f)(double), double a, double b, int n) {
   assert(n > 0 && a <= b);
   double value = f(a)/2 + f(b)/2;
   double xleft;
   double x = a;
   for (int i = 1; i < n; ++i) {
      xleft = x; x = a + i * (b - a) / n;
      value += f(x) + 2 * f((xleft + x)/2);
   }
   value += 2 * f((x + b)/2); value *= (b - a) / n / 3;
   return value;
}
```

- *simpson* setzt die Simpsonregel für das in n gleichlange Teilintervalle aufgeteilte Intervall $[a, b]$ für die Funktion f um:

$$S(f, a, b, n) = \frac{h}{3}\left(\frac{1}{2}f(x_0) + \sum_{k=1}^{n-1} f(x_k) + 2\sum_{k=1}^{n} f(\frac{x_{k-1} + x_k}{2}) + \frac{1}{2}f(x_n)\right)$$

mit $h = \frac{b-a}{n}$ und $x_k = a + k \cdot h$.

simpson.cpp

```
class SimpsonThread {
   public:
      SimpsonThread(double (*f)(double), double a, double b, int n,
                    double& rp) :
            f(f), a(a), b(b), n(n), rp(rp) {
      }
      void operator()() {
         rp = simpson(f, a, b, n);
      }
   private:
      double (*f)(double);
      double a, b;
      int n;
      double& rp;
};
```

- Jedem Objekt werden nicht nur die Parameter der *simpson*-Funktion übergeben, sondern auch noch einen Zeiger auf die Variable, wo das Ergebnis abzuspeichern ist.

simpson.cpp

```
double mt_simpson(double (*f)(double), double a, double b, int n,
      int nofthreads) {
   // divide the given interval into nofthreads partitions
   assert(n > 0 && a <= b && nofthreads > 0);
   int nofintervals = n / nofthreads;
   int remainder = n % nofthreads;
   int interval = 0;

   std::thread threads[nofthreads];
   double results[nofthreads];

   // fork & join & collect results ...
}
```

- *mt_simpson* ist wie die Funktion *simpson* aufzurufen – nur ein Parameter *nofthreads* ist hinzugekommen, der die Zahl der zur Berechnung zu verwendenden Threads spezifiziert.
- Dann muss die Gesamtaufgabe entsprechend in Teilaufgaben zerlegt werden.

simpson.cpp

```
double x = a;
for (int i = 0; i < nofthreads; ++i) {
   int intervals = nofintervals;
   if (i < remainder) ++intervals;
   interval += intervals;
   double xleft = x; x = a + interval * (b - a) / n;
   threads[i] = std::thread(SimpsonThread(f,
      xleft, x, intervals, results[i]));
}
```

- Für jedes Teilproblem wird ein entsprechendes Funktionsobjekt temporär angelegt, an *std::thread* übergeben, womit ein Thread erzeugt wird und schließlich an *threads*[*i*] mit der Verlagerungs-Semantik zugewiesen.
- Hierbei wird auch implizit der Verlagerungs- oder Kopierkonstruktor von *SimpsonThread* verwendet.

simpson.cpp

```
double sum = 0;
for (int i = 0; i < nofthreads; ++i) {
   threads[i].join();
   sum += results[i];
}
return sum;
```

- Wie gehabt erfolgt die Synchronisierung mit der *join*-Methode.
- Danach kann das entsprechende Ergebnis abgeholt und aggregiert werden.

simpson.cpp

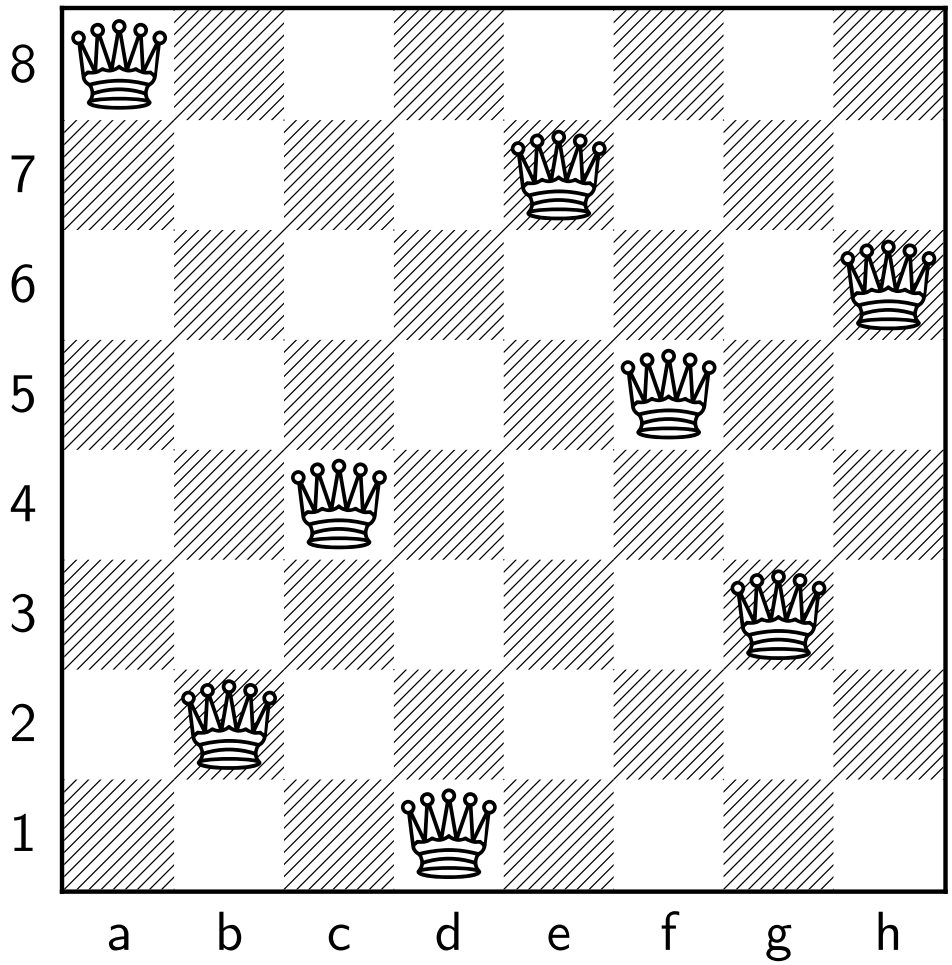
```
cmdname = *argv++; --argc;
if (argc > 0) {
   istringstream arg(*argv++); --argc;
   if (!(arg >> N) || N <= 0) usage();
}
if (argc > 0) {
   istringstream arg(*argv++); --argc;
   if (!(arg >> nofthreads) || nofthreads <= 0) usage();
}
if (argc > 0) usage();
```

- Es ist sinnvoll, die Zahl der zu startenden Threads als Kommandozeilenargument (oder alternativ über eine Umgebungsvariable) zu übergeben, da dieser Parameter von den gegebenen Rahmenbedingungen abhängt (Wahl der Maschine, zumutbare Belastung).
- Zeichenketten können in C++ wie Dateien ausgelesen werden, wenn ein Objekt des Typs *istringstream* damit initialisiert wird.
- Das Einlesen erfolgt in C++ mit dem überladenen >>-Operator, der als linken Operanden einen Stream erwartet und als rechten eine Variable.

simpson.cpp

```
// double sum = simpson(f, a, b, N);
double sum = mt_simpson(f, a, b, N, nofthreads);
cout << setprecision(14) << sum << endl;
cout << setprecision(14) << M_PI << endl;
```

- Testen Sie Ihr Programm zuerst immer ohne Threads, indem die Funktion zur Lösung eines Teilproblems verwendet wird, um das Gesamtproblem unparallelisiert zu lösen. (Diese Variante ist hier auskommentiert.)
- *cout* ist in C++ die Standardausgabe, die mit dem <<-Operator auszugebende Werte erhält.
- *setprecision*(14) setzt die Zahl der auszugebenden Stellen auf 14. *endl* repräsentiert einen Zeilentrenner.
- Zum Vergleich wird hier *M_PI* ausgegeben, weil zum Testen $f(x) = \frac{4}{1+x^2}$ verwendet wurde, wofür $\int_0^1 f(x)dx = 4 \cdot \arctan(1) = \pi$ gilt.

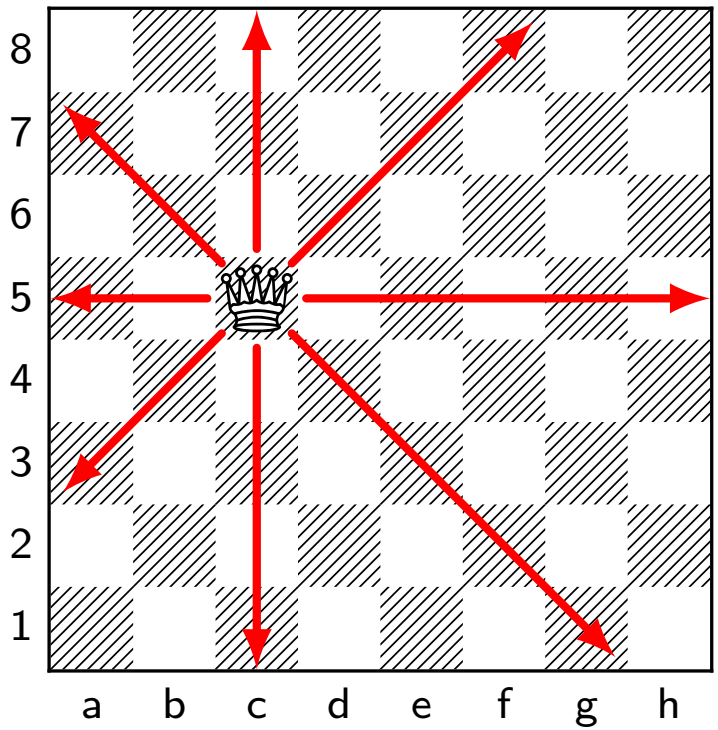


- Problem: Finde alle Stellungen für n Damen auf einem Schachbrett der Größe $n \times n$, so dass sie sich nicht gegenseitig bedrohen.

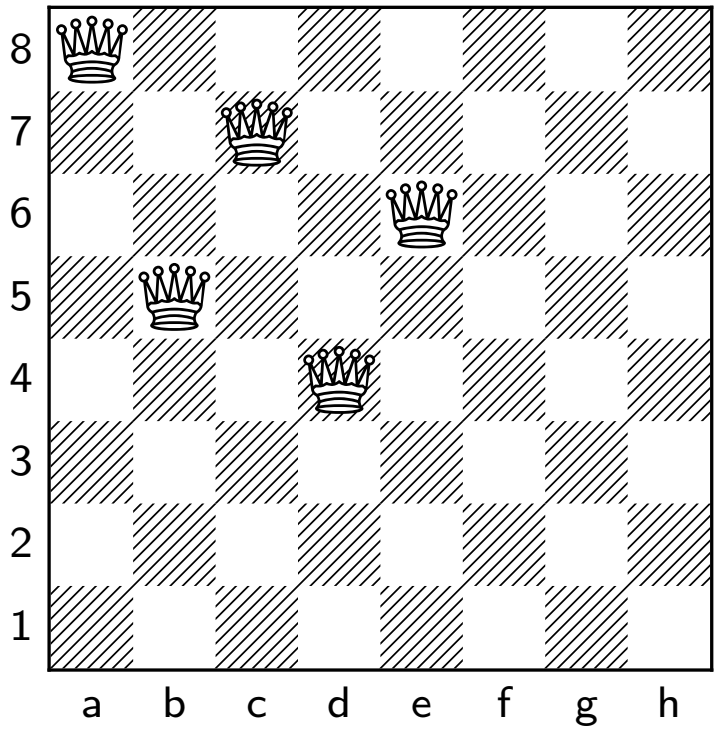
Lösungsansatz:

- Schritt für Schritt eine Lösung aufbauen, indem eine Dame nach der anderen auf das Brett gestellt wird.
- Wenn es keinen zulässigen Platz für die k-te Dame gibt, wird die $(k-1)$-te Dame vom Brett zurückgenommen und eine andere (noch nicht vorher probierte) Position verwendet.
- Wenn eine Lösung gefunden wird, wird sie notiert und die weiteren Möglichkeiten durchprobiert. (Wenn nur eine Lösung relevant ist, kann das Verfahren dann auch abgebrochen werden.)
- Der Verfahren endet, sobald alle Möglichkeiten durchprobiert worden sind.

Zugmöglichkeiten einer Dame



- Eine Dame im Schachspiel bedroht alle Felder
  - in der gleichen Zeile,
  - in der gleichen Spalte und
  - den beiden Diagonalen.



- Verfahren, die schrittweise Möglichkeiten durchprobieren und im Falle von „Sackgassen“ zuvor gemachte Schritte wieder zurücknehmen, um neue Varianten zu probieren, nennen sich Backtracking-Verfahren.
- Eine Sackgasse beim 8-Damen-Problem zeigt das obige Diagramm, da es nicht möglich ist, eine weitere Dame unterzubringen.

- Prinzipiell sind alle Backtracking-Verfahren zur Parallelisierung geeignet.
- Im einfachsten Falle wird für jeden möglichen Schritt ein neuer Thread erzeugt, der dann den Schritt umsetzt und von dort aus nach weiteren Schritten sucht.
- Ein explizites Zurücknehmen von Zügen ist nicht notwendig – stattdessen wird dieser Zweig bzw. dessen zugehöriger Thread beendet.
- Sobald eine Lösung gefunden wird, ist diese synchronisiert auszugeben bzw. in einer Datenstruktur zu vermerken.
- Jeder Thread ist dafür verantwortlich, die von ihm erzeugten Threads wieder mit *join* einzusammeln.

queens.cpp

```
class Thread {
   public:
      // ...
   private:
      unsigned int row; /* current row */
      /* a queen on (row, col) threatens a row, a column,
         and 2 diagonals;
         rows and columns are characterized by their number (0..n-1),
         the diagonals by row-col+n-1 and row+col,
         (n is a shorthand for the square size of the board)
      */
      unsigned int rows, cols; // bitmaps of [0..n-1]
      unsigned int diags1; // bitmap of [0..2*(n-1)] for row-col+n-1
      unsigned int diags2; // bitmap of [0..2*(n-1)] for row+col
      std::list<std::thread> threads; // list of forked-off threads
      std::list<unsigned int> positions; // columns of the queens
};
```

Datenstruktur eines Threads

queens.cpp

```
constexpr bool in_set(unsigned int member, unsigned int set) {
   return (1<<member) & set;
}
constexpr unsigned int include(unsigned int set, unsigned int member) {
   return set | (1<<member);
}
constexpr unsigned int N = 8; /* side length of chess board */

class Thread {
   public:
      // ...
   private:
      // ...
      unsigned int rows, cols; // bitmaps of [0..n-1]
      unsigned int diags1; // bitmap of [0..2*(n-1)] for row-col+n-1
      unsigned int diags2; // bitmap of [0..2*(n-1)] for row+col
      // ...
};
```

- Die von bereits besetzten Damen bedrohten Zeilen, Spalten und Diagonalen lassen sich am einfachsten durch Bitsets repräsentieren.

queens.cpp

```
Thread() : row(0), rows(0), cols(0), diags1(0), diags2(0) {
}
Thread(const Thread& other, unsigned int r, unsigned int c) :
      row(r+1), rows(other.rows), cols(other.cols),
      diags1(other.diags1), diags2(other.diags2),
      positions(other.positions) {
   positions.push_back(c);
   rows = include(rows, r); cols = include(cols, c);
   diags1 = include(diags1, r - c + N - 1);
   diags2 = include(diags2, r + c);
}
```

- Der erste Konstruktor dient der Initialisierung des ersten Threads, der von einem leeren Brett ausgeht.
- Der zweite Konstruktor übernimmt den Zustand eines vorhandenen Bretts und fügt noch eine Dame auf die übergebene Position hinzu, die sich dort konfliktfrei hinsetzen lässt.

queens.cpp

```
std::list<std::thread> threads;
```

- Objekte des Typs *std*::*thread* können in Container aufgenommen werden.
- Dabei ist aber die Verlagerungssemantik zu beachten, d.h. der Thread wandert in den Container und wenn er dort bleiben soll, sollte danach mit Referenzen gearbeitet werden.
- Diese Liste dient dazu, alle in einem Rekursionsschritt erzeugten Threads aufzunehmen, damit sie anschließend allesamt wieder mit *join* eingesammelt werden können.

queens.cpp

```
void operator()() {
   if (row == N) { print_board();
   } else {
      for (unsigned int col = 0; col < N; ++col) {
         if (in_set(row, rows)) continue;
         if (in_set(col, cols)) continue;
         if (in_set(row - col + N - 1, diags1)) continue;
         if (in_set(row + col, diags2)) continue;
         // create new thread with a queen set at (row,col)
         threads.push_back(std::thread(Thread(*this, row, col)));
      }
      for (auto& t: threads) t.join();
   }
}
```

- Wenn alle acht Zeilen besetzt sind, wurde eine Lösung gefunden, die dann nur noch ausgegeben werden muss.
- Ansonsten werden in der aktuellen Zeile sämtliche Spalten durchprobiert, ob sich dort konfliktfrei eine Dame setzen lässt. Falls ja, wird ein neuer Thread erzeugt.

Zugriff auf Threads im Container

queens.cpp

```
    for (auto& t: threads) t.join();
```

- Beim Zugriff auf Threads im Container sollte das versehentliche Kopieren vermieden werden, das zu einer Verlagerung führen würde.
- In solchen Fällen ist es ggf. sinnvoll, mit Referenzen zu arbeiten.
- Bei **auto** übernimmt der C++-Übersetzer die Bestimmung des korrekten Typs. Die Angabe von „&“ ist hier aber noch notwendig, um die Verlagerung der Threads aus dem Container zu vermeiden.

$$
\begin{aligned}
MX &= M \,||\, (P_1 \,|||\, \cdots \,|||\, P_n) \\
M &= (lock \rightarrow unlock \rightarrow M) \\
P_i &= (lock \rightarrow begin\_critical\_region_i \rightarrow \\
&\quad end\_critical\_region_i \rightarrow unlock \rightarrow \\
&\quad private\_work_i \rightarrow P_i)
\end{aligned}
$$

- n Prozesse $P_1 \ldots P_n$ wollen konkurrierend auf eine Ressource zugreifen, aber zu einem Zeitpunkt soll diese nur einem Prozess zur Verfügung stehen.
- Die Ressource wird von M verwaltet.
- Wegen dem $|||$-Operator wird unter den P_i jeweils ein Prozess nicht-deterministisch ausgewählt, der für *lock* bereit ist.

Mutex-Variablen für Threads

queens.cpp

```
#include <mutex>

std::mutex cout_mutex;

// ...

void print_board() {
   std::lock_guard<std::mutex> lock(cout_mutex);
   // cout << ...
}
```

- Bei Threads wird ein gegenseitiger Ausschluss über sogenannte Mutex-Variablen (*mutual exclusion*) organisiert.
- Sinnvollerweise sollten Ausgaben auf den gleichen Stream (hier *std::cout*) synchronisiert erfolgen.
- Deswegen darf nur der Thread schreiben, der exklusiven Zugang zu der Mutex-Variable *cout_mutex* hat.
- Mit *std::lock_guard* gibt es die Möglichkeit, ein Objekt zu erzeugen, das den Mutex bei der Konstruktion reserviert und bei der Dekonstruktion automatisiert freigibt.

queens.cpp

```
#include <mutex>

std::mutex cout_mutex;

// ...

void print_board() {
   std::lock_guard<std::mutex> lock(cout_mutex);
   // cout << ...
}
```

- Code-Bereiche, die nur im gegenseitigen Ausschluss ausgeführt werden können, werden kritische Regionen genannt (*critical regions*).
- Am einfachsten ist es, beim Zugriff auf eine Datenstruktur alle anderen auszuschließen.
- Gelegentlich ist der gegenseitige Ausschluss auch feiner granuliert, wenn sich Teile der gemeinsamen Datenstruktur unabhängig voneinander ändern lassen. Dieser Ansatz führt zu weniger Behinderungen, ist aber auch fehleranfälliger.

Klassen mit Methoden in kritischen Regionen

queens2.cpp

```
class Results {
   public:
      void add(const PositionList& result) {
         std::lock_guard<std::mutex> lock(mutex);
         results.insert(result);
      }
      void print() const {
         std::lock_guard<std::mutex> lock(mutex);
         for (const PositionList& result: results) {
            print_board(result);
         }
      }
   private:
      mutable std::mutex mutex;
      std::set<PositionList> results;
};
```

- Bei der Klasse *Results* sind alle Methoden in kritischen Regionen.
- Das ermöglicht die statische Überprüfung, dass keine konkurrierenden Zugriffe auf die Datenstruktur stattfinden und die Freigabe des Locks nirgends vergessen wird.