

C++ Workshop

12. Block, 20.07.2012

Robert Schneider | 23. Juli 2012

```
БЭИСИК  ДВК  НЦ
001
НУЖНЫ ВАМ РАСШИРЕННЫЕ ФУНКЦИИ ?1
ВЫСОКОСКОРОСТНЫЕ УСТ-СТВА?1
УСТАИ-КА ВНЕШНЕЙ ФН-ЦИИ?1
03У ?48
ЖДУ
5 LET A=1.0000001
10 LET B=A
15 FOR I=1 TO 27
20 LET A=A*A
25 LET B=B-2
30 NEXT I
35 PRINT A,B,"WWW.LENINGRAD.SU/MUSEUM"
RUN
568044 1202420 WWW.LENINGRAD.SU/MUSEUM
ОСТ СТРОКЕ 35
ЖДУ
```

Quelle: Wikimedia Commons
CC-BY-SA Sergei Frolov

Gliederung

- 1 exceptions, ctor und Invarianten
- 2 ownership of data
 - C++ and copying
 - ownership & smart pointer
- 3 variadic templates
- 4 auto
- 5 function pointers
 - Grundlagen, bind
 - member functions
 - polymorphic function wrappers
- 6 lambda
- 7 threading: Einblick
 - Konzeptuelles
 - threads in sequentiellm C++11
 - Einstieg in elegantere concurrency
- 8 Ausblick

1 exceptions, ctor und Invarianten

```
1  struct Socket {
2      // no default-ctor!
3      Socket(Port p) { open(p); }
4      ~Socket()      { close(); }
5
6      void send() {
7          // invariant: socket is usable
8          /* ... */
9      }
10 };
11
12
13
14 try {
15     Socket s(80);
16     s.send();
17 }catch(SocketExcp&) { /* ... */ }
18
19
20 // cannot use 's' outside!
21 // automatic clean-up
```

```
1  struct Socket {
2      Socket()      { open(invalid); }
3      Socket(Port p) { open(p); }
4      ~Socket()     { if(invalid != port) { close(); } }
5
6      void send() {
7          if(invalid == port) { throw NotInitialised(); }
8          /* ... */
9      }
10 };
11
12
13 Socket sock; // invalid socket!
14 try {
15     sock = Socket(80); // moves!
16 } catch(std::bad_alloc&) { /* ... */
17 } catch(SocketExcp&) { /* ... */
18 }
19
20 sock.send();
21 // automatic clean-up
```

```
1  struct Socket {
2      // no default-ctor!
3      Socket(Port p) { open(p); }
4      ~Socket()      { close(); }
5
6      void send() {
7          // invariant: socket is usable
8          /* ... */
9      }
10 };
11
12
13 Socket* pSock = nullptr;
14 try {
15     pSock = new Socket(80);
16 } catch (std::bad_alloc&) { /* ... */
17 } catch (SocketExcp&) { /* ... */
18 }
19
20 pSock->send();
21 delete pSock;
```

```
1  struct Socket {
2      // no default-ctor!
3      Socket(Port p) { open(p); }
4      ~Socket()      { close(); }
5
6      void send() {
7          // invariant: socket is usable
8          /* ... */
9      }
10 };
11
12 char sockBuf[ sizeof(Socket) ];
13 Socket* pSock = nullptr;
14 try {
15     pSock = new(sockBuf) Socket(80);
16 } catch(std::bad_alloc&) { /* ... */
17 } catch(SocketExcp&) { /* ... */
18 }
19
20 pSock->send();
21 pSock->~Socket();
```

```
1  struct Socket {
2      // no default-ctor!
3      Socket(Port p) { open(p); }
4      ~Socket()      { close(); }
5
6      void send() {
7          // invariant: socket is usable
8          /* ... */
9      }
10 };
11
12 Socket createSocket(Port p) {
13     try{
14         return Socket(p);
15     }catch(SocketExcp&) { /* ... */ }
16 }
17
18
19 Socket sock = createSocket(80); // moves!
20 sock.send();
21 // automatic clean-up
```


- 2 ownership of data
 - C++ and copying
 - ownership & smart pointer

Philosophie

- Standard, 5.17/2: Zuweisung ersetzt Wert der linken Seite durch Wert der rechten Seite
- Standard, 12.8/29: Die default-Zuweisung ist eine member-weise Zuweisung (zunächst Zuweisung der base-class subobjects, dann der member)

⇒ Kopie ⇒ „copy-assignment“

Philosophie

- Standard, 5.17/2: Zuweisung ersetzt Wert der linken Seite durch Wert der rechten Seite
- Standard, 12.8/29: Die default-Zuweisung ist eine member-weise Zuweisung (zunächst Zuweisung der base-class subobjects, dann der member)

⇒ Kopie ⇒ „copy-assignment“

```
1 struct MyS {  
2     public:  
3     /* inline */ MyS& operator= (MyS const&);  
4 };  
5  
6 MyS a;  
7 MyS b;  
8 a = b;
```



Philosophie

- Standard, 12.1/9: der copy-ctor kopiert »Dinge«
- Standard, 12.8/16: Der default-copy-ctor kopiert member-weise (zunächst Kopieren der base class subobjects)

⇒ Kopie

```
1  struct MyS {  
2      public:  
3      /* inline */ MyS(MyS const&);  
4  };  
5  
6  MyS a;  
7  MyS b(a);  
8  // oder  
9  MyS b = a;
```

- Referenzen als data members
- Pointer + RAII
- const data members
- class-type data member ohne assignment-op/copy-ctor
- exception-safety

In diesen Fällen (bis auf ptr) wird auch kein default copy-ctor/assignment-op definiert! (dies schlägt fehl)

Achtung!

Der Compiler darf das Kopieren von temporaries vermeiden, selbst wenn assignment-op/copy-ctor Seiteneffekte haben!

Beispiele: function arguments, return value

```
1  MyS foo(MyS p) {  
2      MyS ret( p );  
3      return ret;  
4  }  
5  
6  MyS a;  
7  MyS b = foo(a);
```

assignment mit *strong guarantee*

```
1  struct My {
2      My& operator= (My const& rhs) {
3          Me_can_throw temp;
4          temp = rhs.member; // copy, can and may throw
5
6          std::swap(temp, this->member); // swap, no-throw!
7          return *this;
8      }
9  private:
10     Me_can_thow member;
11 };
```

pro/contra von swap gegenüber doppelter Kopie

- + meist no-throw (`noexcept`)
- + bei STL-Containern in $O(1)$
- expliziter Aufruf notwendig
- kein swap-ctor möglich

Philosophie

- Philosophie: (12.1/9) Verschieben von Inhalten
- 12.8/16, 12.8/29: default move-assignment & move-ctor verschieben member-weise

Philosophie

- Philosophie: (12.1/9) Verschieben von Inhalten
- 12.8/16, 12.8/29: default move-assignment & move-ctor verschieben member-weise

```
1  struct MyS {  
2      public:  
3      /* inline */ MyS(MyS&&);  
4      /* inline */ MyS& operator= (MyS&&);  
5  };  
6  
7  MyS b = MyS();  
8  b = MyS();  
9  
10 MyS a;  
11 b = std::move(a);
```

```
1 vector < int > a = {0, 1, 2, 3};
2 vector < int > b = {0, 1, 2, 3, 4};
3 vector < int > b_pre;
4
5 b_pre = b;
6 //– b_pre == {0, 1, 2, 3, 4}
7 a = move(b);
8 //– a == b_pre
9 //– b == >undefined<
```

```
1 vector < int > a = {0, 1, 2, 3};
2 vector < int > b = {0, 1, 2, 3, 4};
3 vector < int > b_pre;
4
5 b_pre = b;
6 //– b_pre == {0, 1, 2, 3, 4}
7 a = move(b);
8 //– a == b_pre
9 //– b == >undefined<
```

Kopieren: Elementweises Kopieren ($\implies O(n)$)

Verschieben: Pointer tauschen ($\implies O(1)$)

- haben ein part-whole-ownership wie Elemente \leftrightarrow Menge
- hauptsächlich: default-ctor, kopieren
bspw.: `resize + []` vs. `reserve + push_back + lvalue ref`
- Vermeiden von Kopien: move semantics oder Pointer
bspw: `emplace_back`
- Problem bei raw ptrs: bei Zerstörung des Pointers wird das Ziel nicht zerstört!
 - Daher: Ungeeignet für das ownership-Modell der STL-Container (aber möglich)

Problem 1

```
1  int* p = new int[42];
2
3  if( cond ) {
4      delete [] p;
5      return;
6  }
7
8  try{ something() }
9  catch (...) {
10     delete [] p;
11     throw;
12 }
13
14 try{ something() }
15 catch(MyExpT&) { delete [] p; }
16 catch(MyExpT1&) { delete [] p; }
17 catch (...) { delete [] p; }
```

Problem 2

```
1  struct Louvre {
2      MonaLisa* show();
3      ~Louvre() {
4          delete pML;
5      }
6  private:
7      MonaLisa* pML;
8  };
9
10 struct Perugia {
11     void thief(MonaLisa* p) {
12         pML = p;
13     }
14     ~Perugia() {
15         delete pML;
16     }
17 private:
18     MonaLisa* pML;
19 };
```

- wrapper-Objekte für (raw) Pointer
- RAII-Konstrukte (beim Zerstören wird automatisch aufgeräumt)
- explizites ownership-Modell
- exception-„safe“
- alter Repräsentant: `auto_ptr` – nicht verwenden (deprecated)!
- Verwendung mit syntax analog Pointern (* dereferenzieren, `->` member access; via operator overloading)

- ein Besitzer
- move-semantics zum Besitzerwechsel
- thread-safe movement
- einzelne Instanz *nicht* thread-safe

```
1  #include <memory>
2
3  unique_ptr < int > myInt = new int;
4  unique_ptr < int > otherInt = new int;
5
6  otherInt = myInt; // copying is forbidden! (makes no sense)
7
8  *otherInt = 5;
9  cout << *otherInt;
10
11 vector < unique_ptr < LargeData > > myVec; // ok
```

Mit move-semantic kann das Besitzer-Objekt geändert werden:

```
1  struct Louvre {
2      MonaLisa& show();
3      unique_ptr < MonaLisa > breakIn();
4
5      ~Louvre() {}
6  private:
7      unique_ptr < MonaLisa > pML;
8  };
9
10 struct Perugia {
11     void thieve(unique_ptr < MonaLisa > p) {
12         pML = move(p);
13     }
14     ~Perugia() {}
15 private:
16     unique_ptr < MonaLisa > pML;
17 };
18
19 Louvre myMuseum;
20 Perugia myThief;
21
22 unique_ptr < MonaLisa > pML = move( myMuseum.breakIn() );
23 myThief.thieve( move(pML) );
```

shared_ptr

- viele, gleichrangige Besitzer (wie Taxi-Gemeinschaft)
- der letzte macht das Licht aus (ruft `delete` auf)
- thread-safe Besitzer-Operationen (z.B. Erzeugen von Mitbesitzer-Instanzen)
- einzelne Instanz *nicht* thread-safe

```

1  #include <memory>
2
3  shared_ptr < int > myInt = new int;
4  {
5      shared_ptr < int > otherInt = new int;
6
7      otherInt = myInt; // ok, old int of otherInt is destroyed
8
9      *otherInt = 5;
10     cout << *otherInt;
11 } // - otherInt is destroyed, but its content lives on
12
13 vector < shared_ptr < LargeData > > myVec; // ok
  
```

shared_ptr & copying

Mit dem shared_ptr lässt sich oft Kopieren vermeiden:
 Alle, die Zugriff auf die Daten brauchen, werden einfach Mitbesitzer
 (indem sie eine shared_ptr-Instanz speichern).

```

1  struct A { shared_ptr < foo > m; };
2  struct B { shared_ptr < foo > m; };
3
4  void f(shared_ptr < foo > p) {
5      /* do something */
6  }
7
8  shared_ptr < foo > p = new foo;
9  A a = {p};
10 B b = {p};
11 f(p);
12
13 b.m.reset();
14 p.reset();
15 a.m.reset();
  
```

Der kleine Kommunismus in C++

Niemals raw pointer und smart_ptr mischen!

```
1 shared_ptr < foo > sp = p;  
2 delete p;
```

Der kleine Kommunismus in C++

Niemals raw pointer und smart_ptr mischen!

```

1  shared_ptr < foo > sp = p;
2  delete p;
  
```

Defensives Programmieren mit shared_ptr: Factory

```

1  class My {
2  private:
3      My();
4
5  public:
6      typedef shared_ptr < My > Ptr;
7
8      static Ptr create() {
9          return new My;
10     }
11 };
12
13 My* p = new My; // forbidden
14 My::Ptr p = My::create(); // OK
  
```

Situation:

- Ein Besitzer-Objekt `r` hält ein Besitztum `d` mit einem `shared_ptr`.
- `d` benötigt einen Zugang zu `r`.

Situation:

- Ein Besitzer-Objekt `r` hält ein Besitztum `d` mit einem `shared_ptr`.
- `d` benötigt einen Zugang zu `r`.

Möglichkeit 1: `d` hält einen `shared_ptr` auf `r`

Situation:

- Ein Besitzer-Objekt `r` hält ein Besitztum `d` mit einem `shared_ptr`.
- `d` benötigt einen Zugang zu `r`.

Möglichkeit 1: `d` hält einen `shared_ptr` auf `r`

Möglichkeit 2: `d` hält einen `raw ptr` auf `r`

Situation:

- Ein Besitzer-Objekt `r` hält ein Besitztum `d` mit einem `shared_ptr`.
- `d` benötigt einen Zugang zu `r`.

Möglichkeit 1: `d` hält einen `shared_ptr` auf `r`

Möglichkeit 2: `d` hält einen `raw ptr` auf `r`

Lösung: `weak_ptr`

weak_ptr

- hat selbst keinen Besitz am ptr
- kein Dereferenzieren usw.
- kann in einen `shared_ptr` verwandelt werden
- kann auf Zerstörtheit geprüft werden

weak_ptr: Beispiel

```
1  struct Owned;  
2  struct Owner {  
3      shared_ptr < Owned > d;  
4      ~Owner();  
5      void change();  
6  };  
7  
8  struct Owned {  
9      weak_ptr < Owner > r;  
10     void changeOwner() {  
11         shared_ptr < Owner > tempAccess = {r};  
12         tempAccess->change();  
13     }  
14 };  
15  
16 // dtor must be defined AFTER defining the class Owned  
17 // as Owned is the template parameter of the data member d  
18 Owner::~~Owner() {}
```

3 variadic templates

Beispiel printf (von wikipedia)

```
1 void printf(const char *s)
2 {
3     while (*s) {
4         if (*s == '%' && *(++s) != '%')
5             throw std::runtime_error("missing_arguments");
6         std::cout << *s++;
7     }
8 }
9 template<typename T, typename... Args>
10 void printf(const char *s, T value, Args... args)
11 {
12     while (*s) {
13         if (*s == '%' && *(++s) != '%') {
14             std::cout << value;
15             ++s;
16             printf(s, args...);
17             return;
18         }
19         std::cout << *s++;
20     }
21     throw std::logic_error("more_arguments_than_%");
22 }
23 printf("uint:%u, float:%f\n", 42.235123, "hallo");
```

Beispiel prntline

```
1 void prntline ()
2 {
3     std::cout << std::endl;
4 }
5 template<typename T, typename... Args>
6 void prntline(T value, Args... args)
7 {
8     std::cout << T << std::endl;
9 }
10 prntline("uint:\u2013", 42.235123, "\u2013float:\u2013", "hallo");
11 std::cout << "uint:\u2013" << 42.235123 << "\u2013float:\u2013" << "hallo" << std::endl;
```

4 auto

Beispiel: template template template member type

```
1  std::vector<std::unique_ptr<myTemplate<int, std::string, myClass3>>> >::const_iterator
2      it = somecoolFunction();
3
4  typedef std::vector<std::unique_ptr<myTemplate<int, std::string, myClass3>>> >::const_iterator
5      my_it_type;
6
7  my_it_type it2 = somecoolFunction();

1  // it3 is an iterator over mtISC3 pointers
2  auto it3 = somecoolFunction();
```

Da auto für fremde Leser (oder einen selbst in 2 Monaten) vielleicht für Verwirrung sorgt, sollte man immer noch einen kurzen kommentar hinschreiben. Dieser sollte im Zusammenhang für menschliche Leser leicht verständlich sein.

5

function pointers

- Grundlagen, bind
- member functions
- polymorphic function wrappers

Allgemeines

- ein „normaler“ Pointer enthält die Adresse eines »Dinges«
- ein *function pointer* enthalten die Adresse einer Funktion
- Nicht alle Funktionen haben eine Adresse! (z.B. ctor)
- Signatur der Funktion gibt Typen des function ptrs

Syntax grausam! Nicht direkt verwenden!

function objects

- function objects können der (syntaktisch) wie eine Funktion aufgerufen werden (operator overloading)
- enthalten entweder selbst den Funktionsaufruf oder sind wrapper für function ptrs
- für optimale Performance kein spezifizierter Typ / template
- Erzeugung und Speicherung am besten mittels Automatisierungen

Beispiel: function objects

```
1  int doCalc1(int , double , bool);  
2  int doCalc2(int , double , bool);  
3  int doCalc3 ();  
4  
5  
6  #include <functional>  
7  
8  auto myFunc = std::bind( &doCalc1 ); // OK  
9      myFunc = std::bind( &doCalc2 ); // OK  
10     myFunc = std::bind( myFunc ); // OK  
11     myFunc = std::bind( &doCalc3 ); // error!  
12  
13  int result = myFunc(4, 2.0, true);
```

std::bind & ownership

- bind kopiert üblicherweise
- moved, wenn kopieren verboten
- alternativ: std::ref oder (smart) ptr

- bind kopiert üblicherweise
- moved, wenn kopieren verboten
- alternativ: std::ref oder (smart) ptr

```
1 void thiefe( unique_ptr < MonaLisa > );  
2 bool drawAt( MonaLisa& );  
3  
4 MonaLisa ml;  
5 std::bind( &drawAt, std::ref(ml) );  
6  
7 unique_ptr < MonaLisa > pML = new MonaLisa;  
8 std::bind( &thieve, pML );
```

```
1  int doCalc1(int, bool, double);  
2  
3  #include <functional>  
4  
5  auto myFunc = std::bind( &doCalc1, 42, _1, _2 );  
6  //– myFunc has signature 'int(bool, double)'  
7  
8  int result = myFunc(true, 4.2);
```

```
1  int doCalc1(int, bool, double);
2
3  #include <functional>
4
5  auto myFunc = std::bind( &doCalc1, 42, _1, _2 );
6  //– myFunc has signature 'int(bool, double)'
7
8  int result = myFunc(true, 4.2);
9
10 auto myFunc2 = std::bind( &doCalc1, _1, false, 21.42);
11 //– myFunc has signature 'int(int)'  
result = myFunc2(0);
```


- Bisher: »Ding« pointer und function pointer
- Es gibt aber auch pointer-to-member (extrem selten!) und pointer-to-member-function ...
- Veranschaulichung: Auf welcher Instanz soll die member function aufgerufen werden, wenn ich nur ihre Adresse habe?
- Veranschaulichung: Wer setzt den `this`-Pointer?

- Bisher: »Ding« pointer und function pointer
- Es gibt aber auch pointer-to-member (extrem selten!) und pointer-to-member-function ...
- Veranschaulichung: Auf welcher Instanz soll die member function aufgerufen werden, wenn ich nur ihre Adresse habe?
- Veranschaulichung: Wer setzt den `this`-Pointer?

Normale Syntax noch hässlicher als die von function pointers!

Daher wieder mit function objects.

member functions: Beispiel

```
1  struct Calculator {
2      int add(int);
3      int sub(int);
4      int res() const;
5  };
6  struct Calc2 {
7      int add(int);
8  };
9
10 auto myFunc = std::bind( &Calculator::add );
11 //- myFunc has signature akin to 'int (Calculator&, int)'
12     myFunc = std::bind( &Calculator::sub );
13     myFunc = std::bind( &Calculator::res );      // error!
14     myFunc = std::bind( &Calc2::add );          // error!
15
16 shared_ptr < Calculator > spC = new Calculator;
17 Calculator& rC = *spC;
18 Calculator* pC = spC.get();
19
20 int result = myFunc(spC, 5);
21     result = myFunc(rC, 5);
22     result = myFunc(pC, 5);
```

Man kann die zu verwendene Instanz an das function object binden!

```
1  Calculator myCalc;
2  auto myFunc = std::bind( &Calculator::add, myCalc );
3  //– myFunc has signature 'int(int)'
4      myFunc = std::bind( &Calculator::add, std::ref(myCalc) );
5      myFunc = std::bind( &Calculator::add, &myCalc );
6
7  shared_ptr < Calculator > pMyCalc = new Calculator;
8      myFunc = std::bind( &Calculator::add, pMyCalc );
9
10 int result = myFunc(42);
```

polymorphic function wrappers

Wie bei pointern + Polymorphismus kann in einem function object alles mögliche referenziert werden (z.B. member function vs. normale function).

⇒ spezielles wrapper-Objekt, das eben alles mögliche Aufnahmen & speichern kann (auch gebundene Parameter usw.)

polymorphic function wrappers

Wie bei pointern + Polymorphismus kann in einem function object alles mögliche referenziert werden (z.B. member function vs. normale function).

⇒ spezielles wrapper-Objekt, das eben alles mögliche Aufnahmen & speichern kann (auch gebundene Parameter usw.)

```
1  #include <functional>
2
3  struct EventProvider {
4      std::function < int(bool) > observer;
5      void pollEvent() {
6          /* ... */
7          int result = observer(true);
8          /* ... */
9      }
10 };
11
12 int onEvent(bool);
13
14 EventProvider ep;
15 ep.observer = std::bind( &onEvent );
16 /* ... */
17 ep.pollEvent();
```

6 lambda

```
1  #include <functional>
2
3  void doRaycast(int x, int y, int dx, int dy, std::function<bool(int, int, int)> cond)
4  {
5      if (cond(10, 20, 0)) return;
6      if (cond(99, 42, 0)) return;
7      if (cond(0, 0, 1)) return;
8  }
9
10 int main()
11 {
12     int prev_x, prev_y;
13     auto cond = [&prev_x, &prev_y](int x, int y, int pixelval)->bool
14     {
15         // stop if solid
16         if (pixelval > 0) return true;
17         prev_x = x;
18         prev_y = y;
19         // continue raycasting
20         return false;
21     };
22     doRaycast(10, 10, 3, 5, cond);
23     return 0;
24 }
```


7 threading: Einblick

- Konzeptuelles
- threads in sequentiellm C++11
- Einstieg in elegantere concurrency

Hardware

Historische Architektur: Berechnungen auf CPU mit einem oder wenigen Kernen

⇒ sequentielles Programmieren, sequentielle Prozeduren

langsame Steigerung der Zahl Kerne

⇒ unabhängige, parallel ausgeführte Programmsequenzen

Hardware

Historische Architektur: Berechnungen auf CPU mit einem oder wenigen Kernen

⇒ sequentielles Programmieren, sequentielle Prozeduren

langsame Steigerung der Zahl Kerne

⇒ unabhängige, parallel ausgeführte Programmsequenzen

Software

Mehrere Aufgaben (Tasks) gleichzeitig

⇒ multi-tasking

unabhängige Tasks

⇒ Support durch OS

Sequentielles Programmieren

- Programm besteht aus Reihe von Befehlssequenzen
- Befehle einer Sequenz werden strikt nacheinander ausgeführt

Sequentielles Programmieren

- Programm besteht aus Reihe von Befehlssequenzen
- Befehle einer Sequenz werden strikt nacheinander ausgeführt

Thread

- Ein Thread führt vorgegebene Abfolge von Sequenzen aus
⇒ Befehlsfluss
- Mehrere Threads bearbeiten unabhängige Abfolgen ⇒ mehrere „parallele“ Flüsse
- Threads teilen sich bestimmte Dinge ...
- ... haben aber auch individuelle Zuordnungen
- bisher: ein Thread, der die `main` ausführt
- jetzt: *zusätzlich* beliebige Zahl Threads

threads in sequentiellem Programmieren (1)

thread scheduling

- 1 Ein Thread durchläuft das Programm (Ausführung).
- 2 Der Programmablauf wird zu einem unbekanntem Zeitpunkt unterbrochen.
- 3 Andere Threads werden fortgesetzt.
- 4 Irgendwann wird der eigene Thread fortgesetzt.

threads in sequentiellem Programmieren (1)

thread scheduling

- 1 Ein Thread durchläuft das Programm (Ausführung).
 - 2 Der Programmablauf wird zu einem unbekanntem Zeitpunkt unterbrochen.
 - 3 Andere Threads werden fortgesetzt.
 - 4 Irgendwann wird der eigene Thread fortgesetzt.
- Keine Garantie, wann unterbrochen wird.
 - Keine Garantie, wer wann ausgeführt/fortgesetzt wird.
 - Mehrere Threads: weiß nicht ohne Stoppen, wer wann wo ist. (Unschärferelation!)
 - Fortsetzen „als wäre nichts geschehen“ (wie standby)

threads in sequentiellem Programmieren (2)

concurrent threads

Bei mehreren Kernen/CPUs/. . . : gleichzeitiges Ausführen mehrerer Threads möglich. Dieselben (nicht-)Garantien wie bei thread scheduling!

Resultat: Kenne den Zustand eines anderen Threads *nur bei Synchronisierung!*

Analogie: Unbestimmtheit eines Teilchenzustandes bis zur Messung

threads in sequentiellem Programmieren (2)

concurrent threads

Bei mehreren Kernen/CPU's/. . . : gleichzeitiges Ausführen mehrerer Threads möglich. Dieselben (nicht-)Garantien wie bei thread scheduling!

Resultat: Kenne den Zustand eines anderen Threads *nur bei Synchronisierung!*

Analogie: Unbestimmtheit eines Teilchenzustandes bis zur Messung

Kombiniert

- thread != Kern (außer mit Aufwand)
- OS verwaltet Zuweisung thread ↔ Kern
- Viele Threads möglich, u.U. sinnvoll wenn diese oft schlafen

Grundlegende Probleme

Bei sequentiellem Programm-Design!

Kein Problem: Mehrere unabhängige Sequenzen

Probleme bei *Interaktion* und *Überlappungen*

- Schreibzugriff auf Daten
- Synchronisierung (z.B. Warten)
- Übergeben von Daten
- Exceptions (!)

Zusätzlicher Aspekt (neben const correctness, exception safety usw.):

threading safety

Heißt: Was darf ich von welchem thread aus mit einer Funktion / einer Instanz tun?

Grundlegende Probleme

Bei sequentiell Programm-Design!

Kein Problem: Mehrere unabhängige Sequenzen

Probleme bei *Interaktion* und *Überlappungen*

- Schreibzugriff auf Daten
- Synchronisierung (z.B. Warten)
- Übergeben von Daten
- Exceptions (!)

Zusätzlicher Aspekt (neben const correctness, exception safety usw.):

threading safety

Heißt: Was darf ich von welchem thread aus mit einer Funktion / einer Instanz tun?

Viele der Probleme können durch ein besseres Design schon vermieden werden, z.B. message-based, event-driven, callbacks usw.

Zugriff auf ein »Ding« über seinen Namen

- Jeder thread hat seinen eigenen Stack
- automatic storage: unabhängig
- static storage: geteilt
- dynamic storage: n/A (kein Name!)
- thread-local storage: duration wie bei static, aber Daten individuell pro thread

Zugriff über Pointer immer möglich!

Zugriff auf ein »Ding« über seinen Namen

- Jeder thread hat seinen eigenen Stack
- automatic storage: unabhängig
- static storage: geteilt
- dynamic storage: n/A (kein Name!)
- thread-local storage: duration wie bei static, aber Daten individuell pro thread

Zugriff über Pointer immer möglich!

```
1  int stS = 42;
2  thread_local bool tIS;
3
4  void foo(int* ptr) {
5      int auS = 5;
6      stS = 15;
7      tIS = 10;
8
9      *ptr = 21;
10     double* dyS2 = new double(3.0);
11 }
```

threads in sequentiellem C++11

- Vor C++11: diverse threading-Bibliotheken, z.B. pthreads oder boost
- Mit C++11: größte *gemeinsame* Basis im Standard selbst (reicht für alles Grundlegende)
- externe Bibliotheken nicht obsolet für Spezialanwendungen ...
- ... aber besser zu vermeiden

- Vor C++11: diverse threading-Bibliotheken, z.B. pthreads oder boost
- Mit C++11: größte *gemeinsame* Basis im Standard selbst (reicht für alles Grundlegende)
- externe Bibliotheken nicht obsolet für Spezialanwendungen ...
- ... aber besser zu vermeiden

std::thread

Die Thread-Klasse in C++11

- RAII: Erzeugung einer Instanz = Starten eines Threads (außer default-ctor)
- Übergabe einer Funktion wie bei `std::bind`
- member functions `join`, `joinable`, `detach`
- Achtung: dtor terminiert das Programm, wenn der thread noch läuft!
- static member function: `hardware_concurrency`

Beispiel: `std::thread`

```
1 void calc(int,  
2         int& result);  
3  
4  
5  
6 #include <thread>  
7  
8 int result = 0;  
9 std::thread myThread =  
10    {&calc, 5,  
11     std::ref(result)};  
12  
13 /* do something else */  
14  
15 myThread.join();  
16  
17 std::cout << result;
```


Beispiel: `std::thread`

```
1 void calc(int,
2           int& result);
3
4
5
6 #include <thread>
7
8 int result = 0;
9 std::thread myThread =
10    {&calc, 5,
11     std::ref(result)};
12
13 /* do something else */
14
15 myThread.join();
16
17 std::cout << result;
```

```
1 struct Calculator {
2     void calc(int);
3     int result;
4 };
5
6
7 #include <thread>
8
9 Calculator myCalc;
10 std::thread myThread =
11    {&MyCalculator::calc,
12     std::ref(myCalc), 5};
13
14 /* do something else */
15
16 myThread.join();
17
18 std::cout << myCalc.result;
```

- Reines Lesen von (konstanten) Daten ist meist kein Problem.
- Gibt es einen Schreibenden \implies Probleme
- Einfachste Lösung: locking
- Hinweis: Auf Puffern durch Compiler achten (`volatile`)

- Reines Lesen von (konstanten) Daten ist meist kein Problem.
- Gibt es einen Schreibenden \implies Probleme
- Einfachste Lösung: locking
- Hinweis: Auf Puffern durch Compiler achten (`volatile`)

Mutex: Konzeptuell

- Einfachste Form des lockings
- Amortisiert so ziemlich das langsamste locking (da sehr allgemein)
- Mutex = Mutual exclusion
- Es kann zu jedem Zeitpunkt maximal ein Thread das Mutex „besitzen“ (sperrern)
- Andere Threads können auf das Mutex warten und einer der wartenden erhält dann den Besitz

std::mutex

Wichtigste member functions:

- `lock` – lässt den aktuellen Thread auf den Besitz dieses Mutexes warten
- `try_lock` – versucht, den Besitz sofort zu erhalten (falls kein aktueller Besitzer)
- `unlock`

std::mutex

Wichtigste member functions:

- `lock` – lässt den aktuellen Thread auf den Besitz dieses Mutexes warten
- `try_lock` – versucht, den Besitz sofort zu erhalten (falls kein aktueller Besitzer)
- `unlock`

Häufiges Problem: `unlock` vergessen

std locks

- RAII: Wie smart ptr, nur eben für locking
- Im ctor `lock`, im dtor `unlock`
- Wichtigster Vertreter: `std::unique_lock`

Beispiel: mutex + unique_lock

```
1  #include <mutex>
2  shared_ptr < LargeData >  pDat;
3  shared_ptr < std::mutex > pDat_mutex;
4
5  void calc(int p)
6  {
7      shared_ptr < LargeData >  pDat_my      = pDat;
8      shared_ptr < std::mutex > pDat_mutex_my = pDat_mutex;
9
10     /* ... */
11
12     { unique_lock datLock = {*pDat_mutex_my};
13       pDat_my->change();
14     }
15
16     /* ... */
17 }
```

dead-lock

Wechselseitiges Blockieren \implies unendliches Warten

```
1  shared_ptr < AdditionalData >  pAddDat ;
2  shared_ptr < std::mutex >      pAddDat_mutex ;
3
4  void calc0(int p) {
5      shared_ptr < AdditionalData > pAddDat_my      = pDat ;
6      shared_ptr < std::mutex >      pAddDat_mutex_my = ppAddDat_mutex ;
7      /* ... */
8      {unique_lock  addDatLock = {*pAddDat_mutex_my};
9        unique_lock  datLock  = {*pDat_mutex_my};
10         pDat_my->change();
11     }
12 }
13 void calc1(int p) {
14     shared_ptr < AdditionalData > pAddDat_my      = pDat ;
15     shared_ptr < std::mutex >      pAddDat_mutex_my = ppAddDat_mutex ;
16     /* ... */
17     {unique_lock  datLock  = {*pDat_mutex_my};
18       unique_lock addDatLock = {*pAddDat_mutex_my};
19         pMyDat_my->change();
20     }
21 }
```

Einstieg in elegantere concurrency:

async

Ein Traum wie aus einer Script-Sprache:

std::async

- `async` ist eine Funktion, Parameter wie beim ctor von `thread`
- führ die angegebene Funktion aus, wobei die Implementierung/OS entscheidet, ob *sofort* in einem neuen Thread oder *später* (lazy evaluation)
- kann gezwungen werden per Parameter, einen neuen Thread zu starten oder auch nicht (lazy evaluation)
- Rückgabety: ein `future`

Einstieg in elegantere concurrency: promises & futures

Hier nur als Skizze!

promises & futures

aktueller Thread = A , (neuer) Thread für Aufgabe = N

- A erwartet ein Resultat in einer gewissen Form (Typ): future
- N verspricht, das Resultat zu geben, und zwar an einen bestimmten „Ort“: promise

Einstieg in elegantere concurrency: promises & futures

Hier nur als Skizze!

promises & futures

aktueller Thread = A , (neuer) Thread für Aufgabe = N

- A erwartet ein Resultat in einer gewissen Form (Typ): `future`
- N verspricht, das Resultat zu geben, und zwar an einen bestimmten „Ort“: `promise`
- A kann mittels der `future` auf N warten und dann den Rückgabewert auslesen
- ODER, bei lazy evaluation: A berechnet selbst das Ergebnis, aber erst wenn es angefordert wird

Einstieg in elegantere concurrency: promises & futures

Hier nur als Skizze!

promises & futures

aktueller Thread = A , (neuer) Thread für Aufgabe = N

- A erwartet ein Resultat in einer gewissen Form (Typ): `future`
- N verspricht, das Resultat zu geben, und zwar an einen bestimmten „Ort“: `promise`
- A kann mittels der `future` auf N warten und dann den Rückgabewert auslesen
- ODER, bei lazy evaluation: A berechnet selbst das Ergebnis, aber erst wenn es angefordert wird
- Beim Auslesen dieses Rückgabewertes kann eine Exception von N nach A propagiert werden!

Beispiel: async & future

```
1  int calcHard(int i, double d)
2  {
3      int result;
4      /* work hard, maybe throw up */
5      return result;
6  }
7
8
9  #include <future>
10
11 future < int > future_result = std::async(&calcHard, 42, 21.0);
12 /* do something or not */
13 int result = future_result.get();
```

8 Ausblick

?

- kleinere Aufgaben / Projekte
- mehre Beteiligung aller Teilnehmer
- Code-Revision & Analyse
- größere Baustellen (Qt, cmake)
- patterns in action

- mailing list `cpp-workshop@lists.kit.edu` steht auch für Fragen zu C++ zur Verfügung
- euirc `#kit-cpp-workshop`

Viel Ruhe und Erfolg in den Semesterferien!
Vergiss Dein Handtuch nicht.