Advanced Developers Conference
Development for Professionals!

Embedded Programmierung mit C++

Rainer Grimm

Softwarearchitekt

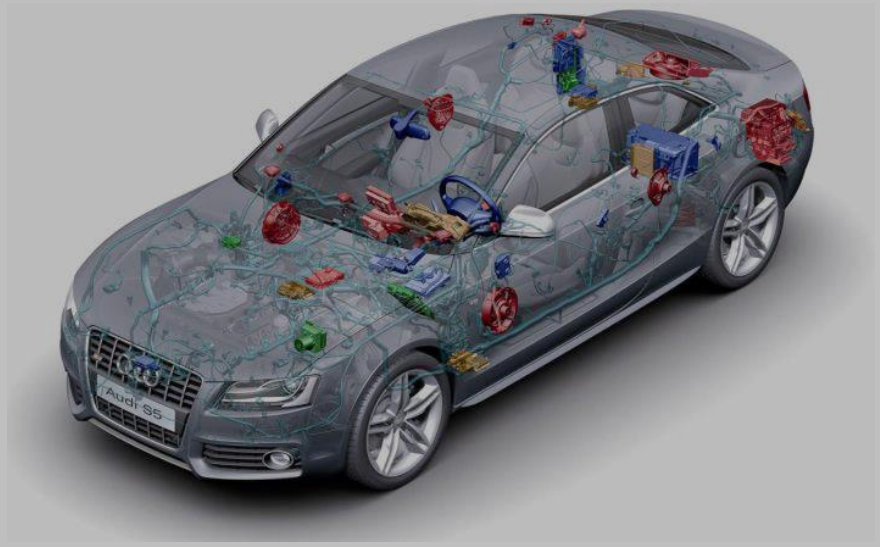
Ziele für C++11

Bjarne Stroustrup

„... make C++ even better for embedded system programming ... „

Überblick

- Charakteristiken
  - Sicherheitskritische Systeme
  - Eingeschränkte Ressourcen
  - Lange Lebenszeit
  - Viele Kerne
- Letzte Gedanken
  - Mythen
  - Fakten



Sicherheitskritische Systeme

{}- Initialisierung verhindert Verengung.

3.14159 ➡ ~~3.14159~~

```
double dou= 3.14159;
int a= dou;
int b(dou);

int c= {dou};        // ERROR
int d{dou};          // ERROR

int8_t f= {2011};    // ERROR
int8_t g= {14};
```

Sicherheitskritische Systeme

Zusicherungen mit Type-Traits und `static_assert` zur Übersetzungszeit

- Type-Traits
  - Typ-Informationen
  - Typ-Vergleiche
  - Typ-Modifikationen
- static_assert
  - Validieren Ausdrücke

➡ Kein Einfluss auf die Laufzeit des Programmes

Sicherheitskritische Systeme

```
template <typename S, typename D>
void smallerAs(S s,D d){
  static_assert(sizeof(S) <= sizeof(D),"S is too big");
}

smallerAs(1.0,1.0L);
smallerAs(1.0L,1.0);   //     with S= long double; D= double
                       //     S is too big

template <typename T>T fac(T a){
  static_assert(std::is_integral<T>::value,"T not integral");
}
fac(10);
fac(10.1);             //   with T= double; T not integral
```

Sicherheitskritische Systeme

Benutzerdefinierte Literale

- Syntax: <built_in Literal>+ _ + <Suffix>
  - Natürliche Zahlen: 101010_b
  - Fließkomma Zahlen: 123.45_km
  - String Literale: “hello”_i18n
  - Zeichen Literale: ‘1’_character

Sicherheitskritische Systeme

```
int main(){

  cout << "1.0_km: " << 1.0_km << endl;          //  1000 m
  cout << "1.0_m: " << 1.0_m << endl;            //  1 m
  cout << "1.0_dm: " << 1.0_dm << endl;          //  0.1 m
  cout << "1.0_cm: " << 1.0_cm << endl;          //  0.01 m

  cout << 1.0_km + 2.0_dm + 3.0_dm + 4.0_cm;    //  1000.54 m

  MyDis myD= 10345.5_dm + 123.45_km - 1200.0_m + 150000.0_cm;
  cout << myD;   //  124785 m

}
```

Sicherheitskritische Systeme

```
namespace Unit{
  MyDis operator "" _km(long double k){
    return MyDis(1000*k);
  }
  MyDis operator "" _m(long double m){
    return MyDis(m);
  }
  MyDis operator "" _dm(long double d){
    return MyDis(d/10);
  }
  MyDis operator "" _cm(long double c){
    return MyDis(c/100);
  }
}
```

Sicherheitskritische Systeme

```
class MyDis{
  private:
    long double m;
  public:
    MyDis(long double i):m(i){}
    friend MyDis operator +(const MyDis& a, const MyDis& b){
      return MyDis(a.m + b.m);
    }
    friend MyDis operator -(const MyDis& a, const MyDis& b){
      return MyDis(a.m - b.m);
    }
    friend ostream& operator<<(ostream &out, const MyDis& myD){
      out << myDis.m << " m";
      return out;
  }
};
```

Sicherheitskritische Systeme

C++14 bringt einen Satz an built_in Literalen mit.

Typ	Präfix/Suffix	Beispiel
bool	0b	0b10
std::string	s	"HELLO"s
complex<double>	i	5i
complex<long double>	il	5il
complex<float>	if	5if
std::chrono::hours	h	5h
std::chrono::seconds	s	5s
std::chrono::milliseconds	ms	5ms
std::chrono::microseconds	us	5us
std::chrono::nanoseconds	ns	5ns

Sicherheitskritische Systeme

Wie lange ist ein Schultag?

```
auto hour= 45min;
auto sBreak= 300s;
auto lBreak= 0.25h;
auto way= 15min;
auto work= 2h;
auto dayInSec= 2*way + 6*hour + 4*sBreak + lBreak + work;

cout << "Day in seconds: " << dayInSec.count();       // 27300
duration<double,ratio<3600>> dayInHours = dayInSec;
duration<double,ratio<60>> dayInMinutes = dayInSec;
duration<double,ratio<1,1000>> dayInMilli= dayInSec;
cout << "Day in hours: " << dayInHours.count();      // 7.58333
cout << "Day in minutes: " << dayInMinutes.count(); // 455
cout << "Day in millisec: " << dayInMilli.count();  // 2.73e+07
```

Eingeschränkte Ressourcen

Konstante Ausdrücke (`constexpr`) können zur Übersetzungszeit ausgewertet werden.

➡ Sie werden im ROM gespeichert.

- 3 Formen
  - Variablen
  - Funktionen
  - Benutzerdefinierte Typen

EPROM

Flash-EEPROM

Eingeschränkte Ressourcen

```
constexpr int myConstExpr= 2;
constexpr int square(int i){ return i*i; }

struct MyInt{
  int myInt;
  constexpr MyInt(int i):myInt(i){}
  constexpr int multiplyBy(int i){ return i*myInt; }
};

constexpr MyInt myInt(5);
constexpr int res= myInt.multiplyBy(myConstExpr);

static_assert(myInt.multiplyBy(2) == 10,"ERROR");
static_assert(res == 10,"ERROR");
cout << myInt.multiplyBy(square(5)) << endl;     // 125
```

Eingeschränkte Ressourcen

Konstante Ausdrücke werden in C++14 deutlich erweitert.

- Variablen sind nicht implizit `const`.
- Funktionen können aus mehreren Anweisungen bestehen.

- gcd (Scott Meyers)

```
constexpr auto gcd(int a, int b){
  while (b != 0){
    auto t= b;
      b= a % b;
      a= t;
  }
  return a;
}
```

Eingeschränkte Ressourcen

Move Semantik: Billiges Verschieben statt teurem Kopieren.

- Vorteile
  - Geschwindigkeit
  - Keine Speicher Allokation und Deallokation
    ➡ Vorhersagbarkeit
  - Unterstützung von nur "verschiebbaren" Typen
    - unique_ptr, Dateien, Locks und Tasks

Eingeschränkte Ressourcen

```
std::vector<int> a, b;
swap(a,b);
```

```
template <typename T>
void swap(T& a, T& b){
  T tmp(a);
  a= b;
  b= tmp;
}
```

`T tmp(a);`

- Allokiert `tmp` und jedes Element von `tmp`.
- Kopiert jedes Element von `a` nach `tmp`.
- Deallokiert `tmp` und jedes Element von `tmp`.

```
template <typename T>
void swap(T& a, T& b){
  T tmp(std::move(a));
  a= std::move(b);
  b= std::move(tmp);
}
```

`T tmp(std::move(a));`

- Verbiegt den Zeiger von `tmp` auf den von `a`.

Eingeschränkte Ressourcen

Perfect Forwarding: Unveränderte Übergabe von Argumenten.

Anwendungsfälle:

- Fabrikfunktionen
- Konstruktoren

- Verketten von Verschiebesemantik
- Übergabe von nur verschiebbaren Typen

Eingeschränkte Ressourcen

```
template <typename T, typename T1>
T createT(T1&& t1){
  return T(std::forward<T1>(t1));
}
auto lValue= createT<int>(2011);
auto i= createT<int>(lValue);

struct NeedOnlyMove{
  NeedOnlyMove(OnlyMove){};
};
struct OnlyMove{
  OnlyMove()= default;
  OnlyMove(const OnlyMove&)= delete;
  OnlyMove& operator= (const OnlyMove&)= delete;
  OnlyMove(OnlyMove&&)= default;
  OnlyMove& operator= (OnlyMove&&)= default;
};
NeedOnlyMove onlyMove2= createT<NeedOnlyMove>(OnlyMove());
```

Eingeschränkte Ressourcen

Volle Kontrolle über den Speicher mit placement new.

- Die Erzeugen eines Objektes besteht aus zwei Schritten.
  1. Allokierung des Speicher ⇨ operator new
  2. Initialisierung des Objektes ⇨ Konstruktor

  ⇨ Durch placement new kann die Allokierung des Speichers für Objekte einer Klasse oder global überladen werden.

- Anwendungsfälle
  - Objekte in eigenen Speicherbereichen anlegen
  - Exceptions bei fehlgeschlagener Allokierung unterbinden
  - Fehlersuche

Eingeschränkte Ressourcen

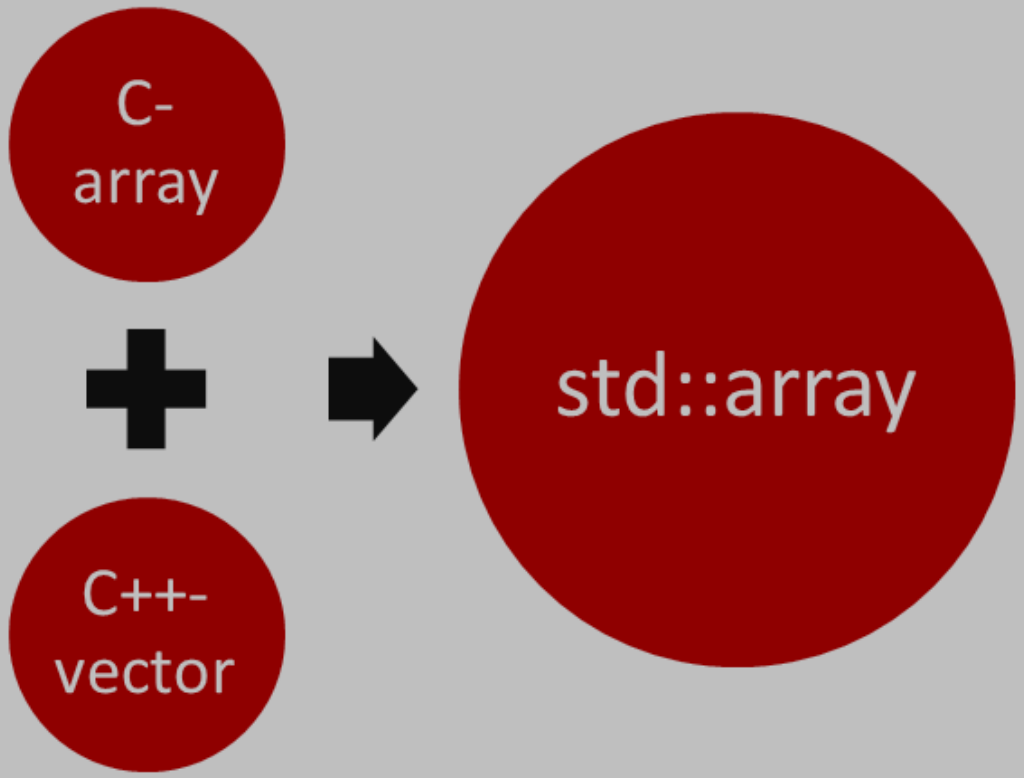
```
class MyInt {
  int i;
  public: MyInt(int ii) : i(ii) {
    cout << "constructor for: " << this << endl;
  }
  ~MyInt() {
    cout << "destructor for: " << this << endl;
  }
  void* operator new(std::size_t){
    cout << "operator new (default) " << endl;
    return static_cast<void*> (malloc(sizeof(MyInt)));
  }
  void* operator new(std::size_t, void* loc) {
    cout << "operator new (placement) " << endl;
    return loc;
  }
};
```

Eingeschränkte Ressourcen

```
char myIntBuf[sizeof(MyInt)];

int main(){
  string* onHeap= new string("on heap");
  char* buf= new char[100];
  string* inBuffer= new(buf) string("in buffer");
  cout << static_cast<void*>(&inBuffer);    // 0x7fffa0dfba40
  cout << static_cast<void*>(&onHeap);      // 0x7fffa0dfba48
  delete[] buf;
  delete onHeap;

  MyInt* myIntHeap= new MyInt(2011);        // operator new (default)
                                            // constructor for: 0xc15010
  delete myIntHeap;                         // destructor for: 0xc15010

  MyInt* myIntBuffer= new (myIntBuf) MyInt(2014); //operator new (placement)
                                                  //constructor for: 0x6021f4
  myIntBuffer->MyInt::~MyInt();                   //destructor for: 0x6021f4
}
```

Eingeschränkte Ressourcen

C++11 erhielt ein `array`.



- Homogener Container fester Länge
- Vereinigt die Performance eines C-Arrays mit dem Interface eines C++-Vektors
- Benötigt keine Heap Allokation

Eingeschränkte Ressourcen

```
std::array<int,6> arrCpp= {1,2,3,4,5,6};
int arrC[]= {1,2,3,4,5,6};

static_assert(sizeof(arrCpp) == 6*sizeof(int),"wrong size");
static_assert(sizeof(arrCpp) == sizeof(arrC),"size differs");

cout << accumulate(arrCpp.begin(),arrCpp.end(),0);  // 21

cout << accumulate(arrCpp.begin(),arrCpp.end(),1,
                   [](int a,int b){return a*b;});  // 720
```

Eingeschränkte Ressourcen

Konstante Zugriffszeit mit den neuen Hashtabellen.

- Auch bekannt als Dictionary oder assoziatives Array
- In C++98 lange vermisst
- Ergänzen die assoziativen Container aus C++98
  - Sehr ähnliches Interface
  - Schlüssel sind nicht geordnet
  - Konstante Zugriffszeit
  - Besitzen den Namenszusatz unordered_
- 4 Variationen

Name	Wert zugeordnet	mehrere gleiche Schlüssel
unordered_map	ja	nein
unordered_set	nein	nein
unordered_multimap	ja	ja
unordered_multiset	nein	ja

Eingeschränkte Ressourcen

```
map<string,int> m {{"Dijkstra",1972},{"Scott",1976}};
m["Ritchie"] = 1982;
cout << m["Ritchie"]; // 1982
m["Ritchie"] = 1983;
for(auto p : m) cout << '{' << p.first << ',' << p.second << '}';
// {Dijkstra,1972}{Ritchie,1983}{Scott,1976}
```

```
unordered_map<string,int>um{{"Dijkstra",1972},{"Scott",1976}};
um["Ritchie"] = 1982;
cout << m["Ritchie"]; // 1982
m["Ritchie"] = 1983;
for(auto p : um) cout << '{' << p.first << ',' << p.second << '}';
//  {Ritchie,1983}{Dijkstra,1972}{Scott,1976}
```

Lange Lebenszeit

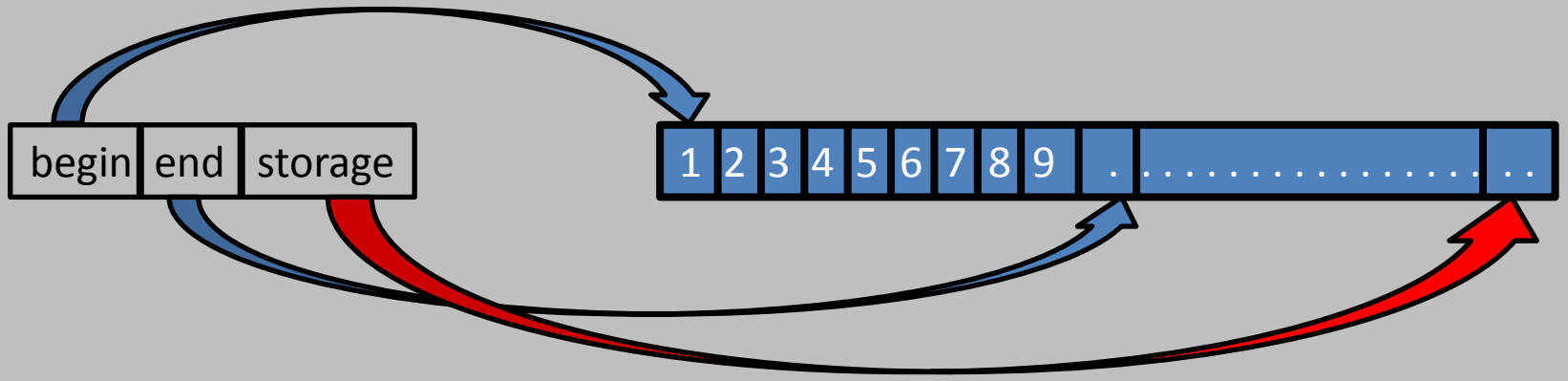
Speichermanagement for free mit C++-Containern.

- std::string

```
string text{"Initial text, "};
text +="which can still grow!";
```

- std::vector

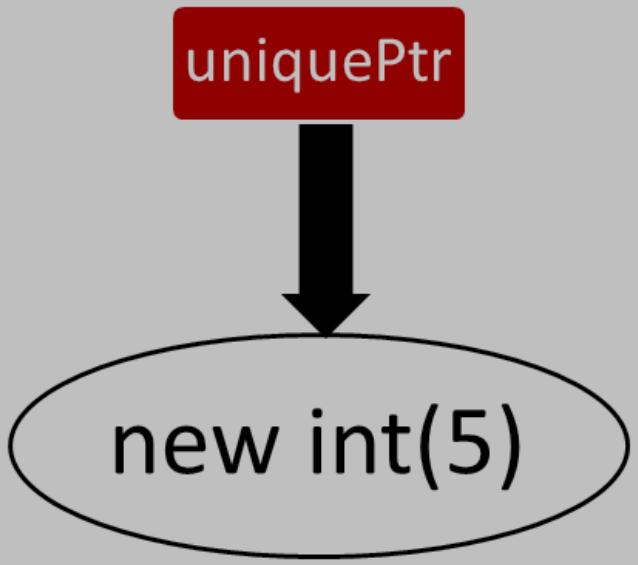
```
vector<int> myIntVec={1,2,3,4,5,6,7,8};
myIntVec.push_back(9);
myIntVec.reserve(100);
```




```
myIntVec.insert(myIntVec.end(),{10,... , 99 });
```

Lange Lebenszeit

Explizite Besitzverhältnisse mit `unique_ptr`.

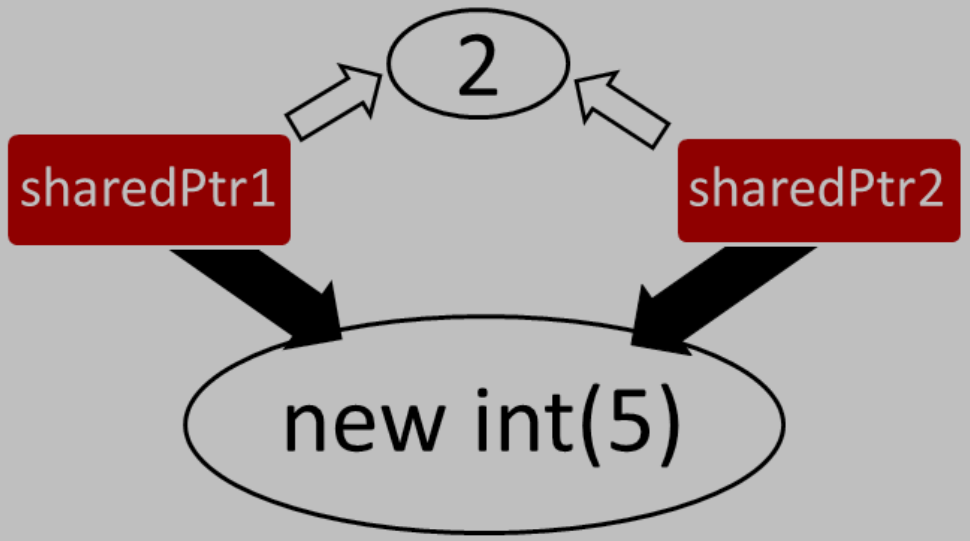


- Lässt sich nur verschieben
- Unterstützt Arrays
- Automatisches Speichermangement

- create and forget
- Minimale zusätzliche Speicher- und Zeitanforderungen
- Unterstützt spezielle Speichermanagement Strategien

Lange Lebenszeit

Geteilte Besitzverhältnisse mit `shared_ptr`.

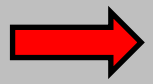


- Besitzt und verwaltet automatisch seinen Referenzzähler und seinen Verweis auf die Ressource
- Automatisches Speichermanagement

- Zusätzlicher Verwaltungsaufwand in Zeit und Speicher
- Reduziert den Speicherbedarf
- Muss mit Zyklen von `shared_ptr` umgehen

Lange Lebenszeit

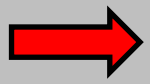
```
std::chrono::duration<double> st = std::chrono::system_clock::now();
for (long long i=0 ; i < 100000000; ++i){
  int* tmp(new int(i));
  delete tmp;
  // std::unique_ptr<int> tmp(new int(i));
  // std::unique_ptr<int> tmp= std::make_unique<int>(i);
  // std::shared_ptr<int> tmp(new int(i));
  // std::shared_ptr<int> tmp= std::make_shared<int>(i);
}
auto dur=std::chrono::system_clock::now() - st();
std::cout << dur.count();
```



Zeigertyp	Zeit	Verfügbarkeit
new	2.93 Sekunden	C++
std::unique_ptr	2.96 Sekunden	C++11
std::make_unique	2.84 Sekunden	C++14
std::shared_ptr	6.00 Sekunden	C++11
std::make_shared	3.40 Sekunden	C++11

Viele Kerne

C++ erhält mit C++11 ein Speichermodell.



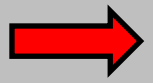
C++ ist sich zum ersten Mal mehrerer Threads bewußt.

- Ein Speichermodell gibt Antworten auf die Fragen
  1. Was sind atomare Operationen?
  2. Welche partielle Ordnung von Operationen ist gewährleistet?
  3. Welche Zusicherung gibt es an die Sichtbarkeit von Variablen?

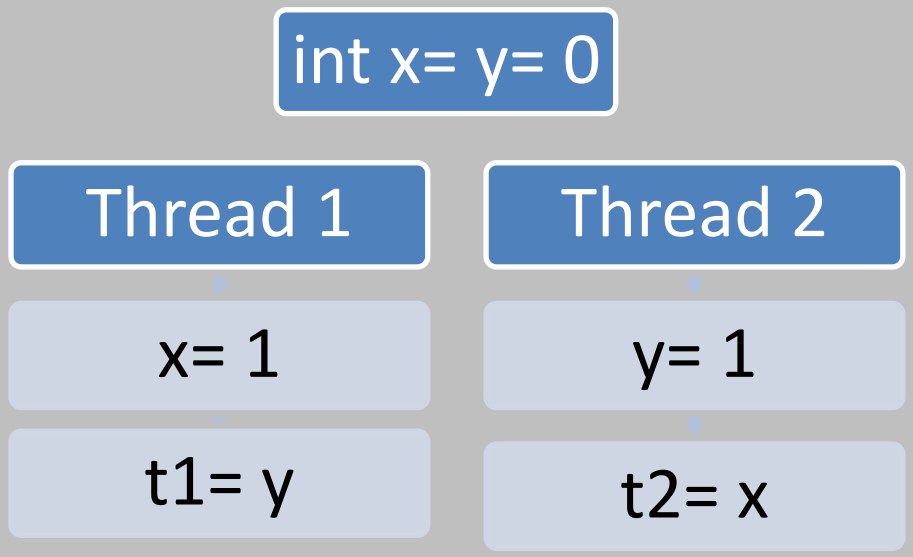
- Speichermodell
  - Ist an Javas Speichermodell angelehnt
  - Erlaubt aber den Bruch der Sequentiellen Konsistenz

Viele Kerne

C++ Standard Speichermodell ist die sequentielle Konsistenz.

 Die Reihenfolge der Operationen in Threads entspricht der Reihenfolge der Anweisungen.

- Wider die Intuition: Welche Werte können t1 und t2 besitzen?



Viele Kerne

Threads versus Tasks.

Thread

```
int res;
thread t([&]{res= 3+4;});
t.join();
cout << res << endl;   // 7
```

Task

```
auto fut= async([]{return 3+4;});
cout << fut.get()  << endl; // 7
```

	Thread	Task
Kommunikation	gemeinsame Variable	Kommunikationskanal
Threaderzeugung	verbindlich	optional durch das System
Synchronisation	der Vater wartet durch den `join`-Aufruf auf sein Kind	der `get`-Aufruf ist blockierend
Ausnahme im neuen Thread	Kind- und Erzeugerthread terminieren	Ausnahme wird an Aufrufer übergeben

Viele Kerne

Promise und Future als Kommunikationskanal.

- Promise
  - Sendet die Daten.
  - Kann mehrere Futures bedienen.
  - Kann Werte, Ausnahmen und Benachrichtigungen schicken.
- Future
  - Empfängt die Daten.

Viele Kerne

Promise und Future in Aktion.

```
a= 2000;
b= 11;
```

- Implizit durch `async`

```
future<int> sum= async([=]{ return a+b; });
sum.get();                                  // 2011
```

- Explizit durch `future` und `promise`

```
void sum(promise<int>&& intProm,int x, int y){
  intProm.set_value(x+y);
}
promise<int> futRes= sumPromise.get_future();
thread sumThread(&sum,move(sumPromise),a,b);
futRes.get();                               // 2011
```

Viele Kerne

Charakteristiken von Threads.

- Der Thread startet sofort sein Ausführung.

```
threadt t([] {cout << "I'm running." << endl; });
```

- Der Vater-Thread muss
  - auf sein Kind warten

    `t.join();`
  - sich von seinem Kind trennen (Daemon-Thread)

    `t.detach();`

  ⇒ Falls nicht, beendet sich der Prozess mit `std::terminate`.

- Der Thread kann private Daten besitzen.

```
thread_local int threadLocalStorage= 2011;
```

Viele Kerne

Gemeinsame Variablen müssen geschützt werden.

- Mutex
  - Stellt den exclusiven Zugriff (**mut**ual **ex**clusion) sicher.
  - Wird in ein Lock verpackt um Deadlocks zu vermeiden.
- Lock
  - `lock_guard`
    - Einfachste Anwendungsfall
  - `unique_lock`
    - Explizites Setzen oder Freigeben eines Locks
    - Verschieben oder Austauschen von Locks
    - Versuchsweise oder verzögertes Locken
  - `shared_lock`
    - Reader-Writer Locks
    - Neu mit C++14

Viele Kerne

Lesend verwendete Daten müssen nur sicher initialisiert werden.

➡ Das teuere Locken der Variable ist nicht notwendig.

- 3 Möglichkeiten
  1. Konstante Ausdrücke

```
constexpr double d(3.5);
```

  2. `call_once` and `once_flag`

```
void onlyOnceFunc(){ . . . }
std::once_flag= onceFlag;
std::call_once(onceFlag, onlyOnceFunc);
```

  3. Statische Variablen mit Blockgültigkeit

```
void func(){ . . . static int a{2011}; . . . }
```

Viele Kerne

Synchronisation zweier Threads.

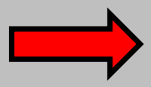
```
std::mutex proVarMutex;
std::condition_variable condVar;
bool dataReady;
```

Thread 1: Sender

```
std::lock_guard<std::mutex> senderLock(proVarMutex);
protectedVar= 2000;
dataReady= true;
condVar.notify_one();
```

Thread 2: Empfänger

```
std::unique_lock<std::mutex> receiverLock(proVarMutex);
condVar.wait(receiverLock,[]{ return dataReady;});
protectedVar += 11;
```



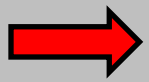
Ein Sender – viele Empfänger (`notify_all`, `wait`)

Was ich noch sagen wollte

- Sicherheitskritische Systeme
  - Streng typisierte Aufzählungstypen
  - Der neue Nullzeiger `nullptr`
  - Der Smart Zeiger `auto_ptr` ist deprecated.
- Eingeschränkte Ressourcen
  - Die Container `tuple` und `forward_list`
  - Erweiterte Plain Old Datas (PODs)
  - `vector<bool>` und `bitset` sind auf Speicherplatz optimiert
- Viele Kerne
  - Alignment Unterstützung
  - Die neue Zeitbibliothek `chrono`

Mythen

- Templates blähen den Code auf.
- Objekte müssen im Heap erzeugt werden.
- Exceptions sind teuer.
- C++ ist langsam und benötigt zu viel Speicher.
- C++ ist zu gefährlich in sicherheitskritischen Systemen.
- In C++ muss man Objekt Orientiert programmieren.
- C++ kann man nur für Applikationen verwenden.
- Die iostream-Bibliothek ist zu groß, die STL-Bibliothek zu langsam.

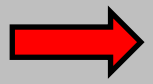


C++ ist ein nettes Spielzeug. Wir beschäftigen uns hier aber mit den richtigen Problemen.

Fakten

- MISRA C++
  - Motor Industry Software Reliability Association
  - Regeln für C++ in kritischen (embedded) Systemen

- TR18015.pdf
  - Technical report über die Performance von C++
  - Spezieller Fokus auf eigebettete Systeme
  - Widerlegen die Mythen



Beide basieren auf C++03, aber C++11 /C++14 ist noch besser für die Softwareentwicklung in eingebetteten Systemen geeignet.

FRAGEN?

Ich freue mich auf Ihr Feedback!

Vielen Dank!

Rainer Grimm