

# C++11: Quo vadis?

**science + computing ag**

IT-Dienstleistungen und Software für anspruchsvolle Rechnernetze

Tübingen | München | Berlin | Düsseldorf

# Bjarne Stroustrup über C++11

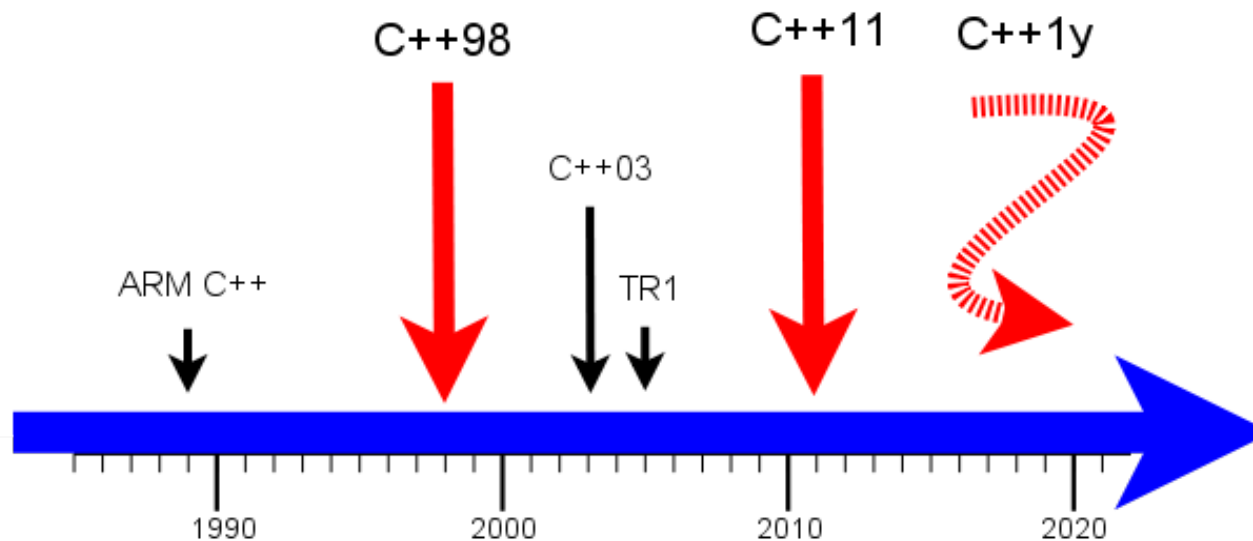


*Bjarne Stroustrup:  
„Surprisingly, C++11 feels like a  
new language - The pieces just fit  
together better.“*



„(Quelle:<http://www.wojcik.net>;2012-02-28)“

- The Past: C++98
- The Present: C++11
  - Kernsprache
  - Multithreading
  - Die Standardbibliothek
- The Future: C++1y



- ARM C++: “The Annotated C++ Reference Manual“
- **C++98**: erster ISO-Standard
  - C++03: technische Korrektur von C++98
  - TR1: Technical Report 1
- **C++11**: aktueller ISO-Standard
- **C++1y**: zukünftiger ISO-Standard



- Prinzipien von C++:
  - Vertraue dem Programmierer.
  - Zahle nicht für etwas, das Du nicht nutzt.
  - Brich keinen funktionierenden Code.
  - Kontrolle zur Übersetzungszeit ist besser als zur Laufzeit.
- Neue Ziele von C++11:
  - Ist die bessere Programmiersprache
    - für die Systemprogrammierung.
    - für das Schreiben von Bibliotheken.
  - Ist einfacher zu lehren und zu lernen.

- GCC
  - seit GCC 4.3 werden sukzessive die C++11-Feature unterstützt
  - der aktuelle GCC 4.7 unterstützt
    - Kernfunktionalität nahezu vollständig
    - Multithreading mit Einschränkungen (Speichermodell)
    - Bibliothek mit Einschränkungen (reguläre Ausdrücke)
- Microsoft Visual C++ 11.0
  - Kernfunktionalität mit großen Einschränkungen
  - Multithreading mit Einschränkungen
  - Bibliothek vollständig

# Automatische Typableitung mit auto

- Der Compiler bestimmt den Typ

```
auto myString= "my String"; // C++11  
auto myInt= 5; // C++11  
auto myFloat= 3.14; // C++11
```

- Erhalte einen Iterator auf das erste Element eines Vektors

```
vector<int> v;  
vector<int>::iterator it1= v.begin(); // C++98  
auto it2= v.begin(); // C++11
```

- Definition eines Funktionszeigers

```
int add(int a,int b){ return a+b; };  
int (*myAdd1)(int,int)= add; // C++98  
auto myAdd2=add; // C++11  
myAdd1(2,3) == myAdd2(2,3);
```

# Automatische Typableitung mit `decltype`

- Der Compiler gibt den Typ eines Ausdruckes zurück

```
decltype("str") myString= "str";           // C++11
decltype(5) myInt= 5;                       // C++11
decltype(3.14) myFloat= 3.14;              // C++11
decltype(myInt) myNewInt= 2011;            // C++11

int add(int a,int b){ return a+b; };
decltype(add) myAdd= add; // (int) (*) (int, int) // C++11
```



- Beispiel für die neue, alternative Funktionssyntax  
func( Argumente ) → Rückgabetyp { Funktionskörper }
- Eine generische add-Funktion durch **auto** und **decltype**

```
template <typename T1, typename T2>  
auto add(T1 first, T2 second) -> decltype(first + second) {  
    return first + second;  
}
```

```
add(1,1);
```

```
add(1,1.1);
```

```
add(1000LL,5);
```

- ➔ Das Ergebnis ist vom Typ

```
int
```

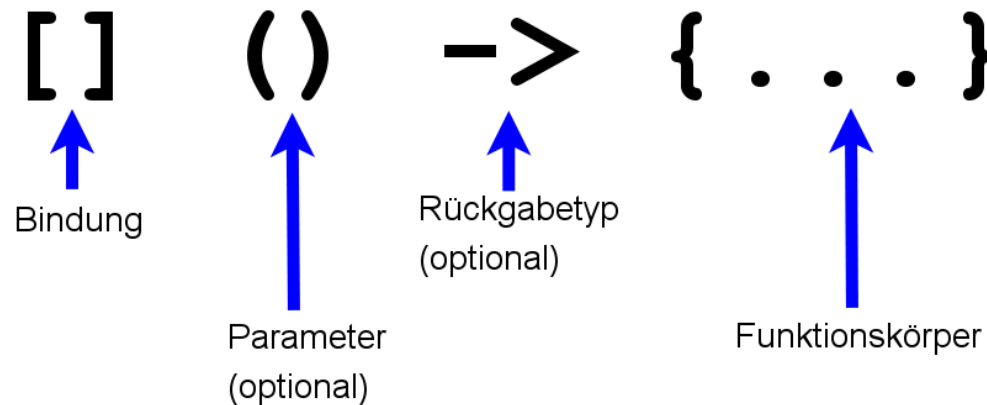
```
double
```

```
long long int
```



- Lambda-Funktionen sind
  - Funktionen ohne Namen
  - definieren ihre Funktionalität an Ort und Stelle
  - können wie Daten kopiert werden
- Lambda-Funktionen sollen
  - kurz und knackig sein
  - selbsterklärend sein

# Lambda-Funktionen: Syntax



- `[ ]` : Bindung der verwendeten Variablen per Copy oder per Referenz möglich
- `( )` : bei Parametern notwendig
- `->` : bei komplexeren Lambda-Funktionen notwendig
- `{ }` : kann mehrere Ausdrücke und Statements enthalten

- Sortiere die Elemente eines Vektors

```
vector<int> vec={3,2,1,5,4};
```

- in C++98 mit einem Funktionsobjekt

```
class MySort{  
public:  
    bool operator()(int v, int w){ return v > w; }  
};  
// a lot of code  
sort(vec.begin(),vec.end(),MySort());
```

- in C++11 mit einer Lambda-Funktion

```
sort(vec.begin(),vec.end(),  
    [](int v,int w){  
        return v > w;  
    });  
sort(vec.begin(),vec.end(), [](int v,int w){return v>w;});
```

- Lambda-Funktionen können noch viel mehr

- Starten eines Threads

```
thread t1([]{cout << this_thread::get_id() << endl;});  
thread t2([]{veryExpensiveFunction();});
```

- Lambda-Funktionen als first-class functions

- Argument einer Funktion

```
auto myLambda= []{return "lambda function";};  
getLambda(myLambda);
```

- Rückgabewert einer Funktion

```
function< string() > makeLambda{  
    return []{return "2011";};  
};
```

# Vereinfachte und vereinheitlichte Initialisierung

- Einfache Datentypen

```
int i{2011};  
string st= {"Stroustrup"};
```

- Container

```
vector<string> vec= {"Scott",st,"Sutter"};  
unordered_map<string,int> um= {{ "C++98",1998},{ "C++11",i}};
```

- Array als Datenelement einer Klasse

```
struct MyArray{  
    MyArray(): myData{1,2,3,4,5} {}  
    int myData[5];  
}
```

- konstantes Heap-Array

```
const float* pData= new const float[5]{1,2,3,4,5};
```

# Die Range-basierte For-Schleife

- Einfaches Iterieren über einen Container

```
vector<int> vec={1,2,3,4,5};  
for (auto v: vec) cout << v << ", ";           // 1,2,3,4,5,
```

```
unordered_map<string,int> um= {{"C++98",1998}, {"C++11",2011}};  
for (auto u:um) cout << u->first << ":" << u->second << " ";  
// "C++11":2011 "C++98":1998
```

- Modifizieren der Containerelemente durch auto&

```
for (auto& v: vec) v *= 2;  
for (auto v: vec) cout << v << " ,";           // 2,4,6,8,10,
```

```
string testStr{"Only for Testing."};  
for (auto& c: testStr) c= toupper(c);  
for (auto c: testStr) cout << c; // "ONLY FOR TESTING."
```

# Konstruktor: Delegation

```
class MyHour{
    int myHour_;
public:
    MyHour(int h){ // #1
        if (0 <= h and h <= 23 ) myHour_ = h;
        else myHour_ = 0;
    }
    MyHour(): MyHour(0){}; // #2
    MyHour(double h): MyHour(static_cast<int>(ceil(h))){}; // #3
};
```

→ die Konstruktoren #2 und #3 rufen den Konstruktor #1 auf



# Konstruktor: Vererbung (`using`)

```
struct Base{
    Base(int) {}
    Base(string) {}
};
struct Derived: public Base{
    using Base::Base;
    Derived(double) {}
};
int main() {
    Derived(2011);           // Base::Base(2011)
    Derived("C++11");      // Base::Base(C++11)
    Derived(0.33);         // Derived::Derived(0.33)
}
```

# Methoden anfordern (default)

- fordere spezielle Methoden und Operatoren vom Compiler an
  - Beispiele: Standard-, Kopierkonstruktor und Destruktor; Zuweisungsoperator, operator new

```
class MyType{  
    public:  
        MyType(int val) {} // #1  
        MyType()= default; // #2  
        virtual ~MyType();  
        MyType& operator= (MyType&)  
};  
MyType::~~MyType()= default;  
MyType& MyType::operator(MyType&)= default;
```

- #1 verhindert das automatische Erzeugen von #2

# Funktionsaufrufe unterdrücken (`delete`)

- eine nicht kopierbare Klasse

```
class NonCopyClass{
    public:
        NonCopyClass()= default;
        NonCopyClass& operator =(const NonCopyClass&)= delete;
        NonCopyClass (const NonCopyClass&)= delete;
};
```

- eine Funktion, die nur `double` annimmt

```
void onlyDouble(double){}
template <typename T> void onlyDouble(T)= delete;
int main(){
    onlyDouble(3);
};
```

➔ Fehler: use of deleted function »void onlyDouble(T) [mit T = int]«

# Explizites Überschreiben (override)

- Kontrolle durch den Compiler

```
class Base {  
    virtual void func1();  
    virtual void func2(float);  
    virtual void func3() const;  
    virtual long func4(int);  
};  
  
class Derived: public Base {  
    virtual void fun1() override;           // ERROR  
    virtual void func2(double) override;  // ERROR  
    virtual void func3() override;        // ERROR  
    virtual int func4(int) override;       // ERROR  
    virtual long func4(int) override;     // OK  
};
```

# Überschreiben unterbinden (`final`)

- für Methoden

```
class Base {  
    virtual void h(int) final;  
};  
class Derived: public Base {  
    virtual void h(int);           // ERROR  
    virtual void h(double);      // OK  
};
```

- für Klassen

```
struct Base final{};  
struct Derived: Base{};          // ERROR
```

- sind spezielle Referenzen, an die ein Rvalue gebunden werden kann
- Rvalues sind
  - temporäre Objekte
  - Objekte ohne Namen
  - Objekte, von denen keine Adresse bestimmt werden kann
- werden durch zwei und-Symbole (&&) erklärt

```
MyData myData;
```

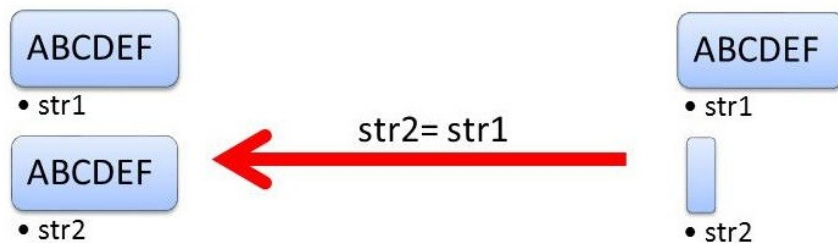
```
MyData& myDataLvalue= myData;
```

```
MyData&& myDataRvalue(MyData());
```

- der Compiler bildet Lvalues auf Lvalue-Referenzen, Rvalues auf Rvalue-Referenzen ab
  - ➔ spezielle Aktionen können für Rvalues hinterlegt werden
- Anwendungsfälle: Move-Semantik und Perfect Forwarding

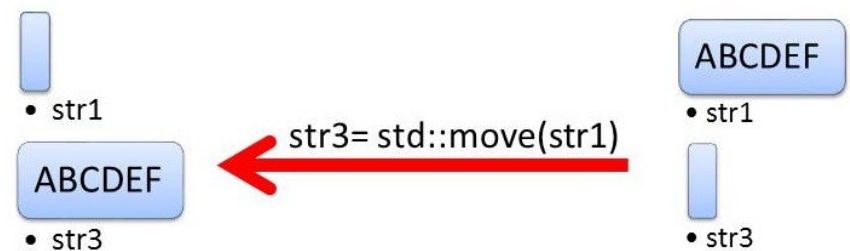
## Copy

```
string str1("ABDEF");  
string str2;  
str2= str1;
```



## Move

```
string str1{"ABCDEF"};  
string str3;  
str3= std::move(str1);
```



- Vorteile

- billiges Verschieben einer Ressource statt teurem Kopieren

```
vector<int> myBigVector;
```

```
.....
```

```
vector<int> myBigVector2 ( move (myBigVector) );
```

- nicht kopierbare aber verschiebbare (*moveable*) Objekte können *by value* einer Funktion übergeben oder von einer Funktion zurückgegeben werden

- Beispiele: `unique_ptr`, Dateien, Mutexe, Promise und Future

```
mutex m;
```

```
unique_lock<mutex> uniqueLock(m);
```

```
unique_lock<mutex> uniqueLock2 (move (m) );
```



# Perfect Forwarding (`forward`)

- ermöglicht Funktionstemplates zu schreiben, die ihre Argumente an eine weitere Funktion übergeben können, so dass deren Lvalue/Rvalue Eigenschaften beibehalten werden.
  - Stroustrup: „... a heretofore unsolved problem in C++.“
- Anwendungsfälle: Fabrikfunktionen oder auch Konstruktoren
- Beispiel: Fabrikfunktion mit einem Argument

```
template <typename T, typename T1>
T create(T1&& t1) {
    return T(forward<T1>(t1));
}

int myFive2= create<int>(5);           // Rvalue
int five=5;
int myFive= create<int>(five);       // Lvalue
```

- Templates, die beliebig viele Argumente annehmen können
- die Ellipse `...` bezeichnet das Template Parameter Pack, das gepackt oder entpackt werden kann
- Anwendung: `std::tuple`, `std::thread`
- Beispiel: eine vollkommen generische Fabrikfunktion

```
template <typename T, typename ... Args>
T createN(Args&& ... args) {
    return T(forward<Args>(args)...);
}

string st= createN<string>("Rainer Grimm");
struct MyStruct{
    MyStruct(int i, double d, string s) {}
};

MyStruct myStr= createN<MyStruct>(2011, 3.14, "Rainer Grimm");
```

# Mehr Kontrolle zur Übersetzungszeit

## (`static_assert`)

- besitzt keinen Einfluss auf die Laufzeit des Programmes
- `static_assert` lässt sich ideal mit der neuen Type-Traits-Bibliothek kombinieren
- stelle sicher,

- dass eine 64-bit Architektur vorliegt

```
static_assert(sizeof(long) >= 8, "no 64-bit code");
```

- dass ein arithmetischer Typ vorliegt

```
template< typename T >  
struct Add{  
    static_assert(is_arithmetic<T>::value, "T is not arith");  
};
```

# Konstante Ausdrücke (`constexpr`)

- stellen ein Optimierungspotential für den Compiler dar
  - können zur Übersetzungszeit ausgewertet werden
  - Compiler erhält einen tiefen Einblick in den evaluierten Code
- Drei Typen

- Variablen

```
constexpr double myDouble= 5.2;
```

- Funktionen

```
constexpr fact (int n){return n > 0 ? n * fact(n-1) : 1;}
```

- Benutzerdefinierte Typen

```
struct MyDouble{  
    double myVal;  
    constexpr MyDouble(double v) : myVal(v) {}  
};
```

# Raw-String-Literale (`r“(raw string)”`)

- unterdrücken das Interpretieren des Strings
- werden durch `r“(raw string)”` oder `R“(Raw String)”` definiert
- sind praktische Helferlein bei

- Pfadangaben

```
string pathOld= "C:\\temp\\newFile.txt";  
string pathRaw= r"(C:\temp\newFile.txt)";
```

- regulären Ausdrücken

```
string regOld= "c\\+\\+";  
string regRaw= r"(c\+\+)";
```

# Was ich noch sagen wollte

- Entwurf von Klassen

- Direktes Initialisieren von Klassenelementen

```
class MyClass{  
    const static int oldX= 5;  
    int newX= 5;  
    vector<int> myVec{1,2,3,4,5};  
};
```

- Erweiterte Datenkonzepte

- Streng typisierte Aufzählungstypen:

- `enum class Color: short { red, green, blue };`

- Unicode-Unterstützung: UTF-16 und UTF-32

- Benutzerdefinierte Literale: `63_s; 123.45_km; "Hallo"_i18n`

- das Nullzeiger-Literal `nullptr`

## C++11's Antwort auf die Anforderungen der Multicore-Architekturen



„(Quelle: <http://www.livingroutes.org>,2012-02-28)“

- ➔ eine standardisierte Threading-Schnittstelle
- ➔ ein definiertes Speichermodell

# Thread versus Task

## ▪ Thread

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

## ▪ Task

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

|                          | Thread  | Task   |
|--------------------------|---|--|
| Kommunikation            | gemeinsame Variable   | Kommunikationskanal                          |
| Threaderzeugung          | verbindlich   | optional                                     |
| Synchronisation          | der Vater wartet durch den <code>join</code> -Aufruf auf das Kind | der <code>get</code> -Aufruf ist blockierend |
| Ausnahme im neuen Thread | Kind- und Erzeugerthread terminieren                              | Rückgabewert des <code>get</code> -Aufrufes  |



# Threads (**thread**)

- ein Thread wird über sein Arbeitspaket parametrisiert und startet sofort
- der Vater-Thread muss auf sein Kind warten

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

- oder sich von seinem Kind trennen (Daemon-Thread)

```
t.detach();
```

- ➔ Daten sollen per Default in einen Thread kopiert werden

```
string s{"undefined behavior"};  
thread t([&]{ cout << s << endl;});  
t.detach();
```

# Thread-lokale Daten (`thread_local`)

- gehören exklusiv einem Thread
- verhalten sich wie statische Variablen

```
void addThreadLocal(string const& s){  
    thread_local threadLocalStr("Hello from ");  
    threadLocalStr += s;  
    cout << threadLocalStr << endl;  
}
```

```
thread t1(addThreadLocal, "t1");
```

```
thread t2(addThreadLocal, "t2");
```

- ➔ Ergebnis: Hello from t1  
Hello from t2

# Schutz von Daten (**mutex**)

- gemeinsam von Threads verwendete Daten müssen geschützt werden um eine Race Condition zu vermeiden
- **race condition**: mindestens zwei Threads verwenden eine gemeinsame Variable, wobei mindestens ein Thread diese modifiziert
- ein Mutex (**mutual exclusion**)
  - stellt den gegenseitigen Ausschluss sicher
  - gibt es
    - rekursiv und nicht rekursiv,
    - ohne und mit relativer oder absoluter Zeitangabe

# Deadlocks mit Mutexen

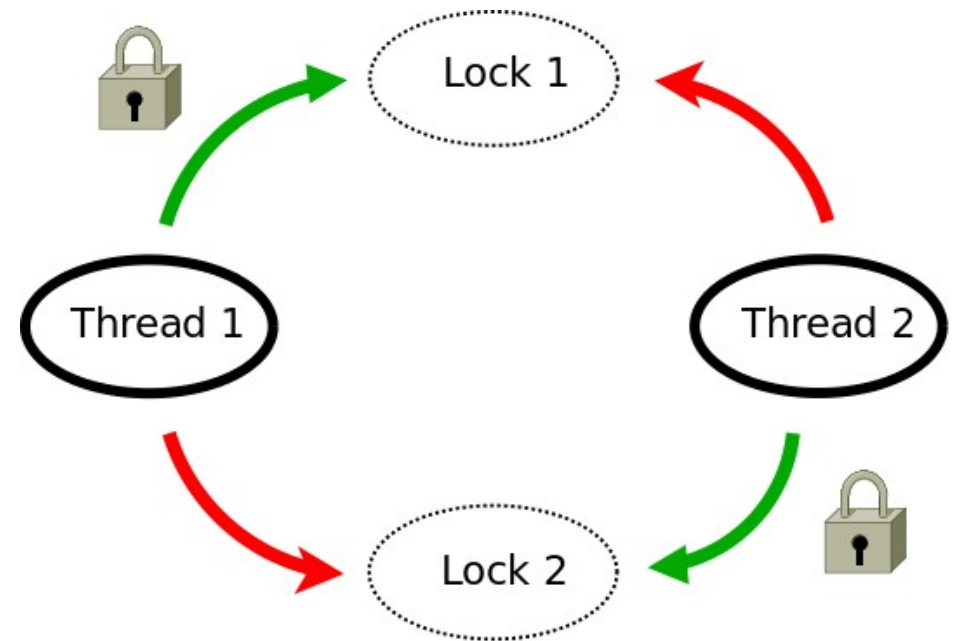
## Ausnahmesituation

- **Mutex im Einsatz:**

```
mutex m;  
m.lock();  
sharedVar= getVar();  
m.unlock();
```

- **Problem:** Eine Ausnahme in `getVar()` kann zu einem Deadlock führen.

## Locken in verschiedener Reihenfolge



➔ Verwenden Sie `lock_guard` und `unique_lock`

# RAII mit `lock_guard` und `unique_lock`

- `lock_guard` und `unique_lock` verwalten die Lebenszeit ihres Mutex automatisch nach dem RAII-Idiom

- `lock_guard`

```
mutex mapMutex;  
{  
    lock_guard<mutex> mapLock (mapMutex) ;  
    addToMap ("white", 0) ;  
}
```

- `unique_lock` für den anspruchsvolleren Anwendungsbereich
  - Explizite Setzen oder Freigeben eines Locks
  - Verschieben oder Austauschen von Locks
  - Versuchsweise oder verzögertes Locken

# Initialisieren gemeinsamer Variablen

- Lesend verwendete Daten müssen nur sicher initialisiert werden
  - ➔ das teure Locken der Variable ist nicht notwendig
- C++11 bietet drei Möglichkeiten an

## 1) konstante Ausdrücke

```
constexpr MyDouble myDouble;
```

## 2) `call_once` und `once_flag`

```
void onlyOnceFunction() { .... };  
once_flag= onceFlag;  
call_once(onceFlag, onlyOnceFunction)
```

## 3) statische Variablen mit Blockgültigkeit

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

# Bedingungsvariablen (**notify\_one**, **wait**)

- Ein Sender - ein Empfänger

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

## Thread 1: Sender

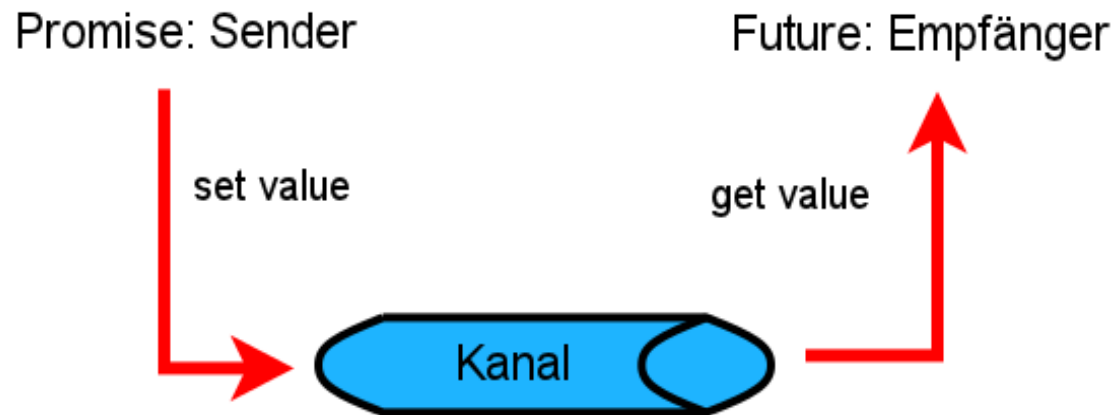
```
lock_guard<mutex> sender_lock(protVarMutex);  
protectedVar= 2000;  
dataReady= true;  
condVar.notify_one();
```

## Thread 2 : Empfänger

```
unique_lock<mutex> receiver_lock(protVarMutex);  
condVar.wait(receiver_lock, []{return dataReady;});  
protectedVar += 11;
```

- Ein Sender - viele Empfänger (**notify\_all** und **wait**)

# Promise und Future als Datenkanal



- der Promise
  - ist der Datensender
  - kann mehrere Futures bedienen
  - kann Werte, Ausnahmen und Benachrichtigungen schicken
- der Future
  - ist der Datenempfänger
  - **der `get`-Aufruf ist blockierend**



# Promise und Future in Aktion

```
a=2000;
```

```
b=11;
```

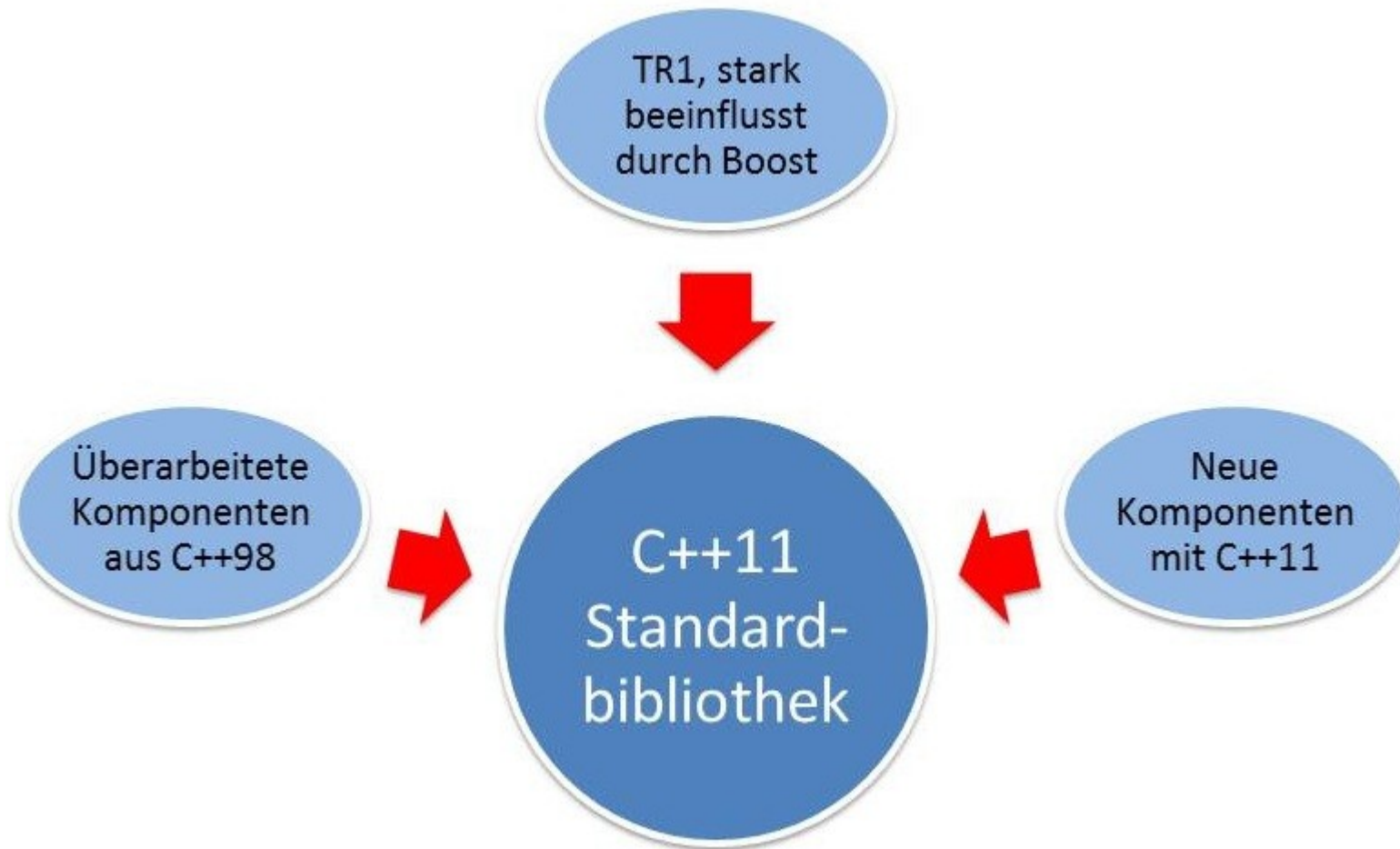
- implizit durch **async**

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

- explizit durch **future** und **promise**

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

# Einflüsse auf die neue Standardbibliothek





- ist die Beschreibungssprache für Zeichenmuster
- ist das Werkzeug für Textmanipulation:
  - entspricht der Text dem Zeichenmuster
  - suche ein Zeichenmuster in dem Text
  - ersetze ein Zeichenmuster in dem Text
  - iteriere über alle Zeichenmuster in dem Text

# Umgang mit Reguläre Ausdrücke



- Suche die erste Zahl in einem Text

```
string text("abc1234def");  
string regexNumber(r"(\d+)"); // 1  
smatch holdResult; // 2  
if ( regex_search(text,holdResult,regexNumber) ) // 3  
    cout << holdResult[0] << endl;  
    cout << holdResult.prefix() << endl;  
    cout << holdResult.suffix() << endl;
```

➔ Ergebnis: 1234

abc

def

- Iteriere über alle Zahlen eines Texts

```
string text="Der bisherige Standard C++98 ist nach 13  
Jahren am 12. August 2011 durch den neuen Standard C++11  
abgelöst worden."  
  
regex regNumb(r"(\d+)");  
sregex_token_iterator it(text.begin(),text.end(),regNumb);  
sregex_token_iterator end;  
while (it != end) cout << *it++ << " ";
```

➔ Ergebnis: 98 13 12 2011 11

- ermöglichen zur Übersetzungszeit

- Typabfragen (`is_integral<T>`, `is_same<T,U>`)

```
template <typename T>
```

```
T gcd(T a, T b) {
```

```
    static_assert(is_integral<T>::value, "T != integral");
```

```
    if( b==0 ) return a;
```

```
    else return gcd(b, a % b);
```

```
}
```

- Typtransformationen (`add_const<T>`)

```
typedef add_const<int>::type myConstInt;
```

```
cout << is_same<const int,myConstInt>::value << endl;
```

- ➔ Ergebnis: true

- ➔ Code, der sich selber optimiert

- kombiniert einen Zufallszahlenerzeuger mit einer -verteilung
  - Zufallszahlenerzeuger
    - erzeugt einen Zahlenstrom zwischen einem Minimum- und Maximumwert
    - Beispiel: Mersenne Twister, `random_device (/dev/urandom)`
  - Zufallszahlenverteilung
    - bildet die Zufallszahlen auf die Verteilung ab
    - Beispiele: Gleich-, Normal-, Poisson- und Gammaverteilung
- Simulieren eines Würfels:

```
random_device seed;  
mt19337 numberGenerator(seed());  
uniform_int_distribution<int> six(1,6);  
cout << six(numberGenerator) << endl; // 3
```



- elementarer Bestandteil der neuen Multithreading-Funktionalität
- Beispiele:
  - lege den aktuellen Thread für 100 Millisekunden schlafen  
`this_thread::sleep_for( chrono::millisecond(100) );`
  - Performancemessung in Sekunden  
`auto begin= chrono::system_clock::now();`  
`// a lot to do`  
`auto end= chrono::system_clock::now() - begin;`  
`auto timeInSeconds= chrono::duration<double>(end).count();`

# Referenz-Wrapper (`reference_wrapper`)

- ist ein kopierkonstruierbarer und zuweisbarer Wrapper um T&
  - ➔ verhält sich wie eine Referenz, kann aber kopiert werden
- Neue Anwendungsfälle für `reference_wrapper<T>`

1. Klassen, die Referenzen enthalten, können kopiert werden

```
struct Copyable{  
    Copyable(string& s) : name(s) {}  
    // string& badName; will not compile  
    reference_wrapper<string> name;  
};
```

2. Referenzen können in Containern der STL verwendet werden

```
int a=1, b=2, c=4;  
vector<reference_wrapper<int>> vec={ref(a), ref(b), ref(c)};  
c = 3;
```

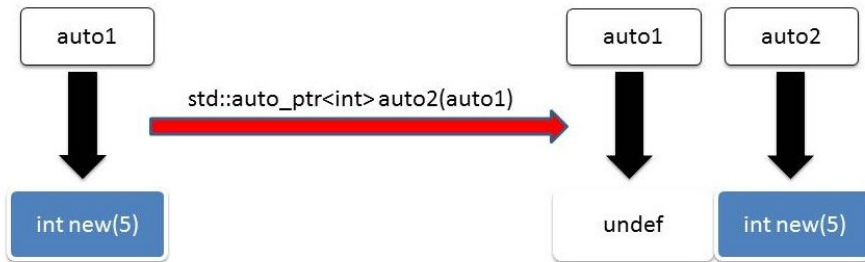
➔ Ergebnis: `vec[2] == 3`

# Smart Pointer: Lebenszyklus verwalten

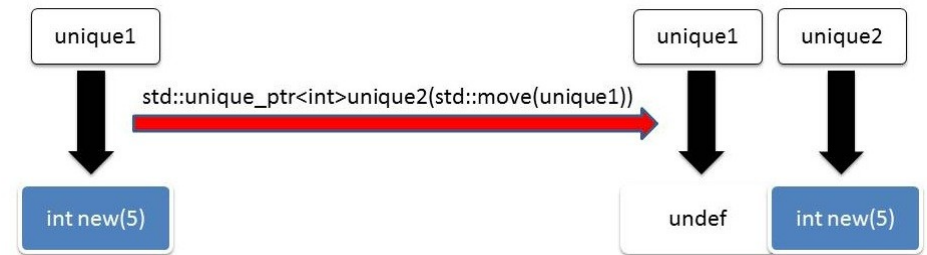
| Name       | Im C++-Standard | Beschreibung   |
|------------|-----------------|--|
| auto_ptr   | C++98           | <ul style="list-style-type: none"><li>• Besitzt die Ressource exklusiv.</li><li>• Transferiert beim Kopieren heimlich die Ressource.</li></ul>   |
| unique_ptr | C++11           | <ul style="list-style-type: none"><li>• Besitzt die Ressource exklusiv.</li><li>• Kann nicht kopiert werden.</li><li>• Verwaltet nicht kopierbare Objekte (Threads, Locks, Dateien ...).</li></ul>                                 |
| shared_ptr | C++11           | <ul style="list-style-type: none"><li>• Bietet einen Referenzzähler auf die gemeinsame Variable an.</li><li>• Verwaltet automatisch den Referenzzähler.</li><li>• Löscht die Ressource, sobald der Referenzzähler 0 ist.</li></ul> |
| weak_ptr   | C++11           | <ul style="list-style-type: none"><li>• Hilft, zyklische Referenzen zu brechen.</li><li>• Modifiziert nicht den Referenzzähler.</li></ul>  |

# Smart Pointer: *Kopieren*

## auto\_ptr



## unique\_ptr



## shared\_ptr



# Smart Pointer im Einsatz

```
shared_ptr<int> sharedPtr(new int(5));  
// Referenzzähler == 1  
{  
    shared_ptr<int> localSharedPtr(sharedPtr);  
    // Referenzzähler == 2  
}  
// Referenzzähler == 1  
weak_ptr<int> weakPtr(sharedPtr);  
// Referenzzähler == 1  
shared_ptr<int> globalSharedPtr= sharedPtr;  
// Referenzzähler == 2  
globalSharedPtr.reset();  
// Referenzzähler == 1
```

# Neue Container (tuple und array)

- **Tupel**

- heterogener Container fester Länge
- Erweiterung von pair aus C++98

```
tuple<string,int,float> tup=("first",1998,3.14);  
auto tup2= make_tuple("second",2011,'c');  
get<1>(tup)= get<1>(tup2);
```

- **Array**

- homogener Container fester Länge
- verbindet die Performance des c-Arrays mit dem Interface eines C++-Vektors

```
array<int,8> arr{{1,2,3,4,5,6,7,8}};  
int sum= 0;  
for_each(arr.begin(),arr.end(), [&sum](int v){sum += v;});
```

# Neue Container: Hashtabellen

- besteht aus (Schlüssel,Wert)-Paaren
- auch bekannt als Dictionary oder assoziatives Array
- in C++98 lange vermisst
- Vergleich mit den assoziativen Containern aus C++98
  - sehr ähnliches Interface
  - Schlüssel nicht geordnet
  - konstante Zugriffszeit
- 4 Variationen:

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

# Neue Container: Hashtabellen

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

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

- Ergebnis: {Dijkstra,1972}{Ritchie,1983}{Scott,1976}  
              {Ritchie,1983}{Dijkstra,1972}{Scott,1976}



- Feature für die funktionale Programmierung
  - `bind` erlaubt einfach Funktionsobjekte zu erzeugen
  - `function` bindet die Funktionsobjekte von `bind`

```
int add(int a, int b){ return a+b;};  
function< int(int)> myAdd= bind(add,2000,_1);  
add(2000,11) == myAdd(11);
```
- beide Bibliotheken werden durch die Erweiterung der Kernsprache nahezu überflüssig
  - `bind` kann durch Lambda-Funktionen ausgedrückt werden
  - `function` kann durch `auto` ersetzt werden

```
auto myAddLambda= [](int v){ return add(2000,v); };  
add(2000,11) == myAddLambda(11);
```

Vorhersagen sind schwierig, besonders wenn sie die Zukunft betreffen.



„(Quelle: [www.nato.int](http://www.nato.int);2012-02-28)“

- Zeitrahmen für C++1y:
  - 2016 + X
- Bibliothekserweiterung:
  - 2016 - X
  - Boost mit dem Technical Report 2
- Inhalt
  - Concepts
  - Multithreading
  - Bibliotheken

# C++11: Quo vadis?



Vielen Dank für Ihre Aufmerksamkeit.

**Rainer Grimm**

science + computing ag

[www.science-computing.de](http://www.science-computing.de)

Telefon 07071 9457-253

[r.grimm@science-computing.de](mailto:r.grimm@science-computing.de)