I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I can no longer figure out how to use my telephone.

Type operator sign (parameters) {/.....//}
The C++ Lands
Its amazing creatures and weird beasts
explorer
Mena [http://alexacpp.blogspot.com]
cartographer
Jim [http://Jimblog.me]
C++ History
C++ History

~ 1980 Bjarne Stroustrup „C with Classes“ Cfront (C++ --> C)
1983 erstmalige Nennung von C++
1994 erster Entwurf ANSI/ISO Standard
1998 Standard ISO/IEC 14882 (C++98)
2003 Standard C++03 (TR1 etc.)
2011 Standard C++11 (zuvor als C++ox als Arbeitstitel)
A taste of C++11

```cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <array>
#include <future>
#include <functional>

using std::string;

string flip (string s) {
    reverse (begin(s), end(s));
    return s;
}

std::future<string> task(string s) {
    return async(std::bind(flip, s));
}
```
```cpp
int main()
{
    std::array<std::future<std::string>, 3> v = {
        task(" sleef ++C"),
        task(" wen a ekil"),
        task(" \negaugnal")
    };

    for (auto& e : v)
        std::cout << e.get();
}
```
#include <iostream>
#include <string>
#include <algorithm>
#include <array>
#include <future>
#include <functional>

using std::string;

string flip (string s) {
    reverse (begin(s), end(s));
    return s;
}

std::future<string> task(string s) {
    return async(std::bind(flip, s));
}
```cpp
int main()
{
    std::array<std::future<std::string>, 3> v = {
        task("sleef ++C"),
        task("wen a ekil"),
        task("\negaugnal")
    };

    for (auto& e : v)
        std::cout << e.get();
}
```
A taste of C++11

// a variant without function task:
...

int main() {
    std::array<std::future<string>, 3> v = {
        async([] { return flip( "sleef ++C" ); }),
        async([] { return flip( "wen a ekil " ); }),
        async([] { return flip( "\negaugnal " ); })
    };
    for (auto& e : v) {
        std::cout << e.get();
    }
}
test your C++ level

reverse(begin(s), end(s));

// oder

std::reverse(begin(s), end(s));

Argument dependent lookup aka ADL aka König lookup:

```
3.4.2 Argument-dependent name lookup

When the postfix-expression in a function call (5.2.2) is an unqualified-id, other namespaces not considered during the usual unqualified lookup (3.4.1) may be searched, and in those namespaces, namespace-scope friend function declarations (11.4) not otherwise visible may be found. These modifications to the search depend on the types of the arguments (and for template template arguments, the namespace of the template argument). [Example:

```cpp
namespace N {
    struct S {};
    void f(S);
}

void g() {
    N::S s;
    f(s);
    (f)(s);
    // OK: calls N::f
    // error: N::f not considered; parentheses
    // prevent argument-dependent lookup
```

— end example]
```
good books

good books

 Bollywood 

finales Standarddokument nur gegen Geld (238 CHF ! ~ 200 €)
frei verfügbar: letzte drafts (~99%) z.B. n3092.pdf
bislang nur wenige (gute) Bücher verfügbar
u.a.


Compiler

- noch keiner mit C++11 kompletter Unterstützung
- es wird besser:
  - Microsoft VS C++11 (beta)
  - g++ > 4.6.0
  - clang > 3.0
  - ....
- häufig sind work-arounds möglich
- fragen Sie auf stackoverflow.com (NACH! der Suche nach einer Antwort)
C++11 im Detail

Abschnitt 1: Kernsprache

Abschnitt 2: Bibliothek

Abschnitt 3: Parallelität
auto -- deduction of a type from an initializer
**auto** -- deduction of a type from an initializer

```c
auto x = 7;
```

*x* ist von Typ *int* wegen des Typs des Literals.

```c
auto x = expression;
```

*x* ist vom Typ des Resultats von *expression*. 

auto -- deduction of a type from an initializer

template<class T> void printall(const vector<T>& v) {
    for (auto p = begin(v); p!=end(v); ++p) cout << *p << "\n";
}

statt C++98:
template<class T> void printall(const vector<T>& v) {
    for (typename vector<T>::const_iterator p = v.begin(); p!=v.end(); ++p)
        cout << *p << "\n";
}

---------------------------------------------------------------------------
template<class T,class U> void f(const vector<T>& vt, const vector<U>& vu){
    // ...
    auto tmp = vt[i]*vu[i]; // whatever T*U yields
    // ...
}
**decltype -- deduction of a type from an expression**

**decltype(E)**

Ist der Typ ("declared type") des Namens oder des Ausdrucks `E` und kann in Deklarationen verwendet werden. Der Ausdruck wird **NICHT** berechnet!

```cpp
void f(const vector<int>& a, vector<float>& b) {
    typedef decltype(a[0]*b[0]) Tmp;
    for (int i=0; i<b.size(); ++i) {
        Tmp* p = new Tmp(a[i]*b[i]);
    }
}
```

**auto** ist oft einfacher. **decltype** wird gebraucht, wenn man einen Typ für etwas benötigt, das keine Variable ist z. B. ein return Typ.
decltype -- deduction of a type from an expression

decltype unterscheidet Werte und Referenzen:

```
int val = 3;
int& ref = val;
```

```
dcltype(val) -----> int
decltype(ref) -----> int&
dcltype(ref+1) -----> int
dcltype((val)) -----> int&
```

(val) ist ein Ausdruck und hat eine Adresse
Wie kann man eine generische Funktion add schreiben?

template <typename X, typename Y> 
add(X x, Y y) 
{ 
    return x + y; 
}
Wie kann man eine generische Funktion `add` schreiben?

template <typename X, typename Y>
dcltype(x+y) add(X x, Y y)
{
    return x + y;
}

geht nicht: x und y nicht bekannt :-(

decltype -- deduction of a type from an expression
decltype -- deduction of a type from an expression

Wie kann man eine generische Funktion add schreiben?

template <typename X, typename Y>
dcltype((*(X*)0)+ (*(Y*)0)) add(X x, Y y)
{
    return x + y;
}
OK, aber extrem hässlich :-(
Lösung: neue Signatur-Syntax

template <typename X, typename Y>
auto add(X x, Y y) -> decltype(x+y)
{
    return x + y;
}

auto nur als Platzhalter (keine Typ-Deduktion)

häufig bei sog. lambda-Funktionen anzutreffen (später...)

dcltype -- deduction of a type from an expression