enum class -- scoped and strongly typed enums

C - **enum**s mit Problemen:

- konvertierbar nach int
- exportieren ihre Aufzählungsbezeichner in den umgebenden Bereich (name clashes)
- schwach typisiert (z.B. keine forward Deklaration möglich)

enum classes ("strong enums") sind stark typisiert und 'scoped':

```
enum Alert { green, yellow, election, red }; // traditional enum
enum class Color { red, blue };              // scoped and strongly typed enum
                            // no export of enumerator names into enclosing scope
                            // no implicit conversion to int
enum class TrafficLight { red, yellow, green };

Alert a = 7;            // error (as ever in C++)
Color c = 7;            // error: no int->Color conversion
int a2 = red;           // ok: Alert->int conversion
int a3 = Alert::red;    // error in C++98; ok in C++11
int a4 = blue;          // error: blue not in scope
int a5 = Color::blue;   // error: not Color->int conversion

Color a6 = Color::blue; // ok
```

enum class

Typ der Repräsentation kann spezifiziert werden

```
enum class Color : char { red, blue }; // compact representation enum class

TrafficLight { red, yellow, green };
          // by default, the underlying type is int

enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };
// how big is an E?
// (whatever the old rules say;
// i.e. "implementation defined")

enum EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U };
// now we can be specific
```

forward Deklaration möglich

```
enum class Color_code : char; // (forward) declaration
void foobar(Color_code* p); // use of forward declaration
// ...
enum class Color_code : char { red, yellow, green, blue }; // definition
```

constexpr -- generalized and guaranteed constant expressions

mehr Konstanten zur Compile-Zeit


```
enum Flags { good=0, fail=1, bad=2, eof=4 };

constexpr int operator|(Flags f1, Flags f2)
{ return Flags(int(f1)|int(f2)); }

void f(Flags x) {
        switch (x) {
                case bad: /* ... */ break;
                case eof: /* ... */ break;
                case bad|eof: /* ... */ break;
                default: /* ... */ break;
        }
}
```

constexpr sagt, dass das Resultat zur Compile-Zeit berechnet werden kann/muss

constexpr

```
constexpr int x1 = bad|eof; // ok

void f(Flags f3) {
  constexpr int x2 = bad|f3; // error can't evaluate at compile time
  int x3 = bad|f3; // ok
}
```

Auch für einfache Objekte:

```
struct Point {
  int x,y;
  constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};
constexpr Point origo(0,0);
constexpr int z = origo.x;
constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2) };
constexpr auto x = a[1].x; // x becomes 1
```

static_assert

Compiler-Assertions (C- assert wirkt zur Laufzeit!)

```
static_assert(constant_expression,string);
```

```
static_assert(sizeof(long) >= 8,
       "64-bit code generation required for this library.");
struct S { X m1; Y m2; }; static_assert(sizeof(S)==sizeof(X)
+sizeof(Y),
       "unexpected padding in S");
```

Nicht zur Prüfung von Laufzeiteigenschaften verwendbar

```
int f(int* p, int n) {
  static_assert(p==0,"p is not null");
  // error: static_assert() expression not a constant expression
}
```

long long -- a longer integer

Integers mit wenigstens 64 bits

```
long long x = 9223372036854775807LL;
```

kein **`long long long`**

kein **`short long long`**

[insb. **`long`** != **`short long long`**]

nullptr -- a null pointer literal

`nullptr` ist ein Literal für den Zeigerwert Null, kein integer!

```
char* p = nullptr;
int* q  = nullptr;
char* p2 = 0; // 0 still works and p==p2
void f(int);
void f(char*);
...
f(0);       // calls f(int)
f(NULL);    // calls f(int)
f(nullptr); // calls f(char*)
void g(int);
g(nullptr); // error nullptr is not an int
int i = nullptr; // error nullptr is not an int
```

uniform initialization syntax and semantics

C++ hat verschiedene Wege zur Initialisierung, je nach Objekttyp und Kontext.
fehleranfällig und nicht konsistent ☹

```
string a[] = { "foo", " bar" }; // ok: initialize array variable
vector<string> v = { "foo", " bar" }; // error: initializer list for non-aggregate vector
void f(string a[]); f( { "foo", " bar" } ); // syntax error: block as argument
```

und

```
int a = 2;           // assignment style
int[] aa = { 2, 3 }; // assignment style with list
complex z(1,2);      // functional style initialization
x = Ptr(y);          // functional style for conversion/cast/construction
```

uniform initialization syntax and semantics

C++11 {}-initializer lists für alle Initialisierungen:

```
X x1 = X{1,2};
X x2 = {1,2}; // the = is optional
X x3{1,2};
X* p = new X{1,2};

struct D : X {
 D(int x, int y) :X{x,y} { /* ... */ };
};

struct S {
 int a[3];
 S(int x, int y, int z) :a{x,y,z} { /* ... */ };
 // solution to an old problem
};
```

uniform initialization syntax and semantics

Auch ein altes (Parse-)Problem ist damit gelöst:

```
struct P
{
    P(){std::cout<<"P::P()\n";}
    P(const P&) {std::cout<<"P::P(const P&)\n";}
};

// C++ most vexing parse – what is:

P p(P()); // ???
// p: P -> P :-(

P p{P()}; // default constructed P
```

initializer lists

Handliche Listen überall

```
vector<double> v = { 1, 2, 3.456, 99.99 };

list<pair<string,string>> languages = { // parse error in C++98
 {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"} };

map<vector<string>,vector<int>> years = { // fine in C++11
  { {"Maurice","Vincent","Wilkes"},{1913,1945,1951,1967,2000} },
  { {"Martin","Ritchards"},{1982,2003,2007} },
  { {"David","John","Wheeler"},{1927,1947,1951,2004} }
};
```

initializer lists

Nicht mehr nur für Felder, Argumente vom Typ `std::initializer_list<T>` möglich.

```
void f( initializer_list<int> );
f( {1,2} );
f( {23,345,4567,56789} );
f({}); // the empty list
f{1,2}; // error: function call ( ) missing
years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});

void print(std::initializer_list<int> ls) {
    for(const auto i: ls) std::cout<<i<<std::endl;
}
... print ({1,2,3,4,5,6,7,8,9});
```

Konstruktoren mit einem einzigen Argument vom Typ **std::initializer_list** heißen initializer-list Konstruktoren. Die Standardcontainer, string, regex etc. haben solche.

initializer lists

initializer-list Konstruktoren sind auch für nutzerdefinierte Typen möglich:

```
class Data {
    std::vector<int> stuff_;
public:
    Data(std::initializer_list<int> data) :
                stuff_{begin(data), end(data)} {}
};

Data data {1,2,3,4,5,6};
```

ACHTUNG:

2 A constructor is an *initializer-list constructor* if its first parameter is of type `std::initializer_list<E>` or reference to possibly cv-qualified `std::initializer_list<E>` for some type `E`, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [*Note:* Initializer-list constructors are favored over other constructors in list-initialization (13.3.1.7). — *end note*] The template `std::initializer_list` is not predefined; if the header `<initializer_list>` is not included prior to a use of `std::initializer_list` — even an implicit use in which the type is not named (7.1.6.4) — the program is ill-formed.

initializer lists

neue Regeln für die Auflösung von Überladung wurden erforderlich:

(quasi in letzter Sekunde im Standard [13.3.1.7], noch nicht in N3092 !)

eine initializer_list-Signatur hat Vorrang auch wenn nur ein Element (als Liste) übergeben wird, außer die Typen sind nicht kompatibel

```
void print(std::initializer_list<int> ls) {
     for(const auto i: ls) std::cout<<i;
}
void print(int i) {
     std::cout<<"int: "<<i<<std::endl;
}
void print(const char* i) {
     std::cout<<"const char*: "<<i<<std::endl;
}

print ({1,2,3,4,5,6,7,8,9});        123456789
print (2);                          int: 2
print ({2});                        2
print ("zwei");                     const char*: zwei
print ({"zwei"});                   const char*: zwei
```

preventing narrowing

```
int x = 7.3; // Ouch!
void f(int);
f(7.3); // Ouch!
```

mit {} Initialisierung nicht:

```
int x1 = {7.3}; // error: narrowing
double d = 7;
int x2{d}; // error: narrowing (double to int)
char x3{7};
     // ok: even though 7 is an int, this is not narrowing
vector<int> vi = { 1, 2.3, 4, 5.6 };
     // error: double to int narrowing
```