attributes

Eine standardisierte Syntax für alle Arten optionaler und Vendor-Spezifischer Informationen im Quelltext (z.B. **__attribute__**, **__declspec**, **#pragma** ...), fast überall erlaubt, betreffen immer das vorhergehende Element.

```
void f  [[ noreturn ]] () { throw "error"; /* OK */ }
// f() will never return
unsigned char c [[ align(double) ]] [sizeof(double)];
// array of characters, suitably aligned for a double
```

Neben **noreturn** und **align** gibt's im Standard noch:

```
struct D : B { void f(); // error };
struct foo* f [[carries_dependency]] (int i); // hint to optimizer
int* g(int* x, int* y [[carries_dependency]] );
```

Denkbar ist z.B. OpenMP-Unterstützung

```
for [[omp::parallel()]] (int i=0; i<v.size(); ++i) { ... }
```

Anders gelöst wurde:

```
struct B { virtual void f [[ final ]] (); // do not try to override };
```

inline namespace

Koexistenz verschiedener Bibliotheksversionen, Insb. für **::std** benötigt

```
// file V99.h:
inline namespace V99 {          void f(int); // does something better than the V98 version
                                void f(double); // new feature
}
// file V98.h:
namespace V98 {                 void f(int); // does something
}
// file Mine.h:
namespace Mine {
#include "V99.h"
#include "V98.h“
}
```

Beide Versionen sind verfügbar: Deklarationen aus inline Namensräumen erscheinen im übergeordneten Namensraum

```
#include "Mine.h"
using namespace Mine;
V98::f(1); // old version
V99::f(1); // new version
f(1); // default version   --- ??? V99::f(1)
```

inline namespace

```
// from
namespace Networking {
    class TCPSocket;
    class UDPSocket;
}
// to versioning
namespace Networking {
    namespace V1 {
        class TCPSocket;
    }
    namespace V2 {
        class TCPSocket;
    }

    class UDPSocket;
}
// problem: breaks any code using TCPSocket

// with inline namespace:
namespace Networking {
    namespace V1 {
        class TCPSocket;
    }

    inline namespace V2 {
        class TCPSocket;
    }

    class UDPSocket;
}
```

```
Networking::TCPSocket *t;
// Networking::V2::TCPSocket, because of the inline namespace
Networking::V1::TCPSocket *t2; // Networking::V1::TCPSocket
Networking::V2::TCPSocket *t3; // Networking::V2::TCPSocket

using namespace Networking;
TCPSocket *t4;
// Networking::V2::TCPSocket, because of the inline namespace
V1::TCPSocket *t5;          // Networking::V1::TCPSocket
V2::TCPSocket *t6;          // Networking::V2::TCPSocket
```

explicit conversion operators

explicit auch für *conversion operators*

```
struct S { S(int) { } };
struct SS {
  int m;
  SS(int x) :m(x) { }
  explicit operator S() { return S(m); }
  // because S don't have S(SS)
};
SS ss(1);
S s1 = ss; // error; like an explicit constructor
S s2(ss);  // ok ; like an explicit constructor
void f(S);
f(ss);     // error; like an explicit constructor
f(S(ss));  // ok
```

noexcept

exception specifications à la `throw(dies, das, nochwas)` haben sich als unpraktisch erwiesen.

Was geblieben ist, ist die Verwendung als `throw()`, wenn doch eine Exception auftaucht:

- *stack unwinding* (dtors aller lokalen Objekte)
- Aufruf von `unexpected()` .... Aufruf von `terminate()` ... Aufruf von `abort()`
- „Viel Aufwand mit wenig Nutzen“

Neu: `noexcept(expression)` ... keine Ausnahme erlauben, wenn `expression` --- ruft direkt `terminate()`

```
// whether foo is declared noexcept depends on if the expression
// T() will throw any exceptions
template <class T>
  void foo() noexcept(noexcept(T())) {}
void bar() noexcept(true) {}
void baz() noexcept { throw 42; }  // noexcept is the same as noexcept(true)
int main() {
    foo<int>();  // noexcept(noexcept(int())) => noexcept(true), so this is fine

    bar();  // fine
    baz();  // compiles, but at runtime this calls std::terminate
}
```

Warum eigentlich nicht `nothrow`?

DEMO: noexcept

Teil 2: Neues in std

- tuple, array (*)
- unordered_[multi]map|set (*)
- forward_list (*)
- unique|shared_ptr (*)
- new algorithms (*)
- new set with const elements
- vector|deque|basic_string::shrink_to_fit
- function bind (*)
- regex (*)
- ratio
- chrono
- random (*)

(*) schon in std::tr1 (2003)

tuple

schon alles wichtige gesagt ...

```
tuple<A,B> != pair<A,B>
```

Umwandlung aber nahtlos möglich

unordered_[multi]map|set

Container für Typen ohne Ordungsrelation

basieren auf Hash-Funktionen

bisherige (sortierte) Varianten mit Operationen inO(log n)

diese im Idealfall O(1) aber keine Garantie: abhängig von der Hashfunktion

Elemente mit gleichen Hashwerten kommen in Bucket-Listen

Hash-Funktionen implizit (sicher möglichst gut - aber ohne Vorgaben im Standard) bereitgestellt, ansonsten als 2. Template-Parameter spezifizierbar:

```
struct bad_hash {
  size_t operator()(int i) const {return i%7;}
};
unoredered_set<int,bad_hash> uos;
```

forward_list

einfach (vorwärts) verkettete Listen, sparen einen Zeiger pro Knoten

aber dafür

- ohne `size()`, nur `empty()` abfragbar
- kein `insert()/erase()`, nur `insert_after()/erase_after()`, kein `push_back()` nur `push_front()`
- Iteratoren nur vorwärts bewegbar ++, nicht --
- `forward_list::sort()` in O(n log n)

array

Felder fester (unveränderlicher) Größe (ohne Allocator)

noch kompakter als `vector`

```
template <class T, size_t N > struct array;
```

aber

`array::swap(array<T,N>& other)` und

`::swap(array<T,N>& a, array<T,N>& b)` mit O(n)

ansonsten wie `vector` +

```
T * data();
const T * data() const;
```

unique_ptr

semantisch korrekter Ersatz für `auto_ptr` (depricated!)

`auto_ptr` konnten NICHT in Containern gehalten werden, `unique_ptr` wohl

Owning Pointers (exklusiver Besitz) mit move-Semantik, immer besser als 'blanke' Zeiger

einfachste „Smart Pointer" Klasse: `operator*()`, `operator->()`, `get()`, `release()`, `reset()`

```
#include <memory>
std::unique_ptr<int> factory(int i) {
   return std::unique_ptr(new int{i});
}
void give_up(std::unique_ptr<int> p) { // by_value: move
   std::cout<<*p<<std::endl;
   // automatic delete(&*p) !!!
}
```

unique_ptr

```
int main() {
   std::unique_ptr<int> p = factory(66); //*p == 66
   p.reset(new int{99}); // delete(&*p), *p==99
   give_up(factory(33));
   // give_up(p); ERROR rvalue required
   give_up(std::move(p)); // ok, p ~ nullptr
}
```

unique_ptr sehr gut für incomplete Types geeignet:

```
struct A; // incomplete – no details in header
class B {
   unique_ptr<A> pimpl_; // Pimple Pattern
public:
   B(); ~B();
   A& get() { return *pimpl_; }
};
```

unique_ptr

auch mit (C-)Feldern verträglich (anders als `shared_ptr` !) dann ohne ->:

```
int main() {
    std::unique_ptr<int[]> p {new int[3];}
    p[0] = 0;
    p[1] = 1;
    p[2] = 2;
    // automatic delete[] p.get();
}
```

bevorzugt benutzen (im Standard „vergessen“), vgl. make_shared:

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

so kommt man ganz ohne new aus

shared_ptr

echte *Smart Pointers* (geteilte Ressourcen, referenzgezählt)

beim Kopieren wird Zähler inkrementiert, wenn letzte Referenz verschwindet: automatische Freigabe

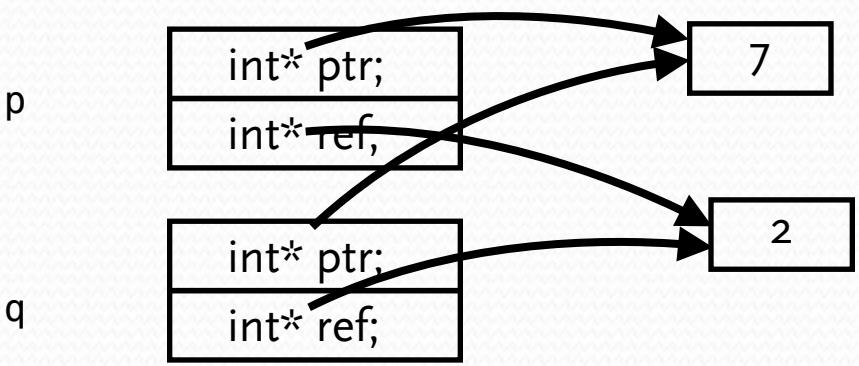
bevorzugt benutzen: make_shared (im Standard definiert!):

```
template<typename T, typename... Args>
std::shared_ptr<T> make_shared(Args&&... args)
```

so kommt man ganz ohne new aus

UND eine effizienztere Implementation ist möglich:

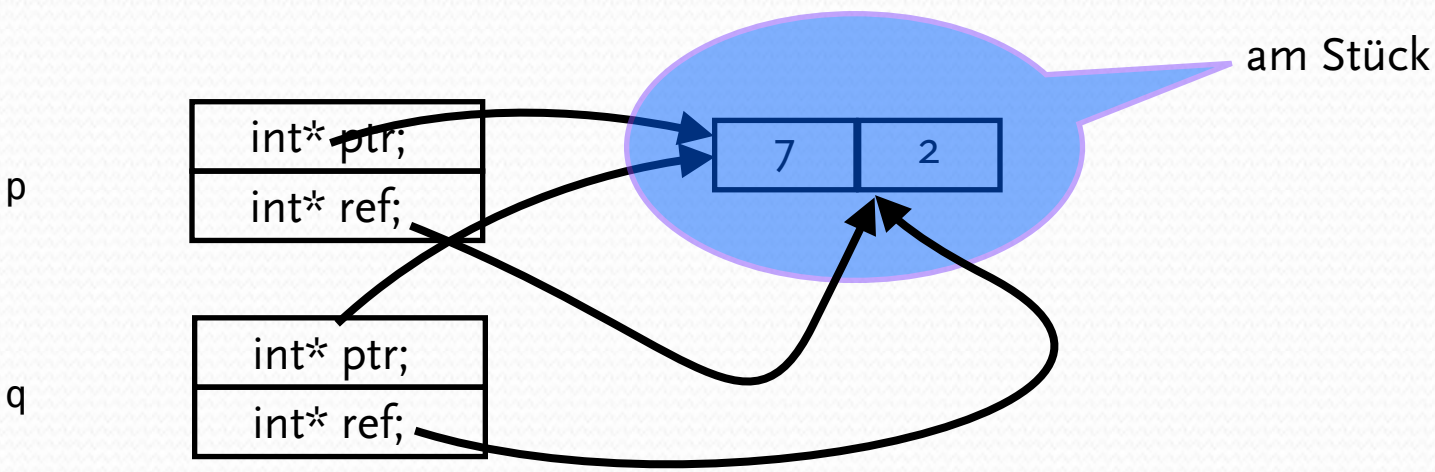
```
std::shared_ptr<int> p (new int{7};)
std::shared_prt<int> q (p); // copy! ref_count == 2
```



shared_ptr

UND eine effizienztere Implementation ist möglich:

```
auto p make_shared<int>(7);
auto q (p); // copy! ref_count == 2
```



weak_ptr

zyklische Strukturen von shared_ptr würden nie freigegeben werden:

es gibt auch noch ‚non-owning-pointers'

`weak_ptr` - nur Verweis ohne Referenzzählung !

dabei könnte das eigentliche Objekt aber schon freigegeben sein:

weak_ptr hat weder operator* noch operator->

stattdessen kann man explizit abfragen, ob das Objekt noch gültig ist:

```
auto sp = make_shared<int>(123);
std::weak_ptr<int> wp = sp;
if (!wp.expired()) {
    std::shared_ptr<int> alive (wp);
    // use *alive
}
// or
std::shared_ptr<int> maybe (wp); // throws bad_weak_ptr if (wp.expired())
// or
std::shared_ptr<int> maybe (wp.lock()); // nullptr if (wp.expired())
if (maybe)... // use *maybe
```

weak_ptr

```
#include <iostream>
#include <memory>

std::weak_ptr<int> gw;

void f()
{
    if (auto spt = gw.lock()) { // Has to be copied into a shared_ptr before usage
        std::cout << *spt << "\n";
    }
    else {
        std::cout << "gw is expired\n";
    }
}

int main()
{
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;

        f();
    }

    f();
}
```