range for statement

range for erlaubt es, über beliebige Bereiche zu iterieren:

alle Standard Container, std::string, initializer lists, array, (alles was begin() und end() hat).

```
void f(const vector<double>& v) {
  for (auto x : v) cout << x << '\n'; // copy of v[i]‘s
  for (auto& x : v) ++x; // v[i]‘s themselves
}

for (const auto x : { 1,2,3,5,8,13,21,34 })
  cout << x << '\n';
```

die Länge muss verfügbar sein:

```
void print(int nums[]){ for(auto i : nums) .... /*error*/ }
```

range for statement

auch für nutzerdefinierte Typen, wenn diese `begin()` und `end()` haben:

```
#include <iostream>

class X {
    int i[] = {1,2}; // what's that ?
public:
    int* begin(){return i;}
    int* end()  {return i+2;}
};

int main() {
    X x;
    for (auto i : x)
        std::cout<<i<<std::endl;
}
```

free-standing begin/end

`begin(v)` und `end(v)` auf allen ‚Bereichen'

für Container (generisch) auf `begin(v)` / `end(v)` abgebildet

für andere Typen explizit spezialisiert

falls noch nicht unterstützt, *work-around* leicht möglich:

```
// Container templates
template <typename T>
auto begin/end([const] T& t) -> decltype(t.begin/end()) {
       return t.begin/end();
}
```

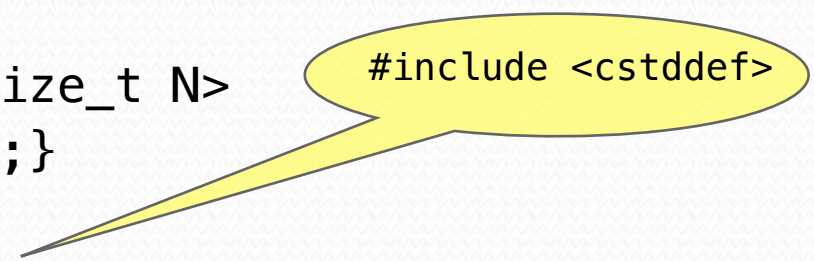
free-standing begin/end

falls noch nicht unterstützt, *work-around* leicht möglich

```
// C arrays
// NOT:
// template <typename T>
// T* begin(T* b) {return b;} // how to do end then ?

// BUT:
template <typename T, std::size_t N>
T* begin(T(&b)[N]) {return b;}

template <typename T, std::size_t N>
T* end(T(&b)[N]) {return b+N;}
```



cbegin/cend -- const ranges

`begin()` und `end()` gibt es schon immer in zwei Ausprägungen:

```
iterator container::begin ();
const_iterator container::begin () const;
```

Problem: auf nicht konstante Container nur lesend zugreifen?

```
std::vector<int> v = {2,3,5,7,11};
for(auto it = v.begin(), it!= v.end(), ++it) {
     read(*it);
     write(*it); // ok !
}
```

`... const auto it` ... hilft hier nicht! Warum nicht?

ein const iterator ist etwas anderes als ein const_iterator

cbegin/cend -- const ranges

Lösung: ist nur lesender Zugriff gewünscht, benutze man

`cbegin()` und `cend()` liefern immer const_iterator

gibt es auch in den reverse-Versionen `crbegin()` und `crend()`

```
for(auto it = v.cbegin(), it!= v.cend(), ++it) {
     read(*it); // ok!
     write(*it); // error !
}
```

defaulted and deleted functions

Das Idiom “Kopieren verboten” kann nun direkt ausgedrückt werden:

```
class X {
// ...
  X& operator=(const X&) = delete; // Disallow copying
  X(const X&) = delete;
};
```

Auch die Nutzung der vom Compiler implizit erzeugten *default members* kann spezifiziert (und modifiziert) werden. Dies sind in C++11:

- `A::A()` nur wenn keine eigenen Konstruktoren definiert sind
- `A::~A()` nur wenn kein eigener Destruktor definiert sind
- Kopieroperationen: `A::A(const A&)` und `A& A::operator=(const A&)` nur wenn keine Verschiebeoperationen definiert sind
- Verschiebeoperationen: `A::A(A&&)` und `A& A::operator=(A&&)` nur wenn keine Kopieroperationen definiert sind (siehe move - Semantik)

defaulted and deleted functions

Die vom Compiler implizit erzeugten *default members* können dabei modifiziert werden:

```
class Y {
  explicit Y() = default;  // + explicit
  Y& operator=(const Y&) = default;
  Y(const Y&)  = default;
  virtual ~Y() = default;  // + virtual
protected:
  Y(Y&&)       = default;  // + protected
};
```

defaulted and deleted functions

ACHTUNG: `default` kann auch `delete` bedeuten, wenn der Compiler die Implementation nicht bereitstellen darf:

```
class B {
public:
  B(const B&) = delete;
};

class D : public B {
    D(const D&) = default; // means delete
};
```

... oder Member, dito für Move-Konstruktoren, (Copy/Move)-Zuweisungen

defaulted and deleted functions

Auch zur Eliminierung unerwünschter Umwandlungen ...

```
struct Z {
// ...
  Z(long long); // can initialize with a long long
  Z(long) = delete; // but not anything less
};
```

... und Template-Spezialisierungen

```
template <typename ALL> foo(ALL a) { ... }
template <> foo(BUT b) = delete;
```

final and override

finale Klassen: keine Ableitung möglich

finale Methoden: keine Redefinition in Ableitungen

```
class X {
public:
    virtual void foo();
};

class Y final : public X {
public:
    virtual void foo() override {}
};

void call (Y* p)
{
    p->foo(); // can bind statically !
}

// class Z : public Y {}; // not possible
```

```
// seems to be a defect:

class Nonsens final {
public:
    virtual void pure() = 0;
};
```

final and override

finale Methoden müssen virtuell sein !

```
class Z {
    void virtual foo() const {}
};

class ZZ : public Z {
    void foo() const final override {};
} final, override;
```

sind **KEINE** neuen Schlüsselworte, sondern *kontext-sensitiv*

override: Fehler bei der Redefinition besser finden

```
class B {
public:
    virtual void bar() {}
};

class D: public B {
    //  void baz() override {}
    //  virtual void baz() override {}
    virtual void bar() override {}
};
```