Deklarationen

Deklarationen vereinbaren Namen in der aktuellen Übersetzungseinheit (ab dem Deklarationspunkt). Im Unterschied zu Definitionen wird *kein* Speicherplatz reserviert.

Zweck:

Vereinbarung später in der Übersetzungseinheit definierter Funktionen (z.B. bei wechselseitig rekursiven Funktionen).

Vereinbarung von Namen, die in einer anderen Übersetzungseinheit (z.B. einer Programmbibliothek) definiert sind.

Speicherklasse

Die genaue Wirkung einer Vereinbarung hängt auch von der Speicherklasse ab, die zu den Deklarationsangaben gehört.

Speicherklassen regeln v.a. zwei unterschiedliche Eigenschaften:

Dauer der Speicherreservierung bei lokalen Variablen

temporär („automatisch“)	nur während der Funktions- oder Blockausführung	Initialisierung bei jedem Funktionsaufruf oder Blockeintritt (nur falls explizit angegeben)
permanent („statisch“)	während der gesamten Programmausführung	Initialisierung nur beim Erstaufruf (implizit mit 0 oder explizit), Weiterverwendung des Werts beim erneuten Aufruf

Bindungswirkung von Namen

Programm („extern“)	Der Name bezieht sich innerhalb des Programms immer auf dieselbe Größe. Definition: Zugriff auf die Größe auch in anderen Übersetzungseinheiten möglich (sofern Name dort deklariert). Deklaration: Name bezieht sich auf Namen in einer anderen Übersetzungseinheit (dort Def. erforderlich)
Übersetzungseinheit („intern“)	Der Name bezieht sich innerhalb der Übersetzungseinheit immer auf dieselbe Größe. Definition: Zugriff auf die Größe nur in derselben Übersetzungseinheit möglich Deklaration: Name bezieht sich auf eine Definition in derselben Übersetzungseinheit
keine	Name bezieht sich nur innerhalb seines Gültigkeitsbereichs auf dieselbe Größe.

Leider haben in C++ die Speicherklassenangaben zum Teil unterschiedliche Bedeutungen, abhängig davon, ob sie auf Funktionen oder Variablen angewendet werden und ob diese sich außerhalb oder innerhalb von Funktionen befinden. (Variablen und Funktionen innerhalb von Klassen werden hier nicht betrachtet.)

Speicherklassenangabe	*Anwendungsbereich (Auszug)*	*Bedeutung*[2]
`extern`	Var./Fkt. außerhalb von Fkt. u. Kl.	externe Bindung, perm. Speich.
`static`	Var./Fkt. außerhalb von Fkt. u. Kl	interne Bindung, perm. Speich.
	Var. innerhalb von Fkt.	keine Bindung, perm. Speich.
`auto`[1]	Var. innerhalb von Fkt.	keine Bindung, temp. Speich.
`register`	Var. innerhalb von Fkt., Parameter	wie auto, Zugriffszeit minimieren

Anwendungsbereich	*Voreingestellte Speicherklasse*[2]
Funktionen	`extern`
Variablen außerhalb von Funktionen und Klassen	`extern`
Variablen innerhalb von Funktionen	`auto`

Definition und Deklaration globaler Namen und Namen in Namensräumen (Auszug)

Namen werden in C++ als *global*[3] bezeichnet, wenn Sie außerhalb von Funktionen und Klassen (auf der obersten Programmebene) vereinbart sind und *keinem* Namensraum angehören[4].
Der Gültigkeitsbereich des Namens erstreckt sich vom Vereinbarungspunkt bis zum Ende der Übersetzungseinheit *(file scope)*.
Gehören solche Namen zu einem Namensraum, so erstreckt sich der Gültigkeitsbereich ab dem Vereinbarungspunkts bis zum Ende des Namensraums inkl. der Namensräume, in die sie durch using-Direktiven eingeführt werden *(namespace scope)*.

Vereinbarung	*Bedeutung*[2]
`f(...){...}`	Funktionsdefinition mit externer Bindung
`extern f(...){...}`	Funktionsdefinition mit externer Bindung
`static f(...){...}`	Funktionsdefinition mit interner Bindung
`extern f(...);`	Funktions*deklaration* mit externer Bindung
`class C{...};`	Klassendefinition (externe Bindung)
`extern class C[{...}];`	Dekl. u. Def. verboten
`static class C[{...}];`	Dekl. u. Def. verboten
`T x`	Variablendefinition mit externer Bindung
`T x = a;`	Variablendefinition mit externer Bindung
`extern T x = a;`	Variablendefinition mit externer Bindung
`extern T x;`	Variablen*deklaration* mit externer Bindung
`static T x;`	Variablendefinition mit interner Bindung
`static T x = a;`	Variablendefinition mit interner Bindung
`const T x = a;`	Konstantendefinition (interne Bindung)
`static const T x = a;`	Konstantendefinition (interne Bindung)
`extern const T x = a;`	Konstantendefinition (externe Bindung)
`extern const T x;`	Konstanten*deklaration* (externe Bindung)

[1] nicht mehr ab C++11
[2] ohne unbenannte Namensräume
[3] Auch Namen mit interner Bindung werden als global bezeichnet, obwohl sie nur in der jeweiligen Übersetzungseinheit sichtbar sind.
[4] auch als globaler Namensraum bezeichnet

Lebensdauer und Initialisierung

Statische Variable (d.h. Variable außerhalb von Funktionen und Klassen auf oberster Ebene und `static`-Variable) haben permanente Lebensdauer, sie werden zu Programmbeginn initialisiert (per Default oder automatischem Konstruktoraufruf) und am Programmende freigegeben (ggf. per automatischem Destruktoraufruf).

Bsp.: Die ersten sieben Zeilen einer Programmdatei

```
#include <iostream>
using namespace std;
int main()
{
  cout << "moeglich";
  return 0;
}
```

Ausgabe:

```
Wie ist das moeglich?
```

Auflösung: Der Rest der Programmdatei

```
class seltsam {
  public:
     seltsam() { cout << "Wie ist das "; }
    ~seltsam() { cout << "?" << endl; }
} X;
```

`static`-Variablen in der Standardbibliothek (<cstring>)

Bsp.: Zerlegung einer Zeichenkette in Felder

```
#include <cstring>
char *strtok(char *s1, const char *s2)
```

Liefert jeweils einen Zeiger auf die nächste Teilzeichenkette von *s1*, die kein Zeichen von *s2* enthält, oder `0`. Beim ersten Aufruf ist für *s1* die zu untersuchende Zeichenkette einzusetzen, bei allen weiteren Aufrufen `0`.

Programm:

```
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
  char *wort, zk[]=" wort1 :wort2::: :wort3  wort4:wort5";
  wort = strtok(zk," :");
  while ( wort != 0 ) {
    cout << '|' << wort << '|' << endl;
    wort = strtok(0," :");
  }
  return 0;
}
```

Ergebnis:

```
|wort1|
|wort2|
|wort3|
|wort4|
|wort5|
```

Vorwärts- und Frienddeklarationen von Klassen

Vorwärtsdeklarationen ermöglichen die wechselseitige Benutzung von Klassen. Die Deklaration `friend class` *B* innerhalb einer Klasse *C* macht alle Komponenten von *C* für *B* zugreifbar. Mit `friend` *T B::comp(...)* bzw. `friend` *B::B(...)* in *C* werden einzelne Komponentenfunktionen oder Konstruktoren von *B* als mit *C* befreundet erklärt.

Bsp. aus dem C++-Standard:

```
class Vector;

class Matrix {
  // ...
  friend Vector operator*(Matrix&, Vector&);
};

class Vector {
  // ...
  friend Vector operator*(Matrix&, Vector&);
};
```

inline-Funktionen

Die Deklarationsangabe `inline` kann bewirken, daß beim Übersetzen statt eines Funktionsaufrufs wirkungsäquivalente Anweisungen eingefügt werden.

Bsp.:

```
          :
inline double sqr(double x)
{ return x*x; }

int main()
{
  double x,y,z;
          :
  z = sqr(x+y);
          :
```

Der Compiler könnte im Hauptprogramm den Funktionsaufruf `z = sqr(x+y)` durch `double tmp; z = ((tmp=x+y),(tmp*tmp))` ersetzen.

Inline-Funktionen außerhalb von Klassen haben nach Voreinstellung interne Bindung, d.h. sie sind nur innerhalb ihrer Übersetzungseinheit bekannt. Sind sie als extern deklariert, so müssen sie in allen Übersetzungseinheiten, in denen sie benutzt werden, identisch *definiert* werden.

Funktionen, die in Klassen *definiert* sind, sind automatisch Inline-Funktionen.