Kapitel 4

Qualifier

4.1 Qualifier static und extern

static	extern
Nur den Programmteilen bekannt, die die Definition sehen.	Allen Programmteilen bekannt, die dazugelinkt werden.
Standard für Variablen	Standard für Funktionen
`static int abs(int i);`	`extern node_array xpos;`
besser: private Funktion einer Klasse	besser: als Parameter Übergeben

Tabelle 4.1: static und extern in C/C++

So ähnlich wie es private- und public-Funktionen gibt, die außerhalb der Klasse unsichtbar bzw. sichtbar sind, so kann man auch Funktionen unterscheiden, die außerhalb der Quelldatei oder genauer gesagt der Objektdatei unsichtbar bzw. sichtbar sind. Beim Linken ist es also nicht möglich, die „unsichtbaren“ Funktionen zu verwenden. Unglücklicherweise nennt man diese dann aber nicht `private` und `public` sondern `static` und `extern`. Gesagtes gilt übrigens mutatis mutandis für Variablen. Eine Übersicht bietet Tabelle 4.1.

Wie bei vielen Dingen, ist dies ein Relikt von C, für die es in C++ meist die bessere Alternative der Kapselung durch Objekte gibt. Statische Funktionen sind daher nahezu überflüssig geworden. Wenn eine Klasse aber nur aus einer Funktion bestünde, dann erscheint es wenig sinnvoll, diese in eine Klasse zu packen. Statische Variablen sortiert man aber besser zu den zugehörigen Funktionen in ein Objekt. Externe Variablen sind allerdings ganz schlechter Stil. Man sollte ihren Inhalt besser als Parameter übergeben.

4.2 static – eine andere Bedeutung

Besonders unglücklich ist die Namensgebung für `static` in Abschnitt 4.1, da es auch noch eine weitere Bedeutung hat, die von Java bereits bekannt sein sollte:

Beispiel:

```
class Vektor {
    ⋮
  static int maximalLength;
  static Vektor generateRandom();
    ⋮
}
```

So eine Variable existiert nur einmal pro Klasse (und nicht pro Instanz) und so eine Funktion ist unabhängig von der Instanz.

4.3 Qualifier inline

Zusätzlich zum Erbe `static` und `extern` aus C gibt es in C++ noch einen Qualifier, der sich aber als sehr nützlich erweist. Die Rede ist von `inline`:

Beispiel

```
inline int abs(int i)
{ return i<0 ? -i : i; }
```

`inline` ist ein Hinweis an den Kompiler, den Code direkt einzufügen anstatt einen wirklichen Funktionsaufruf zu machen. Das spart einen Funktionsaufruf (welcher Zeit kostet), vergrößert aber (eventuell) das Programm. Selbstverständlich ist das Verhalten das gleiche, als wenn eine Funktion aufgerufen wird. Der Kompiler muss sich allerdings nicht an den Hinweis halten. (Was wir aber nie feststellen werden, wenn wir nicht in den erzeugten Assembler-Code reinschauen). Eine `inline`-Funktion muss natürlich automatisch `static` sein, weil eventuell gar keine Funktion generiert wird. Daher ist eine `inline`-Funktion automatisch `static`.

Darüberhinaus gibt es noch einen weiteren Automatismus. Wenn eine Funktion in einer Klasse nicht nur deklariert, sondern auch definiert wird, dann ist sie automatisch inline (und damit static). Das Beispiel `vektor.h` aus Abschnitt 3.1 ist daher auch in seiner ursprünglichen Form korrekt:

```
class Vektor {
private:
  double myX,myY;
public:
  Vektor(double x, double y):myX(x),myY(y) {};

Vektor addiere(Vektor v)
  { return Vektor(myX+v.myX,myY+v.myY); }
};
```

Für derart kurze Funktionen macht dies durchaus Sinn. Die Funktionen in eine cc-Datei auszulagern war insofern ein etwas akademisches Beispiel.

4.4 Aufgaben

1. Schreiben Sie eine statische Funktion, die das Maximum zweier Werte zurückgibt. Muss die Definition der Funktion in der Header- oder in der cc-Datei stehen, wenn man sie in zwei verschiedenen cc-Dateien benutzen will? Probieren Sie aus, ob ihre Überlegung stimmt.

2. Deklarieren Sie ihre Funktion als `inline` und untersuchen Sie, ob das einen Unterschied macht.