Deklaration oder Definition im Objekttyp

Wie bisher kann man Methoden und Freund-Funktionen innerhalb der Objektdefinition definieren (= man programmiert sie komplett mit dem ganzen Code in `{...}`) oder nur deklarieren. Eine Deklaration legt nur die Signatur der Funktion fest, enthält keinen Code und endet mit einem Strichpunkt:

```
struct Komplexe_Zahl) {
     double re, im;            // Datenattribute
     double abs() const;       // konstante abs()-Methode wird deklariert
     double arg()  const       // konstante arg()-Methode wir definiert
     { return atan2(im, re); }      // kein Strichpunkt
     friend print(Komplexe_Zahl);       // friend–Deklaration
     friend double real_part(Komplexe_Zahl z)
     { return z.re; }          // friend–Definition: Hidden friend
     ...
};
```

Steht nur eine Deklaration in der Typdefinition, muss die Methode/Funktion später in derselben Datei oder in einer anderen definiert werden. Eine Definition darf nur einmal im gesamten Programm enthalten sein (ODR), Deklarationen können beliebig oft eingefügt werden, solange sie konsistent sind.

Vorteile der Definition von Methoden/Freunden in der Typdefinition:

1) Man findet den Code gleich an Ort und Stelle.
2) Die Methoden werden automatisch `inline`, d.h. der Compiler kann sie besonders schnell aufrufen bzw. den Aufruf sogar wegoptimieren.
3) Alles steht an einer einzigen Stelle. Deshalb sind Änderungen einfacher zu machen.
4) Manche Methoden haben sogar einen leeren Funktionsrumpf. Warum also woanders Platz dafür verschwenden.
5) Die Definition von Methoden ist einfacher (man muss z.B. keinen Scope angeben).
6) Es gibt riesige Vorteile, wenn es sich um eine Template-Typdefinition handelt. Wer schon einmal eine korrekte `friend`-**Deklaration** geschafft ist, ist ein Meister seines Faches. Hingegen sind `friend`-**Definitionen** auch leicht zu erstellen.
7) Wird eine `friend`-Funktion innerhalb eines Objekttyps **definiert**, gilt sie als „**hidden friend**“ und sie wird nur mehr für solche Objekte herangezogen. Das ist oft nützlich, wenn man den Funktionsnamen oft überladen hat: z.B. die Streamausgabe `operator<<()`. Schreibt man diesen Operator als hidden friend, so wird er nicht für andere Objekttypen in Erwägung gezogen. Das beschleunigt das Kompilieren, spart sinnlose Fehlermeldungen ein und wird derzeit - im Gegensatz zu früher - sogar empfohlen:

globaler `friend`, wird daher für alle Objekte in Betracht gezogen:

```
struct A {…
     friend ostream& operator<<(ostream& , A);// Deklaration
};

ostream& operator<<(ostream& os , A x)  // Definition außerhalb
{     …  }
```

hidden `friend`, wird nur für **`B`** in Betracht gezogen:

```
struct B { …
     friend ostream& operator<<(ostream& os , B x)
     {     … )                              // Definition innerhalb
};
…
cout << 3; // egal was
```

C++ erkennt den Aufruf von `operator<<(ostream&, int)`. Es muss jetzt

1) alle bekannten `operator<<()` suchen (findet auch den für `A`, der global ist, aber nicht den von `B`, weil der im Scope `B` liegt.
2) Alle unpassenden ausscheiden
3) den am besten passenden des Rests bestimmen

Jede unpassende globale Streamausgabe (wie von `A)` schafft es immer in Stufe 1 und manchmal auch in Stufe 2 von oben und verlangsamt das Kompilieren.

```
cout << B{…};
```

Hier wird die Streamausgabe im ersten Durchlauf gar nicht gefunden (da im Scope `B`), allerdings macht C++ weitere Suchläufe in den Scopes aller Argumenttypen (hier `std::ostream` und `B`) = ADL (Argument Dependend Lookup) und findet daher die richtige Streamausgabe für `B`.

Nachteile der Definition von Methoden/Freunden innerhalb der Typdefinition:

1) Das galt früher als unelegant oder sogar hässlich.
2) Es kann die Typdefinition durch lange Codeblöcke unübersichtlich machen.
3) Sie gestatten nicht die Aufteilung in eine Headerdatei `.h` und eine Code-Datei `.cpp` Dieser Nachteil ist nicht gegeben, wenn man überwiegend Templates programmiert, die ohnehin in der Headerdatei stehen müssen.

Beim Erstellen eines neuen Typs incl. Methoden wird man sicher mit inline-Definitionen beginnen, weil die Vorteile überwiegen. Die Trennung in Deklaration/Definition macht man, wenn alles funktioniert (oder nie :-)