

Erklärung zum Programm „potenzieren“

Erst der Quelltext:

```
#include <iostream>
#include <conio.h>

using namespace std;

int potenzieren(int x, int y)
{
    int erg = x;
    if(y==0) return 1;
    if (y==1) return x;

    for(int i = 1; i<y; i++)
    {
        erg = erg * x;
    }
    return erg;
}
```

```
main()
{
    cout << "x: ";
    int x, y;

    cin >> x;
    cout << "y: ";
    cin >> y;
```

```
int ergebnis = potenzieren(x,y);
```

```
cout << ergebnis;
getch();
}
```

Wir beginnen in der main-Funktion:

Sagen wir, die Basis ist x und die Potenz ist y.

Der Benutzer soll natürlich die Möglichkeit haben, sowohl Basis als auch Exponent anzugeben. Damit er weiß, dass er die Basis eingeben soll, kommunizieren wir über die Konsole mit dem Benutzer durch die Zeile

```
cout << "x: ";
```

Nun muss das, was der Benutzer eingeben soll, irgendwo gespeichert werden. Natürlich in einer Variable. Wir legen also die Variable x an (und wenn wir schonmal dabei sind, dann auch gleich y).

```
int x, y;
```

Auf die Eingabe des Benutzers für den x-Wert warten wir in der nächsten Zeile:

```
cin >> x;
```

Das gleiche passiert jetzt mit dem y-Wert:

Dem Benutzer sagen, dass er den y-Wert eingeben soll:

```
cout << "y: ";
```

Auf Eingabe des Benutzers warten und Eingabe in die Variable y speichern:

```
cin >> y;
```

Nun haben wir Basis und Exponent, nämlich in den Werten x (Basis) und y (Exponent) abgespeichert. Wir können also jetzt die Funktion aufrufen. Wenn wir der Funktion sagen: "Berechne mal eine Potenz für uns", dann würde sie völlig zurecht sagen: "Kein Problem, dazu bin ich ja da. Aber ich bräuchte dann auch zwei Werte, mit denen ich rechnen soll. Übergib mir doch bitte die Basis und den Exponenten, dann sag ich Dir das Ergebnis".

Die Funktion erwartet also zwei sogenannte "Parameter", also zwei Werte, die wir ihm übergeben.

Klar, wir übergeben ihm beim Aufruf die beiden Werte, die er braucht, um x hoch y zu rechnen - also übergeben wir ihm x und y.

Da wir mit einem Ergebnis rechnen, das uns die Funktion ja zurückgeben soll, wollen wir dieses Ergebnis ja auch irgendwo speichern, damit wir den Wert jederzeit aufrufen können.

Eine Funktion, die etwas zurückgibt, gibt das, was sie zurückgibt, genau an die Stelle zurück, wo sie aufgerufen wurde.

Die Zeile

```
int ergebnis = potenzieren(x,y);
```

bedeutet also:

"ich lege eine Variable vom Typ int an und nenne sie ergebnis. In diese Variable möchte ich den Wert speichern, den mir die Funktion "potenzieren" zurückliefert, wenn ich ihr die Werte x als Basis und y als Exponent übergebe".

Das Programm springt, sobald eine Funktion aufgerufen wird, in die aufgerufene Funktion und macht dort weiter.

Also geht es jetzt in der Funktion int potenzieren(int x, int y) weiter:

```
int potenzieren(int x, int y)
```

Eine Funktion, die einen Wert "zurückgibt", also ein Ergebnis liefert, muss dies deutlich machen, in dem sie den Rückgabebetyp ganz zu Beginn angibt. Da bei Potenzberechnungen mit ganzen Zahlen ohnehin nur ganze Zahlen rauskommen rauskommen können, gibt die Funktion also eine Ganzzahl, also eine Zahl vom Typ int zurück.

`int potenzieren(int x, int y)`

Irgendwie muss die Funktion ja heißen. Mit diesem Namen wird sie aufgerufen...

`int potenzieren(int x, int y)`

Wie bereits erwähnt, muss eine Funktion auch angeben, was sie braucht, um ihre Aufgabe zu erfüllen. Eine Funktion, die eine Potenz errechnen soll, braucht natürlich den Basis- und den Potenzwert, also zwei Werte. Beide sollen ganzzahlig sein, also vom Typ `int`. Das muss mit angegeben werden. Diese beiden Werte heißen "Parameter". Wer diese Funktion aufruft, muss also in die Klammer zwei Werte (mit einem Komma getrennt) schreiben.

Das Ergebnis, das die Funktion zurückgeben soll, speichern wir in einer Variablen, die wir erstmal anlegen. Wir weisen dieser Variablen gleich mal den Anfangswert zu (wir werden später sehen, wozu das so sinnvoll ist).

```
int erg = x;
```

Stellen wir uns vor, der Exponent wäre 0 gewesen (also $y = 0$). Egal, was den Exponenten 0 hat, es kommt immer 1 raus (x hoch 0 gleich 1).

Wenn der Exponent 1 ist, dann kommt x selbst raus (klar, denn x hoch 1 = 1).

Deswegen die folgenden zwei Zeilen:

```
if(y==0) return 1;
if (y==1) return x;
```

Wenn y also 0 ist, dann soll er als Wert die 1 zurückgeben.

Wenn $y = 1$ ist (hier mit `==` dargestellt, als Vergleichsoperator, denn `=` wäre ja der Zuweisungsoperator), dann soll er den x -Wert zurückgeben.

Mit diesen zwei Zeilen haben wir den Fall für den Exponenten 0 und 1 behandelt (bei `return` ist die Funktion beendet).

Für den Fall, dass der Exponent aber größer ist als 1 wird die `return`-Anweisung ja nicht ausgeführt und es geht "weiter im Programm".

Jetzt müssen wir uns folgendes überlegen:

Wenn der Exponent gleich 2 ist, dann muss x EINMAL mit sich selbst multipliziert werden (also $x*x$).

Wenn er 3 ist, dann muss x ZWEIMAL (also $x*x*x$) mit sich selbst multipliziert werden.

Also immer einmal weniger als der Exponent groß ist.

Diese Beobachtung eröffnet zwei sinnvolle Varianten für die `for`-Schleife:

Wir lassen die Schleifenvariable bei $i = 1$ beginnen.

In die `for`-Schleife wird das Programm eh nur gehen, wenn y (also der Exponent) mindestens 2 ist, denn der Fall für $y == 0$ oder $y == 1$ ist ja bereits vorher abgefangen.

Wenn i zu Beginn 1 ist und y 2, dann soll er das, was im Schleifenkörper steht, EINMAL ausführen. Also darf die Bedingung nach der ersten Ausführung nicht mehr stimmen. Deshalb schreiben wir bei der Bedingung, dass er das ganze so lange machen soll, wie i kleiner als y ist (was, wie jetzt mehrmals angedeutet, mindestens einmal der Fall ist).

Wenn i nach dem ersten Durchlauf um eins erhöht wird, jetzt also 2 ist, dann stimmt die Bedingung, dass i kleiner als y sein soll nicht mehr.

Und es wurde genau einmal das x mit sich selbst multipliziert.

Kommen wir zur alles entscheidenden Zeile:

```
erg = erg * x;
```

Diese Zeile ist der Grund, warum wir der vorher angelegten Variable als Anfangswert x zugewiesen haben.

Was genau soll bei der Durchführung dieser Zeile passieren?

Der bisherige Wert, der sich hinter erg befindet (zu Anfang x) soll durch einen neuen überschrieben werden. Deswegen steht er links vom Gleichzeichen, denn ihm soll jetzt etwas zugewiesen werden.

Was soll ihm zugewiesen werden?

Das, was momentan in erg gespeichert ist (also zu Beginn der x-Wert) soll mit x multipliziert werden. Deswegen also

```
erg * x
```

Denn dann haben wir ja x^2 .

In erg ist nach dem ersten Durchlauf also x^2 gespeichert.

Wenn wir das bei einem weiteren Durchlauf noch einmal machen, wird aus x^2 ein x^3 und so weiter...

Die Klammer der for-Schleife wird geschlossen.

Was der Funktion noch übrig bleibt ist natürlich die Rückgabe des errechneten Wertes. Etwas zurückgeben ist in der Programmiersprache **return**.

Was soll er zurückgeben?

Das, was in erg gespeichert ist, also erg.

Deswegen steht da

```
return erg;
```

Somit springt das Programm zurück in die main-Funktion.

Das, was die Funktion zurückgegeben hat, wird in der int-Variable ergebnis gespeichert.

Diesen Wert können wir jetzt also noch ganz einfach ausgeben lassen:

```
cout << ergebnis;
```

Kleine "Rumprobieraufgabe":

Wie müsste sich die Schleife ändern, wenn wir eine andere Variante benutzt hätten, also nicht

```
for(int i = 1; ...;...)
```

sondern

```
for(int i = 2; ...;...)
```

Ich hoffe, diese Aufgabe ist für euch so billig, dass ihr sie gar nicht erst anfangen wollt. Falls doch: Viel Erfolg!!!