Grundbegriffe aus der Kombinatorik

Fakultät

Für $n \in \mathbb{N}$ definieren wir: $n! = \prod_{k=1}^{n} k = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.

Binomial-Koeffizienten

Für $n, k \in \mathbb{N}$ definieren wir: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

Beobachtung: $\binom{n}{k}$ ist die Zahl der Möglichkeiten, aus n verschiedenen Elementen genau k Elemente auszuwählen.

Fibonacci-Zahlen

Definiere $(F_n)_{n \in \mathbb{N}}$ über $F_0 = 0$, $F_1 = 1$ und $F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$. F_n ist dann die n-te Fibonacci-Zahl.

Funktionen

Eine Funktionsdefinition hat folgende Form:

```
RÜCKGABETYP FUNKTIONSNAME( PARMTYP 1 PARMNAME 1,
                           PARMTYP 2 PARMNAME 2,
                           ...,
                           PARMTYP n PARMNAME n )
{
  BEFEHLE
}
```

Aufruf einer Funktion:

```
FUNKTIONSNAME ( PARM 1, ..., PARM n )
```

Rückgabe:

```
return RÜCKGABEWERT;
```

Funktionen

Beispiel:

```
1  #include <iostream>
2
3  int absolut(int x)
4  {
5     if (x < 0)
6     {
7        return -x;
8     }
9     return x;
10 }
11
12 int main()
13 {
14    int n = 0;
15
16    std::cout << "Bitte eine ganze Zahl: " << std::endl;
17    std::cin >> n;
18    std::cout << "Der Absolutwert ist: " << absolut(n) << std::endl;
19    return 0;
20 }
```

Funktionen

- Funktionen können selbst wieder andere Funktionen aufrufen.
- Funktionen können auch sich selbst aufrufen (Rekursion).
- Wenn eine Funktion nichts zurückgeben soll, kann man ihr den Rückgabetyp `void` geben.

Funktionen

Beispiel: Was ist die Ausgabe dieses Programms?

```
1  #include <iostream>
2
3  int addiere_37(int x)  // Eine wichtige Funktion
4  {
5     x = x + 37;
6     std::cout <<  "Wert von x ist " << x << std::endl;
7     return x;
8  }
9
10 int main()
11 {
12    int a = 42;
13    int result;
14
15    result = addiere_37(a);
16    std::cout <<  "Wert von a ist " << a << std::endl;
17    std::cout <<  "Wert von result ist " << result << std::endl;
18    return 0;
19 }
```

Funktionen

Die Ausgabe ist:

```
Wert von x ist 79
Wert von a ist 42
Wert von result ist 79
```

Der Wert von `a` wird nicht verändert.

Denn: Nicht die Variable `a` wird übergeben, sondern nur ihr *Wert* (der dazu irgendwo anders hin gespeichert wird).
⇒ Call by Value

Anders wäre es, wenn die Funktion so definiert wäre:

```
1 int addiere_37(int &x)   // Eine wichtige Funktion
2 {
3    x = x + 37;
4    std::cout <<  "Wert von x ist " << x << std::endl;
5    return x;
6 }
```

⇒ Call by Reference

Funktionen

Eine Funktions*deklaration* hat folgende Form:

```
RÜCKGABETYP FUNKTIONSNAME( PARMTYP 1 PARMNAME 1,
                           PARMTYP 2 PARMNAME 2,
                           ...,
                           PARMTYP n PARMNAME n );
```

- Kann z.B. verwendet werden, wenn zwei Funktionen sich gegenseitig aufrufen sollen.
- Ebenfalls nützlich bei der Aufteilung des C++-Codes auf mehrere Dateien.

Modularisierung

- C++-Programme können in mehrere Dateien aufgeteilt werden.
- Die einzelnen `.cpp`-Dateien werden dann separat kompiliert (aber in einem Aufruf) und erst später durch einen *Linker* verbunden.
- Schnittstellen zwischen den einzelnen Modulen werden über Header-Dateien (typischerweise mit der Endung `.h`) definiert.

Beispiel

```
1  #include <iostream>
2
3  double maximum(double a, double b)
4  {
5     if(a > b)
6     {
7        return a;
8     }
9     return b;
10 }
11
12 double durchschnitt(double a, double b)
13 {
14    return 0.5 * (a + b);
15 }
16
17 int main()
18 {
19    double a = 0.0, b = 0.0;
20    std::cin >> a >> b;
21
22    std::cout << "Maximum  ist " << maximum(a,b) << std::endl;
23    std::cout << "Durchschnitt ist " << durchschnitt(a,b) << std::endl;
24    return 0;
25 }
```

Beispiel: In Module ausgelagerte Funktionen

Datei: `rechnen.h`

```
1 double maximum(double a, double b);
2 double durchschnitt(double a, double b);
```

Datei: `rechnen.cpp`

```
1  #include "rechnen.h"
2
3  double maximum(double a, double b)
4  {
5     if(a > b)
6     {
7        return a;
8     }
9     return b;
10 }
11
12 double durchschnitt(double a, double b)
13 {
14    return 0.5 * (a + b);
15 }
```

Beispiel

Dann ist folgendes `main.cpp` möglich:

```
1  #include <iostream>
2  #include "rechnen.h"
3
4  int main()
5  {
6     double a = 0.0, b = 0.0;
7     std::cin >> a >> b;
8
9     std::cout << "Maximum  ist " << maximum(a,b) << std::endl;
10    std::cout << "Durchschnitt ist " << durchschnitt(a,b) << std::endl;
11
12    return 0;
13 }
```

Compiler-Aufruf:

```
g++ -std=c++11 -Wall -Wpedantic rechnen.cpp main.cpp
```

Der Präprozessor

- Vor dem eigentlichen Compiler läuft der Präprozessor, der das Einbinden von Code aus anderen Dateien regelt.
- Kommandos für den Präprozessor beginnen mit `#` (z.B. `#include <iostream>`).
- Mit
  `#define MACRO REPLACE`
  kann man dem Präprozessor mitteilen, dass `MACRO` durch `REPLACE` ersetzt werden soll.
- If-Abfrage sind möglich, z.B.:

```
#if !defined(MACRO)
...
#endif
```

Der Präprozessor

⇒ Der folgende Code sorgt dafür, dass die Header-Datei `rechnen.h` nur einmmal eingebunden wird:

```
1 #if !defined(RECHNEN_H)
2 #define RECHNEN_H
3
4 double maximum(double a, double b);
5 double durchschnitt(double a, double b);
6
7 #endif
```