Software-Technik: Vom Programmierer zur erfolgreichen …

1. Von der Idee zur Software
2. Funktionen und Datenstrukturen
3. Organisation des Quellcodes
4. Werte- und Referenzsemantik
5. Entwurf von Algorithmen
6. Fehlersuche und –behandlung
7. Software-Entwicklung im Team
8. Abstrakte Datentypen: Einheit von Daten und Funktionalität
9. Vielgestaltigkeit (Polymorphie)
10. Entwurfsprinzipien für Software

Anhang A: Die Familie der C-Sprachen

Anhang B: Grundlagen der C++ und der Java-Programmierung

Software-Technik: Vom Programmierer zur erfolgreichen …

2. Funktionen und Datenstrukturen

2.1 Funktionale Abstraktion (Funktionen)
2.2 Funktionale Abstraktion bei der Netzplanung
2.3 Datenabstraktion: Strukturierte Datentypen
2.4 Generische Programmierung, 1. Teil
2.5 Zusammenfassung

Lehrbuch: 1.4
Kompendium, 3. Auflage: 1.7
Kompendium, 4. Auflage: 1.5

Folien mit gelben Punkten ● am oberen rechten Rand sind weniger wichtig für das Verständnis der nachfolgenden Kapitel.

Funktionen (Beispiel)

```
void summe();
```

Eine Funktion hat

- einen Namen (summe),
- einen Rückgabetyp (hier void (keine Rückgabe)) und
- optional Argumente bestimmten Typs (hier keine).

Dieses ist die Deklaration einer Funktion (der Prototyp).
Eine Funktion muss jedoch auch **definiert** werden.

Funktionen (Beispiel)

```
void summe() {
   int s=0;
   for (int i=1; i<101; ++i) {
      s += i;
     }
cout<<"Summe ist"<< s << "\n";
}
```

```
public class Mathe {
   public static void summe() {
   int s = 0;
   for (int i=1; i<101; ++i) {
      s += i;
      }
   System.out.println("Summe ist "+s);
   };
};
```

Funktionsaufruf

Ein Funktionsaufruf bewirkt, dass die Programmsteuerung an die aufgerufene Funktion übergeben wird; die Ausführung der momentan aktiven Funktion wird unterbrochen. Nach dem Abschluss der Auswertung der aufgerufenen Funktion wird die Ausführung der ausgesetzten Funktion direkt hinter dem Aufruf fortgesetzt.

Echte Funktionen und Anweisungs-Funktionen

```
int summeP(int a, int b) {
   int s=0;
   for (int i=a; i<b+1; ++i) {
     s += i;
     }
   return s;
}
```

C++

Beispiel für eine
Echte Funktion:

```
public static int summeP(int a, int b) {
   int s = 0;
   for (int i=a; i<b+1; ++i) {
     s += i;
     }
   return s;
};
```

Java

Echte Funktionen und Anweisungs-Funktionen (2)

```
void summeR(int a, int b, int& s)
{
   s = 0;
   for (int i=a; i<b+1; ++i) {
     s += i;
    }
}
```

C++

Beispiel für eine ***Anweisungs-Funktion***:

```
public static void summeR(
                          int a, int b, int s[]) {
   s[0] = 0;
   for (int i=a; i<b+1; ++i) {
      s[0] += i;
      }
}
```

Java

Aufbau von Funktionen

Allgemeine Form, vereinfacht:

Rückgabetyp Funktionsname **(** *Typ1 Param1, Typ2 Param2 ...* **)** **{** . . . **}**

Funktionen ohne Parameter sind auch möglich:

int funk() oder **int funk(void)**

Werte- und Referenzparameter

```
void summeR(int a, int b, int& s)
{
   s = 0;
   for (int i=a; i<b+1; ++i) {
     s += i;
    }
}
```

C++

```
public static void summeR(
                        int a, int b, int s[]) {
   s[0] = 0;
   for (int i=a; i<b+1; ++i) {
      s[0] += i;
      }
}
```

Java

Für Werteparameter dürfen Konstanten, Werte, Ausdrücke oder Variablen als aktuelle Parameter eingesetzt werden. Dagegen müssen für Referenzparameter immer Variablen übergeben werden, denn die Referenzparameter werden lediglich als anderer Name für die angegebenen Variablen verwendet.

Werte- und Referenzparameter (2)

Der folgende Codeausschnitt zeigt die Übergabe von aktuellen Parametern an Referenzparameter. Der Wert der Variablen „e“ wird durch Verwendung der Anweisungsfunktion berechnet.

```
int st=22;
int f;
summeR(st, 8, f);
e = f;
summeR(st, 9, f);
e += f;
```

C++

```
int st=22;
int f[] = {8};
Mathe.summeR(st, 8, f);
e = f[0]
Mathe.summeR(st, 9, f);
e += f[0];
```

Java

Clicker-“Abstimmung“

```
void funk1(int a, int b) {
 a = 33; b = 66;
}
void funk2(int& a, int& b) {
  a = 33; b = 66;
}
```

```
void test () {
int i=10, j=9;
funk1(i, j); cout << i << " " << j << " " ;
i=10; j=9;
funk2(i, j); cout << i << " " << j;
}
```

Was wird auf dem Bildschirm ausgegeben?

1. 10 9 10 9
2. 10 9 33 66
3. 33 66 10 9
4. 33 66 33 66

Prototypen

Java und C++-Programme werden von einem Compiler erst auf korrekte Syntax geprüft, bevor sie übersetzt und damit ausführbar werden. In Java ist jede Funktion, wurde sie erst einmal *definiert* (d.h. implementiert), grundsätzlich überall bekannt.

In C++ muss entweder die Definition *vor* dem Aufruf erfolgen oder der Funktionskopf, bestehend aus Rückgabetyp, Funktionsname und formalen Parametern, wird in Form eines sog. *Prototypen* vorher *deklariert*.

```
void swapWert(int a, int b);
void swapRef(int& a, int& b);
void testRef() {
int i=10, j=9;
swapWert(i, j); cout << i << " " << j;
swapRef(i, j); cout << i << " " << j;
}
```

C++

```
void swapWert(int a, int b) {
   int tmp=a; a=b; b=tmp;
}
void swapRef(int& a, int& b) {
   int tmp=a; a=b; b=tmp;
}
```

Default-Parameter von Funktionen

C++ bietet die Möglichkeit der **Default-Werte** für Funktionsparameter:

```
double Summe(double x1, double x2, double x3=0.0, double x4=0.0)
{
  return x1 + x2 + x3 + x4;
}
. . .
y1 = Summe(1.1, 2.2, 3.3, 4.4);  // i.O.: y1 == 11.0
y2 = Summe(1.1, 2.2, 3.3);       // i.O.: y2 ==  6.6
y3 = Summe(1.1, 2.2);            // i.O.: y3 ==  3.3
```

Beim Weglassen von aktuellen Parametern geht die Reihenfolge stets von rechts nach links, dadurch wird **y2 = Summe(1.1, 2.2, 3.3);** eindeutig: der dritte Parameter (von links gezählt) hat den Wert **3.3**, und der vierte hat den Wert **0.0**.

Clicker-“Abstimmung“

```
int min(int b=-1, int c=0) {
   if (b<c) { return b;}
   else {return c;}
}
…
cout << min(4,5) << " " min(4) <<
" " min() << " " min(-4) ;
```

Was wird auf dem Bildschirm ausgegeben?

1. 5 4 0 -4
2. 4 0 -1 -4
3. 4 0 0 0
4. 4 5 -1 -4

Überladen von Funktionen und Methoden

Eine Funktion wird in C++ nicht allein über ihren Namen identifiziert, sondern auch über die Anzahl und den Typ der Argumente, ihrer sog. *Signatur*:

```
int Mult(int a, int b);                 // 1
int Mult(int a, int b, int c);                  // 2
int Mult(double a, double b);           // 3
double Mult(double a, int b)            // 4
int Mult(int a, int b, int c = 17);     // Fehler nicht von 2, 3 unterscheidbar
double Mult(int a, int b);              // Fehler Rückgabewert wird nicht
                                        //    zur Unterscheidung verwendet
```

Überladen von Funktionen und Methoden

```
int Mult(int a, int b);                 // 1
int Mult(int a, int b, int c);                  // 2
int Mult(double a, double b);           // 3
double Mult(double a, int b)            // 4

int i = Mult(17, 14);                   // Aufruf von 1
int j = Mult(17, 14, 15);               // Aufruf von 2
double d=Mult(17.4,  28.3);             // Aufruf von 3 trotz Rückgabe
double d=Mult(17, 16.4);                // keine Unterscheidung von 1,3,4
```

Clicker-“Abstimmung“

```
void pr(int a, int b) {cout << " PrIntInt "; }
void pr(int a, double b)
   {cout << " PrIntDouble "; }
void pr(int a, int b, int c)
    {cout << " PrIntIntInt "; }

…
pr(4,5);  pr(-9, 6.1);  pr(6.0, 7.2, 8.4);
```

Was wird auf dem Bildschirm ausgegeben?

1. PrIntInt PrIntDouble PrDoubleDoubleDouble
2. PrIntDouble PrIntIntInt PrIntDouble
3. PrIntInt PrIntDouble PrIntIntInt
4. PrIntInt PrIntInt PrIntIntInt

Ergebnis:

___ 1 __ 2 __ 3 __ 4

Übungen zu Arrays

Implementieren Sie eine Funktion *MinMaxDurch* der beliebig viele ganzzahlige Argumente in einem Array übergeben werden und die auf dem Bildschirm das maximale und das minimale Arrayelement ausgibt sowie den Durchschnitt aller Arrayelemente zurückliefert:

```
double MinMaxDurch ( int arrayGr,  const int array[] );
```

Weitere Übung:
Implementieren Sie eine Funktion Sort der beliebig viele ganzzahlige Argumente in einem Array übergeben werden und die die Arrayelemente sortiert in absteigender oder wahlweise aufsteigender Reihenfolge auf den Bildschirm ausgibt. Die Arrayelemente (ihre Reihenfolge) sollen in der Funktion nicht verändert werden.
Wie sehen die Argumente und der Rückgabewert der Funktion aus?

Hauptprogramm zur Lösung

```
int main()
{
   int array[]={1, 4, 11, 6, 44, 1, 9, 22,-100, 12};

   cout << "Durchschnitt des Arrays ist "
        << MinMaxDurch( sizeof(array)/sizeof(int),  array);

   return 0;
}

// sizeof liefert die Anzahl Bytes, die ein Typ bzw. eine
// Variable im Speicher belegt.
```

Übung zu Referenzparameter von Funktionen

Übung

Schreiben Sie eine Funktion MinMaxDurch, der beliebig viele ganzzahlige Argumente in einem Array übergeben werden und die das maximale, das minimale Arrayelement UND den Durchschnitt aller Arrayelemente über Referenzparameter zurückliefert.

```
void MinMaxDurch( int arrGr,  const int arr[], /* da fehlt was */)
…
void main ()
{  // Anwendungsbeispiel
int min, max;
double durch;
int arr[] = {1, 2, 3, 4, 5, 6};
MinMaxDurch(6, arr, min, max, durch);
```

Übung zu Referenzparameter von Funktionen

```
void MinMaxDurch( int arrGr,  const int arr[], /* da fehlt was */)
…
void main ()
{  // Anwendungsbeispiel
int min, max;
double durch;
int arr[] = {1, 2, 3, 4, 5, 6};
MinMaxDurch(6, arr, min, max, durch);
```

Übungen zu Zeichenketten

1) Implementieren Sie eine Funktion (und Tests dazu)

```
void ersetze(char text[], char ch_alt, char ch_neu);
```

die in text alle Zeichen ch_alt durch das Zeichen ch_neu ersetzt.

2) Übung für zu Hause (nicht ganz einfach):
Implementieren Sie eine Funktion (plus Tests dazu)

```
void ersetze(char text[], int max_text_len,
                    const char text_alt[], const char text_neu[]);
```

die in text alle Strings text_alt durch den String text_neu ersetzt.
Die maximale Länge, die text annehmen darf, ist nun inklusive des '\0' Zeichens max_text_len.

Refactoring -- Nachtrag

Refactoring dient somit „*lediglich*“ dazu, übersichtlichen, wartungsfreundlichen und damit weniger fehleranfälligen Code zu erhalten. Genau das haben wir hier durchgeführt.

Eine Art des Refactoring ist die einfache Aufspaltung einer großen Funktion in mehrere kleinere Funktionen – im Original wird jede dieser Funktionen einmal aufgerufen.

.

Refactoring – Nachtrag (2)

Eine zweite Art des Refactoring ist die Zusammenfassung gleichartiger Funktionalität zu einer Funktion, die dann an mehreren Stellen aufgerufen wird.

Das haben wir bspw. bei der Funktion plane praktiziert, die nun in jedem Test aufgerufen wird.

Wenn man beide Möglichkeiten zur Auswahl hat, welche soll man dann zuerst anwenden?

Die Antwort ist schwierig und einfach zugleich. Man sollte es gar nicht erst so weit kommen lassen, sondern viel früher eine Aufspaltung in Funktionen durchführen, und nicht erst, wenn es gar nicht mehr anders geht und die Funktion Hunderte von Codezeilen enthält.

Wenn man dann aber doch beide Möglichkeiten hat, ist es meist gleichgültig, womit man beginnt.

Man sollte die Schritte allerdings nacheinander durchführen, da sonst die Gefahr des versehentlichen Einbaus von Fehlern zu groß ist.

Einleitendes Beispiel

Schreiben Sie ein Programm *CntWords*, dem eine Datei übergeben wird, und das alle Worte (beginnend mit einem Buchstaben und länger als 2 Zeichen) zählt (d.h. wie oft jedes Wort vorkommt).

Ausgeben werden sollen alle Worte, die mindestens 3mal vorkommen und zwar in absteigender Reihenfolge des ASCII-Codes.

Sie sind dran.

Sie können die Lösung aus der ersten Veranstaltung verwenden.
Erweitern Sie anschließend z.B. durch Aufteilung in Funktionen.

```
          zur :    4
           zu :    4
         wird :    4
       werden :    4
          von :    4
          und :    8
         oder :    3
          nur :    3
        nicht :    3
          ist :    5
           in :    4
         fuer :    3
        durch :    3
          die :    8
          der :   12
         dass :    5
          das :    3
          auf :    4
       Klasse :    5
  Hilfsklasse :    3
          C++ :    4
```

Rekursion

Rekursive Funktionen sind in C++ ebenfalls möglich.

Beispiel Fakultät:

N ! = N * (N-1) * (N-2) ... * 3 * 2 * 1;
0! = 1
5! = 5* 4 * 3 * 2 * 1 = 120

```
int fak(int n) {
/* gilt nur fuer n >= 0 und fuer
   "kleine" Werte von n */
  if (n > 0)
       return n * fak(n-1);
  else
       return 1;
}
```

Übung

Schreiben Sie eine Funktion fibbo, die die N-te Fibbonacci Zahl $fib_{(n)}$ berechnet.

$fib_{(0)} = 1$
$fib_{(1)} = 1$

$$fib_{(n)} = fib_{(n-1)} + fib_{(n-2)}$$

Also 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

Versuchen Sie hierfür eine **rekursive** und eine **iterative** Lösung zu implementieren.
Ändern Sie die Funktion fibbo, sodass sie void zurückliefert und das Ergebnis in einem Referenzparameter geliefert wird.
fibbo(6,wert); → wert ist anschließend 13.

Motivation von Inline-Funktionen

```
void swap(int& a, int& b) {
int tmp=a; a=b; b=tmp;}

void HeapSort(int a[], int n) {
/*Aufbau des Heaps */
  int i;
  for (i=1; i<n; ++i) {
     int j = i;
     while (j > 0) {
        int p = (j-1) / 2;
        if (a[p] < a[j]) {
                        swap(a[p], a[j]); }
        j = p;
     }
  }
```

```
 /* Sortieren des errichteten Heaps */
  for (i=n-1; i>0; --i) {
     swap(a[0], a[i]);
     int m = i-1; int j=0;
     while ( 2*j+1 <= m ) {
        int c = 2*j+1;
        if ((c+1) <= m)
              if ( a[c] < a[c+1] ) { ++c; }
        if (a[j] < a[c]) {
              swap(a[c], a[j]); }
        j = c;
} } }
```

Motivation von Inline-Funktionen

```
void HeapSort(int a[], int n) {
/*Aufbau des Heaps */
  int i;
  for (i=1; i<n; ++i) {
     int j = i;
     while (j > 0) {
        int p = (j-1) / 2;
        if (a[p] < a[j]) {
          int tmp(a[p]); a[p]=a[j];
          a[j] = tmp; }
        j = p;
     }
  }
```

Welche Version ist zu bevorzugen?

```
 /* Sortieren des errichteten Heaps */
  for (i=n-1; i>0; --i) {
     int tmp(a[0]); a[0]=a[i]; a[i] = tmp;
     int m = i-1; int j=0;
     while ( 2*j+1 <= m ) {
        int c = 2*j+1;
        if ((c+1) <= m)
              if ( a[c] < a[c+1] ) { ++c; }
        if (a[j] < a[c]) {
           int temp(a[c]); a[c]=a[j];
           a[j] = temp; }
        j = c;
} } }
```

Inline-Funktionen (2)

Der Compiler erzeugt bei normalen Funktionen aus *swap(i, j);*

- j und i auf den Stack packen
- Rücksprungadresse merken
- Funktion swap aufrufen
- Funktion swap ausführen
- Zur Rücksprungadresse (Aufrufstelle) zurückspringen
- i und j vom Stack entfernen
- (Evtl.) Rückgabewert auswerten (Hier kein Rückgabewert)

Clicker-“Abstimmung“

Welche Variante der HeapSort-Implementierung sollte gewählt werden?

1. Die Version mit der Funktion swap, weil sie übersichtlicher ist
2. Die Version mit der Vertauschung direkt im Code,
   weil sie schneller ist.
3. Die Version mit der Funktion swap, weil es mehr Code ist
   (Lines of Code werden bezahlt).
4. Weder noch, sondern was Besseres.
5. Egal, Hauptsache die Sonne scheint.

Ergebnis:

___ 1 __ 2 __ 3 __ 4 __5

Inline-Funktionen

Eine weitere Neuerung von C++ gegenüber ANSI/ISO-C sind **Inline-Funktionen;**

```
inline void swap(double& a, double& b) {
  double temp;
  temp = a; a = b; b = temp;
}
...
double x = 1.5, y = 3.3;
swap(x, y);
```

Es wird kein Unterprogramm aufgerufen, sondern der definierte Code der aufgerufenen Inline-Funktion wird direkt eingefügt (Makro-Expansion). Das kann bei sehr kleinen Funktionen sinnvoll sein, um ggf. ein *speed-up* zu erzielen, indem der Verwaltungsüberhang (Parameter-übergabe, Sprung ins Unterprogramm und zurück) eliminiert wird.

Inline-Funktionen kombinieren beide Vorteile

```
inline void swap(int& a, int& b) {
int tmp=a; a=b; b=tmp;}

void HeapSort(int a[], int n) {
/*Aufbau des Heaps */
  int i;
  for (i=1; i<n; ++i) {
     int j = i;
     while (j > 0) {
        int p = (j-1) / 2;
        if (a[p] < a[j]) {
                        swap(a[p], a[j]); }
        j = p;
     }
  }
```

```
 /* Sortieren des errichteten Heaps */
  for (i=n-1; i>0; --i) {
     swap(a[0], a[i]);
     int m = i-1; int j=0;
     while ( 2*j+1 <= m ) {
        int c = 2*j+1;
        if ((c+1) <= m)
              if ( a[c] < a[c+1] ) { ++c; }
        if (a[j] < a[c]) {
              swap(a[c], a[j]); }
        j = c;
} } }
```

Inline-Funktionen (3)

Das Attribut **inline** gibt dem Compiler die Empfehlung, das Makrokonzept zu verwenden, der Compiler kann diese Empfehlung aber ignorieren. Rekursive Aufrufe können aus naheliegenden Gründen nicht korrekt abgearbeitet werden.

Die **Definitionen** von *inline-Funktionen* sollten in Header-Dateien stehen!

Sie werden dann per #include-Anweisung in die Dateien eingefügt, in denen sie benutzt werden. Wenn die entsprechenden Definitionen wie die Definitionen *normaler* Funktionen in C/C++-Quellcode-Dateien ausgelagert werden, dann sind Probleme beim Linken zu erwarten!

Wo werden die Funktionen definiert?

In C++ ist es nicht wie in Pascal oder Oberon möglich, lokale Funktionen zu definieren, d.h. Folgendes geht **nicht**

```
/* Es ist gleichgültig, ob die folgende Deklaration vorhanden ist oder
   nicht. Lokale Funktionsdefinitionen sind nie erlaubt. */
int funk1(int i); // Deklaration

int main()
{
   int funk1(int i) { ...} // Definition
   . . .  // Rumpf von main mit Benutzung von funk1
}
```

Eine Funktionsdefinition ist immer global.
Im obigen Beispiel in Pascal wäre funk1 nur in main bekannt.

Wo werden Funktionen definiert? (2)

funk1 wird daher vor oder nach main definiert oder auch in einer eigenen Datei.

```
int funk1(int i); // Deklaration
int funk1(int i)
{ ...} // Definition

int main()
{
... // Rumpf von main mit Benutzung von
von funk1
}
```

```
int funk1(int i); // Deklaration

int main()
{
... // Rumpf von main mit Benutzung von
von funk1
}

int funk1(int i)
{ ...} // Definition
```

Wo werden Funktionen definiert? (3)

Definition in einer eigenen Datei. Eine Header-Datei, z.B. „funk1.h“ enthält die Deklaration von funk1, und in der Quelldatei wird dann funk1definiert.

```
// Datei funk1.h:
int funk1(int i); // Deklaration

// Datei funk1.cpp:
#include "funk1.h"
int funk1(int i)  // Definition
{ ...}
```

In der Datei, in der funk1 benutzt wird, wird funk1.h eingebunden, d.h. die Deklaration von funk1:

```
// Datei main.cpp:
#include “funk1.h“
int main() {
  ... int rueck=funk1(17); ...
}
```

main.cpp und funk1.cpp müssen nun beide übersetzt werden und dann mit dem *Linker* verbunden werden.