Funktionen

Zusammenfassung von Befehlssequenzen als aufrufbare/wiederverwendbare Funktionen in einem Programmblock mit festgelegter Schnittstelle (Signatur)

Derartige prozedurale Programmierung erlaubt Dekomposition komplexer Probleme

Im Folgenden:

- Definition von Funktionen
- Deklaration vs. Definition
- Parameterübergabe
  - Pass by Value
  - Pass by Reference
  - Defaults, Overloading, variable Parameteranzahl
- Rekursion

83

Funktionen: Definition

Syntax zur Definition einer Methode

```
[<Speicherklasse>] <Typ> <Funktionsname>(<Parameterliste>)
{
   <Anweisungen>
}
```

Wichtigste Speicherklassen

`static`: Funktion kann nur innerhalb dieser Quelldatei benutzt werden

`extern` (Default): Funktion kann aus anderen Quelldateien benutzt werden, speziell **`extern "C"`** und **`extern "C++"`** für Benutzung aus C- bzw. C++-Programmen

`inline`: Quelltext der Funktion kann als Optimierung an aufrufender Stelle direkt eingesetzt werden, „Empfehlung“ für den Compiler

84

Funktionen: Rückgabetyp

Funktionen haben einzelnen typisierten Rückgabewert, da Funktionsaufrufe dadurch einfach Ausdrücke sind

Bisher: Rückgabewerte von einfachen Basisdatentypen
Alternativen zur Modifikation von mehreren Variablen bzw. komplexerer Ergebnisdaten

- Pass by Reference (Veränderung der Parametervariablen)
- Ergebnis als Feld oder selbstdefinierter komplexer Typ (z.B. **`struct`**)
- Zeiger auf Feld oder komplexen Datentyp

Wenn kein sinnvoller Rückgabewert existieren kann, spezieller Typ **`void`**, zum Beispiel

```
void ausgabe(char* text) { printf("%s", text);}
```

wobei entsprechende Methoden kein **`return`** benötigen

85

Funktionen: Deklaration

Deklaration einer Funktion entspricht deren „Bekanntmachung“, d.h. Funktion existiert, Implementierung ist aber „an einer anderen Stelle“

```
[<Speicherklasse>] <Typ> <Funktionsname>(<Parameterliste>);
```

Auch Signatur bzw. Prototyp der Funktion

Erlaubt Verwendung der Funktion im Quelltext, auch wenn **noch** keine Implementierung angegeben wurde

86

Funktionen: Deklaration /2

Gebräuchliche Anwendungen

- Vorab-Deklaration einer Funktion, die schon im Quelltext benutzt wird, deren Implementierung aber erst an späterer Stelle folgt (zwingend erforderlich z.B. wenn Funktionen sich gegenseitig aufrufen, aber in einer bestimmten Reihenfolge im Quelltext definiert werden müssen)
- Deklaration von Methoden in Header-Dateien, deren Implementierung aus anderen Quelldateien erst später vom Linker hinzugefügt wird
- Deklaration von Methoden (Klassenfunktionen) in einer Klassendefinition (in der Regel auch in Header-Dateien)

87

Vorab-Deklaration: Beispiel (C) /1

Vorab-Deklaration notwendig, damit Programm übersetzbar:

```
#include <stdio.h>

void ungerade(int z);

void gerade(int z) {
   if (z%2==0) printf("Zahl ist gerade. \n");
   else ungerade(z);
}

void ungerade(int z) {
   if (z%2!=0) printf("Zahl ist ungerade. \n");
 else gerade(z);
}
```

```
int main()
{
   int x;  printf("x =
   ");
   scanf("%i",&x);
   gerade(x);
   return 0;
}
```

Funktionen: Übergabe von Parametern

Bisher: Parameter können an eine Funktion als literale Werte oder Werte einer Variable übergeben werden

```
float add(float a, float b) { return a+b; }
...

   float x,y,z;
   ...
   x = add(32.3,9.7); // literale Werte
   x = add(y,z); // Variablenwerte
```

Werte werden dazu in die Speicherbereiche der lokalen Variablen der Funktion kopiert

1. Aufruf: a erhält Wert 32.3, b erhält Wert 9.7
2. Aufruf: a erhält Wert von y, b erhält Wert von z

Werte von y und z bleiben unverändert

→ *Pass by Value* (auch *Call by Value*)

Beispiel Test-function.c

89

Funktionen: Pass by Value oder by Reference

Pass by Value: Übergabe der Parameter durch Kopieren der Werte, wodurch eine Modifikation der ursprünglichen Variablen ausgeschlossen wird

Pass by Reference: Übergabe einer Referenz auf die Ursprungsvariable, wodurch deren Wert in der Funktion manipuliert werden kann

Übergabe von referenzierter Variable als Zeiger auf Speicherbereich der Variable

90

Einschub: Beispiel für Arbeit mit Zeigern

```
int i = 13;
```

`i` ist normale Variable mit Wert 13 an Adresse x

```
int* p;
```

p ist undefinierter (!) Zeiger (Pointer) auf `int`-Wert

```
p = &i;
```

Referenzierung (Bildung eines Zeigers auf Adresse der dahinter stehenden Variable) mit &, p zeigt jetzt auf Adresse x

```
*p = 7;
```

Dereferenzierung mit `*`, Wert an Adresse x ist jetzt 7

```
*p==7 && i==7;
```

Ist jetzt wahr ...

91

Pass by Reference: C-style

Basiert auf der Übergabe von Zeigerwerten statt eigentlichen Datenwerten

Zeiger (→) oder auch *Pointer* von einem bestimmten Typ entsprechen Speicheradresse an dem ein Wert von diesem Typ steht

Entsprechende Deklaration der Methode mit Zeigerparametern, z.B.

```
void tausche(int* x, int* y) { ... }
```

Beim Aufruf mit normalen Variablen muss vorher die Adresse jeder Variable berechnet werden

```
tausche(&a,&b);
```

Technik bisher schon verwendet, z.B. bei `scanf()` aus `stdio` zum Einlesen von Werten für Variablen

92

Pass by Reference: C-Style Beispiel

```
#include <stdio.h>

void tausche(int* x, int* y)  {
   int hilfe = *x;
   *x = *y;
   *y = hilfe;
}

int main() {  int a=4;  int
   b=7;
   printf("Vorher: a=%i b=%i \n",a,b);
   tausche(&a,&b);
   printf("Nachher: a=%i b=%i \n",a,b);
   return 0;
}
```

```
Vorher: a=4 b=7  Nachher: a=7 b=4
```

Beispiel call-by-reference.c

93

Zusammenfassung: Funktionen

- Funktionen als wiederverwendbare, zusammenhängende Folge von Anweisungen zur Strukturierung des Quelltextes
- Definition einer Funktion mit Speicherklasse, Rückgabetyp, Name, Parameter und deren Typen sowie eigentlicher Implementierung als Programmblock
- Deklaration einer Funktion als Bekanntmachung der Schnittstelle erlaubt Benutzung im Quelltext, ohne dass Implementierung bis dahin gegeben
- Parameterübergabe als kopierte Werte (Pass by Value) oder modifizierbare Variablen (Pass by Reference)

94

Mehr zu Datentypen: Union, Strukturen, Felder, Zeiger

- Arbeiten allein mit einzelnen Variablen jeweils eines Basisdatentyps wird für komplexere Aufgaben schnell unpraktisch.
- Verminderte Verständlichkeit und erhöhter Umfang des Codes
- Deshalb Zusammenfassung nach ihrer Bedeutung zusammenhängender Daten möglich:

  Union (Overlay): Besondere Struktur, bei der die Variablen auf identische Speicherbereiche zeigen. Wir nutzen union, um eine Speicherstelle zu interpretieren (Sonderfall).

  Strukturen (Structures, kurz Structs): Zusammenfassung von Variablen unterschiedlichen Typs mit Zugriff über separate Namen

  Felder (Arrays): Zusammenfassung von Variablen des gleichen Typs in einer Sequenz mit Zugriff über deren Position. Außerdem flexibles Arbeiten mit Daten im Hauptspeicher über

  Zeiger (Pointer): typisierte Zeiger verweisen auf beliebige Stellen im Speicher und erlauben so Arbeit mit verschiedenen Daten über eine Zeigervariable

95

union (Overlay Typ)

Unions erlauben es, auf eine einzelne Speicherstelle mit mehreren Variablentypen zuzugreifen. Das erlaubt z.B., die Bitfolge eines float-Typs als unsigned int darzustellen und darauf zu arbeiten. Das spart Speicherplatz, falls das notwendig ist.
Achtung: Jeder schreibende Zugriff auf eine Variable der union verändert auch die Inhalte der anderen Variablen. Daher wird union nur in Sonderfällen genutzt. Lesender Zugriff ist unschädlich und verändert nicht den Speicherinhalt.
[typ_name] ist optional, wird benötigt, wenn später weitere union_var angelegt werden sollen. Zugriff auf die internen Variablen variable1, variable2,… über union_var.variable1,…

```
union [typ_name] {
   datentyp variable1;
   datentyp variable2;
   ...
} union_var1, ... ;
```

96

union (Overlay struct)

```
int main()
{
 union {        // alle drei Variablen f,i,b[4]
  float f;      // auf einem 4 Byte Speicherbereich!
  unsigned int i;
  unsigned char b[4];
 } a;   // union mit Namen a;

 printf("Gib Float-Zahl ein: ");
 scanf("%f",&a.f);  //nur float wird eingelesen

printf("Gib Float %f als \nHex %X und als \n4 Bytes
[%2X][%2X][%2X][%2X] aus (little endian, LSB first)
\n", a.f, a.i, a.b[0], a.b[1], a.b[2], a.b[3]); //
                 // alle Werte werden dargestellt

return 0;
}
```

Beispiel union.c

97

Strukturen (`struct`)

Strukturen dienen der Zusammenfassung inhaltlich zusammengehöriger Variablen, die aber verschiedenen Typs sind

Anwendungsbeispiele:

- Daten zu einer Person (Name, Vorname, Alter etc.)
- Adressen (Straße, Hausnummer, Postleizahl, Ort)
- Daten einer Datei (Dateiname und/oder Pfad, Zustand, Zeiger auf aktuelle Lese- oder Schreibposition etc.)
- Daten eines GUI-Fensters (Koordinaten, Ausdehnung, Zustand, Komponenten etc.)
- ...

Vorstufe der Objektorientierung: Strukturen fassen vergleichbar mit Klassen von Objekten Daten zusammen

→ können in C++ ähnlich verwendet werden und bieten dort zusätzliche Funktionalität

98

Strukturen: Deklaration und Initialisierung

Einfachste Form der Deklaration inklusive einer Benennung des Strukturtyps (für spätere Wiederverwendung) und der sofortigen Deklaration von Variablen dieses Typs

```
struct typ_name {
   datentyp variable1;
   datentyp variable2;
   ...
} struct_variable1, ... ;
```

Initialisierung ähnlich Arrays über Werteliste (verschiedenen Typs) möglich

```
struct typ_name struct_variable = {...};
```

zum Beispiel

```
struct studenten_typ s = {"Meier","Karl","123456",25};
```

99

Strukturen: Zugriff auf Komponenten

Zugriff auf die einzelnen Komponentenvariablen erfolgt über Punkt-Operator **.** und können dann wie normale Variablen verwendet werden, zum Beispiel

```
struct studenten_typ s1;
s1.name = "Schmidt";
```

Bei Zugriff auf über Zeiger entweder Dereferenzierung des Zeigers oder Verwendung des Pfeil-Operators **->**, zum Beispiel

```
struct studenten_typ* s2 = &s1;
s2->alter=28;
```

100

Strukturen: Einfaches Beispiel (C)

```
#include <stdio.h>

int main()
{
   struct studenten_typ {
      char* name;
      char* vorname;
      char matrnr[7];
      int alter;
   } s = {"Meier","Karl","123456",25};
   printf("Name: %s \n",s.name);
   printf("Vorname: %s \n",s.vorname);
   printf("Matrikelnummer: %s \n",s.matrnr);
   printf("Alter: %i \n",s.alter);
   return 0;
}
```

101

Wiederverwendbare Strukturdefinitionen

Strukturen können auch unabhängig von konkreten Variablen global oder in einem gegebenen Scope definiert und dann später beliebig oft zur Definition von Variablen wiederverwendet werden

Definition über **`typedef`** möglich

102

Wiederverwendbare Strukturen: Beispiel (C)

```
#include <stdio.h>

typedef struct {
   char* name;
   char* vorname;
   char matrnr[7];
   int alter;
} studenten_typ ;

int main()
{
   studenten_typ s =
   {"Meier","Karl","123456",25 };
   printf("Name: %s \n",s.name);
   printf("Vorname: %s \n",s.vorname);
   printf("Matrikelnummer: %s
   \n",s.matrnr);
   printf("Alter: %i \n",s.alter);
   return 0;
}
```

103

Felder (Arrays)

Zusammenfassung von Variablen des gleichen Typs in einer ein- oder mehrdimensionalen Anordnung fester Größe

Anwendungsbeispiele:

- Folge von Messwerten
- Spalte einer Tabelle
- Koeffizienten einer Gleichung/Funktion
- Vektoren (eindimensionale Felder)
- Matrizen (zweidimensionale Felder)
- ...

Im Folgenden:

- Eindimensionale Felder
- Mehrdimensionale Felder
- Zeichenketten als Felder von **`char`**

104

Eindimensionale Felder: Aufbau und Zugriff

```
float f[7];
```

f[0]	f[1]	f[2]	f[3]	f[4]	f[5]	f[6]

Syntax für Deklaration:

```
datentyp feldname[elementanzahl];
```

Bei einer Feldgröße von n sind die Feldelemente über

ihren Index (Position) von 0 bis $n - 1$ in eckigen Klammern wie eine normale Variable zugreifbar

```
f[0]=42;
f[6]=f[0]+24;
```


Eindimensionale Felder: Verwaltung im Hauptspeicher

```
float f[7];
```

4 Byte	4 Byte	...	4 Byte

```
sizeof(f)= 7 * sizeof(float)
```

Felder werden im Hauptspeicher als zusammenhängende Folge von Bytes gespeichert

Gesamtgröße kann über im Sprachumfang vordefinierte Funktion **`sizeof`**() berechnet werden

Beginnt Feld bei Startadresse x kann die Adresse des Elements mit Index i berechnet werden über $x + i * sizeof(datentyp)$ → erlaubt einfaches Arbeit mit Felder über Zeiger und Zeigerarithmetik (→)

Eindimensionale Felder: Initialisierung /1

Bei der Definition von Feldvariablen ist wie bei normalen Variablen ein Setzen der initialen Werte möglich

Zuweisung erfolgt über Werteliste in geschweiften Klammern in der Form `{Wert1, Wert2, ...}`

```
int feld2[4] = {1,2,3,4};
```

Wird bei der Initialisierung die Feldgröße ausgelassen, ergibt sich die Feldgröße implizit aus der Anzahl der Werte in der Initialisierungsliste

```
int feld2[] = {1,2,3,4};
```

Diese Art der Wertzuweisung ist nur bei der Initialisierung (Definition) des Feldes zulässig

107

Eindimensionale Felder: Initialisierung /2

Ist die Werteliste unvollständig, d.h. enthält sie weniger Werte als nach der angegebenen Größe des Arrays erforderlich, werden die verbleibenden Werte mit 0 initialisiert

```
int feld3[10] =  {1,2,3};
int feld4[10] =  {0}; // alles auf 0
```

Wird keine Initialisierung vorgenommen, sind die Werte der einzelnen Feldelemente zufällig (und damit verschieden von 0!)

108

Eindimensionale Felder: Beispiel (C)

```
#include <stdio.h>

int main()
{
   float messwert[5];
   for (int i=0; i < 5; i++)
   {
     printf("Messwert[%i] =      ",i);
         scanf("%f", &messwert[i]);
   }
   float sum = 0;
   printf("Die Messwerte sind: ");
   for (int i=0; i < 5; i++) {
      printf("%g, ", messwert[i]);  sum
      = sum + messwert[i];
   }
   printf(" \nDer Mittelwert ist: %g \n", sum/5);
   return 0;
}
```

109

Verbesserung des Beispiels

Feste Länge von Arrays wird im Code an vielen Stellen verwendet

Bei Änderung der Programmlogik müssen deshalb alle diese Stellen geändert und vorher gefunden (!) werden

Deshalb gängige Programmierpraxis in C/C++

Literale Werte für Felddimensionen vermeiden

Definition der Größe an zentraler Stelle als Präprozessor-Konstante

```
#define MAX_MESSWERTE 5
```

oder als **const**-Variable

```
int const max_messwerte=5;
```

Array-Größe kann leider im Gegensatz zu anderen Programmiersprachen (z.B. Java mit `length()`-Methode für jedes Array) in C/C++ auch nicht problemlos dynamisch ermittelt werden

110

Eindimensionale Felder: verbessertes Beispiel (C)

```
#include <stdio.h>
#define MAX_MESSWERTE 5

int main()
{
   float messwert[MAX_MESSWERTE];
   for (int i=0; i < MAX_MESSWERTE; i++) {
      printf("Messwert[%i] = ",i);
        scanf("%f", &messwert[i]);
   }
   float sum = 0;
   printf("Die Messwerte sind: ");
   for (int i=0; i < MAX_MESSWERTE; i++) {
      printf("%g, ", messwert[i]);
        sum = sum + messwert[i];
   }
   printf(" \nDer Mittelwert ist: %g \n", sum/MAX_MESSWERTE);
   return 0;
}
```

111

Initialisierung: Beispiel (C) /1

```
#include <stdio.h>
int main()
{
   int feld1[10];
   for (int i=0; i < 10; i++)
   {
     printf("feld1[%i] = %i \n",i,feld1[i]);
   }
   printf("----------------- \n");

   int feld2[] = {1,2,3,4};

   int groesse_feld2 =
   sizeof(feld2)/sizeof(feld2[0]);
     printf("Groesse des Feldes: %i
   \n",groesse_feld2);

   for (int i=0; i < groesse_feld2; i++)
   {
      printf("feld2[%i] = %i \n",i,feld2[i]);
   }  ...
```

112

Initialisierung: Beispiel (C) /2

```
   ...
   printf("----------------- \n");
   int feld3[10] =  {1,2,3};
   for (int i=0; i < 10; i++)
   {
     printf("feld3[%i] = %i \n",i,feld3[i]);
   }
   printf("----------------- \n");
   int feld4[10] = {0};
   for (int i=0; i < 10; i++) {
      printf("feld4[%i] = %i \n",i,feld4[i]);
   }
}
```

113

Mehrdimensionale Felder

Deklaration entsprechend eindimensionalen Feldern
mit Sequenz von Dimensionsangaben in eckigen Klammern

```
datentyp feldname[dim1][dim2]...;
```

Speicherung als (nicht-zusammenhängende) Anzahl von Arrays, zum Beispiel 2-dimensionales Array:

- Dimension 1: Array von Pointern auf Arrays der 2. Dimension
- Dimension 2: Arrays mit eigentlich Werten

Initialisierung entsprechend eindimensionalen Arrays möglich über geschachtelte Listen oder Gesamtfolge aller Werte, zum Beispiel

```
float matrix1[3][4]= {{1,1,1,1},{2,2,2,2},{3,3,3,3}};
float matrix2[3][3]= {1,2,3,4,5,6,7,8,9};
```

114

Mehrdimensionale Felder: Beispiel (C) /1

```
#include <stdio.h>

int main()
{
   float matrix1[3][4]= {{1,1,1,1},{2,2,2,2},{3,3,3,3}};
   float matrix2[3][3]= {1,2,3,4,5,6,7,8,9};
   printf("Matrix 1: \n");
   for (int i=0; i < 3; i++) {
      for (int j=0; j < 4; j++)
         printf("%g ",matrix1[i][j]);
      printf(" \n");
   }
   printf("Matrix 2: \n");
   for (int i=0; i < 3; i++) {
      for (int j=0; j < 3; j++)
           printf("%g ",matrix2[i][j]);
      printf(" \n");
   }
   Return 0;
   }
```

Beispiel Test-2DFeld.c

115

Mehrdimensionale Felder: Beispiel (C) /2

Multiplikation der Matrizen und Ausgabe des Ergebnisses:

```
   ...
   for (int i=0; i < 3; i++)
      for (int j=0; j < 3; j++)
         for (int k=0; k < 3; k++)
            matrix3[i][j] += matrix1[i][k]*matrix2[k][j];
   printf("Matrix 1 * Matrix 2 : \n");
   for (int i=0; i < 3; i++) {
      for (int j=0; j < 3; j++)
         printf("%g ",matrix3[i][j]);
      printf(" \n");
   }
   return 0;
}
```

116

Zeichenketten als **`char`**-Array

Umsetzung von Zeichenketten (engl. *Strings*) als Folge von **`char`**-Werten (1 Byte), welche ASCII-Zeichen kodieren

Größe bestimmt **maximale** Länge der Zeichenkette → tatsächliches Ende der Zeichenkette ist durch speziellen Wert 0 gekennzeichnet (null-terminierte Zeichenketten)

Deshalb: tatsächliche maximale Länge ist Größe des Arrays minus ein Zeichen für Endemarkierung

Initialisierung als Liste wie normale Arrays oder als literaler Stringwert in Hochkommas

```
char s[6]= {'H','A','L','L','O','\0'};
```

ist äquivalent zu

```
char s[] = "HALLO";
```

117

Arbeiten mit Zeichenketten in C

Standardbibliothek `string.h` liefert zahlreiche Funktionen zur Arbeit mit Zeichenketten

Arbeiten auf **`char`**-Arrays und Zeiger darauf (**`char`*** →)

Funktion	Beschreibung
`strlen(s)`	Tatsächliche Länge der Zeichenkette `s` ohne die terminierende Null
`strcmp(s1,s2)`	Vergleicht `s1` und `s2` - Ergebnis ist 0, wenn Zeichenketten gleich sind positiv, wenn `s1` lexikographisch nach `s2` negativ, wenn `s1` lexikographisch vor `s2`
`strcat(s1,s2)`	Hängt `s2` an `s1` an, wobei genügend Platz in `s1` zur Verfügung stehen muß
`strcpy(s1,s2)`	Kopiert Inhalt von `s2` in `s1`, wobei genügend Platz in `s1` zur Verfügung stehen muß
...	...

118

Zeichenketten als **char**-Array: Beispiel (C) /1

```
#include <stdio.h>
#include <string.h>

int main()
{
   char s1[6]= {'H','A','L','L','O','\0'};
   char s2[] = "HALLO";
   char s3[3][5] =  {"Eins","Zwei","Drei" };
   ...
```

119

Zeichenketten als **char**-Array: Beispiel (C) /2

```
   ...
   printf("Die Stringlaenge von %s ist %i. \n",
         s1,strlen(s1));
   char s4[20]="";
   strcat(s4,s3[2]);
   strcat(s4,s3[1]);
   strcat(s4,s3[0]);
   printf("Konkatenation: %s  {\}n",s4);
   printf("Vergleich von %s und %s liefert %i. \n",
         s1,s2,strcmp(s1,s2));
   printf("Vergleich von %s und %s liefert %i. \n",
         s1,s4,strcmp(s1,s4));
   printf("Vergleich von %s und %s liefert %i. \n",
         s4,s1,strcmp(s4,s1));
   return 0;
}
```

120

Felder als Parameter

Arrays können prinzipiell als Funktionsparameter verwendet werden:

```
int f(float a[], int size_a, float b[][4],int size_b) { ...}
```

Aber Vorsicht:

- Werden immer als Referenz übergeben (→ Funktionen, Pass by Reference), d.h. die Werte werden nicht kopiert und bei schreibendem Zugriff ändern sich die Werte auch für die aufrufende Funktion
- Beschränkungen bei mehrdimensionalen Feldern: außer für die erste Dimension müssen in der Signatur konkrete Größen angegeben
- Größe des Feldes sollte am besten immer als separates Parameter übergeben werden, da sie nur umständlich bestimmbar ist

Arrays können nicht als Rückgabetyp einer Funktion verwendet werden (aber Zeiger auf ein Array)

121

Zeiger (Pointer)

Typisierte Zeiger speichern statt Wert des Datentyps Adresse als Verweis auf Speicherbereich, wo Daten dieses Typs stehen

Erlauben über Derefenzierung der Adresse direkten Zugriff auf Speicher von Daten

Nutzung für flexiblere Manipulation von Daten

Einsatzfeldern von Zeigern

- Pass by Reference in C (→ Funktionen)
- Zeichenketten als `char*`
- Flexibles Arbeiten mit Feldern
- Dynamische Speicherverwaltung auf dem *Heap*
- Komplexe Datenstrukturen
- ...

Arbeit mit Zeigern: Zeigerarithmetik mit speziellen Operatoren

122

Zeiger: Deklaration und Initialisierung

Deklaration eines

```
datentyp* zeigername[=<initialisierung>];
```

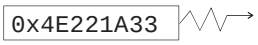
Möglichkeiten der Initialisierung

- Keine Initialisierung: Zeiger hat zufälligen Wert → kann zu schweren Programmfehlern führen
- **NULL**: setzt Zeiger auf vordefinierten unbenutzten Wert → kann abgefragt werden, d.h. p==**NULL** bedeutet „Ist der Zeiger benutzt?“
- Referenzierung einer existierenden Variablen oder eines Feldes vom Zeigertyp
- Dynamische Allokation von Speicher für die Daten über `malloc()` oder `calloc` in C

123

Zeiger: Initialisierung

```
int* p;
```



Nicht initialisierter Zeiger verweist auf zufällige Stelle im Hauptspeicher

```
int* p = NULL;
```

Auf **NULL** gesetzter Zeiger gilt als nicht definiert

```
int* p = &i;
```

Über Referenzierung gesetzter Zeiger verweist auf Folge von **sizeof**(**int**) Bytes an der Speicherstelle `0x0F116A00` der Variablen `i=42`

124

Arbeiten mit Zeigern

Referenzierung und Dereferenzierung:

- Dereferenzierungsoperator `*` liefert zu einem Zeiger den eigentlichen Wert an der referenzierten Speicherstelle
  Referenzierungsoperator `&` liefert die Adresse einer Variablen zurück

Zeigerarithmetik: Operatoren haben bei der Arbeit mit Zeigern spezielle Bedeutung, z.B. für `float* p` und `sizeof(float)=4`

- `p++` setzt Zeiger auf nächste `float`-Wert, d.h. um 4 Byte nach vorn
  `p--` setzt Zeiger um 4 Byte zurück
  `p+=4` setzt Zeiger um 4 `float`-Werte, d.h. 16 Byte, nach vorne
  `p-=4` setzt Zeiger um 16 Byte zurück

125

Zeiger: Arbeit mit Feldelementen

Zeiger erlauben flexibles Arbeiten mit Feldern
Feldvariable ohne Elementzugriff (Position in eckigen Klammern) hat als Wert Zeiger auf Anfang des Feldes → kann einer Zeigervariablen zugewiesen werden, zum Beispiel

```
float vektor[]= {1,2,3};
float* v = vektor;
```

Einzelne Feldelemente können dann über Zeigerarithmetik und Dereferenzierung zugegriffen werden (Positionieren des Zeigers auf einzelne Werte)

```
*v = 4;
*(v+1) = 2;
```

entspricht

```
vektor[0] = 4;
vektor[1] = 2;
```

126

Zeiger und Felder: Beispiel (C) /1

```
#include <stdio.h>
#include <math.h>

float norm(float* v, int dimension)
{
   float tmp=0;
   for (int i=0; i<dimension; i++) {
      tmp+=*v * *v;
      v++;
   }
   return sqrt(tmp);
}

...
```

127

Zeiger und Felder: Beispiel (C) /2

```
...

int main()
{
   float vektor[]= {1,2,3 };
   float n=norm(vektor,3);
   printf("Die Norm des Vektors (%g %g %g) ist %g. \n" ,
      vektor[0],vektor[1],vektor[2],n);
   return 0;
}
```

128

Zeiger und Felder: Erklärung des Beispiels

Funktion `norm()` berechnet die euklidsche Norm oder „Länge“ eines Vektors v beliebiger Dimension d aus den Vektorkomponenten v_i:

$$|v| = \sqrt{\sum_{i=1}^{d} v_i^2}$$

Durch Typkompatibilität kann Feld als Zeiger auf Anfang übergeben werden

Berücksichtigung der Dimension des Vektors/Größe des Arrays durch explizite Übergabe als Parameter

`tmp` zum Berechnen der Summe der Quadrate. Zeiger `v` verweist jeweils auf aktuelle Position im Feld

- `*v` greift auf aktuelle Vektorkomponente zu
- `v++` setzt Zeiger auf nächste Vektorkomponente

129

Zeiger auf Zeichenketten: **`char*`**

`char`* Zeiger auf **`char`**-Felder (→ **`char`**-Arrays) sind in C und C++ für Zeichenketten gebräuchlich

Eigenschaften entsprechend Zeigern auf Felder

- Zuweisungskompatibilität von **`char`**-Arrays
- Deshalb: alle Funktionen aus `string.h` entsprechend anwendbar
- Deshalb: Initialisierung wie bei **`char`**-Arrays möglich
- Zugriff über Zeigerarithmetik möglich

130

Zeiger und Zeichenketten: Beispiel (C)

```
#include <stdio.h>
#include <string.h>

int main()
{
   char s1[] = ”Torsten”;
   char* s2 = ”Mario”;
   char* s3 = ”Torsten”;
   s2 = s1;
   printf(”s2 (%X) verweist jetzt auf s1 (%X). \n”,
          s2,s1);
   if (strcmp(s2,s3)==0)
      printf(”s2 (%X) und s3 (%X) mit gleichem Wert %s. \n”,
         s2,s3,s2);
   return 0;
}
```

131

Dynamische Felder und Zeiger

Bisher: Größe eines Feldes muss zum Übersetzungszeitpunkt feststehen, d.h. folgt explizit aus dem Quelltext durch Größenangabe oder Initialisierung

Oft ergibt sich notwendige Größe eines Feldes aber erst zur Laufzeit → während der Ausführung

Lösung: jedes Programm hat einen dynamischen Speicherbereich (engl. *Heap*, dt. etwa Halde oder Haufen), auf dem Speicher beliebiger Größe angefordert werden kann und freigegeben werden muss → Verwaltung obliegt Programm/Programmierer

Zugriff auf dynamischen Speicherbereich mit `malloc()`

132

Dynamische Felder und Zeiger in C

Funktionen in `stdlib.h` zur Verwaltung des dynamischen Speichers, zum Beispiel

Funktion	Beschreibung
`malloc(size)`	Allokiert `size` Byte auf dem Heap
`calloc(number,size)`	Allokiert `number*size` Byte auf dem Heap
`realloc(pointer,size)`	Vergrößert ein an Stelle `pointer` existierenden Speicher auf die neue Größe `size`
`free(pointer)`	Gibt zuvor durch eine der oben genannten Funktionen reservierten Speicher frei
...	...

Ergebnis von `malloc`, `calloc` und `realloc` sind untypisierte Zeiger der Form **`void`*** → müssen durch explizites Casting auf benötigten Zeigertyp umgewandelt werden, z.B.

```
float* feld = (float*) malloc(100*sizeof(float));
```

133

Dynamische Felder und Zeiger in C: Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int groesse;
   float* feld;
   printf("Groesse des Feldes = ");
   scanf("%i",&groesse);
   feld = (float*) malloc(groesse*sizeof(float));
   for (int i=0; i<groesse; i++) {
      printf("Feld[%i] = ",i);
      scanf("%f",feld+i);
   }
   for (int i=0; i<groesse; i++) printf("%g ",*(feld+i));
   printf("\n");
   free(feld);
   return 0;
}
```

134

Komplexe Datenstrukturen

Komplexere Datenstrukturen können durch die zuvor eingeführten Felder, Strukturen und Zeiger zusammengesetzt werden

Klassische Beispiele:

- Listen, Warteschlangen etc.
- Bäume (Suchbäume, Indexe) und Hierarchien (z.B. Verzeichnisse im Dateisystem)
- Graphen allgemein (z.B. Straßennetze, Schaltpläne etc.)
- ...

Sind anwendungsspezifisch und daher nicht Teil des Sprachumfangs → müssen vom Programmierer entwickelt werden oder ggf. vorhandene Bibliothek kann genutzt werden

135

Beispiel: einfach verkettete Liste

Eine **Liste** ist eine Folge beliebiger Länge von Werten eines Typs → Array nicht problemlos anwendbar, da feste Länge → Liste ist eine **dynamische Datenstruktur**

- Kann zur Laufzeit beliebig wachsen und schrumpfen
- Erfordert spezielle Funktionen zum Arbeiten mit der Liste

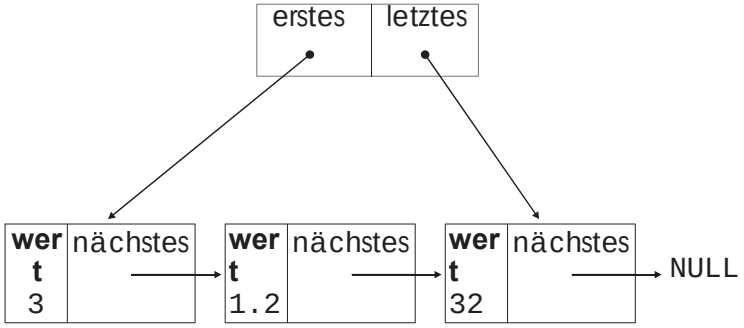
Einfach verkettet: Implementierung benutzt nur „Verkettung“ in eine Richtung, d.h. Zeiger zum jeweils folgenden Element, wodurch zum Beispiel Suchoperationen erschwert werden

136

Einfach verkettete Liste: Implementierung

Implementierung der Liste besteht aus

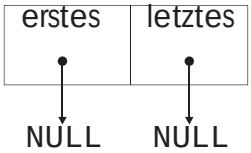
- Einer Struktur `listen_typ` für eigentliche Liste, welche aber nur aus Zeigern auf Anfang und Ende der Liste besteht
- Einer Struktur `listen_element_typ`, die die eigentlichen Werte und einen Zeiger auf das jeweils nächste Element umfaßt (beim letzten Element ist Zeiger **`NULL`**)



137

Leere einfach verkettete Liste

Eine leere Liste (direkt nach dem Erzeugen) wird hier durch zwei **`NULL`**-Pointer auf die nicht existierenden ersten und letzten Listenelemente dargestellt



138

Einfach verkettete Liste: Datenstruktur (C)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct  {
   float wert;
   listenelement_typ* naechstes;
} listenelement_typ;

typedef struct {
   listenelement_typ* erstes;
   listenelement_typ* letztes;
} listen_typ;

...
```

139

Einfach verkettete Liste: Funktionen

Im folgenden nur zwei Funktionen implementiert

- **Einfügen** eines neuen Wertes am Listenende umfaßt
  - Speicherung des neuen Listenelementes auf dem Heap
  - Wenn Liste leer, zeigen Anfangs- und Endzeiger der Liste auf dieses neue Element
  - Andernfalls wird das neue Element an das Ende angehängt
- **Ausgabe** der Liste, indem die Liste von Anfang bis Ende durchlaufen wird und alle Werte aus Listenelementen ausgegeben werden

Weitere denkbare Funktionen: Einfügen am Anfang, Einfügen an bestimmter Position, lesender Zugriff auf bestimmte Position, Suchen in der Liste, Sortieren der Liste, ...

140

Einfach verkettete Liste: Funktionen (C) /1

```
...

void einfuegen(listen_typ* l, float wert)
{
   listenelement_typ* neu = (listenelement_typ*)
      malloc(sizeof(listenelement_typ));
   neu->wert = wert;
   neu->naechstes = NULL;
   if (l->erstes == NULL) {
      l->erstes = neu;
      l->letztes = neu;
   }
   else {
      l->letztes->naechstes = neu;
      l->letztes = neu;
   }
}
...
```

141

Einfach verkettete Liste: Funktionen (C) /2

```
...

void ausgabe(listen_typ l)
{
   printf(”Listenelemente: (”);
   listenelement_typ* aktuell = l.erstes;
   while (aktuell != NULL) {
      printf(”%g ”,aktuell->wert);
      aktuell = aktuell->naechstes;
   }
   printf(”) \n”);
}

...
```

142

Einfach verkettete Liste: Hauptprogramm (C)

```
...

int main()
{
   listen_typ l = {NULL,NULL };
   einfuegen(&l,3.0);
   einfuegen(&l,4.678);
   einfuegen(&l,345.2);
   ausgabe(l);
   return 0;
}
```

143

Aufzählung (**enum**)

Weiterer einfacher Typkonstruktor in C und C++:
Aufzählung (Enumeration) als **enum**

Erlaubt Definition eines eigenen Wertebereichs, der intern auf Integer-Zahlen abgebildet wird

Beispiel:

```
enum studiengang { inf , bwl , mb , wmb };
studiengang s = bwl;
```

Wenn nicht anders angegeben, entspricht erster Wert in der Aufzählung 0, zweiter Wert 1 usw.

Andernfalls Definition möglich

```
enum zugriffsrecht { besitzer=1 , gruppe=2 , alle=4 };
```

144

Zusammenfassung: Datentypen

Flexibleres Arbeiten mit Daten durch

- Felder als Zusammenfassung von Daten gleichen Typs
- Strukturen als Zusammenfassung von Daten unterschiedlichen Typs
- Zeigervariablen zum Arbeiten mit Daten an verschiedenen Stellen im Hauptspeicher

Kombination dieser Möglichkeiten zur Schaffung komplexerer Datenstrukturen

145

Funktionen: Rekursion

Definition (Rekursion)

Rekursion (vom lateinischen *recurrere*: zurückführen, zurücklaufen) bezeichnet die Technik, eine Funktion durch sich selbst zu definieren.

In der Programmierung: Funktionen können direkt oder indirekt (über eine weitere Funktion) sich selbst aufrufen

Konzept entliehen aus der funktionalen Programmierung, im Gegensatz zu imperativer Programmierung als nur Folge von Befehlen

146

Funktionen: Rekursion

Anwendbar für viele Aufgaben: Berechnungsmöglichkeiten äquivalent zu Schleifen

Rekursion kann Schleifen immer ersetzen bzw. von diesen ersetzt werden

Vorteile:

- Oft einfache, elegante Lösungen möglich
- Praktisch zum Beispiel bei komplexeren Datenstrukturen (z.B. wenn diese rekursiv definiert sind: Hierarchien, Bäume), Optimierung, etc.

Nachteile:

- Funktionsaufrufe verursachen erhöhten Aufwand bei der Programmausführung → Performance
- Führt mitunter zu komplexeren Programmstrukturen, die schwerer verständlich bzw. deren Korrektheit schwerer verifizierbar ist

147

Rekursion: Beispiel Fakultätsberechnung

Mathematische Funktion für natürliche Zahlen n:

Fakultät (n): $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{i=1}^{n} i$

Kann so als Schleife (Zählschleife mit Produktbildung) umgesetzt werden

Aber auch rekursive Definition möglich:

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n(n-1)! & \text{für } n > 0 \end{cases}$$

Umsetzbar als rekursive Funktion

148

Rekursion: Beispiel (C)

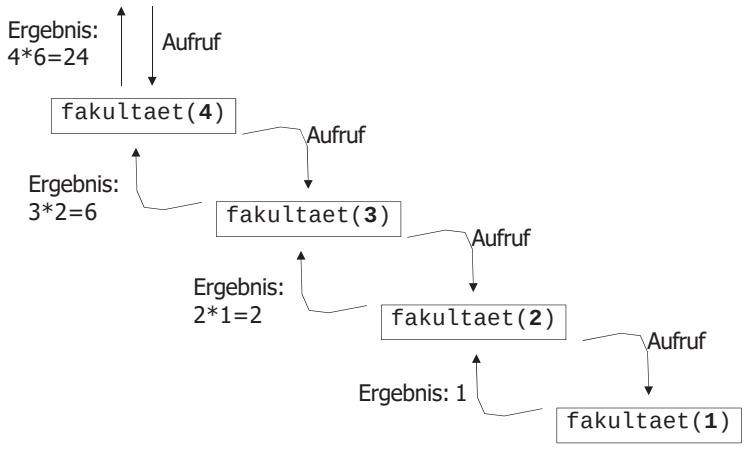
```
#include <stdio.h>

int fakultaet(int i)  {
   if (i<=1) return 1;
   else return i*fakultaet(i-1);
}

int main()
{
int x,f;
printf("x = ");
  scanf("%i",&x);
  f = fakultaet(x);
printf("x! = %i \n",
f);
  return 0;
}
```

149

Rekursion: Beispiel - Ablauf



150