Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Strukturierte Typen und ihre Codierung

- Typen und Daten in Quell-, Meta- und Objektsprache
- Primitive Typen und Daten
- Strukturierte Typen: *first-* oder *second class*
- Produkt- und Summen-Typen: Records / Unions
- Arrays
- Code für *second-class* Arrays, Records und Unions

Typen und Daten in Quell- Meta- und Objektsprache

Typ- und Datenrepräsentation

Drei Sprachen

- **Objekt-Sprache: Die Sprache die der Compiler übersetzt, oder der Interpreter interpretiert**
- **Meta-Sprache: Die Sprache in der der Compiler oder Interpreter geschrieben ist**
- **Ziel-Sprache: Die Sprache in die der Compiler übersetzt**

Typen und Werte in drei Welten

- **Jede Sprache kommt mit ihren eigenen Konzepten zu Typen und Werten. Diese können sich drastisch unterscheiden**
- **Beispiel:**

  Objektsprache dynamisch typisiert:

  keine Typdefinitionen

  Metasprache statisch typisiert

  Alle Variablen und Ausdrücke haben einen Typ aus dem Repertoire der Metasprache

  Zielsprache Assembler

  Alle Werte sind Bytesequenzen unterschiedlicher Länge (Byte, Word, ...)

Typen und Daten in Quell- Meta- und Objektsprache

Datenrepräsentation

Werte

- existieren zur Laufzeit
- haben einen
  - Typ der Objekt-Sprache und eine
  - Darstellung in der Ziel-Sprache (Compiler) oder Meta-Sprache (Interpreter)

Repräsentation der Werte

Werte der Objektsprache können auf unterschiedliche Art repräsentiert werden:

- Zum Wert / Typ der Objektsprache gibt es einen Wert / Typ der Ziel (oder Meta-) Sprache die dem Wert / Typ direkt entspricht.

  Beispiel:

  Objektsprache: `int x = 5;`

  Meta-Sprache: `var x : Int = 5`

- Im Regelfall gibt es aber mehr oder minder erhebliche Differenzen zwischen Objekt-, Meta- und Zielsprache

  Werte der Objektsprache werden durch Datenstrukturen in der Meta- oder Zielsprache dargestellt

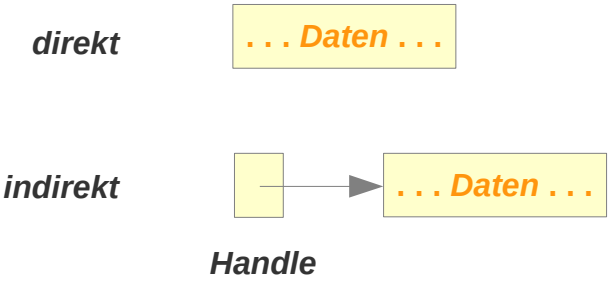
Typen und Daten in Quell- Meta- und Objektsprache

Datenrepräsentation

Direkt oder Indirekt

Werte der Objektsprache können
- direkt oder
- indirekt repräsentiert über ein *„Handle“*

werden



Mit oder ohne Typ-Info

Werte der Objektsprache
- ohne explizite Typinformation oder mit
- mit expliziter Typinformation oder mit

repräsentiert werden

Primitive Typen und Werte

Primitive Typen

- **Primitive Typen sind die Typen, der Werte aus aus anderen Werten zusammengesetzt sind.**
- **Beispiel: Integer, Boolean, Double, ...**

Repräsentation primitiver Typen

Primitive Typen werden meist so gewählt, dass direkt unterstützt werden können

- **Interpreter:**

  Metasprache unterstützt int und double

  => Objektsprache bietet int- und double-Typen / -Werte
- **Compiler:**

  Hardware unterstützt Ganzzahl- und Fließkomma-Arithmetik

  => Objektsprache bietet Ganzzahl und Fließkomma-Typen / -Werte

Literale für Werte primitiver Typen

Für die Werte primitiver Typen stehen stets Literale zur Verfügung die

- **den Wertebereich der Typen abdecken und**
- **eindeutig interpretierbar sind**

Primitive Typen und Werte in C

Primitive Typen

C bietet eine exorbitante Fülle an primitiven Datentypen:

- Ganzahl-Typen

```
char, signed char, unsigned char,
int, signed int, unsigned int,
long, signed long, unsigned long,
long long, signed long long, unsigned long long,
unsigned long long int
```

- Fließkomma-Typen

```
float,
double,
long double
```

Drüber hinaus werden in der Standard-Bibliothek noch weitere Typen definiert.

Ziel des Typsystems von C

Das Typsystem von C hat das Ziel maschinennah zu sein, ohne sich direkt auf eine bestimmte Maschine zu beziehen, und gleichzeitig auch noch anwendungsnah.

Beispiel:

- **char: ist definiert als die kleinste addressierbare Speichereinheit.**
  Normalerweise, aber nicht zwingend notwendig, ist das ein Byte
- **short: ist nicht mit Bezug auf adressierbare Speichereinheiten definiert sondern als Wertebereich mindestens: –32768, +32767**

Primitive Typen und Werte in C

Char und Int

- C bietet die Typen `char` und `int`.
- Beides sind Ganzzahl-Typen mit unterschiedlichen Definitionen:
  - `char` : kleinste adressierbare Speichereinheit, interpretiert als Ganzzahl
  - `int` : Ganzzahl im Wertebereich von mindestens –32768, +32767

Die genau Interpretation der beiden Typen ist plattform-abhängig

Numerische Literale und Char-Literale

- Eine numerisches Literal in C steht immer für einen `int`-Wert

  `123` => Int-Wert ***hundert-drei-und-zwanzig***
- C bietet „char-Literale“

  'a' => char-Wert ***sieben-und-neunzig***

  Achtung: In C++ stehen char-Literale für int-Werte!
- In C und in C++ gibt es
  - weder einen Datentyp für Zeichen,
  - noch irgendwelche Literale die für ein Zeichen sehen.

man kann aber in der Ausgabe eine Zahl als Zeichen interpretieren und

Zeichen-artige Literale für char-Werte verwenden

`'a'` : eine andere Art um 97 zu sagen

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

int main(void) {

  int i = 'a';
  char c = 'a';

  printf("i = %i as number \n", i);
  printf("i = %c as character \n", i);
  printf("c = %i as number \n", c);
  printf("c = %c as character \n", c);

  return 0;
}
```

```
i = 97 as number
i = a as character
c = 97 as number
c = a as character
```

Strukturierte Typen

Werte mit Struktur

- Records (ein Wert und noch ein Wert)
- Unions (ein Wert oder ein anderer Wert)
- Arrays (viele gleichartige Werte)

Typen mit Struktur

- Produkt-Typen (Typ von Records)
- Summen-Typen (Typ von Unions)
- Sequenz-Typen (Typ von Arrays)

First-Class oder *Second-Class*

- Typen sind *first-class* wenn
  - es Ausdrücke gibt, deren Wert diesen Typ hat
    Beispiel: In Sprachen ohne „*Lambda*-s“ sind Funktionen *second-class*
  - wenn diese Typen wie alle alle anderen Typen verwendet werden können:
    als Typen von Variablen,
    als Typen von Parametern und Funktions-Ergebnissen,
    etc.
- Strukturierte Typen sind nicht immer *first-class* Typen
  - Strukturierte Variable: Strukturierter Typ ist nur als Typ von Variablen (und Referenz-Parametern) erlaubt ~> *second class*
  - Strukturierter Wert: Strukturierter Typ ist überall erlaubt ~> *first class*

Produkt- und Summentypen

Records

auch: Produkt-Typ / Structure / Struct / Verbund

Ein Record (Wert mit Produkt-Typ) besteht aus mehreren Komponenten (*Fields*)

„Produkt“: wie „kartesisches Produkt“, Ein Wert besteht aus mehreren Komponenten

Jede Komponente

- **hat einen Namen (*field-selector*) und**
- **(in statisch typisierten Sprachen) einen Typ**

Beispiel struct in C:

```c
int main(void) {

      struct {
            char * name;
            int age;
      } hugo;

      hugo.name = "Hugo";
      hugo.age = 65;

      printf("name = %s, age = %d\n", hugo.name, hugo.age);

      return 0;
}
```

Structure mit den *field-selectors* name und age

Records: Strukturierte Werte / strukturierte Variablen

Strukturierte Variablen

Ein Record / Produkt kann sein ein

- **ein strukturierter Wert, oder**
- **eine strukturierte Variable**

Structs in C

- **sind strukturierte Variablen**
- **es ist auch möglich einen Struktur-Typ zu definieren, der diese Struktur repräsentiert:**

```
struct {
  char * name;
  int age;
} hugo;

struct {
  char * name;
  int age;
} egon;

hugo.name = "Hugo";
hugo.age = 65;

egon = hugo;
```

assigning to struct from incompatible type struct

Der Variablentyp der Structure wird benannt. Der Name des Typs wird verwendet.

```
struct Person {
  char * name;
  int age;
} hugo;

struct Person  egon;

hugo.name = "Hugo";
hugo.age = 65;

egon = hugo;
```

OK

Geht auch: Variablentyp definieren ohne gleichzeitig eine Variable zu definieren.

```
struct Person {
  char * name;
  int age;
};

struct Person  hugo;
struct Person  egon;

hugo.name = "Hugo";
hugo.age = 65;

egon = hugo;
```

Records: Speicherdarstellung

Ein Produkt / Record

als strukturierte Variable

- ist eine Sequenz von Variablen
- die jeweils die einzelnen Komponenten (*fields*) enthalten

Zugriff auf die Komponenten über

- eine Abbildung: Selektor-Name ~> Distanz-Adresse

Records: Darstellung

Die Implementierung von Records benötigt 2 Klassen von Informationen:

- Gespeicherte Daten

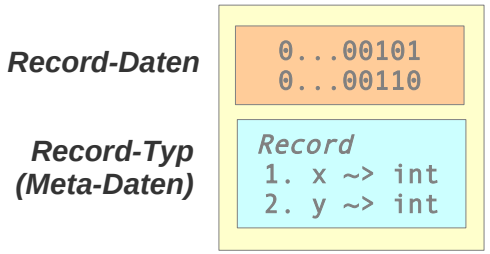
  z.B. die Werte 5 und 6,die der x- der y-Komponente in einem Punkt zugeordnet sind

- Meta-Informationen (Typ-Informationen)

  z.B.: die erste Komponente heisst `x`, und ist vom Typ `int`

  die zweite Komponente heisst `y`, und ist vom Typ `int`

  Die Meta-Information wird benötigt, um Zugriffe und Zuweisungen korrekt ausführen zu können



Ein Record

Produkt- und Summentypen

Union

auch: Summen-Typ / Disjoint Union / varianter Record

Eine Union (Wert mit Summen-Typ) besteht aus einer von mehreren möglichen Komponenten (*Fields*)

„Summe“: wie „oder“, Ein Wert besteht aus der Überlagerung möglicher Komponenten

Jede Komponente

- **hat einen Namen (*field-selector*) und**
- **(in statisch typisierten Sprachen) einen Typ**

Beispiel union in C:

```c
int main(void) {

  union {
    char * name;
    int age;
  } nameOrAge;

  nameOrAge.name = "hugo";

  printf("nameOrAge.name = %s  \n", nameOrAge.name);
  printf("nameOrAge.age = %i  \n", nameOrAge.age);

  return 0;
}
```

```
nameOrAge.name = hugo
nameOrAge.age = 106917698
```

age: Der Zeiger auf den String wird als Zahl interprtetiert

Produkt- und Summentypen

Union Beispiel union in C,

Union-Typen können definiert werden,

Unions werden meist in Structs eingebettet, um mit einer Markierung die aktuelle Variante bestimmen zu können:

```c
struct Point {
  int x;
  int y;
};
struct Line {
  struct Point start;
  struct Point end;
};
struct Geo {
  int isPoint; // Marker: 1 = Point, 0 = Line
  union {
    struct Point point;
    struct Line  line;
  };
};

int main(void) {
  struct Point start = {0, 0};
  struct Geo geo;
  geo.isPoint = 1;
  geo.point = start;
  if (geo.isPoint) {
    printf("geo = Point(%d, %d)  \n", geo.point.x, geo.point.y);
  }
  return 0;
}
```

```
geo = Point(0, 0)
```

Produkt- und Summentypen

Union

Summentypen werden in OO-Sprachen durch Vererbung realisiert

```c
struct Point {
  int x;
  int y;
};
struct Line {
  struct Point start;
  struct Point end;
};
struct Geo {
  int isPoint; // 1 = Point, 0 = Line
  union {
    struct Point point;
    struct Line  line;
  };
};

int main(void) {
  struct Point start = {0, 0};
  struct Geo geo;
  geo.isPoint = 1;
  geo.point = start;
  if (geo.isPoint) {
    printf("geo = Point(%d, %d)  \n", geo.point.x, geo.point.y);
  }
  return 0;
}
```

C

```java
abstract class Geo{}

class Point extends Geo {
  Point(int x, int y) {this.x = x; this.y = y; }
  int x;
  int y;
}
class Line extends Geo {
  Point start;
  Point end;
}

public class Test {
  public static void main(String[] args) {
    Point start = new Point(0,0);
    Geo geo = start;
    System.out.println(geo);
  }
}
```

Java

Union: Speicherdarstellung

Eine Union

als strukturierte Variable

- ist eine Variable
- in einer bestimmten Variante

Zugriff auf die Komponenten über

- eine Abbildung: Selektor-Name ~> Anfangs-Adresse

Union: Darstellung

Die Implementierung von Unions benötigt 2 Klassen von Informationen:

- Gespeicherte Daten

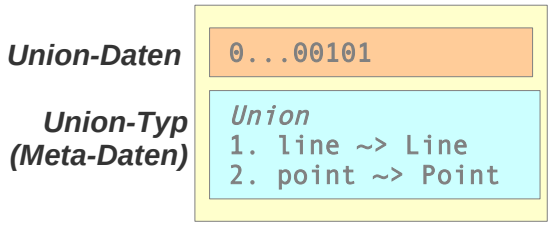
  z.B. die Werte Punkt Linie, die der point- der line-Komponente in einer Geo-Union zugeordnet sind

- Meta-Informationen (Typ-Informationen)

  z.B.: die erste Alternative heisst `point`, und ist vom Typ `Point`

  die zweite Komponente heisst `line`, und ist vom Typ `Line`

  Die Meta-Information wird benötigt, um Zugriffe und Zuweisungen korrekt ausführen zu können



Eine Union

Arrays

Ein Array als strukturierte Variable

- ist eine Sequenz von Variablen
- die alle vom gleichen Typ sind

Zugriff auf die Komponenten über

- eine Abbildung: Natürliche Zahl ~> Anfangs-Adresse der Komponente

Beispiel Array in C

```
int main(void) {

  char a[5] = {97, 98, 99, 100, 0};

  a[1] = a[0] + 5;

  printf("a = %s\n", a);

  return 0;
}
```

→ `a = afcd`

Arrays haben eine fixe Größe. a wird hier auf dem Laufzeit-Stack allokiert..

Strings sind char-Arrays die mit 0 enden.

Arrays in C

In C sind Arrays lediglich eine Unterstützung für die Allokation von Variablen-Sequenzen und die Adressrechnungen in diesen Sequenzen von Variablen.

D.h. in C gibt es eigentlich keine Arrays, sie ist lediglich eine Kurz-Notation für Pointer-Operationen. (Sie werden vor der Codegenierung eliminiert)

```
int main(void) {

  char a[5] = {97,98,99,100,0};

  *(a+1) = *a + 5;  //entspricht a[1] = a[0] + 5;

  printf("a = %s\n", a);

  return 0;
}
```

Arrays: Speicherdarstellung

Die Implementierung von Arrays benötigt 2 Klassen von Informationen:

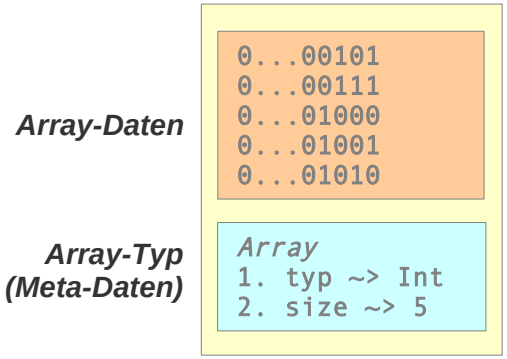
- **Gespeicherte Daten**

  die Sequenz der gespeicherten Daten

- **Meta-Informationen** (**Typ-Informationen**)

  Die Meta-Information wird benötigt, um Zugriffe und Zuweisungen korrekt ausführen zu können:

  - Typ der Komponenten
  - Anzahl der Komponenten

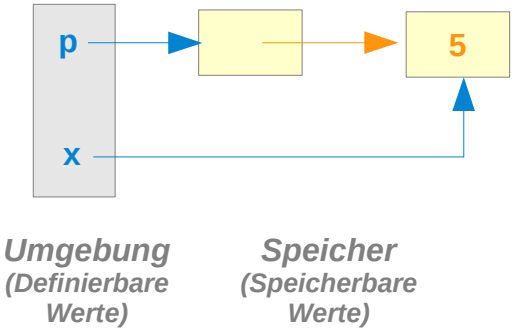


Ein Array

Referenzen und Zeiger

Zeiger

- Ein Zeiger ist eine Adresse als Wert
- Mit dem Zeiger-Konzept werden Speicheradressen zu Werten die vom Programm manipuliert werden können
- Definierbare Werte und speicherbare Werte werden damit vermischt!
- Beispiel C:




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

int main(void) {

  int x = 5;
  int *p = &x;

  x = x + *p;

  printf("%d\n", x);

  return 0;
}
```

&x : die Adresse von x als Wert

**p : das was an der Adresse zu finden ist, die in p steht: der Wert als Adresse*

Ausgabe: 10

Zeiger

Zeiger sind gefährlich und böse – zu böse für einfache Anwendungsprogrammierer: sie könnten sich oder andere damit verletzen

- **Zeiger sind ein mächtiges und riskantes Mittel der Programmierung**
- **Beispiel:**

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

int main(void) {

  int x = 5;
  int y = 6;

  int *p = &x;

  p--;

  printf("*p = %d\n", *p);

  return 0;
}
```

Ausgabe: `*p = 6`

Referenzen-Variablen

Die Referenz ist die brave kleine Schwester des Zeigers

- **Zeiger sind ein mächtiges und riskantes Mittel der Programmierung**
- **Referenzen sind eine gezähmte Variante der Zeiger:**
  - **Referenzen sind Zeiger,**
  - **und damit Adressen als speicherbare Werte**
  - **aber es sind Werte zweiter Klasse die durch praktische keine Operation manipuliert werden dürfen**
- **Beispiel C++:**

```
#include <iostream>
using namespace std;

int main() {

  int x = 5;
  int & p = x;

  p--;

  cout << "*p = " << p << endl;

  return 0;
}
```

& p : p ist eine Referenz-Variable: sie enthält Adressen

Das worauf p zeigt (x) wird erniedrigt – nicht p selbst!

Ausgabe: `*p = 4`

***Adressen** treten jetzt zwar auch als **Werte** auf, es ist aber nicht möglich mit ihnen „zu rechnen“*

Referenzen: Referenz-Variablen und Referenz-Parameter

- Referenzen werden in Java intensiv genutzt, ohne explizit erwähnt zu werden
  Jede Variable mit einem Klassentyp ist einer Referenz-Variable
- In manchen Sprachen können Referenz-Variablen explizit definiert werden (C++)
- Referenz-Parameter / Referenz-Übergabe:
  Übergib nicht den Wert eines Ausdrucks sondern die Adresse einer Variablen an die Funktion
- Viele Sprachen beschränken den Einsatz von Referenzen auf Parameter
- C++ hat Zeiger, Referenzen-Variablen und Referenz-Parameter
- Beispiel Referenz-Parameter in C++:

```
#include <iostream>
using namespace std;

void f(int &a) {
  a++;
}

int main() {
  int x = 5;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
  return 0;
}
```

Referenz-Parameter

Ausgabe:

```
x = 5
x = 6
```

Referenz-Parameter

- **Referenz-Parameter / Referenz-Übergabe: Übergib nicht den Wert sondern die Adresse einer Variablen an die Funktion**
- **Referenz-Parameter sind sehr nützlich bei der Programmentwicklung**

  Beispiel: Sortiertes Einfügen in eine verketteten Liste in C++

```
struct Node {
  int    v;
  Node * next;
};

void insert(int v, Node * &p) {
  if (p == 0) { // last
    p = new Node;
    p->v = v;
    p->next = 0;
  } else {
    if (p->v <= v) {
      insert(v, p->next);
    } else {
      Node * n = new Node;
      n->v = v;
      n->next = p;
      p = n;
    }
  }
}
```

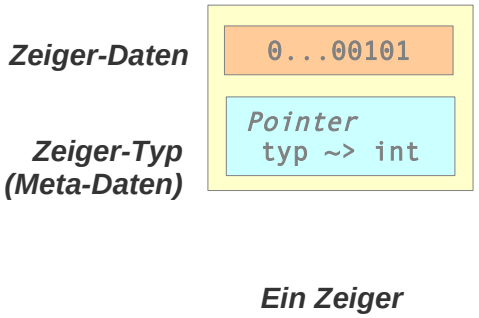
& vor Parameter: Referenz-Übergabe.

Die Referenz-Übergabe erlaubt es, an eine Funktion einen eventuell zu ändernden Speicherplatz zu übergeben.

Referenzen und Zeiger: Speicherdarstellung

Die Implementierung von Arrays benötigt 2 Klassen von Informationen:

- **Gespeicherte Daten**
  Adresse
- **Meta-Informationen (Typ-Informationen)**
  Typ der Daten auf die gezeigt wird.



Ein Zeiger

Code für Array und Record-Zugriffe

Code für Array-Zugriffe

wir gehen davon aus, dass die strukturierten Typen der Quellsprache *second-class* sind

Sie treten also nur

- **als Typen von Variablen und**
- **als Typen von Referenz-Parametern auf**

Code den Zugriff auf Array-Komponenten

- Berechne den Offset: Index-Wert multipliziert mit dem Speicherbedarf der Array-Elemente
- Berechne die Adresse des Arrays
- Addiere die Adresse des Arrays und den Offset

Code den Zugriff auf Record-/Union-Komponenten

- Bestimme den Offset der Komponenten: der Offset von Record-Komponenten kann statisch bestimmt werden: Entspricht dem Platzbedarf der vorherigen Komponenten.

  Unions: der Offset ist immer 0
- Berechne die Adresse des Records / der Union
- Addiere die Adresse des Records / der Union und den Offset

Code für Definitionen

- Berechne des Platzbedarf
  - **Array / Record: Summe des Platzbedarfs der Komponenten**
  - **Union: Maximum des Platzbedarfs der Komponenten**
- Reserviere den notwendigen Speicherplatz