

5. Abgeleitete Datentypen

Prof. Dr. Markus Gross

Informatik I für D-ITET (WS 03/04)

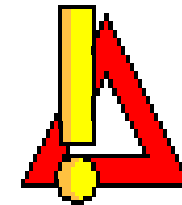
- Felder (Arrays)
- Zeichenketten (Strings)
- Strukturen (Structs)

Arrays

- Ein *Array* ist ein Datentyp, der mehrere Werte des gleichen Typs speichern kann
 - ◆ 60 ints oder 120 chars
- Die Definition eines Arrays besteht aus 3 Elementen:
 - ◆ Typ der zu speichernden Elemente
 - ◆ Name des Arrays
 - ◆ Anzahl der Elemente
- `int students[241];`
- Legt ein Array mit 241 Variablen des Typs `int` an und reserviert entsprechend Speicherplatz
- Allgemeine Form:
`typeName arrayName[arraySize];`

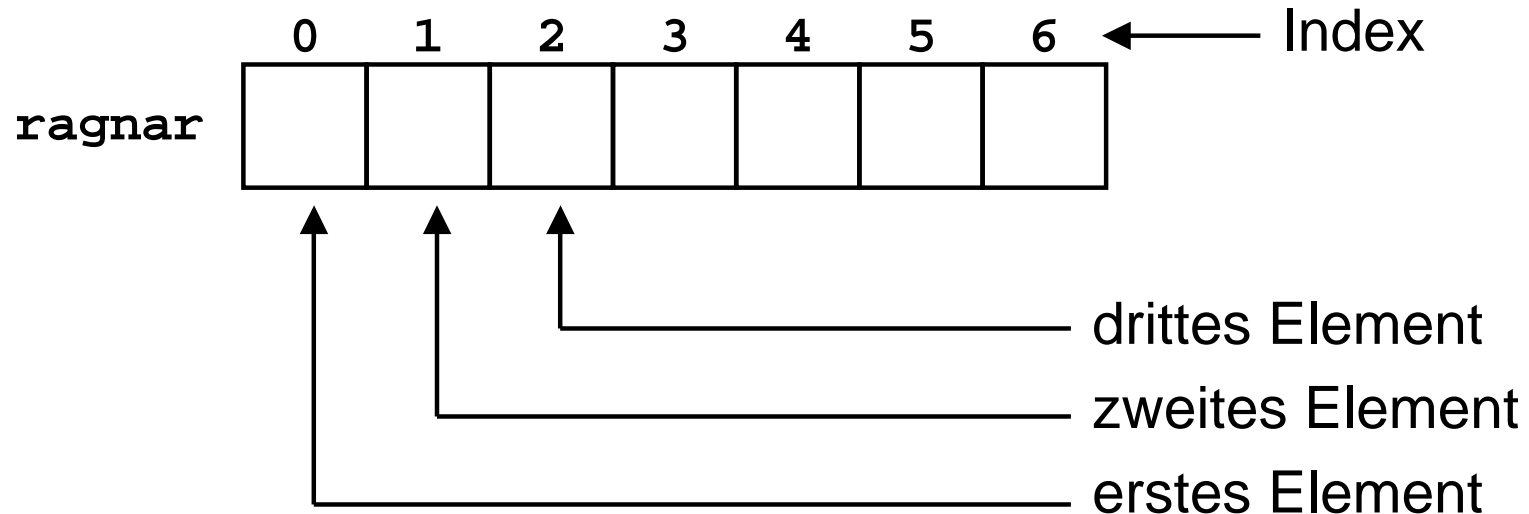
Arrays

- *arraySize* muss **const** sein
- *arraySize* darf keine Variable sein
- Ein *Array* ist ein *abgeleiteter* Typ, da er auf anderen Typen aufbaut
- Der Zugriff auf Elemente eines Arrays erfolgt durch Indizierung
- Der *index* beginnt immer bei 0!
- **students[111] = 64;**
- Setzt also das 112. Element des Arrays
- Dies führt leicht zu Fehlern



Illustration

```
int ragnar[7];
```



`ragnar` ist ein Array mit 7 Elementen vom Typ `int`

Beispiel_1: Array anlegen

```
// arrayone.cpp _ small arrays of integers
#include <iostream>
using namespace std;
int main()
{
    int yams[3]; // creates array with three elements
    yams[0] = 7; // assign value to first element
    yams[1] = 8;
    yams[2] = 6;

    int yamcosts[3] = {20, 30, 5};

    // create, initialize array
    // NOTE: If your C++ compiler or translator can't initialize
    // this array, use static int yamcosts[3] instead of
    // int yamcosts[3]

    cout << "Total yams = ";
    cout << yams[0] + yams[1] + yams[2] << "\n";
    cout << "The package with " << yams[1] << " yams costs ";
    cout << yamcosts[1] << " cents per yam.\n";
    int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
    total = total + yams[2] * yamcosts[2];
    cout << "The total yam expense is " << total << " cents.\n";

    cout << "\nSize of yams array = " << sizeof yams;
    cout << " bytes.\n";
    cout << "Size of one element = " << sizeof yams[0];
    cout << " bytes.\n";
    return 0;
}
```

Definition des Arrays

Initialisierung

Initialisierungsliste

Adressierung

Grösse eines Elementes

Initialisierung

- Initialisierung kann auf zwei Arten erfolgen
 - ◆ Initialisierungsliste bei Definition
`int hotel[2] = {1,2};`
 - ◆ Explizit
`hotel[0] = 1;`
- Arrays können auch partiell initialisiert werden
 - ◆ `int hotel[5] = {1,2};`
 - ◆ `int hotel[500] = {0};`
- Compiler setzt die restlichen Elemente zu 0
- Compiler berechnet Arraygröße automatisch bei Initialisierung
 - ◆ `int hotel[] = {0,1,2,3};`

Mehrdimensionale Arrays

- Arrays können in beliebiger Dimension angelegt werden
- Für jede Dimension muss die Arraygrösse getrennt angegeben werden
- Definition erfolgt analog zu 1D
 - ◆ `int hotel[3][2];`
- Initialisierung ebenso
 - ◆ Initialisierungsliste bei Definition
`int hotel[3][2] = {{1,2},{3,4},{5,6}};`
 - ◆ Explizit
`hotel[0][1] = 2;`
- Zugriff durch Index-Vektor
 - ◆ `int ndArray[2][2][2][2];`
 - ◆ `ndArray[1][1][1][1] = 3;`

Strings

- Ein String ist eine Reihe von Zeichen, welche in aufeinanderfolgenden Bytes im Speicher gehalten werden
- Ein C++ String muss immer mit `\0` (*null character*) enden
 - ◆ `\0` benötigt ein zusätzliches Byte
 - ◆ `char stud[5] = {'b','i','l','l','\0'};`
 - ◆ `char stud[5] = {'b','i','l','l','y'};`
Kein String!
- Initialisierung auf zwei Arten
 - ◆ `char stud[5] = {'b','i','l','l','\0'};`
 - ◆ `char stud[] = "billy";`
- `"billy"` ist ein *String Literal*

Strings

- Bei String Literalen wird der Null Character automatisch eingefügt
- Einzelnes Zeichen darf nicht mit String verwechselt werden
 - ◆ `char stud = 'b'; //O.K.`
 - ◆ `char stud = "b"; //not O.K. Type mismatch`
- `cout` kann Strings sinnvoll auf der Konsole ausgeben
 - ◆ `cout << "Markus ist ein Lieber \n";`
- Strings können mit Hilfe von Arrays gespeichert und verwaltet werden
- `<cstring>` *Bibliothek* (*library*) stellt viele wichtige String-Funktionen zur Verfügung

Beispiel_2: String und Array

```
// strings.cpp _ storing strings in an array
#include <iostream>
#include <cstring> // for the strlen() function
using namespace std;
int main()
{
    const int Size = 15;
    char name1[Size]; // empty array
    char name2[Size] = "C++owboy"; // initialized array

    // NOTE: some implementations may require the static keyword
    // to initialize the array name2

    cout << "Howdy! I'm " << name2;
    cout << "! What's your name?\n";
    cin >> name1;

    cout << "Well, " << name1 << ", your name has ";
    cout << strlen(name1) << " letters and is stored\n";
    cout << "in an array of " << sizeof name1 << " bytes.\n";

    cout << "Your initial is " << name1[0] << "...\n";
    name2[3] = '\0'; // null character

    cout << "Here are the first 3 characters of my name: ";
    cout << name2 << "\n";
    return 0;
}
```

Definition einer Konstanten

Array für String

Initialisierung

Eingabe

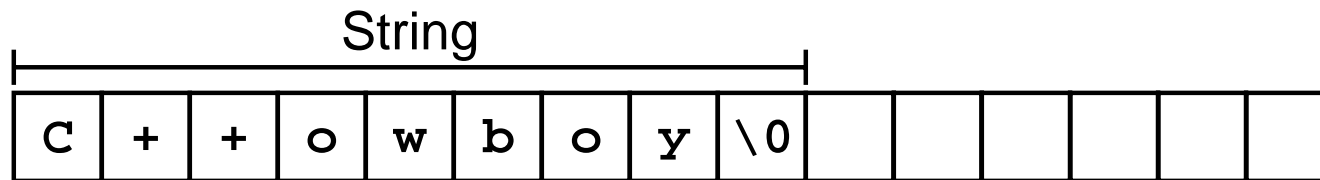
Funktion aus cstring

Ausgabe

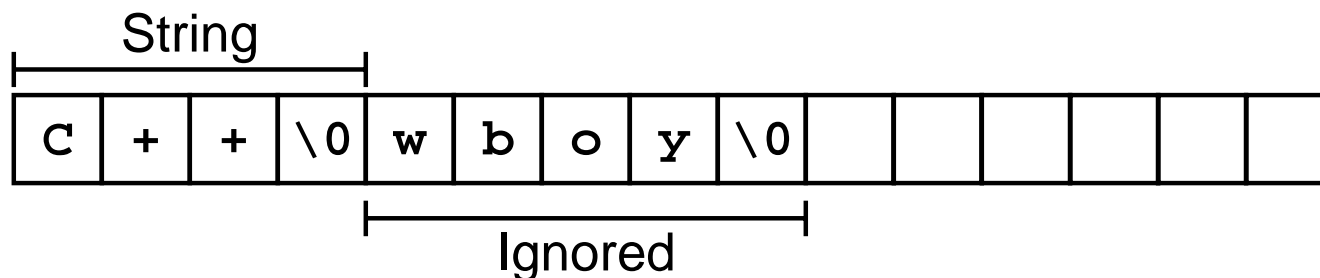
Initialisierung des Arrays

- Bei der Initialisierung werden die restlichen Stellen des Arrays unbesetzt belassen

```
const int ArSize = 15;
char name2[ArSize] = "C++owboy";
```



```
name2[3] = '\0';
```



- String Input pp. 100-109 bitte selbstständig anschauen

Structs

- Ein *Struct* ist ein Datentyp, der mehrere Werte verschiedener Typen speichern kann
- Vielseitiger als Array und enorm mächtig
- Ein Struct ist ein Vorläufer einer Klasse
- Trennung von Deklaration des Typs und Definition einer Variable des Typs
- Deklaration des Typs erfolgt durch
 - ◆ Schlüsselwort **struct**
 - ◆ Name (*Name Tag*)
 - ◆ Aufzählung der *Struct Member* Variablen
- Definition einer Variablen dieses Typs erfolgt durch Verwendung des Name Tags

Structs

- Beispiel für Deklaration

```
struct product
{
    char name[20];
    float volume;
    double price;
};
```

- Deklaration erfolgt oft ausserhalb von Funktionen
 - ◆ Externe Deklaration - Globale Sichtbarkeit (scope)
 - ◆ Interne Deklaration - Lokale, auf Funktion beschränkte Sichtbarkeit
- Definition einer Variablen des Typs
 - ◆ `struct product p1; // C-Stil`
 - ◆ `product p1; // C++ Stil...ist einfacher`

Initialisierung und Zugriff

- Initialisierung erfolgt explizit durch Liste

```
product p1 =  
{  
    "Informatik",  
    500.0,  
    6.7  
};
```

- Komma als Separator verwendet
- Zuweisungsoperator kann auf Structs angewandt werden
 - ◆ `product p2 = p1;`
 - ◆ Jede einzelne Mitgliedsvariable wird zugewiesen

Initialisierung und Zugriff

- Deklaration und Definition können auch kombiniert werden

```
struct product
{
    char name[20];
    float volume;
    double price;
} p1, p2;
```

- Deklariert Typ und legt zwei Variablen des Typs an
 - ◆ `p2, p1;`
 - ◆ Variablen haben hier gleichen Gültigkeitsbereich, wie Typendeklaration

Beispiel_3: Structs

```
// assgn_st.cpp -- assigning structures
#include <iostream>
using namespace std;

struct inflatable
{
    char name[20];
    float volume;
    double price;
};

int main()
{
    inflatable bouquet =
    {
        "sunflowers",
        0.20,
        12.49
    };

    inflatable choice;

    cout << "bouquet: " << bouquet.name << " for $";
    cout << bouquet.price << "\n";

    choice = bouquet; // assign one structure to another
    cout << "choice: " << choice.name << " for $";
    cout << choice.price << "\n";

    return 0;
}
```

Externe Deklaration

Initialisierung

Zugriff auf Members

Zugriff

- Zugriff auf Mitglieder durch `.`-Operator
 - ◆ `p1.price = 5.3;`
- Ebenso können Arrays von Structs angelegt werden
 - ◆ `product pArray[10];`
 - ◆ Definiert Array von 10 Variablen vom Typ `product`
- Initialisierung ebenso durch Listen

```
product pArray[2] = {{"Informatik I", 500.0, 6.7}, {"Informatik I", 300.0, 3.7}};
```
- *Bitfelder* für hardwarenahe Programmierung
- Variablen sind dabei auf bestimmte Anzahl von Bits beschränkt

Unions

- Ein *Union* ist ein Struct, der jeweils nur *EINE* der Mitgliedsvariablen speichern kann
- Die Grösse von Union-Variablen entspricht somit der Grösse der grössten Mitgliedsvariablen
- Variablen teilen die gleiche Speicheradresse

```
union one
{
    int i;
    float f;
    double d;
} var_1;
```

- Zuweisung, wie bei Structs
 - ◆ `var_1.i = 10;`

Aufzählungstypen (Enumerations)

- Eine *Enumeration* erlaubt es, elegant symbolische Konstanten zu definieren
 - ◆ `enum colors = {red, green, blue};`
 - ◆ Legt neuen Typ des Namens colors an
 - ◆ Definiert `red`, `green`, `blue` als symbolische Integer-Konstanten der Werte `0,1,2`
 - ◆ Heissen auch *Enumerators*
- Im Standard-Fall (default) weist ein enum Integer-Werte von `0` an aufsteigend zu
- Zuweisung nur sehr beschränkt möglich
 - ◆ `colors c1 = red; // O.K.`
 - ◆ `colors c1 = 4; // not O.K.`
- Konstanten können auch explizit gesetzt werden
 - ◆ `enum {one = 1, three = 3};`

typedef

- **typedef** wird benutzt um ein Synonym für bestehende Datentypen zu erzeugen.
- Beispiel: **typedef int Integer;**
- Ab dieser Anweisung kann **Integer** als Type benutzt werden und ist identisch mit **int**.
- Achtung: **typedef** erzeugt keinen neuen Datentyp, nur ein Synonym (Alias) für einen bestehenden Datentyp!
- Verwendung:
 - ◆ Häufig im Zusammenhang mit Structs (C-Style).
 - ◆ Flexibilität, z.B. ein Algorithmus wird mit dem Typ **Number** programmiert, und je nachdem ob man **typedef float Number;** oder **typedef double Number;** festlegt, verändert sich die Genauigkeit der Berechnung.