

Strukturen

Sebastian Stabinger

SS2020

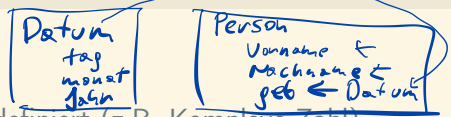
Standarddatentypen

- C bietet eine Reihe von Standarddatentypen an
- `int` ←
- `double` ←
- ...

Benutzerdefinierte Datentypen

- In vielen Fällen ist es sinnvoll, **eigene Datentypen** zu definieren
- z.B. Komplexe Zahlen, Koordinaten, Vektoren/Matrizen, Spielfigur, ...

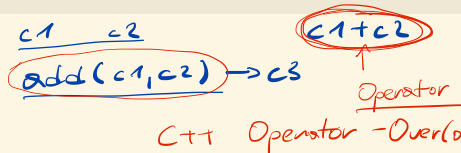
Wie werden solche Datentypen normalerweise verwendet?



- Es wird ein **neuer Datentyp** definiert (z.B. Komplexe Zahl)
 - Hier wird entschieden welche Daten gespeichert werden müssen
 - Für komplexe Zahl z.B. Real- und Imaginärteil als double
- Man entscheidet wie diese **Daten verarbeitet** werden
 - z.B. zwei komplexe Zahlen addieren, multiplizieren, formatiert auf dem Bildschirm ausgeben, ...

Implementierung in C

- Ein **neuer Datentyp** wird mittels Strukturen implementiert
- Die **Verarbeitung** dieser Daten wird über Funktionen realisiert welche Strukturen entgegennehmen und zurück geben



Neuer Datentyp

Strukturen

Syntax

struct Name {
 // Enthaltene Daten im Format:
 // datentyp name;
};

void f() {
 // ...
}

Beispiel

Wir definieren einen neuen Typ namens Complex:

struct Complex {
 double real; ←
 double imag; ←
};

real und imag sind Teil des neuen Datentyps

Hinweis

Strukturen werden vor der main-Funktion deklariert!

Erzeugen einer uninitialisierten Variable

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

```
struct Complex {  
    double real;  
    double imag;  
};
```

```
int main() {  
    // Deklaration einer Variablen des neuen Typs  
    struct Complex c;  
}
```

int c;

struct Complex c

c benötigt!

C++ nicht benötigt => Complex c;

Typealias

Es ist etwas unpraktisch, dass man bei einem Strukturdatentyp immer explizit **struct** davor schreiben muss (Hinweis: In C++ ist dies nicht mehr der Fall)

Lösung mit typedef

- Mit Hilfe des Befehls **typedef** können alternative Namen (ein sogenannter Typealias) für Datentypen vergeben werden

Format: typedef datentyp neuer_name;

int
unsigned int
struct Complex → Complex

Typealias — Beispiel

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

```
struct Complex {  
    double real;  
    double imag;  
};
```

"alten" Name

"neuer" Name

```
typedef struct Complex Complex; // "Complex" -> "struct Complex"
```

```
int main() {  
    Complex c; // Statt struct Complex c  
}
```


Struktur und Typealias in einem Rutsch

Die Deklaration einer Struktur und die Vergabe eines Typealias können auch **in einem Rutsch** erledigt werden

Beispiel

```
#include <stdio.h>

// Typealias und Deklaration der Struktur in einem Schritt
typedef struct Complex {alte Datentyp
    double real;
    double imag;
}Complex neue Datentyp

int main() {
    Complex c;
}
```

Zugriff auf Komponenten einer Struktur

Geschieht mittels Punkt (.) zwischen Variablenname und Komponentename der Struktur:

Beispiel

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

```
typedef struct Complex {  
    double real;  
    double imag;  
} Complex;
```

```
int main() {  
    Complex c;  
    // Schreibender Zugriff  
    c.real = 12.3;  
    c.imag = 2.3; } Initialisierung  
    // Lesender Zugriff  
    printf("%f + %fi", c.real, c.imag);  
}
```

12.300000 + 2.300000i

Initialisierung

- Wie die meisten anderen Datentypen auch sind Variablen mit einem Strukturdatentyp nicht automatisch initialisiert
- Häufig steht 0 in den Komponenten einer Struktur, aber **man kann sich nicht drauf verlassen!**

Beispiel

```
#include <stdio.h>

typedef struct Complex {
    double real;
    double imag;
} Complex;

int main() {
    Complex c;
    // Ausgabe an dieser Stelle kann alles sein ...
    printf("%f + %fi", c.real, c.imag);
}
```

Initialisierung

- Beim Deklarieren einer Variable (und nur dort) mit einem Strukturdatentyp können Startwerte in Form einer Initialisierungsliste (wie bei Arrays) angegeben werden.
- Es gilt die gleiche Reihenfolge wie bei der Definition der Struktur

Beispiel

```
#include <stdio.h>

typedef struct Complex {
    double real;
    double imag;
} Complex;

int main() {
    Complex c = {1.2, 0.234};
    printf("%f + %fi", c.real, c.imag);
}
```

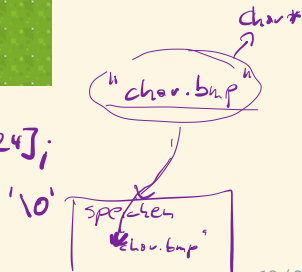
1.200000 + 0.234000i

Wir schreiben unseren Spieleprototyp so um, dass die Informationen einer Spielfigur in einer Struktur gespeichert sind



```
struct  
  x: int  
  y: int  
  img: char*
```

string => char[1024];



Mit den Datentypen arbeiten

Strukturen als Datenpaket

- Die Verwendung von Strukturen als eine Sammlung von zusammengehörenden Variablen ist an sich schon nützlich
- Strukturen werden aber speziell dann ein mächtiges Werkzeug zur Abstraktion, wenn die Verarbeitung von den darin enthaltenen Daten in Funktionen passiert.

1.) $f \leftarrow \text{struct}$

2.) $f \rightarrow \text{struct}$

3.) Allgemeiner Verwendung

Strukturen als Parameter von Funktionen

So wie sie einen int als Parameter in eine Funktion schicken können, können Sie auch eine Struktur als Parameter an eine Funktion übergeben.

Beispiel

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

```
typedef struct Complex {  
    double real;  
    double imag;  
} Complex;
```

*void f(int i) {
 ...
}*



```
void print(Complex num) { printf("%f + %fi\n", num.real, num.imag); }
```

```
int main() {  
    Complex c = {1.2, 0.234};  
    print(c);  
}
```


Rückgabe von Strukturen von Funktionen

Genauso wie Sie einen `int` von einer Funktion mittels `return` zurückgeben können, können Sie auch eine Struktur mit `return` zurück geben

Beispiel

Nächstes Slide

```
#include <stdio.h>

typedef struct Complex {
    double real;
    double imag;
} Complex;

void print(Complex num) { printf("Xf = %f\n", num.real, num.imag); }

Complex add(Complex c1, Complex c2) {
    Complex res;
    res.real = c1.real + c2.real;
    res.imag = c2.imag + c2.imag;
    return res;
}

int main() {
    Complex c1 = {1.2, 0.234};
    Complex c2 = {12.5, -1.33};
    Complex c3 = add(c1, c2);
    print(c3);
    // Diese Zeilenzeile ist Variable
    print(add(c1, c2));
}
```

Rückgabe von Strukturen von Funktionen — Beispiel

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

```
typedef struct Complex {  
    double real;  
    double imag;  
} Complex;
```

```
void print(Complex num) { printf("%f + %fi\n", num.real, num.imag); }
```

```
Complex add(Complex c1, Complex c2) {  
    Complex res;   
    res.real = c1.real + c2.real;  
    res.imag = c1.imag + c2.imag;  
    return res;  
}
```

int i;
i = 23;
return i;

=> return 23;

```
int main() {
```

```
Complex c1 = {1.2, 0.234};
```

```
Complex c2 = {12.5, -1.33};
```

```
Complex c3 = add(c1, c2);
```

```
print(c3);
```

```
// Ohne Zwischenspeicherung in Variable
```

```
print(add(c1, c2));
```

```
}
```

printf(4)

in C nicht möglich

↗ in C++ möglich

⇒ [c1+c2]

Rückgabe — Beispiel ohne temporäre Variable

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

```
typedef struct Complex {  
    double real;  
    double imag;  
} Complex;
```

```
void print(Complex num) { printf("%f + %fi\n", num.real, num.imag); }
```

```
Complex add(Complex c1, Complex c2) {  
    return (Complex){c1.real + c2.real, c1.imag + c2.imag};  
}
```

```
int main() {  
    Complex c1 = {1.2, 0.234};  
    Complex c2 = {12.5, -1.33};  
    Complex c3 = add(c1, c2);  
    print(c3);
```

// Ohne Zwischenspeicherung in Variable

```
    print(add(c1, c2));  
}
```

$\text{int } i1 = 5j$

$\text{int } i2 = 10j$

$\text{int } i3 = i1 + i2j$

$\Rightarrow \text{int } i3 = 5 + 10j$

Rückgabe — Beispiel komplett ohne Variablen

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

```
typedef struct Complex {  
    double real;  
    double imag;  
} Complex;
```

```
void print(Complex num) { printf("%f + %fi\n", num.real, num.imag); }
```

```
Complex add(Complex c1, Complex c2) {  
    return (Complex){c1.real + c2.real, c1.imag + c2.imag};  
}
```

```
int main() {  
    print(add((Complex){1.2, 0.234}, (Complex){12.5, -1.33}));  
}
```

Handwritten annotations: The arguments `(Complex){1.2, 0.234}` and `(Complex){12.5, -1.33}` are grouped with red curly braces and labeled `c1` and `c2` respectively below them.

Ändern der Werte einer Struktur innerhalb einer Funktion

Wenn Sie Strukturen als Parameter an eine Funktion übergeben, können Sie die Werte darin zwar ändern, aber **diese Änderungen haben keine Auswirkungen außerhalb der Funktion**

Beispiel

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

```
typedef struct Complex {  
    double real;  
    double imag;  
} Complex;
```

```
void print(Complex num) { printf("%f + %fi\n", num.real, num.imag); }
```

```
void init(Complex num) { num.real = num.imag = 0.0; }
```

Complex

```
int main() {
```

```
    Complex c = {23.0, 42.27};
```

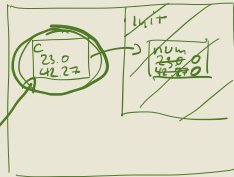
```
    c = init(c);
```

// c ist immer noch 23.0 + 42.27i und nicht 0.0 + 0.0i !

```
    print(c);
```

```
}
```

23.000000 + 42.270000i



(Complex){0,0}

Übergabe von Strukturen als Zeiger

Um Werte in einer Struktur nach aussen hin sichtbar zu ändern, muss die Struktur als Zeiger an die Funktion übergeben werden

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

```
typedef struct Complex {  
    double real;  
    double imag;  
} Complex;
```

Warum nicht #num.real?

$2 + (5 * c)$ (*num)...

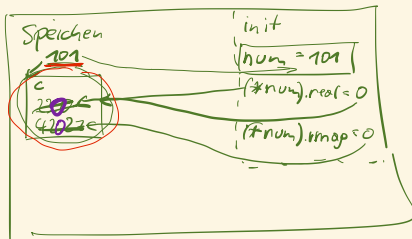
```
void print(Complex num) { printf("%f + %fi\n", num.real, num.imag); }
```

```
void init(Complex *num) { (*num).real = (*num).imag = 0.0; }
```

Complex num

```
int main() {  
    Complex c = {23.0, 42.27};  
    init(&c);  
    // c ist jetzt 0.0 + 0.0i !  
    print(c);  
}
```

0.000000 + 0.000000i



Zugriff auf Komponenten eines Strukturzeigers

- Der Zugriff mit einem Punkt nach dem Dereferenzieren (z.B. `(*num).real`) ist etwas umständlich.
- Syntactic Sugar um das ganze leserlicher zu machen:
 - Statt `(*num).real` kann auch `num->real` geschrieben werden

Beispiel

```
#include <stdio.h>

typedef struct Complex {
    double real;
    double imag;
} Complex;

void print(Complex num) { printf("%f + %fi\n", num.real, num.imag); }
void init(Complex *num) { num->real = num->imag = 0.0; }

int main() {
    Complex c = {23.0, 42.27};
    init(&c);
    print(c);
}
```

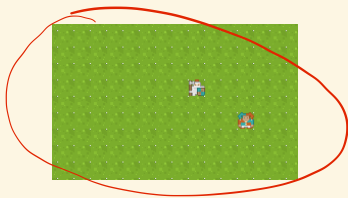
Übung

Schreiben Sie folgende Funktionen für unser auf Strukturen umgeschriebenes Spielebeispiel:

draw_figure Zeichnet die Figur mit der richtigen Grafik an der richtigen Stelle

are_colliding Übernimmt zwei Figur-Strukturen und überprüft ob diese gerade kollidieren

move_up, move_down, move_left, move_right Bewegt eine Figur nach Oben, Unten, Links, Rechts und stellt sicher, dass sich diese nicht vom Spielfeld bewegt



Verwenden Sie die geschriebenen Funktionen an geeigneter Stelle in unserem Spiel