# Funktionen und Typen

# Ausblick

Wir beschäftigen uns nun damit wie ...

- ... ein paar zusätzliche Anweisungstypen funktionieren
- ... wir verschiedene *Datentypen* benutzen können
- ... wir komplett neue *Funktionen definieren* können
- ... *Aufzählungstypen*, *Zeiger*, *Felder*, und *Strukturen* funktionieren
- ... man auf *undefiniertes Verhalten* achtet

# Weitere Schleifen

Schleifen vom Type `while` und `do ... while`

```
1 /* führe "anweisung" aus, solange "ausdruck" war ist */
2 while (ausdruck)
3     anweisung;
4
5 /* wie oben, aber "ausdruck" wird nach "anweisung" geprüft */
6 do anweisung;
7 while (ausdruck);
```

# Die `switch`-Anweisung

```c
/* springt zur case-Marke, die zum Ausdruck passt */
switch (wochentag) {
case 1: printf("Montag"); break;
case 2: printf("Dienstag"); break;
/* ... */
case 7: printf("Sonntag"); break;
default: printf("ungültiger Wochentag %d!", wochentag);
}
```

# Sprunganweisungen

| | |
|---|---|
| `goto` *`marke`* | Sprungmarke `marke` anspringen |
| `return` *`foo`* | Funktion beenden und `foo` zurückgeben |
| `return` | Funktion beenden in `void`-Funktionen |
| `continue` | sofort zur nächsten Schleifeniteration gehen |
| `break` | Schleife / `switch`-Anweisung beenden |
| *`foo`*`:` | Sprungmarke für `goto`-Anweisung |
| `case` *`foo`*`:` | Sprungmarke für Fall `fooin` lstinlineswitch-Block |
| `default:` | Sprungmarke für Fall „andernfalls“ in `switch`-Block |

# Funktionen

- Eine weitere Unterteilung des Programmes bieten Funktionen
- Diese berechnen anhand ihrer als Argumente übergebenen Parameter ihren Rückgabewert
- viele Funktionen in C haben aber auch Nebenwirkungen
- z. Bsp. `puts()`:
  - probiert den übergebenen String auf der Konsole auszugeben
  - bei Fehler gibt es die Konstante `EOF` zurück und setzt div. Fehlervariablen
  - sonst gibt es eine nicht-negative Zahl aus.
- Die Argumente des Funktionsaufrufes müssen in der Reihenfolge und vom Typen mit der Spezifikation übereinstimmen

# Funktionsdefinitionen

- Häufig haben Funktionsdefinitionen die Form:
  *Typ Funktionsname*(*Typ Name*, ...) *Block*
- Bspw.: `int leapyear(int year)  /* ... */`
- Wenn die Funktion keine Argumente nehmen soll, hat sie den speziellen Parameter `void`!
- Das Gleiche gilt für den Rückgabewert, sollte die Funktion nichts „berechnen“
- Falls es jedoch einen gibt, kann der in der Funktion mit dem Statement `return (/* 42? */);` zurückgegeben werden.
- Ansonsten einfach `return` zum vorzeitigen Beenden der Funktion

# Funktionsdeklaration

- Häufig haben Funktionsdefinitionen die Form:
  *Typ Funktionsname*(*Typ Name*, ...) *Block*
- Allgemeiner jedoch, kann der sog. *Deklarator* (zwischen Typ und Block) deutlich komplexer sein!
- Bewusst abschreckendes Beispiel:

```
1 void (*signal(int sig, void (*func)(int)))(int);
```

- Ich habe jetzt an Stelle eines Blockes ein Semikolon am Ende gesetzt
- ... das ist damit nur eine Deklaration.

# Wie der Kompiler funktioniert: Prädeklarationen

- der C-Compiler beginnt am Anfang eurer C-Datei und liest sie ein
- Für jede neue Deklaration: Hinterlegt Namen des Objekts in einer Tabelle
- Wird Objekt später benutzt, kann der Compiler überprüfen, ob die Typen stimmen (was soll eine Division einer Funktion durch einen String sein?)
- Wird das Objekt aber benutzt *bevor* es bekannt ist, kann der Compiler nicht arbeiten.
- Am Rande: Funktionsdefinitionen können nicht in Blöcke geschrieben werden
- Sind also erstmal „global“ sichtbar!

```c
int main(void)
{
    foo(42, 24); // Unbekannte Funktion foo!
    /* ... */
}

int foo(int bar, int baz) // Definition mit Deklaration
{
    return (bar+baz);
}
```

```c
int foo(int bar, int baz) // Definition mit Deklaration
{
    return (bar+baz);
}

int main(void)
{
    foo(42, 24); // Bereits bekannte Funktion foo
    /* ... */
}
```

```c
int foo(int bar, int baz); // Deklaration

int main(void)
{
    foo(42, 24); // Bereits bekannte Funktion foo
    /* ... */
}

int foo(int bar, int baz) // Definition mit Deklaration
{
    return (bar+baz);
}
```

# Datentypen

- Der *Typ* einer Variable gibt an *was* dieser *wie* speichert.
- Grundlegend gibt es in C eigentlich nur Zahlen
- Dies spiegelt den Aufbau eines Computers wieder
- Buchstaben sind bspw. nur Zahlen, von denen wir wissen, dass wir sie als Buchstaben zu interpretieren haben!
- Das nennt man: „Sie sind codiert“, s. ASCII-Code
- Alle anderen Datentypen können aus den primitiven zusammengesetzt werden

# Datentypen: Beispiel

- Explizit kennen wir schon den Datentypen `int`, für den Rückgabewert von `main()`
- Ein `int` (Integer) ist eine Ganzzahl, kann also negativ, positiv oder $0$ sein
- Die Bedeutung des Rückgabewerts von `main()` ist speziell für C-Programme: Exit-Code des gesamten Programms
- Es gibt noch andere, weniger offensichtliche Typen in diesem Beispiel
- Wir beschäftigen uns aber erstmal mit den grundlegenden primitiven Typen

# Primitive Datentypen

Einfachste Datentypen:

- `char`: Ein Byte, häufig für ASCII-Buchstaben verwendet
- `_Bool`: Boolean, kann nur $0$ oder $1$ halten, trotzdem so groß wie `char`
- `int`: Ganzzahl, mit negativen Zahlen
- `float`: Gleitkommazahl einfacher Präzision
- `double`: Gleitkommazahl doppelter Präzision

Modifikatoren für `int`:

- mit `unsigned` speichert der Typ nur natürliche Zahlen
- mit `short`, `long` und `long long` kann die sog. Größe des Typs verändert werden

# Größe eines Typs

- Üblicherweise sind ein Byte acht Bit, es können also maximal $2^8$ unterschiedliche Zahlen repräsentiert werden
- Brauchen wir mehr, brauchen wir ggf. `int`
- Größe ist plattformabhängig! Kleine Microchips haben ggf. andere Standardgrößen für Ganzzahlen als „normale“ Rechner
- Es gibt auch Typen mit festen Größen, je nach Anwendung bietet sich das an.
- Diese fallen unter die Kategorie der abgeleiteten Datentypen

# Abgeleitete Datentypen

Aus `<stdint.h>`

`int`$N$`_t` hat genau $N$ Bits, wobei $N \in \{8, 16, 32, 64\}$.

`uint`$N$`_t` analog, vorzeichenlos

`(u)intmax_t` größter verfügbarer Datentyp für Ganzzahlen

Aus `<stddef.h>`

`size_t` ebenfalls vorzeichenlos, groß genug um die Größe jedes Objektes als Zahl zu speichern

`ptrdiff_t` groß genug, um die Differenz zwischen zwei Zeigern zu speichern wissen!

# Enumerationen

- Syntax: `enum` [ *Bezeichner* ] `{` *Bezeichner* [ = *Wert* ] [ `,` …] `};`
- Definiert symbolische Ganzzahlkonstanten vom Typ `int`
- Kann auch als Typ verwendet werden
- Wird *Wert* weggelassen, so wird die nächste Zahl verwendet

# Enumerationen

```
1 enum {
2   FOO,     // Wert:  0
3   BAR,     // Wert:  1
4   BAZ,     // Wert:  2
5   QUUX = 42,  // Wert: 42
6   LOL,     // Wert: 43
7   NOPE = -1,  // Wert: -1
8   YES,     // Wert:  0
9 };
```

# Zeiger

- *Zeiger* zeigen auf Objekte
- Deklaration: *Typ* * *Bezeichner* [ = *Wert* ] ;
- Der *-Operator *dereferenziert* einen Zeiger
- Der &-Operator *referenziert* ein Objekt
- * und & löschen einander aus, `&*E` ist immer äquivalent zu `E`
- Der `NULL`-Zeiger zeigt nirgendwo hin

# Zeiger

```
1 int x = 0, y = 1, *p;
2 p = &x;                   // p zeigt auf x
3 printf("%d\n", *p);       // gibt 0 aus
4 *p = 42;                  // weise 42 indirekt zu
5 printf("%d\n", x);        // gibt 42 aus
6 p = &y;                   // p zeigt auf y
7 printf("%d\n", *p);       // gibt 1 aus
8 *p = 23;                  // weise 23 indirekt zu
9 printf("%d\n", y);        // gibt 23 aus
```

# Eingabe lesen mit Zeigern

- die Funktion `scanf` ist wie `printf`, ließt aber, anstatt zu schreiben
- fast die gleichen Platzhalter werde unterstützt
- statt Wert wird aber ein Zeiger auf eine Variable übergeben
- Rückgabewert: wie viele Elemente erfolgreich gescannt wurden

```
1 int i, result;
2 result = scanf("%d", &i);
3 if (result != 1)
4         printf("Scanning failed!\n");
```

# Arrays (Felder)

- *Arrays* sind stetige Regionen gleichartiger Objekte
- Deklaration: *Typ Bezeichner* `[` [ *Ausdruck* ] `]` `;`
- Ausdruck in eckigen Klammern bezeichnet die Arraylänge
- Zugriff auf Array-Elemente mit dem `[]`-Operator
- `array[index]` ist äquivalent zu `*(array + index)`
- Arrays verhalten sich fast überall wie Zeiger auf ihr erstes Element.
- Arraygröße muss meistens eine Konstante sein

# Arrays

```
1 int foo[5] = { 1, 2, 3, 4, 5 }, *bar;
2 printf("%d\n", foo[3]);     // gibt 4 zurück
3 bar = &foo[4];              // Zeiger auf 4. Element
4 bar = foo + 4;              // dito
5
6 char test[] = "soso";       // Strings sind Arrays
7 printf("%d\n", test[4]);    // gibt 0 zurück
```

# hilfreiche Funktionen

```c
char buf[20];
memset(buf, '\0', sizeof buf); // Puffer mit 0 initialisieren
strcpy(buf, "Hallo ");         // Zeichenkette kopieren
strcat(buf, "Welt!\n");        // Zeichenkette anhängen
printf("%s", buf);             // Ausgabe: "Hallo Welt!\n"
```

# Strukturen

- eine *Struktur* sammelt benannte Felder verschiedener Typen in einem Objekt
- Syntax: `struct` [ *Bezeichner* ] `{` *Deklaration* `;` …`}`
- auf Strukturfelder kann mit dem Operator `.` zugegriffen werde
- `a->b` ist Abkürzung für `(*a).b`

# Strukturen

```c
/* Deklaration der Struktur */
struct mensch {
    char *vorname, *nachname;
    int groesse, alter;
};

struct mensch hans, *kunde = &hans;

hans.vorname = "Hans";
hans.nachname = "Meier";
kunde->alter = 42;
```

# Unions

- eine *Union* ist wie eine Struktur, hält aber nur eines seiner Felder gleichzeitg
- alle Felder einer Union liegen auf dem gleichen Stück Speicher
- ansonsten funktionieren sie exakt wie Strukturen
- wir gehen nicht näher darauf ein

# Unzulässiges Zeugs

Was passiert...

- wenn man durch Null teilt?
- wenn man eine uninitialisierte Variable ließt?
- wenn man hinter das Ende eines Arrays schreibt?
- wenn man auf freigegebenen Speicher zugreift?
- ...

# Undefiniertes Verhalten

Antwort: *Das Verhalten ist undefiniert.*

- die Sprache C trifft keine Aussagen darüber, was dann passiert
- alles ist erlaubt, von »garnichts« über »Festplatte formatieren« bis »kleine Teufelchen fliegen aus deiner Nase«
- korrekte C-Programme enthalten kein undefiniertes Verhalten
- passt also beim Programmieren auf!

# sonstiges uneindeutiges Zeugs

Es gibt noch zwei weitere Formen uneindeutigen Verhaltens:

- bei *implementationsabhängigem Verhalten* (z. B. Größe von Typen, negative Zahlen nach rechts schieben) muss die Implementierung der Sprache C eine Entscheidung treffen und dokumentieren. Das Verhalten ist dann immer so wie dokumentiert.
- bei *unspezifiziertem Verhalten* (z. B. Reihenfolge der Termauswertung und Variableninitialisierung) gibt es mehrere erlaubte Verhaltensweisen, der Kompiler darf sich jedes mal eine Verhaltensweise beliebig auswählen.