

Funktionen

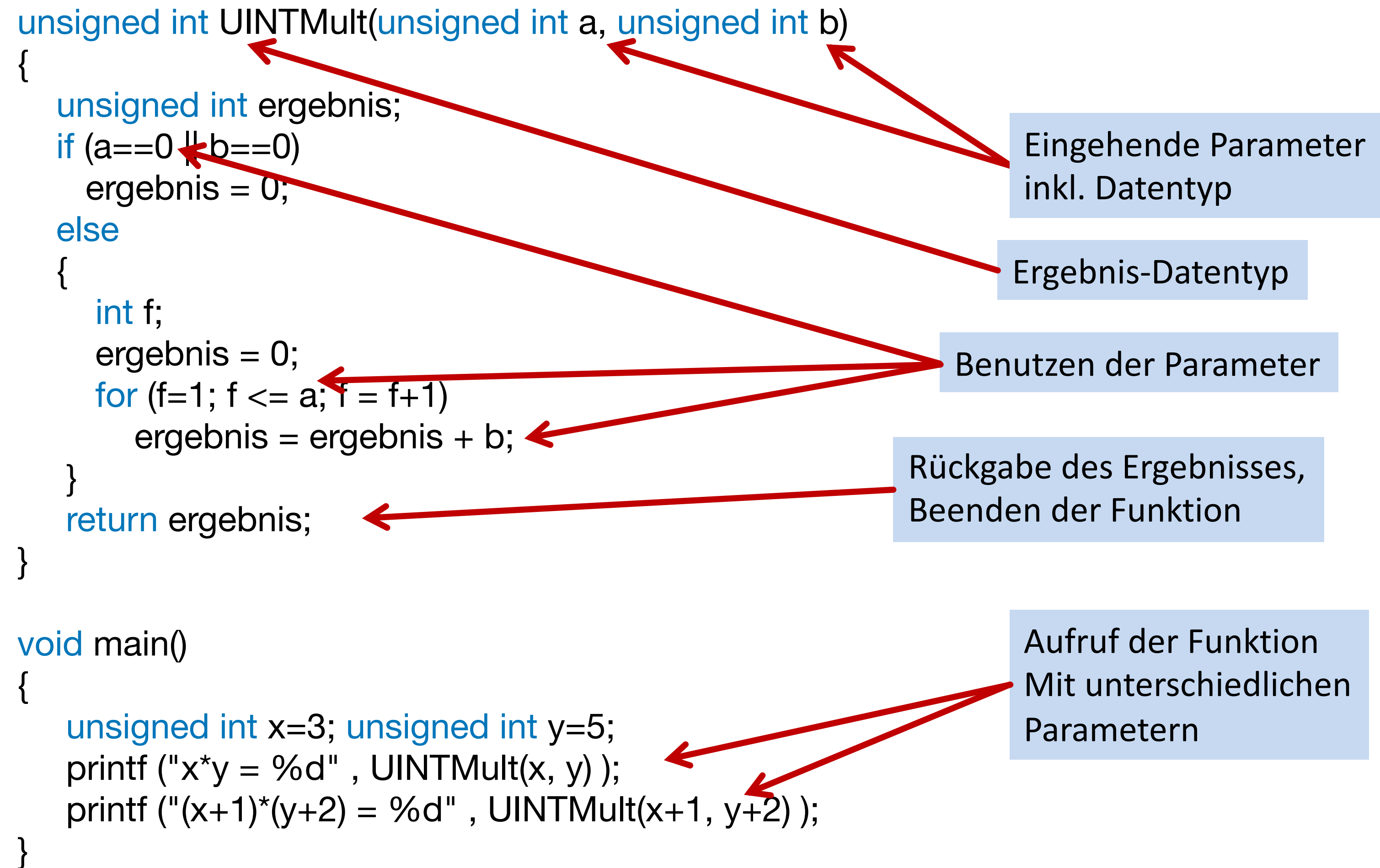
Funktionen

- Wenn Programme groß werden, werden sie sehr schnell unübersichtlich
- Wenn andere Programmierer bereits Algorithmen implementiert haben, möchte man die einfach wiederverwenden können
- -> Strukturierung erforderlich!
- Einführung von Funktionen (Unterprogramme, Prozeduren)
- Mit Funktionen gelingt eine Strukturierung und Wiederverwendung
 - Programme lösen Probleme
 - Funktionen lösen Teilprobleme des Problems

Funktionen

- Funktionen werden über **Aufrufe** aktiviert und ausgeführt
- Funktionen werden **unabhängig von konkreten Werten der Daten geschrieben** (Implementierung der Multiplikation von **int**-Zahlen muss nicht wissen, welche konkreten Zahlen zu multiplizieren sind, sondern nur, dass es **int**-Zahlen sind)
- Funktionen können daher **keine, ein** oder **mehrere** Daten übergeben werden
- Funktionen liefern **genau ein** Datum zurück (**return**)

Beispiel Funktion



Ablauf beim Aufruf

- Aufruf einer Funktion ist eine Expression!
- Der Aufruf einer Funktion bewirkt das folgende
 - Parameter der Funktion werden mit den Übergabewerten initialisiert
 - Programmausführung geht bei der ersten Anweisung der Funktion weiter
 - Funktion berechnet ein Ergebnis
 - Ergebniswert wird mittels **return** zurückgegeben und gleichzeitig wird der Rest der Anweisung im aufrufenden Programmteil weiter ausgeführt

Ablauf beim Aufruf - Beispiel

```
unsigned int UINTMult(unsigned int a, unsigned int b)
{
    unsigned int ergebnis;
    if (a==0 || b==0)
        ergebnis = 0;
    else
    {
        int f;
        ergebnis = 0;
        for (f=1; f <= a; f = f+1)
            ergebnis = ergebnis + b;
    }
    return ergebnis;
}
```

```
void main()
{
    unsigned int x=3; unsigned int y=5;
    printf ("x*y = %d" , UINTMult(x, y) );
    printf ("(x+1)*(y+2) = %d" , UINTMult(x+1, y+2) );
}
```

- 1. Aufruf: a = 3 (von x) und b = 5 (von y)
- 2. Aufruf: a = 4 und b = 7
- Die Werte werden als neue Variablen in der Funktion initialisiert
- unsigned int a = 3
- unsigned int b = 5

Technischer Ablauf

- Für jeden Aufruf einer Funktion wird ein sogenannter Aktivierungsblock im Speicher (Frame) angelegt
- Dieser enthält:
 - Platz für alle lokalen Variablen der Funktion
 - Platz für alle Parameter einer Funktion
 - Platz für die Rücksprungsadresse
- Beim Aufruf einer Funktion wird dieser Frame angelegt, die Werte der Parameter an ihren Platz geschrieben und die Rücksprungsadresse notiert
- Bei Beendigung der Funktion wird der Frame wieder abgebaut
- Die Rücksprungsadresse wird gebraucht, weil das Laufzeitsystem wissen muss, von wo aus die Funktion aufgerufen wurde
- Anmerkung: main ist auch nur eine Funktion, aufgerufen vom BS

Frame-Beispiel

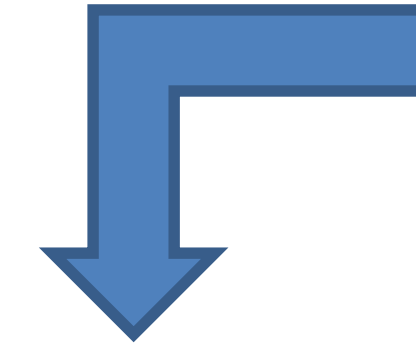
UINTMult(x,y) ;

```
unsigned int UINTMult(unsigned int a, unsigned int b)
```

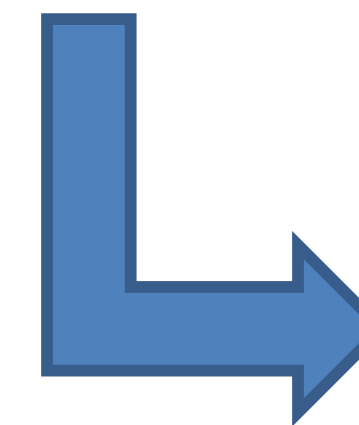
```
{  
    unsigned int ergebnis;  
    if (a==0 || b==0)  
        ergebnis = 0;  
    else  
    {  
        int f;  
        ergebnis = 0;  
        for (f=1; f <= a; f = f+1)  
            ergebnis = ergebnis + b;  
    }  
    return ergebnis;  
}
```

```
void main()
```

```
{  
    unsigned int x=3; unsigned int y=5;  
    printf ("x*y = %d" , UINTMult(x, y) );  
    printf ("(x+1)*(y+2) = %d" , UINTMult(x+1, y+2) );  
}
```



UINTMult			
a = 3	b = 5	ergebnis = 15	Rücksprung Adresse



return ergebnis;

main		
x = 3	y = 5	Rücksprung Adresse

main		
x = 3	y = 5	Rücksprung Adresse

main		
x = 3	y = 5	Rücksprung Adresse

Lokalität von Parametern

- Eingehende Parameter werden in der Funktion immer lokal behandelt, haben also keine Seiteneffekte

- Beispiel:

```
void func(int a, int b)
{
    a = a + 42;
    b = 42;
}
```

Ergebnis-Datentyp leer
(keine Rückgabe)

Parameter-Inhalte verändern

```
void main()
{
    int x,y;

    x = 3; y = 5;
    func(x,y);
    printf(„x=%d und y=%d“,x,y);
}
```

Funktionsaufruf

Keine Seiteneffekte,
Ausgabe: *x=3 und y=5*

Definition des Interfaces

- Die sogenannte Signatur einer Funktion klärt,
 - Welche Parameter gehen rein
 - In welcher Reihenfolge
 - Von welchem Datentyp
 - Welcher Datentyp wird zurückgeliefert
- Datentypen beim Instanzieren der Parameter müssen grundsätzlich kompatibel sein
- Auf die Reihenfolge achten (siehe printf-Beispiel)
- Der erwartete Datentyp des Aufrufers muss kompatibel dem Rückgabe-Datentyp der Funktion sein

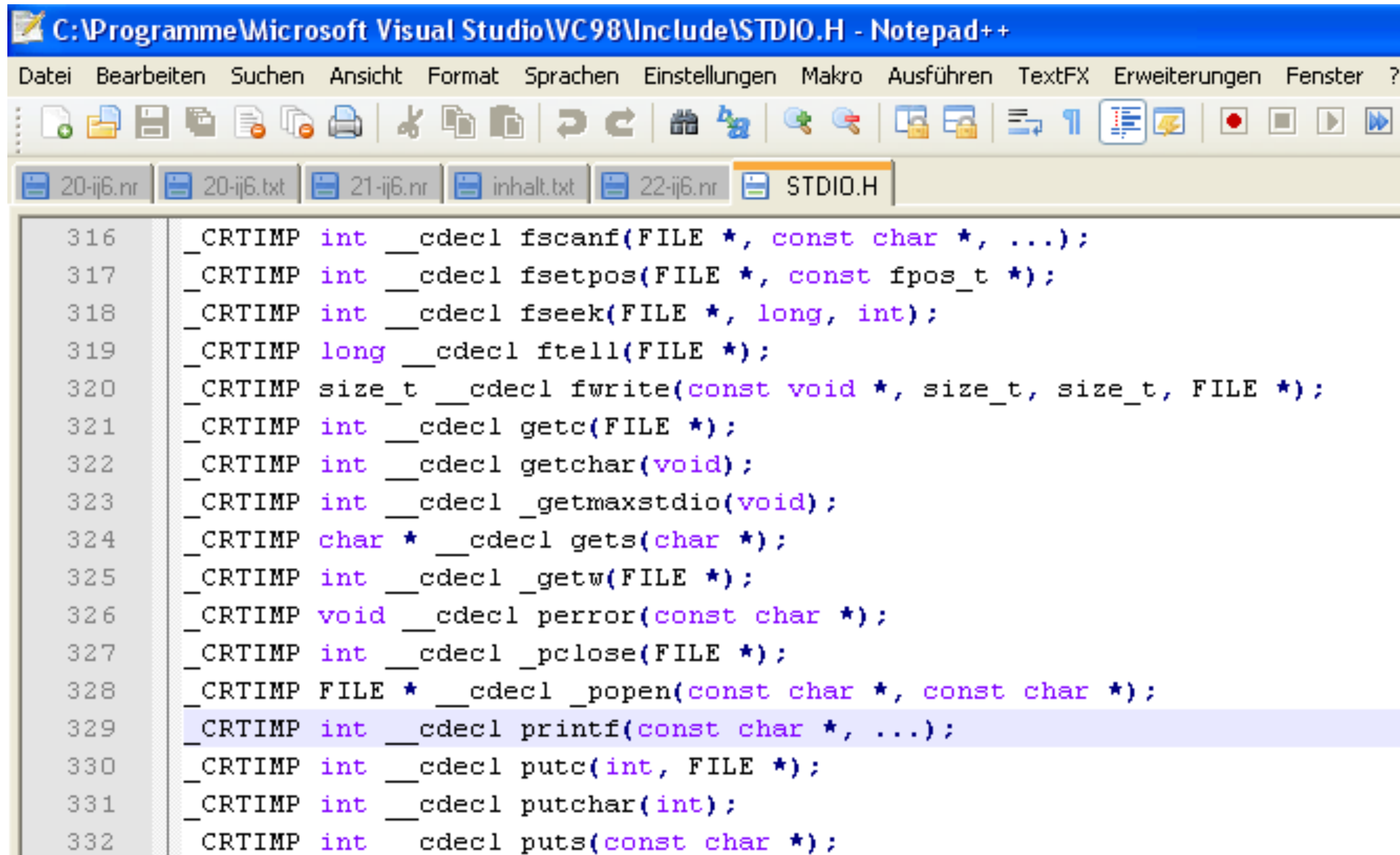
```
unsigned int UINTMult(unsigned int a, unsigned int b)
```

Definition des Interfaces

- Wohin mit der Implementierung einer Funktion?
 - Compiler muss die Signatur kennen (wie alle Dinge mit Namen), damit er das übersetzen kann
- 3 Möglichkeiten:
 1. Vor die main()-Funktion Compiler kennt Signatur u. Implementierung
 2. Hinter die main()-Funktion Compiler kennt nur Signatur.
Implementierung wird später übersetzt
oder vom Linker eingebunden
 3. In einer anderen Datei
- In Fall 2 und 3: woher kennt der Compiler dann die Signatur?
- Vorabdeklaration (ohne Implementierung) vor main()

```
unsigned int UINTMult(unsigned int a, unsigned int b)
```

Definition des Interfaces

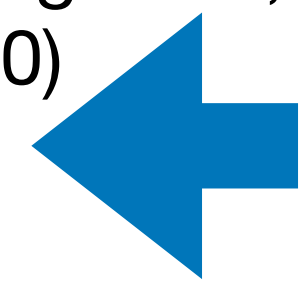


```
316 _CRTIMP int __cdecl fscanf(FILE *, const char *, ...);
317 _CRTIMP int __cdecl fsetpos(FILE *, const fpos_t *);
318 _CRTIMP int __cdecl fseek(FILE *, long, int);
319 _CRTIMP long __cdecl ftell(FILE *);
320 _CRTIMP size_t __cdecl fwrite(const void *, size_t, size_t, FILE *);
321 _CRTIMP int __cdecl getc(FILE *);
322 _CRTIMP int __cdecl getchar(void);
323 _CRTIMP int __cdecl _getmaxstdio(void);
324 _CRTIMP char * __cdecl gets(char *);
325 _CRTIMP int __cdecl _getw(FILE *);
326 _CRTIMP void __cdecl perror(const char *);
327 _CRTIMP int __cdecl _pclose(FILE *);
328 _CRTIMP FILE * __cdecl _popen(const char *, const char *);
329 _CRTIMP int __cdecl printf(const char *, ...);
330 _CRTIMP int __cdecl putc(int, FILE *);
331 _CRTIMP int __cdecl putchar(int);
332 _CRTIMP int __cdecl puts(const char *);
```

Beenden einer Funktion: **return**

- **return** liefert nicht nur Ergebnis an den Aufrufer, sondern beendet auch Funktion
- Damit sind auch Ausstiege an jeder Stelle der Schleife möglich
- Auch innerhalb von wie tief auch immer geschachtelten Schleifen möglich!

```
unsigned int UINTMult(unsigned int a, unsigned int b)
{
    unsigned int ergebnis;
    if (a==0 || b==0)
        return 0;
    else
    {
        int f;
        ergebnis = 0;
        for (f=1; f <= a; f = f+1)
            ergebnis = ergebnis + b;
        }
    return ergebnis;
}
```



```
void main()
{
    unsigned int x=3; unsigned int y=5;
    printf ("x*y = %d" , UINTMult(x, y) );
    printf ("(x+1)*(y+2) = %d" , UINTMult(x+1, y+2) );
}
```


Geschachtelte Aufrufe

- Genau wie Schleifen geschachtelt werden können, können Funktionen andere Funktionen aufrufen

```
int A(int x) { ... }
```

```
int B(int a, int b)
{
    ...
    v = A(a);
}
```

```
void C(int y)
{
    ...
    B(A(w),y);
}
```

Aufruf-Reihenfolge für: C(B(12));

B für Parameter von C

A von B aus

C

A von C aus für 1. Parameter

B von C aus

A von B aus

- Vorsicht bei Aufruf-Zyklen, z.B. in A wird B aufgerufen!

Geschachtelte Aufrufe

- Genau wie Schleifen geschachtelt werden können, können Funktionen andere Funktionen aufrufen

```
int A(int x) { ... }
```

```
int B(int a, int b)
```

```
{
```

```
    v = A(a);
```

```
}
```

```
void C(int y)
```

```
{
```

```
    ...
```

```
    B(A(w),y);
```

```
}
```

Aufruf-Reihenfolge für: C(B(12));

B für Parameter von C

C

A von C aus für 1. Parameter

B von C aus

A von B aus

Daher: Keine dummen Namen geben!!! Sonst wird es unverständlich!

- Vorsicht bei Aufruf-Zyklen, z.B. in A wird B aufgerufen!

Standard-Funktionen

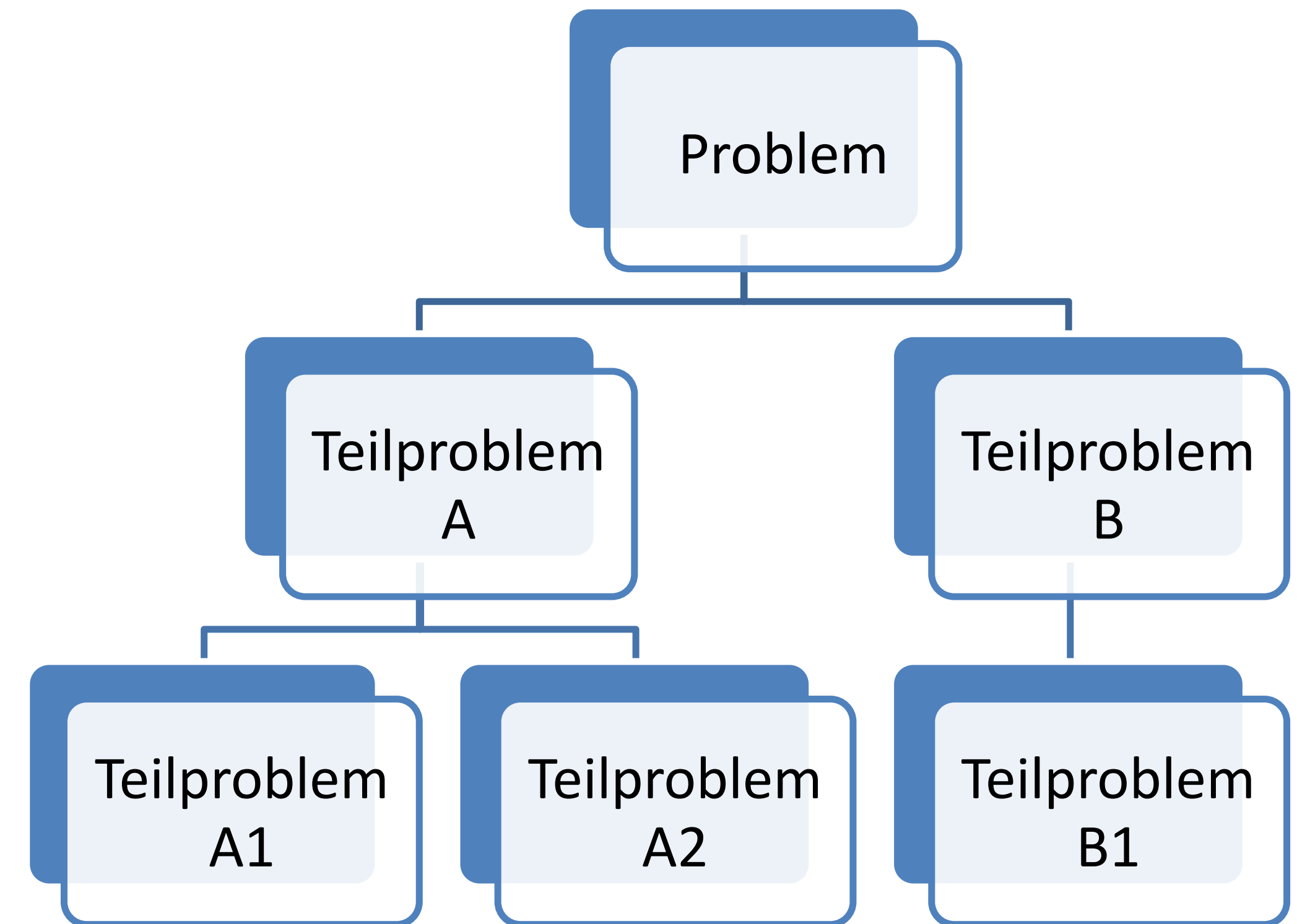
- Funktionen werden auch benutzt, bereits vorbereitete Implementierungen dem Programmier zur Verfügung zu stellen
- Beispiele:
 - Benutzung des Betriebssystems für Ein- und Ausgabe (z.B. printf, scanf) auf Bildschirm/Tastatur, sowie File-System (stdio.h bietet hier eine Vielzahl an Funktionen an)
 - Entwicklung graphischer Oberflächen (man möchte nicht jeden Bildpunkt selbst immer wieder berechnen)
 - Operationen auf Gleitkomma-Zahlen (z.B. math.h)

Mathematische Funktionen aus math.h

<code>sin(x)</code>	Sinus
<code>cos(x)</code>	Cosinus
<code>tan(x)</code>	Tangens
<code>asin(x)</code>	Arcussinus für $-1 \leq x \leq +1$
<code>acos(x)</code>	Arcuscosinus für $-1 \leq x \leq +1$
<code>atan(x)</code>	Arcustangens für $-\pi/2 < x < \pi/2$
<code>sinh(x)</code>	Sinus hyperbolicus
<code>cosh(x)</code>	Cosinus hyperbolicus
<code>tanh(x)</code>	Tangens hyperbolicus
<code>exp(x)</code>	Exponentialfunktion
<code>log(x)</code>	natürlicher Logarithmus für $x > 0$
<code>pow(x,y)</code>	x hoch y
<code>sqrt(x)</code>	Quadratwurzel
<code>ceil(x)</code>	Kleinste ganze Zahl i mit $x \leq i$
<code>floor(x)</code>	Größte ganze Zahl i mit $i \leq x$
<code>fabs(x)</code>	Absolutbetrag von x

Anwendung von Funktionen

- Üblicherweise löst man Probleme durch Aufteilung des Problems in einfachere Teilprobleme
- EVA-Prinzip
 - Eingabe
 - Verarbeitung
 - Ausgabe
- Zerteilung nennt man Funktionsblockzerlegung
 - Übersichtlichere Programme
 - Schrittweise Verfeinerung bis zur Lösung des Problems



Zusammenfassung Funktionen

- Ein Programm kann neben Anweisungen auch Funktionsaufrufe abarbeiten
- Funktionen haben formale Parameter, denen ein Wert übergeben wird (und nur der Wert, nicht das Objekt)
- Funktionen implementieren eine Lösung für bestimmte Probleme unabhängig vom Kontext in dem sie aufgerufen werden
- Funktionen dienen
 - zur Strukturierung eines Programmes (Lesbarkeit!)
 - dazu, Probleme durch Anwendung von Lösungen für Teilprobleme zu lösen
 - zur Wiederverwendung von bereits vorhandenen Lösungen für Probleme