



Grundlagen der Programmierung in C

Pointer & Funktionen

Wintersemester 2005/2006
G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de



- Aufruf einer Funktion mit Parameter bisher

```
void foo( Typ x )  
{  
    ...  
}  
  
int main( )  
{  
    Typ obj;  
    ...  
    foo( obj );  
}
```

- Was genau passiert dabei?
- Kopieren und Wegwerfen



Call-by-Value

- *Call-by-Value* : tatsächliche Parameter werden kopiert und hinterher weggeworfen
- Problem: "große" Parameter, z.B. **structs**

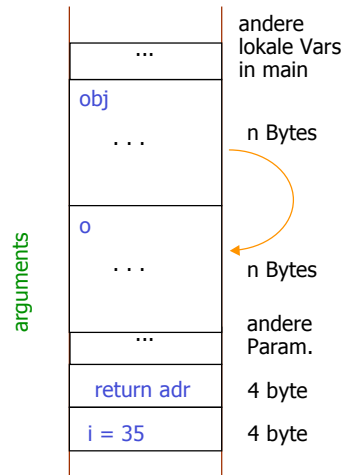
```

struct someStruct { ... };

void foo( someStruct o, ... )
{
    int i = sizeof( o );
    ...
}

int main( )
{
    ...
    someStruct obj; ...
    foo( obj, ... );
}

```



Call-by-Reference

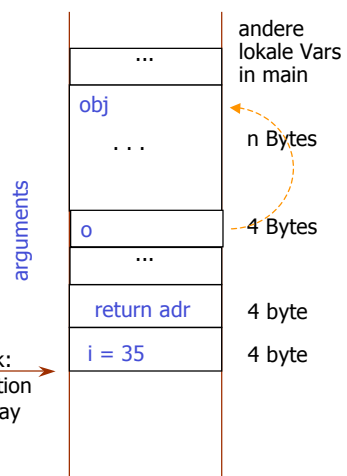
- Lösung: übergebe nur Pointer
- Heißt *Call-by-reference*

```

void foo( someStruct * o, .. )
{
    int i = sizeof( *o );
    ...
}

int main( )
{
    ...
    someStruct obj; ...
    foo( &obj, ... );
}

```





Beispiel: Nochmal Anhängen an Liste



- Selbe Aufgabe wie im vorigen Kapitel (dyn. Speicher)
- Operationen auf Listen sind fast immer Kandidaten für Funktionen!
- Erinnerung:

```
// n int's lesen und an list anhaengen
for ( int i = 0; i < n; i ++ )
{
    int a;
    scanf( "%d", &a );

    // neues ListElement mit a generieren
    ListElement* e = new ListElement;
    e->data = a;
    e->next = NULL;

    // e an list anhaengen
    list->last->next = e;
    list->last = e;
}
```



- Besser als Funktion:

```
for ( int i = 0; i < n; i ++ )
{
    int a;
    scanf( "%d", &a );
    append( list, a );
}
```

```
ListElement* append( List * list, int a )
{
    ListElement* e = new ListElement;
    e->data = a;
    append( list, e );
    return e;
}
```

```
void append( List * list, ListElement * e )
{
    list->last->next = e;
    list->last = e;
    e->next = NULL;
}
```



Rückgabe von Pointern und Referenzen



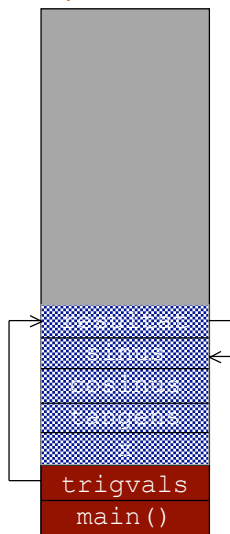
- Vorsicht! Beispiel:

```
struct TrigVals
{
    double sinus, cosinus, tangens;
};

TrigVals* calcTrigVals( double x )
{
    TrigVals resultat;
    resultat.sinus = sin( x );
    resultat.cosinus = cos( x );
    resultat.tangens = tan( x );
    return & resultat;           // BUG!
}
```



The Stack



```
int main()
{
    TrigVals* trigvals;
    trigvals = calcTrigVals( 0.5 );

    TrigVals * calcTrigVals( double x )
    {
        TrigVals resultat;
        . . .
        return & resultat;
    }

    // back in main()
    printf("sin(x) = %f\n", trigvals->sin );
}
```

BOOM!

- Bessere Lösung:

```
TrigVals* calcTrigVals( double x )
{
    static TrigVals resultat;
    resultat.sinus = sin( x );
    resultat.cosinus = cos( x );
    resultat.tangens = tan( x );
    return & resultat;           // besser
}
```

- Aber: nicht thread-safe!

- Beste Lösung: "return by value"

```
TrigVals calcTrigVals( double x )
{
    TrigVals resultat;
    resultat.sinus = sin( x );
    ...
    return resultat;
}
```

- Oder: "Out-Parameter"

```
void calcTrigVals( double x ,
                  TrigVals * trigvals )
{
    resultat->sinus = sin( x );
    ...
}
```

- Aufrufer muß Speicherplatz reservieren



- Fazit:
 - Gebe nie Pointer (oder Referenz) auf lokale Variable zurück!
 - Referenzen sind noch schlimmer (man sieht es ihnen nicht an)
 - Gebe auch keinen Pointer auf eine lokale `static` Variable zurück (auch nicht als Out-Parameter)
- Return-by-Value macht heute keine Kopie mehr (war früher ein Performance-Problem)