

# Programmiertechniken 1

Pointer

# Learning Outcomes

---

Sie können ...

- ... die Adresse einer Variablen bestimmen.
- ... über eine Adresse auf einen Inhalt zugreifen.
- ... Parameter per Pointer übergeben.
- ... Funktionen mit Rückgabeparameter erstellen.
- ... einen Buffer an eine Funktion übergeben.

# Probleme...

---

- Erstellen Sie eine Funktion, die den Inhalt zweier Variablen tauscht (swap).
- Erstellen Sie eine Funktion, die die Anzahl aller Vokale in einem String bestimmt und zurückgibt.
- Erstellen Sie eine Funktion, die das häufigste Zeichen in einem 1 GB großen String bestimmt.

## Probleme in C:

- Kopieren kostet Zeit.
- Funktionen arbeiten auf Kopien → Kopien vertauschen ...
- Mehr als einen Rückgabewert lässt `return` nicht zu.

# Überlegungen

---

Alle **globalen Variablen** liegen irgendwo im Speicher.

Alle **lokalen Variablen** einer Funktion in der Aufrufkette liegen irgendwo im Speicher.

Den Variablen werden **Speicherplätze** zugeordnet. Über die **Bezeichner** spricht man die **Speicherplätze** an. Für den Zugriff werden intern die **Adressen der Speicherplätze** verwendet.

➔ Kennt man die **Speicheradresse** einer Variablen, so kann man bei Unkenntnis des Bezeichner trotzdem auf den Inhalt der Variablen zugreifen.

Speicheradressen kann man wieder in Variablen ablegen, sogenannten **Pointer-Variablen**.

➔ Eine **Pointer-Variable** **ist** eine Variable, die eine Adresse speichert!

# Pointer

---

Eine **Pointer-Variable** **ist** eine Variable, die eine Adresse speichert!

Damit das Ziel richtig interpretiert wird, benötigt man einen Typ.

Definition:

`<type> *<pointer name>;`

Der \* markiert die Variable als  
Pointer des Typs.

Beispiel:

`char *letter;`

„Pointer auf ein char“

Pointer werden auch initialisiert!

Beispiel:

`char *letter = NULL;`

(Hinweis: Intern haben Pointer immer die gleiche Größe, z. B. 4 Byte, egal welchen Typ sie referenzieren.)

# Referenzierung/Dereferenzierung

---

Die **Adresse** einer Variablen kann mittels **Referenzoperator** „&“ bestimmt werden. Beispiel:

```
int x = 5;  
Int *value = &x;    /* value referenziert nun x */
```

Auf den **Wert der Speicherstelle**, die ein Pointer referenziert, kann über den **De-Referenzierungsoperator** „\*“ zugegriffen werden.

Beispiel:

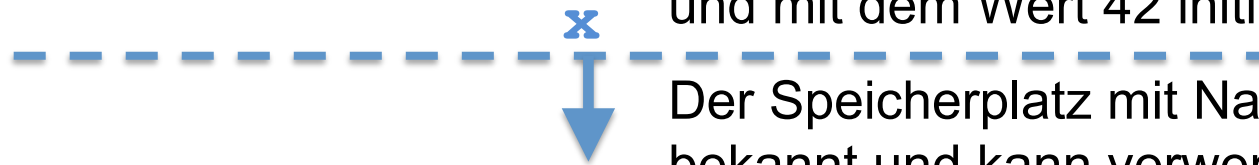
```
int x = 5;  
int *value = &x;  
int y = *value; /* y wird der Wert des referenzierten  
                Elements zugewiesen */  
*value = 23;    /* Dem referenzierten Element wird  
                23 zugewiesen */
```

# Pointer

Problemstellung: Ein Speicherplatz soll über zwei Bezeichner angesprochen werden.

```
int x = 42;
```

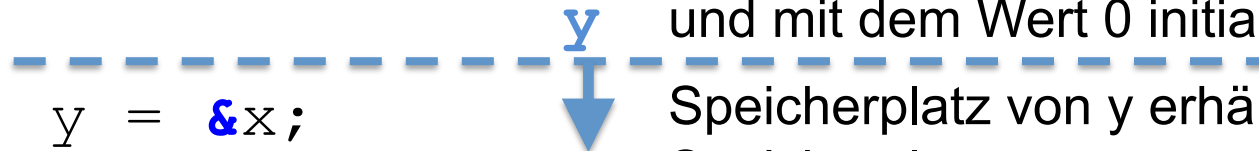
Speicherplatz mit dem Namen „x“ wird belegt und mit dem Wert 42 initialisiert.



Der Speicherplatz mit Namen „x“ ist ab hier bekannt und kann verwendet werden.

```
int *y = Null;
```

Speicherplatz mit dem Namen „y“ wird belegt und mit dem Wert 0 initialisiert.



```
y = &x;
```

Speicherplatz von y erhält die Adresse des Speicherplatzes von x zugewiesen.

➔ y **verweist** auf Speicherplatz von x!

```
*y = 21;
```

Interpretiere Wert von y als Speicherplatz und weise dem Speicherplatz den Wert 21 zu.

➔ Speicherplatz mit Namen x wird modifiziert.

```
printf("%i\n", x);
```

# Pointer als Parameter

Ist die Adresse eines Wertes (Variablen) bekannt, kann dieser unabhängig vom Scope manipuliert werden!

→ Werden Adressen an Funktionen übergeben, können sie den referenzierten Bereich verändern (mit allen schädlichen Konsequenzen).

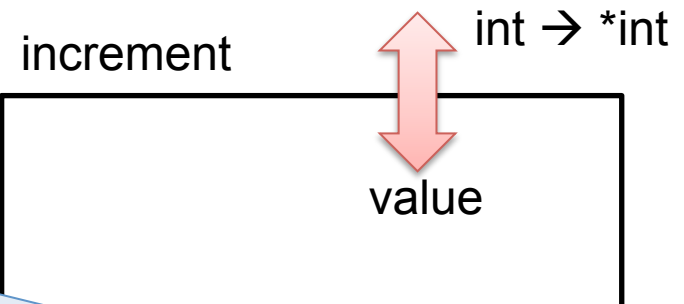
→ Funktion erwartet eine zu manipulierende Adresse (**Call by Reference**)

Beispiel:

```
void increment( int* value ) {  
    (*value) += 1;  
}
```

```
increment(&x);
```

```
int* referenz = &x;  
increment(referenz);
```



Lesung: „Es wird eine Referenz auf einen Integerwert erwartet.“

Lesung: „Die Adresse von x wird an die Funktion increment übergeben.“



# Argument zu groß

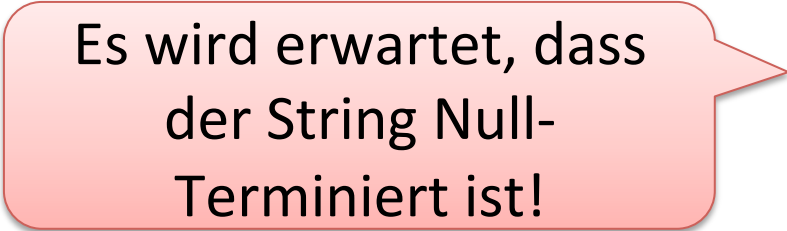
Bei der String-Verarbeitung ist es oft teuer eine Kopie zu erstellen. Auch ist die benötigte Größe unbekannt.

→ Lesendes Arbeiten auf dem Original. Absicherung über `const`.

Compiler prüft, ob es schreibende Zugriffe gibt.

Hinweis: Der Typ `char[]` wird als `char*` behandelt.

```
int length( const char* textString) {  
    int counter = 0;  
    while( textString[counter] != 0) {  
        counter ++;  
    }  
    return counter;  
}
```



Es wird erwartet, dass  
der String Null-  
Terminiert ist!

```
int main(void) {  
    char text[] = "Hallo";  
    int x = length(text);  
    return 0;  
}
```

# Buffer als Rückgabewert

---

Das Ergebnis kann nicht als ein einzelner Wert zurückgegeben werden. Beispielsweise in der String-Verarbeitung. Es wird ein Buffer als Ziel übergeben. Die maximale Größe sollte auch übergeben werden!

```
void firstWord( const char* textString,
               char* buffer,
               int size){
    int counter = 0;
    while( textString[counter] != ' ' &&
           counter < size-1 &&
           textString[counter] != 0 ){
        buffer[counter] = textString[counter];
        counter ++;
    }
    buffer[counter] = '\\0'; // zero-termination
}
```