

## 1. Elementares C++

### 1.2. Datentypen

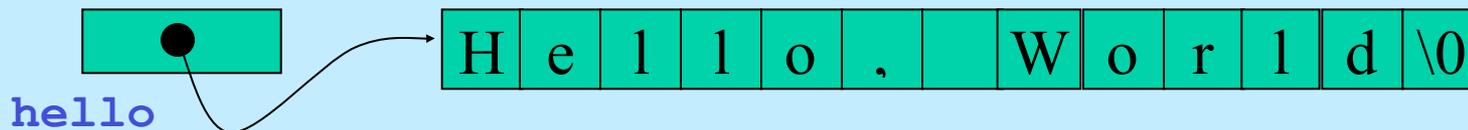
Zeichenketten:

Zeichenkettenlitterale wie »üblich«

"eine Zeichenkette\nmit Doppelapostroph \" und Backslash \\"

werden als 0-terminierte **char**-Felder realisiert !

```
char* hello = "Hello, World";
```

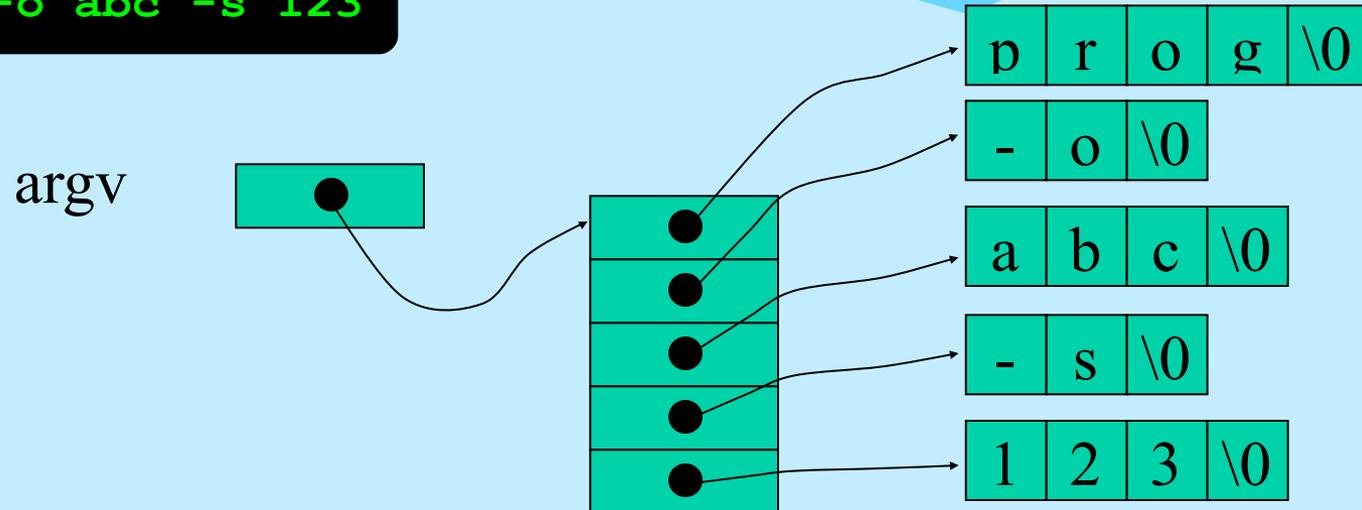


## 1. Elementares C++

### 1.2. Datentypen (Zeichenketten)

```
int main (int argc, char* argv[]) // bzw.  
int main (int argc, char** argv)
```

```
$ prog -o abc -s 123
```



## 1. Elementares C++

### 1.2. Datentypen (Zeichenketten)

Damit ist bereits der Umgang mit Zeichenketten implizit mit allen Problemen der Zeiger belastet (und zusätzlich mit allen buffer overflow Problemen bei Operationen auf Zeichenfeldern)

Außerdem sind die möglichen Operationen auf `char[]` C-legacy (`<cstring>`) und primitiv, z.B. `strcpy` == Kopieren von Zeichenketten

etwa:

```
void strcpy (const char* source, char* dest)
{ while (*dest++=*source++); }
```

## 1. Elementares C++

### 1.2. Datentypen (std::string)

AUSWEG: Datentyp `std::string` (<string>)

- eine Standardklasse zur Verarbeitung von Strings
- etwa auf dem Niveau von `java.lang.String` mit der
- Möglichkeit der Initialisierung aus C-Strings

```
std::string vorname = "bjarne";
```

- und einer Vielzahl von Operationen (a la Java):

```
std::string nachname = "Stroustrup";
```

## 1. Elementares C++

### 1.2. Datentypen (std::string)

```
std::string name; // noch leer !  
  
vorname[0]='B'; // unchecked !  
vorname.at(0) = 'B'; // checked  
name = vorname + " " + nachname;  
if (name != "")  
    std::cout << name << std::endl;  
int l = name.length(); // ohne 0-Byte !  
  
"hallo" + ", World\n"; // ERROR  
string("hallo") + ", World\n"; // OK  
  
const char* cstring = name.c_str();
```

... Vergleich, Suche, I/O ... ↪ <http://de.cppreference.com>

## 1. Elementares C++

### 1.2. Datentypen

#### Referenztypen:

Eine Neuerung gegenüber C, Aliasnamen für Objekte mit Referenzsemantik ähnlich zur primären Objektsemantik von Java, aber

- für alle Typen (incl. build-in Typen)
- es gibt KEINE 'Nullreferenz'

in Anlehnung an die Syntax von Zeigervereinbarungen

```
int i=42;  
// int& ri;  
// ERROR: Referenzen MÜSSEN initialisiert werden  
int& ri = i; // i alias ri  
class X{...} x; X& y=x; // zwei Namen für ein Objekt
```

## 1. Elementares C++

### 1.2. Datentypen

#### Konstantentypen:

Ein Typ **T** wird durch den Präfix **const** zu einem Konstantentyp, Objekte solcher Typen sind unveränderlich (per statischer Kontrolle durch den Compiler)

für Argumente von Funktionen bedeutet dies, dass die Funktion

1. die (nachprüfbare) Zusicherung gibt, dieses Argument NICHT zu verändern
2. beim Aufruf für das Argument auch konstante Objekte benutzt werden dürfen (was für non-const nicht erlaubt ist, weil ja die Funktion keine Zusicherung gegeben hat und daher ...)

```
const double pi=3.1415926; double someMathFkt(double);  
const double x = someMathFkt(pi); // call by value !
```

## 1. Elementares C++

### 1.2. Datentypen (Konstantentypen)

Konstante Objekte müssen initialisiert werden (weil eine spätere Zuweisung nicht erlaubt ist)

konstante Objekte können auch über Zeiger nicht verändert werden, weil die Adresse einer `const T` Variablen vom Typ `const T*` ist

```
double* dp = &pi; // ERROR  
*dp = 33.3;
```

```
const double* cdp = &pi;  
*cdp = 33.3; // ERROR
```

## 1. Elementares C++

### 1.2. Datentypen (Konstantentypen)

bei Zeigern ist wohl zu unterscheiden zwischen der constness des Zeigers selbst

```
int * const constant_pointer = &someint;
```

und der constness des referenzierten Objektes (Feldes)

```
const int * pointer_to_constant;
```

```
const int * const constant_pointer_to_constant = ...;
```

## 1. Elementares C++

### 1.2. Datentypen (Konstantentypen)

Referenzen (selbst) sind implizit const, es gibt jedoch Referenzen auf Konstantentypen

Wichtigste Anwendung: call by reference in-parameter

```
T t;  
void foo(T& pt)  
{  
    pt.change();  
}  
foo(t); // call by reference: t itself changes  
const T ct;  
foo(ct); // ERROR
```

## 1. Elementares C++

### 1.2. Datentypen (Konstantentypen)

```
void foo(const T& pct)
{
    // pct.change(); ERROR
    pct.read_only();
}
```

```
const T ct;
foo(ct); // OK
```

```
class X{ public: void foo() const; void bar(); };
X x; const X cx; x.foo(); x.bar();
cx.foo(); /* OK */ cx.bar() // ERROR !
```

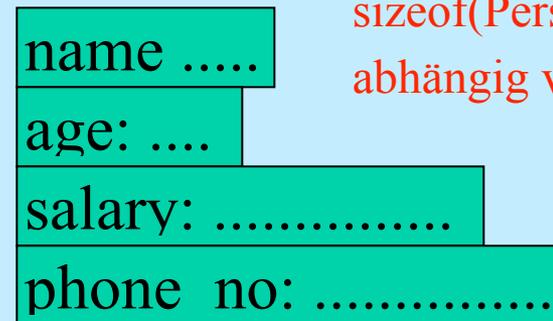
## 1. Elementares C++

### 1.2. Datentypen

Strukturtypen (a la C):

heterogene Wertekombinationen unter einem Typnamen

```
struct Person {  
    std::string name;  
    int age;  
    double salary;  
    long phone_no;  
} p;
```



`sizeof(Person)`

abhängig vom *alignment* !