

Algorithmen und Datenstrukturen in C/C++

Prof. Dr. rer. nat. Peter Gerwinski

7. April 2022

1.3 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //
- Konstante:
`const int n = 5;`
`int prime[n] = { 2, 3, 5, 7, 11 };`
- Ab C++11: `constexpr`-Funktionen
C++11: darf nur aus einem einzigen `return`-Statement bestehen
→ `?:` statt `if`, Rekursion statt Schleife
C++-14: auch Verzweigungen und Schleifen erlaubt
- leere Parameterliste: `void` optional
in C: ohne `void` = Parameterliste wird nicht geprüft
- Operatoren `new` und `delete`
als Alternative zu den Funktionen `malloc()` und `free()`

1.4 Referenz-Typen

```
void calc_answer (int &answer)
{
    answer = 42;
}
```

... als Alternative zu ...

```
void calc_answer (int *answer)
{
    *answer = 42;
}
```

- Zeiger „verborgen“, übersichtlicher und sicherer
- Es gibt keinen **NULL**-Wert.
→ Für verkettete Listen u. ä.: Tricks erforderlich

1.5 Überladbare Operatoren und Funktionen

```
#include <iostream>
```

```
int main ()  
{  
    std::cout << "Hello, world!" << std::endl;  
    return 0;  
}
```

Bemerkungen:

- Compilieren mit `g++` statt `gcc`:
C++-Bibliotheken mit einbinden
- Der Operator `<<` hat normalerweise keinen Seiteneffekt, hier schon.

1.5 Überladbare Operatoren und Funktionen

```
#include <iostream>
```

```
struct vector
```

```
{  
    double x, y, z;  
};
```

```
vector operator + (vector u, vector v)
```

```
{  
    vector w = { u.x + v.x, u.y + v.y, u.z + v.z };  
    return w;  
}
```

- ++ wird zum Präfix-Operator.
- ++ mit zusätzlichem (ungenutzten) **int**-Parameter wird zum Postfix-Operator.

1.5 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

Für den Linker:
veränderte, eindeutige Namen

Wenn man das nicht will:
extern "C" { ... }

Optionale Parameter:

```
void print (const char *s = "\n")
```

```
{  
    printf ("%s", s);  
}
```

wird vom Compiler erledigt

1.6 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    cout << "Hello, _world!" << endl;
```

```
    return 0;
```

```
}
```

```
namespace my_output
```

```
{
```

```
    ...
```

```
}
```

```
using namespace my_output;
```

1.7 Objekte

```
struct TBase  
{  
};
```

```
struct TInteger: public TBase  
{  
    int content;  
};
```

```
struct TString: public TBase  
{  
    char *content;  
};
```


1.7 Objekte: Zugriffsrechte

- `public`, `private`, `protected`
nicht nur Bürokratie, sondern auch Kapselung
(Maßnahme gegen „Namensraumverschmutzung“)
- **`struct`**: standardmäßig `public`
`class`: standardmäßig `private`
- `friend`-Funktionen und -Klassen
- Klasse als Namensraum:
`static`-„Member“-Variable
`static`-„Methoden“
Deklarationen von z. B. Konstanten und Typen

1.7 Objekte: Konstruktoren und Destruktoren

- leerer Standard-Konstrutor
- *Copy-Konstruktor*
- Konstruktor-Aufruf als „Initialisierung“
- Konstruktor-Aufruf mit `new`
Destruktor-Aufruf mit `delete`
- automatischer Destruktor-Aufruf
beim Verlassen des Gültigkeitsbereichs

1.8 Strings

- **#include** <string>
- String-Klasse
- String-Konstante sind **const char ***
- C-kompatiblen String extrahieren: `c_str ()`
- In String schreiben: **#include** <sstream>, `ostringstream`