Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

20. September 2019

C Grundlagen

Inhalt

- Vergleich C und C++ und Java
- Hauptprogramm
- Kontrollstrukturen
- Typen
- Standard-Library
- Eigene Typen definieren
- Memory Management
- Funktionen
- Makros

Vergleich C und C++

C

- kompiliert zu Binärdatei
- Präprozessor
- primitive Typen
- schwach typisiert
- Prozedural
- Pointer
- `structs`
- Function-Pointer
- Standard-Library

C++

- kompiliert zu Binärdatei
- Präprozessor
- primitive Typen
- stark typisiert
- Prozedural, objektorientiert, funktional
- Pointer & Referencen
- Klassen, Vererbung, Polymorphie
- Function-Pointer, std::function, Lambdas
- Standard-Library mit Container und Algorithmen und etc.
- Namespaces, Exceptions
- Templates

Vergleich zu Java

- Binärdatei
- keine VM nötig
- hohe Performanz
- C/C++ Runtime (`libc` und `libc++`)
- hohe Plattformabhängigkeit
- direkter Zugriff auf System und Speicher
- kein Garbage-Collector
- dafür Smart-Pointer mit Reference-Counting

Dateinamen

`*.c`	C-Source-Files. Implementation von Funktionen.
`*.h`	C/C++-Header-Files. Enthält mehrfach benötigte Deklarationen (`struct`, globale Variablen) und wird nicht direkt kompiliert sondern in Source inkludiert.
`*.cpp`	C++-Source-Files. Implementation von Funktionen, Methoden, etc.
`*.hpp`	C++-Header-Files. Wird heute nur noch für Header-Only-Implementationen verwendet.

Hauptprogramm

```
1  int main(int argc, char* argv[]) {
2    ...
3  }
```

`argc`	Anzahl der Programm-Parameter.
`argv`	Array mit den Programm-Parametern als C-Strings. Der erste Parameter (`argv[0]`) ist immer der Pfad zum aufgerufenen Executable.

Die Parameter können auch weggelassen werden.

Kontrollstrukturen

Scope

```
{
  ...
}
```

Conditional

```
if (...) {
  ...
} else if (...) {
  ...
} else {
  ...
}
```

While-Loop

```
int i = 5;
while (i > 0) {
  i--;
}
```

Do-While-Loop

```
int i = 5;
do {
  i--;
} while (i > 0);
```

Kontrollstrukturen

For-Loop

```
for (int i = 0; i < 5; ++i) {
  ...
}
```

Switch-Case

```
switch (n) {
case 1:
  ...
  break;
case 2:
  return ...;
default: {
    ...
  }
  break;
}
```

Primitive Typen

Typ	x86	x64	Bemerkung
`bool`	1 Byte	1 Byte	`!= 0` $\rightarrow$ `true`
`char`	1 Byte	1 Byte	
`short`	2 Byte	2 Byte	
`int`	4 Byte	4 Byte	
`long`	4 Byte	8 Byte	z.B. auf Ubuntu 18.04 x64
`long long`	8 Byte	8 Byte	
`float`	4 Byte	4 Byte	
`double`	8 Byte	8 Byte	
`long double`	10 Byte	10 Byte	nicht überall gleich

Der Modifier `unsigned` kann bei allen Integern als prefix oder inplace verwendet werden.

Fixed width Typen

Typ	Size
`uint8_t`	1 Byte
`int8_t`	1 Byte
`uint16_t`	2 Byte
`int16_t`	2 Byte
`uint32_t`	4 Byte
`int32_t`	4 Byte
`uint64_t`	8 Byte
`int64_t`	8 Byte
`size_t`	unsigned max int
`uintptr_t`	unsigned Pointer (max int)
`intptr_t`	signed Pointer (max int)

Diese und weitere sind bereits definiert in: `stdint.h` / `stddef.h`

Standard-Library

```cpp
// C style
#include <stdint.h>

// C++ style
#include <cstdint>
```

- C sowie auch C++ bringen eine Standard-Library mit, welche Hilfs-Typen und Funktionen enthalten.
- Als Erweiterung von C hat auch C++ Zugriff auf alle C-Header.
- C-Header in der Standard-Library sind mit `.h` gespeichert.
- In einem C++-Projekt werden die Pendants ohne Extension dafür mit vorangehendem ‘c’ verwendet.
- Die C++ Pendants kapseln die Funktionen etc. in den Namespace `std`.

Arrays

- Nur statische Arrays möglich.
- Nur auf dem Stack möglich.
- Deklaration in Form: `<type> <name>[<size>]`
- `<size>` kann weggelassen werden, wenn direkt initialisiert.
- Auch Arrays von nicht-primitiven Typen möglich.

```
1  int manual[4];
2  int automatic[] = { 1, 2, 3 }
3
4  int third = automatic[2];
5  manual[3] = automatic[0];
```

enum

- Der ersten Konstante wird by-default 0 zugewiesen.
- Nachfolgende Werte werden inkrementiert.
- Jeder Konstante kann manuell ein Wert zugewiesen werden.
- Werte können auch doppelt vergeben werden.
- Konstanten sind im umgebenden Scope sichtbar.

```
1  enum Color {
2    blau,
3    gelb = 5,
4    gruen,
5  };
```

```
1  Color c = blau;
2  if (c != gruen) {
3    ...
4  }
```

struct

- Strukturen in C vereinen Daten-Felder als Member-Variablen.
- Zugriff auf Member über den `.`-Operator
- Variablen mit demselben Typ können zugewiesen werden.
- Strukturen können nicht nur primitive Typen enthalten.
- Der Speicherbedarf ist mindestens die Summe aller Member.

```c
struct Point {
  int x;
  int y;
  int z;
};
```

```c
Point p;
p.x = 1;
p.y = 2;
int tmp = p.x;

Point o;
o = p;
```

union

- Fast gleich wie `struct`, die Member teilen aber den Speicher.
- Es kann somit immer nur ein Member gesetzt werden.
- Gelesen kann aber ohne Gefahr von allen.
- Unions können nicht nur primitive Typen enthalten.
- Der Speicherbedarf ist gleich dem grössten Member.

```
1  union SharedData {
2    float value;
3    uint8_t raw[4];
4  };
5
6  SharedData data;
7  data.value = 3.1415;
8
9  uint8_t byte1 = data.raw[0];
```

typedef

- Definieren von Typ-Aliasen
- Kein echter neuer Typ
- Letztes Token definiert den Namen des neuen Alias

```
1  // bereits definiert in stdint.h
2  typedef unsigned short uint16_t;
3
4  uint16_t var;
5  var = 65000;
```

Memory-Management

- Stack und Heap
- Statisch → Stack
- Dynamisch → Heap

```
1  // stack
2  int a[4];
3  a[0] = 1;
4
5  // heap
6  int* b = malloc(4 * sizeof(int));
7  assert(b != nullptr);
8  b[0] = 1;
9  free(b);    // nicht automatisch
```

Funktionen

- Deklaration in Header
- Definition / Implementation in Source-Datei
- Inkludieren vor Verwendung

```
1  // func.h
2  int CalcSum(int a, int b);
3
4  // func.cpp
5  int CalcSum(int a, int b) {
6    return a + b;
7  }
8
9  // main.cpp
10 #include "func.h"
11 int main() {
12   return CalcSum(1, 3);
13 }
```

Makros

- Text-Ersetzung
- wird durch den Präprozessor durchgeführt
- kann schlecht gewartet und debugged werden
- kann nicht-offensichtliche Effekte haben

```
1  #define PRINT_INT(x) std::printf("%i", (x))
2
3  // Verwendung
4  PRINT_INT(12);
5
6  // Compiler-Intern
7  std::printf("%i", 12);
```