Naturwissenschaftlich Technische Fakultät

UNIVERSITÄT SIEGEN

Universität Siegen

C++ Workshop

Jonas Pöhler & Daniel Busch

28. Januar 2019

Grundlegendes

Sprache

- Sprachaufbau
- Datenstrukturen
- Klassen
- Input/Output

Projekt

Buildsysteme

Es gibt verschiedene „Buildsysteme“, mit denen man C++-Projekte kompilieren kann:

- Autotools
- CMake
- . . .

Der Einfachheit halber verwenden wir CLion, das uns einige Arbeit abnimmt und auf CMake aufsetzt.

Hello C++

Listing 1: „Hello World“-Progamm

```
#include <iostream>

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

Kontrollstrukturen

If-Statement

Listing 2: Beispiel für ein If-Statement

```
float n = 0;
std::cin >> n;
if (n < 10)
{
    std::cout << "Kleiner 10!" << std::endl;
}
else if (n < 20)
{
    std::cout << "Kleiner 20, aber groesser gleich 10!" << std::endl;
}
else
{
    std::cout << "Groesser gleich 20!" << std::endl;
}
```

Listing 3: Beispiel für eine While-Schleife

```
int n = 5;
int f_1 = 0;
int f_2 = 1;
while (n > 1)
{
    int f_tmp = f_2;
    f_2 = f_1 + f_2;
    f_1 = f_tmp;

    n--;
}

std::cout << "f_n = " << f_2 << std::endl;
```

Listing 4: Beispiel für eine For-Schleife

```
int sum = 0;
int n = 10;
for (int i = 1; i < 10; i++)
{
    sum += i; // entspricht sum = sum + i
}

std::cout << "Summe: " << sum << std::endl;
```

Kontrollstrukturen

Foreach-Schleife

Oder, wenn man einfach über ein Array (dazu später mehr) iterieren möchte:

Listing 5: Beispiel für eine Foreach-Schleife

```
std::vector<std::string> drinks = { "water", "juice", "tea", "coffee" };

for (std::string d : drinks)
{
    std::cout << d << std::endl;
}
```

Kontrollstrukturen

Logische Ausdrücke

	Operator
Logisches UND	a && b
Logisches ODER	a || b
Negation	!a
Gleich	a == b
Ungleich	a != b
Kleiner (gleich)	a < b (a <= b)
Größer (gleich)	a > b (a >= b)

Funktionen

Jede Funktion hat mindestens mit einem Rückgabetypen, einen Namen und, wenn der Rückgabetyp nicht void ist, einen Rückgabewert.

Listing 6: Prototyp einer C/C++-Funktion

```
<type> <function name>(<type> <parameter name>, ...)
{
    ...

    return <value>;
}
```

Funktionen

Beispielsweise könnte die Fakultät rekursiv so aussehen:

Listing 7: Fakultätsfunktion

```
unsigned int fac(unsigned int n)
{
    return n == 0 ? 1 : n ∗fac(n − 1);
}
```

Funktionen
Call-by-Value

Die eben betrachtete Fakultätsfunktion ist ein Beispiel für eine Funktion mit Call-by-Value-Parametern:

- Der Wert wird der an fac übergeben wird, wird immer kopiert (sofern es sich um Variablen handelt)
- Der Wert kann von fac nicht überschrieben werden

Funktionen
Call-by-Reference

Angenommen wir haben jedoch einen z. B. Datentyp Grid, der sehr viele Daten in einem Gitter angordnet enthält.

- Der Wert, der an double_grid übergeben wird, ist eine Referenz und wird nicht extra kopiert
- Der Wert kann von double_grid verändert werden
- Eine Referenz wird durch ein & hinter dem Datentyp und vor dem Bezeichner gekennzeichnet

Listing 8: Funktion, um alle Elemente des Grids zu verdoppeln

```
void double_grid(Grid& my_grid)
{
    for (int i = 0; i < my_grid.x_size; i++)
    {
        for (int j = 0; j < my_grid.y_size; j++)
        {
            my_grid.data[j][i] *= 2;
        }
    }
}
```

Das spart, je nach Datentyp, sehr viel Rechenzeit, die sonst zum kopieren benötigt würde!

Pointer

Pointer lassen sich im Wesentlichen wie Referenzen verwenden, aber ihr Wert ist tatsächlich eine **Speicheradresse**.

- Mit Pointern kann gerechnet werden, da Speicheradressen auch nur (maschinenabhängige) Zahlen sind
- Pointer werden mit eine * hinter dem Datentyp gekennzeichnet
- Pointer sind mit Vorsicht zu genießen, ein Fehler beim Arbeiten mit Pointern führt sehr schnell zu Programmabstürzen, meistens Segmentation faults (Speicherzugriffsfehler)
- Bevor du einen Pointer benutzt, frage dich immer, ob eine Referenz nicht den gleichen Effekt hat

Pointer

Oft möchte man einen Pointer auf eine bereits deklarierte Variable haben (das & sollte nicht mit dem bei Referenzen verwendeten & verwechselt werden):

Listing 9: Pointer auf eine bestehende Variable

```
int n = 42;
int*pointer_to_n = &n;
```

Pointer

Um an die Daten, auf die ein Pointer zeigt, zu kommen, wird wie bei der Deklaration ein * verwendet:

Listing 10: Dereferenzieren eines Pointers

```
int n = 42;
int*pointer_to_n = &n;
int n_again = *pointer_to_n; // n == n_again
```

Pointer

Was könnte schiefgehen?

Listing 11: Hmmmmm

```
int n = 23;
int*m = &n;
m = m + 19;
printf("%d", *m);
```

Pointer

Das Programm stürzt nicht einmal ab! Ein solcher Fehler wäre unglaublich schwierig zu identifizieren gewesen.

So ist es besser. . .

Listing 12: :)

```
int n = 23;
int*m = &n;
*m = *m + 19;
printf("%d", *m);
```

Datenstrukturen
Arrays

C++ wurde ursprünglich so designt, dass bestehender C-Code damit weiter benutzt werden kann.

- Es gibt C-Arrays
- Es gibt Arrays aus der Standard-Library von C++ (std::array, vector)

C-Arrays sind von der Syntax her vermutlich am vertrautesten:

Listing 13: Initialisieren eines C-Strings

```
char c_style_string[23];
c_style_string = "Hello strings!";
```

Oder:

Listing 14: Initialisieren eines anderen C-Arrays

```
int some_numbers[10] = { 1, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Über diese kann in ganz normal iteriert werden.

C-Arrays sind **Pointer**! Die Syntax versteckt diese Tatsache, aber der Datentyp int[] ist (strukturell) identisch zu int* und kann auch so verwendet werden, dieser Pointer zeigt einfach nur auf das **nullte Element** des Arrays!

Datenstrukturen
Arrays

Nun zu C++. Ein sehr praktischer Array-Datentyp in C++ ist std::vector:

- Kann im Gegensatz zu C-Arrays auf einfache Weise dynamisch erweitert werden und hat einige weitere Annehmlichkeiten
- Verwaltet im Hintergrund ein C-Array, auf das bei Bedarf mit v.data() zugegriffen werden kann
- Es kann wie bei jedem Array mit der Syntax v[i] auf die Elemente zugegriffen werden

Quizfrage: Wie könnte man außer mit v.data() noch auf das zugrundeliegende C-Array zugreifen?

Quizfrage: Wie könnte man außer mit v.data() noch auf das zugrundeliegende C-Array zugreifen?

Listing 15: Veraltete Art, auf das C-Array in std::vector zuzugreifen

```
std::vector<int> v = { 23, 42 };
int*data = &v[0];
```

Datenstrukturen
Arrays

Das war leider nur die halbe Wahrheit:

- Viele Libraries implementieren Datenstrukturen selbst
- Es gibt Libraries wie „Boost“, die viele Dinge beinhalten, die der C++-Standard-Library ursprünglich fehlten, inzwischen aber enthalten sind, trotzdem ist z. B. Boost noch sehr verbreitet
- Letztlich müss man immer die Dokumentation lesen

Datenstrukturen
Arrays

Die C++-Standard-Library (Standard Template Library, STL) enthält noch deutlich mehr Datentypen, die je nach Anwendung hilfreich sein können.

Dokumentation dazu: http://cppreference.com/

Klassen

Ein großer Unterschied zu C ist, dass C++ objektorientiert ist.

In gut geschriebenen C++-Programmen, werden Klassen aufgeteilt in

- eine Header-Datei, in der ausschließlich **deklariert** wird, welche Variablen und Methoden die Klasse enthält, und
- eine Source-Datei, die all diese Dinge **definiert**.

C ist ohne Präprozessor Macros nicht zu denken und ebenso nicht C++. Zum Beispiel sind #includes solche Macros.

Um zu verhindern, dass endlos rekursive #includes entstehen, sollte jede Header-Datei folgendermaßen aussehen:

Listing 16: Header-Datei header_file.hpp

```
#ifndef __HEADER_FILE_HPP__
#define __HEADER_FILE_HPP__

... alles andere dazwischen ...

#endif
```

In den folgenden Folien lassen wir diese Zeilen aus Platzgründen weg.

Klassen

Listing 17: Header-Datei complex.hpp

```
#include <math.h>

class Complex
{
public:
  Complex(); // Konstruktor
  ~Complex(); // Destruktor

  float abs();

private:
  float _x, _y;
};
```

Listing 18: Source-Datei complex.cpp

```
#include "complex.hpp"

Complex::Complex()
{
  _x = _y = 0;
}

Complex::~Complex() {}

float Complex::abs()
{
  return sqrt(pow(_x, 2) +
       pow(_y, 2));
}
```

Listing 19: Header-Datei complex.hpp

```
#include <math.h>

class Complex
{
public:
...
  Complex operator∗(Complex c);
...
};
```

Listing 20: Source-Datei complex.cpp

```
#include "complex.hpp"

...
Complex Complex::operator∗
(Complex c)
{
   return Complex(
      _x ∗c.re() − _y ∗c.im(),
      _y ∗c.re() + _x ∗c.im());
}
...
```

Die Klasse Complex ließe sich mit einigen weiteren Funktionen und einem zusätzlichen Konstruktor, der als Parameter Real- und Imaginärteil hat, wie folgt benutzen:

Listing 21: Anwenden der Klasse Complex

```
Complex a(1.0, 2.0);
Complex b(1.0, 2.0);
Complex c = a ∗b;
float c_abs_val = c.abs();
```

IO

Input und Output haben wir bereits in Form von cin und cout gesehen. Diese nennt man **Streams**.

Dateien sind auch in Form von Streams realisiert und können genauso verwendet werden!

Eine Datei kann zum Beispiel folgendermaßen Zeilenweise eingelesen werden:

Listing 22: Datei einlesen

```
ifstream file("some_file_name.txt");

if (file.is_open())
{
   string value;
   while (!file.eof())
   {
      getline(file, value, '\n');
      cout << value << endl;
   }
   file.close();
}
```

Und so geschrieben werden:

Listing 23: Datei schreiben

```
ifstream file("some_file_name.txt");

if (file.is_open())
{
   file << "..." << std::endl;
   file.close();
}
```

Projekt

Zeit für ein kleines Projekt...