C++

Proseminar "Effiziente Programmierung in C"

Jannes Noack

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

01. Juli 2021

Gliederung (Agenda)

Was ist C++? [1]

- ursprünglich eine Erweiterung von C
- Programmierkonzept:
  - imperativ (aus C)
  - strukturiert (aus C)
  - prozedural (aus C)
  - objektorientiert
  - generisch
- Anwendungs- und Systemprogrammierung
- Code ist Architektur- und Compilerspezifisch

Ursprung und Entwicklung von C++ [1] [2]

- ab 1979: Bjarne Stroustup entwicklet "C mit Klassen"
- 1983: Name jetzt C++
- 1985: erste C++ Version erscheint
- ab 1990: Standardisierungsprozesse
- 1998: C++98 Norm
- ...
- 2020: C++20 Norm (neueste Version)
- Nächste Norm für 2023 geplant

Gemeinsamkeiten mit C [1]

- C++ abwärtskompatibel
  $\hookrightarrow$ C-Code meist in C++ ausführbar
- gleiche Mächtigkeit
- kein gegebener Garbage-Collector
- maschinennah $\Rightarrow$ schnell
- vergleichbarer Overhead erreichbar
- viele Konzepte aus C übernommen

Standardbibliothek [3] [4]

```
1 #include <Headername>
2 std::Paketfunktion;
```

- Häufig genutzte header:
  - <iostream>
  - <string>
  - <array>
  - <vector>
  - <list>
  - <functional>
- bestimmte C-Header mit Präfix c und ohne .h
  - <stdlib.h> ⇒ <cstdlib>
- Weitere C++ Bibliotheken verfügbar ⇒ z.B.: boost

Konsolenbefehle IO [5]

- enthalten in <iostream>
- std::cin >> für Eingaben
- std::cout << für Ausgaben
- std::cerr << für Ausgaben von Errors
- std::clog << für Ausgaben von logs
- std::endl Alternative zu \n

Ein- und Ausgabe über die Konsole [5]

```
1  #include <iostream>
2
3  int main()
4  {
5      int a;
6      std::cin >> a;
7      std::cout << a << std::endl;
8      std::cerr << "Ein Fehler ist aufgetreten" <<
           ↪ std::endl;
9      std::clog << "Ein Fehler ist aufgetreten";
10     return 0;
11 }
```

Listing 1: Beispiel: IO über die Konsole

Namespace [6]

```
1  #include <iostream>
2  using namespace::std;
3
4  namespace first
5  { int var = 5; }
6
7  namespace second
8  { double var = 3.14; }
9
10 int main () {
11     char var = 'a';
12     cout << first::var << endl;   //5
13     cout << second::var << endl; //3.14
14     cout << (int)var << endl;     //97
15     return 0;
16 }
```

Listing 2: Beispiel: Namespaces

Globale Variablen [7]

```
1  #include <iostream>
2  using namespace::std;
3
4  int a = 12;
5  int main()
6  {
7      int a = 42;
8      cout << "a (lokal) ist: " << a << endl;     //42
9      cout << "a (global) ist: " << ::a << endl; //12
10     return 0;
11 }
```

Listing 3: Beispiel: globale Variablen ansprechen

Strings [8]

```
1  #include <iostream>
2  #include <string>
3  using namespace::std;
4
5  int main()
6  {
7      string a("Hallo ");
8      a += ("Welt");
9      cout<<a;                    //Hallo Welt
10     return 0;
11 }
```

Listing 4: Strings in C++

Container [9]

	Größe	Zugriff
array	fest	Index
vector	dynamisch	Index
deque	dynamisch	Index
list	dynamisch	Iteration über Listenbeginn/-ende
stack	dynamisch	LIFO
queue	dynamisch	FIFO
set	dynamisch	Iteration über Setbeginn/-ende
map	dynamisch	Iteration über Mapbeginn/-ende

new/delete [10] [11] [12]

- new Alternative zu malloc
  $\Rightarrow$ ruft Konstruktor auf und erzeugt ein Objekt
- delete Alternative zu free
  $\Rightarrow$ ruft Destruktor auf und löscht ein Objekt
- Speicherplatz von mit new erstellten Objekten immer mit delete freigeben

new und delete [10] [11] [12]

```
1  #include <iostream>
2  using namespace::std;
3
4  int main()
5  {
6      int* intArray = NULL;
7      intArray = new int[5];
8
9      if(!intArray)
10     {
11         cerr << "bad memory allocation" << endl;
12     }
13     ...
14     delete[] intArray;
15     return 0;
16 }
```

Listing 5: Beispiel: new/delete

impliziter Kon- und Destruktor [10] [11] [12]

```
1  class SummeIm {
2      public:
3          int summe;
4          void summiere(int x) {
5              summe += x;
6          }
7  };
8
9  int main()
10 {
11     SummeIm summe1; //benutzt den default Konstruktor
12     return 0;
13 }
```

Listing 6: Beispiel: impliziter Kon-/Destruktor

expliziter Kon- und Destruktor [10] [11] [12]

```
1  class SummeEx {
2      public:
3          int* summe;
4          SummeEx() : summe(new int) {}
5          SummeEx(const int& start) :
6              summe(new int(start)) {}
7          ~SummeEx() { delete summe; }
8          void summiere(int x) { summe += x; }
9  };
10
11 int main()
12 {
13     SummeEx summe2(10);
14     return 0;
15 }
```

Listing 7: Beispiel: expliziter Kon-/Destruktor

RAII [13]

- "resource acquisition is initialization"
- Ressourcenbelegung ist Initialisierung
- Ein Objekt belegt nur solange Ressourcen, wie es im scope/am Leben ist
- Konstruktor: belegt die Ressourcen
- Destruktor: gibt Ressourcen frei

Intelligente Zeiger [14] [15]

- enthalten in <memory>
- vereinfachen die Freigabe von Ressourcen
- unique_ptr: Zeiger mit einem Besitzer
- shared_ptr: Zeiger mit Referenzzähler
- weak_ptr: Zeiger auf shared_ptr-Instanz, ohne den Zähler zu beeinflussen

Intelligente Zeiger [14] [15]

```
1  void UseRawPointer()
2  {
3      Song* pSong = new Song(L"Titel", L"Interpret");
4      ...
5      delete pSong; //muss manuell geloescht werden
6  }
7
8  void UseSmartPointer()
9  {
10     unique_ptr<Song> song2(new Song(L"Titel",
           ↪ L"Interpret"));
11     ...
12     wstring s = song2->duration_;
13     ...
14 } //song2 wird automatisch geloescht
```

Listing 8: Beispiel: Intelligente Zeiger

Null-Zeiger: nullptr [16]

- enthalten in <cstddef>
- eigener Datentyp: *null pointer constant type (std::nullptr_t)*
- implizite Umwandlung zu raw Pointer Typen möglich
- verhindert Compilerfehler durch Nutzung von 0, NULL oder void*

```
1 void foo(char*) {
2 ...
3 }
4
5 int main() {
6     foo(nullptr);
7     return 0;
8 }
```

Listing 9: Beispiel: nullptr

Pass by reference [17]

```
1  void swap(int& x, int& y)
2  {
3      int z = x;
4      x = y;
5      y = z;
6  }
7
8  int main()
9  {
10     int a = 11, b = 22;
11     swap(a, b);              //a = 22, b = 11
12 }
```

Listing 10: Beispiel: pass by reference

Exceptions [18]

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      try
6      {
7          throw 20;
8      }
9      catch (int e)
10     {
11         cout << "Fehler aufgetreten. Fehler Nr. " << e
               ↪ << endl;
12     }
13     return 0;
14 }
```

Listing 11: Beispiel: try-catch

Exceptions unterscheiden [18]

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      try {
6          //Auszufuehrender Code
7      }
8      catch (int param) { cout << "int exception"; }
9      catch (char param) { cout << "char exception"; }
10     catch (...) { cout << "default exception"; }
11     return 0;
12 }
```

Listing 12: Beispiel: Mehrere catch-Anweisungen

Weitere interessante Funktionen

- Lambdas
- Templates
- Assertions
- Module
- ...

Fazit C vs C++

- Aspekt Effizienz:
  - stark Compilerabhängig
  - abhängig von den genutzten Funktionen
  - Pauschal nicht beantwortbar
  - C++ verleitet zur Nutzung weniger performanter Funktionen
- Abhängig vom Anwendungsfall
- Persönliche Präferenzen

Zusammenfassung

- Ursprünglich Erweiterung von C
- Viele neue Möglichkeiten und Funktionen
- Objektorientierung realisierbar
- Programme sind Architektur- und Compilerspezifisch
- C-Code benutzbar (am besten als Bibliothek)

Literatur I

[1] "A Brief Description," 2000.

[2] "History of C++," 2000.

[3] "Übersicht über die C++-Standardbibliothek," 2016.

[4] "boost.org," 2004.

[5] "Input/Output," 2000.

[6] "Namespaces," 2000.

[7] "Gültigkeitsbereich (C++)," 2018.

[8] "std::string," 2000.

[9] "Containers," 2000.

Literatur II

[10] "<new>," 2000.

[11] "Konstruktoren," 2019.

[12] "Destruktoren," 2019.

[13] "Objektlebenszeit und Ressourcenverwaltung (RAII)," 2019.

[14] "<memory>," 2000.

[15] "Intelligente Zeiger (Modern C++)," 2019.

[16] "nullptr_t," 2000.

[17] "Verweistyp-Funktionsargumente," 2018.

[18] "Exceptions," 2018.