Kapitel 1

Die Grundlagen

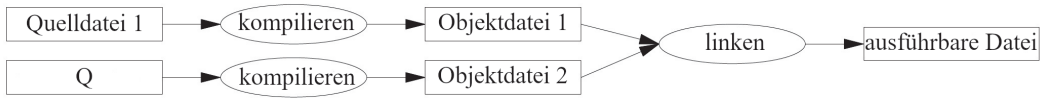
The first thing we do, let's
kill all the language lawyers.
– Henry VI, Part II

1.1 Einführung

Dieses Kapitel präsentiert ganz formlos die Notation von C++, das Speicher- und Berechnungsmodell von C++ sowie die grundlegenden Mechanismen, nach denen Code zu einem Programm zusammengefügt wird. Dies sind die Komponenten, die man vor allem in C sieht und die einen Programmierstil bilden, der als *prozedurale Programmierung* bezeichnet wird.

1.2 Programme

C++ ist eine kompilierte Sprache. Damit ein Programm ausgeführt werden kann, muss sein Quelltext durch einen Compiler verarbeitet werden. Dabei werden Objektdateien erzeugt, die dann ein Linker zu einem ausführbaren Programm kombiniert. Ein C++-Programm besteht typischerweise aus vielen Quellcodedateien (meist einfach *Quelldateien* genannt).



Ein ausführbares Programm wird für eine bestimmte Hardware/System-Kombination erzeugt; es kann nicht von z. B. einem Android-Gerät auf einen Windows-PC übertragen werden. Wenn es um die Portabilität von C++-Programmen geht, dann meinen wir üblicherweise die Portabilität des Quellcodes; das heißt, dass der Quellcode erfolgreich auf einer Vielzahl von Systemen kompiliert und ausgeführt werden kann.

Der ISO-C++-Standard definiert zwei Arten von Entitäten:

- *Elemente der Kernsprache*, wie integrierte Typen (z. B. **`char`** und **`int`**) und Schleifen (z. B. **`for`**- und **`while`**-Anweisungen)
- *Komponenten der Standardbibliothek*, wie etwa Container (z. B. **`vector`** und **`map`**) und I/O-Operationen (z. B. **`<<`** und **`getline()`**)

Bei den Komponenten der Standardbibliothek handelt es sich um völlig normalen C++-Code, der von jeder C++-Implementierung bereitgestellt wird. Das heißt, die C++-Standardbibliothek kann selbst in C++ implementiert werden, was auch so ist (mit sehr geringfügigem Einsatz von Maschinencode für Dinge wie **`thread`**-Kontextwechsel). Das impliziert, dass C++ für die anspruchsvollsten Aufgaben im Bereich der Systemprogrammierung ausreichend ausdrucksstark und effizient ist.

C++ gehört zu den statisch typisierten Sprachen. Das heißt, der Typ jeder Entität (wie etwa Objekt, Wert, Name und Ausdruck) muss dem Compiler an der Stelle bekannt sein, an der sie benutzt wird. Der Typ eines Objekts bestimmt die Menge der Operationen, die darauf angewendet werden können, sowie seine Anordnung im Speicher.

1.2.1 Hello, World!

Das kleinstmögliche C++-Programm ist

```
int main(){}        // das kleinstmögliche C++-Programm
```

Es definiert eine Funktion namens **`main()`**, die keine Argumente entgegennimmt und nichts tut.

Geschweifte Klammern, **`{}`**, drücken in C++ eine Gruppierung aus. Hier kennzeichnen sie den Anfang und das Ende des Funktionskörpers. Der doppelte Schrägstrich, **`//`**, startet einen Kommentar, der bis zum Zeilenende reicht. Ein Kommentar ist für die menschlichen Leserinnen und Leser vorgesehen; der Compiler ignoriert Kommentare.

Jedes C++-Programm muss genau eine globale Funktion namens **`main()`** besitzen. Das Programm startet, indem es diese Funktion ausführt. Der Integer-Wert **`int`**, der von **`main()`** zurückgegeben wird, falls er vorhanden ist, ist der Rückgabewert des Programms an »das System«. Wird kein Wert zurückgegeben, erhält das System einen Wert, der einen erfolgreichen Abschluss des Programms signalisiert. Ist der von **`main()`** zurückgegebene Wert ungleich null, bedeutet dies ein Fehlschlagen des Programms. Nicht alle Betriebssysteme und Ausführungsumgebungen machen Gebrauch von diesem Rückgabewert: Linux/Unix-Systeme tun es, Windows-Umgebungen dagegen nur selten.

Üblicherweise erzeugt ein Programm irgendeine Ausgabe. Hier ist ein Programm, das **`Hello, World!`** schreibt:

```
import std;

int main()
{
     std::cout << "Hello, World!\n";
}
```

Die Zeile **`import std;`** weist den Compiler an, die Deklarationen der Standardbibliothek zur Verfügung zu stellen. Ohne diese Deklarationen wäre der Ausdruck

```
std::cout << "Hello, World!\n"
```

sinnlos. Der Operator **`<<`** (»ausgeben«) schreibt sein zweites Argument auf sein erstes. In diesem Fall wird das String-Literal **`"Hello, World!\n"`** auf den Standard-Ausgabe-Stream **`std::cout`** geschrieben. Ein String-Literal ist eine Folge von Zeichen, die von doppelten Anführungszeichen umgeben sind. In einem String-Literal kennzeichnet der Backslash **`\`** gefolgt von einem anderen Zeichen ein einzelnes »Sonderzeichen«. Hier ist **`\n`** das Newline-Zeichen. Es werden also die Zeichen **`Hello, World!`** geschrieben, gefolgt von einem Newline, also dem Steuerzeichen für eine neue Zeile.

`std::` gibt an, dass der Name (Bezeichner) **`cout`** im Namensraum der Standardbibliothek (§3.3) zu finden ist. Ich lasse das **`std::`** normalerweise weg, wenn es um Standardeigenschaften geht. §3.3 zeigt, wie man Namen aus einem Namensraum auch ohne explizite Qualifizierung sichtbar machen kann.

Die Direktive **`import`** ist neu in C++20. Es ist noch nicht im Standard verankert, dass die gesamte Standardbibliothek als Modul **`std`** vorhanden ist. Das wird in §3.2.2 erklärt. Falls Sie Probleme mit **`import std;`** haben, probieren Sie das altmodische und herkömmliche

```
#include <iostream>        // bindet die Deklarationen für die
                           // I/O-Stream-Bibliothek ein

int main()
{
     std::cout << "Hello, World!\n";
}
```

Das wird in §3.2.1 erklärt und hat in allen C++-Implementierungen seit 1998 funktioniert (§19.1.1).

Im Prinzip wird der gesamte ausführbare Code in Funktionen gepackt und direkt oder indirekt aus **`main()`** heraus aufgerufen. Zum Beispiel:

```
import std;            // importiert die Deklarationen für die
                       // Standardbibliothek
using namespace std;    // macht die Namen aus std auch ohne
                        // std:: sichtbar (§3.3)
double square(double x)    // quadriert eine Gleitkommazahl mit doppelter
                           // Genauigkeit
{
     return x*x;
}

void print_square(double x)
{
     cout << "das Quadrat von " << x << " ist " << square(x) << "\n";
}

int main()
{
     print_square(1.234) // Ausgabe: das Quadrat von 1,234 ist 1,52276
}
```

Der »Rückgabetyp« **`void`** zeigt an, dass die Funktion keinen Wert zurückgibt.

1.3 Funktionen

Die wichtigste Möglichkeit, irgendetwas in einem C++-Programm erledigen zu lassen, besteht darin, dafür eine Funktion aufzurufen. Über das Definieren einer Funktion legen Sie fest, wie eine Operation durchgeführt werden soll. Eine Funktion kann nur aufgerufen werden, wenn sie zuvor deklariert wurde.

Eine Funktionsdeklaration legt den Namen der Funktion, den Typ des zurückgelieferten Werts (falls vorhanden) und die Anzahl und Typen der Argumente fest, die in einem Aufruf angegeben werden müssen. Zum Beispiel:

```
Elem* next_elem();       // kein Argument, liefert einen Zeiger auf
                         // Elem (einen Elem*) zurück
void exit(int);          // int-Argument, liefert nichts zurück
double sqrt(double);     // double-Argument, liefert einen double zurück
```

In einer Funktionsdeklaration steht der Rückgabetyp vor dem Namen der Funktion; die Argumenttypen stehen hinter dem Namen und werden in Klammern eingeschlossen.

Die Semantik der Argumentübergabe ist identisch mit der Semantik der Initialisierung (§3.4.1). Das heißt, die Argumenttypen werden geprüft und falls notwendig findet eine implizite Konvertierung der Argumenttypen statt (§1.4). Zum Beispiel:

```
double s2 = sqrt(2);  // Aufruf von sqrt() mit dem Argument double{2}
double s3 = sqrt("three");  // Fehler: sqrt() verlangt ein Argument des
                            // Typs double
```

Man sollte den Wert einer solchen Prüfung und Typkonvertierung zum Compile-Zeitpunkt nicht unterschätzen.

Eine Funktionsdeklaration könnte Argumentnamen enthalten. Dies kann für den Leser eines Programms hilfreich sein, doch der Compiler ignoriert solche Namen einfach, solange die Deklaration nicht auch eine Funktionsdefinition ist. Zum Beispiel:

```
double sqrt(double d);        // gibt die Quadratwurzel von d zurück
double square(double);        // gibt das Quadrat des Arguments zurück
```

Der Typ einer Funktion besteht aus ihrem Rückgabetyp, gefolgt von einer Abfolge ihrer Argumenttypen in runden Klammern. Zum Beispiel:

```
double get(const vector<double>& vec, int index);   // Typ: double(const
                                                     // vector<double>&,int)
```

Eine Funktion kann Member (Mitglied) einer Klasse sein (§2.3, §5.2.1). Bei einer solchen Member-Funktion ist der Name ihrer Klasse ebenfalls Teil des Funktionstyps. Zum Beispiel:

```
char& String::operator[](int index);    // Typ: char& String::(int)
```

Wir wollen, dass unser Code verständlich ist, weil dies den ersten Schritt auf dem Weg zur Wartungsfreundlichkeit bedeutet. Um Verständlichkeit zu erreichen, zerlegt man als Erstes die Berechnungsaufgaben in sinnvolle Einheiten (dargestellt als Funktionen und Klassen) und benennt diese. Solche Funktionen bilden dann das Grundvokabular der rechnerischen Verarbeitung, genau wie die (integrierten und benutzerdefinierten) Typen das Grundvokabular der Daten bilden. Die C++-Standardalgorithmen (z. B. **`find`**, **`sort`** und **`iota`**) sind ein guter Start (Kapitel 13).