Eine Tour durch C++

Der praktische Leitfaden für modernes C++

Kapitel 1

Die Grundlagen

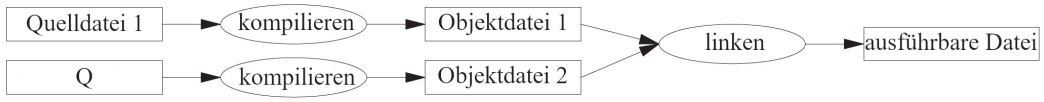
The first thing we do, let's
kill all the language lawyers.
– Henry VI, Part II

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).

Anschließend können Sie Funktionen, die gängige oder spezialisierte Aufgaben repräsentieren, zu größeren Verarbeitungseinheiten zusammensetzen.

Die Anzahl der Fehler in Code korreliert stark mit der Menge und der Komplexität des Codes. Beiden Problemen können Sie begegnen, indem Sie mehr und kürzere Funktionen verwenden. Eine Funktion zu benutzen, die eine bestimmte Aufgabe erledigt, erspart es oft, mitten in irgendwelchem Code ein spezialisiertes Stück Code schreiben zu müssen; wenn Sie daraus eine Funktion bauen, sind Sie gezwungen, die Aktivität zu benennen und ihre Abhängigkeiten zu dokumentieren. Können Sie keinen passenden Namen finden, dann ist es sehr wahrscheinlich, dass Sie ein Designproblem haben.

Falls zwei Funktionen mit demselben Namen, aber unterschiedlichen Argumenttypen definiert sind, wählt der Compiler bei jedem Aufruf die Funktion, die am passendsten erscheint. Zum Beispiel:

```
void print(int);           // nimmt ein Integer-Argument entgegen
void print(double);        // nimmt ein Gleitkomma-Argument entgegen
void print(string);        // nimmt ein String-Argument entgegen

void user()
{
     print(42);            // ruft print(int) auf
     print(9.65);          // ruft print(double) auf
     print("Barcelona");   // ruft print(string) auf
}
```

Falls zwei alternative Funktionen aufgerufen werden könnten, aber keine von beiden besser als die andere ist, dann gilt der Aufruf als mehrdeutig und der Compiler gibt einen Fehler aus. Zum Beispiel:

```
void print(int, double);
void print(double, int);

void user2()
{
     print(0,0);        // Fehler: mehrdeutig
}
```

Das Definieren mehrerer Funktionen mit demselben Namen wird als *Überladen der Funktion* bezeichnet. Es ist ein wesentlicher Bestandteil der generischen Programmierung (§8.2). Wenn eine Funktion überladen wird, dann sollten alle Funktionen mit demselben Namen die gleiche Semantik implementieren. Die **`print()`**-Funktionen sind ein Beispiel dafür; jedes **`print()`** gibt sein Argument aus.

1.4 Typen, Variablen und Arithmetik

Jeder Name und jeder Ausdruck hat einen Typ, der bestimmt, welche Operationen darauf ausgeführt werden dürfen. So legt zum Beispiel die Deklaration

```
int inch;
```

fest, dass **inch** vom Typ **int** ist; das heißt, **inch** ist eine Integer-Variable.

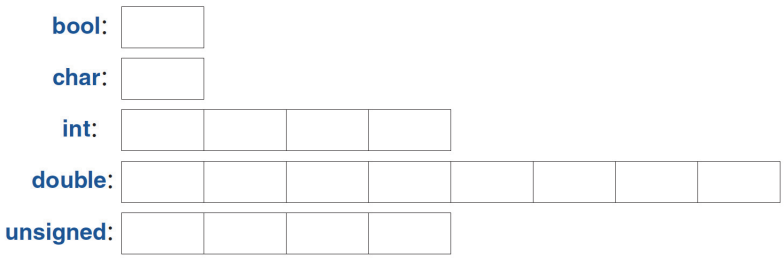
Eine Deklaration ist eine Anweisung, die eine Entität in das Programm einführt und ihren Typ festlegt:

- Ein *Typ* definiert eine Menge an möglichen Werten und eine Menge an Operationen (für ein Objekt).
- Ein *Objekt* ist ein Speicherbereich, der einen Wert eines bestimmten Typs enthält.
- Ein *Wert* ist eine Menge an Bits, die entsprechend einem Typ interpretiert werden.
- Eine *Variable* ist ein benanntes Objekt.

C++ bietet eine ganze Reihe grundlegender Typen, die ich hier aber nicht alle aufführen will. Sie können sie in Referenzquellen finden, etwa in [Cppreference] im Netz. Hier nur einige Beispiele:

```
bool        // Boolean, mögliche Werte sind true und false
char        // Zeichen, zum Beispiel 'a', 'z' und '9'
int         // Integer, zum Beispiel -273, 42 und 1066
double      // Gleitkommazahl doppelter Genauigkeit, zum Beispiel
            // -273.15, 3.14 und 6.626e-34
unsigned    // nichtnegativer Integer, zum Beispiel 0, 1 und 999
            // (wird für bitweise logische Operationen benutzt)
```

Jeder grundlegende Typ besitzt direkt eine Hardware-Entsprechung und hat eine feste Größe, die den Wertebereich festlegt, der darin gespeichert werden kann:



Eine **char**-Variable hat die natürliche Größe eines Zeichens auf einem bestimmten Computer (üblicherweise handelt es sich um ein 8 Bit langes Byte). Die Größen der anderen Typen sind Vielfaches der Größe eines **char**. Die Größe eines Typs ist von der Implementierung abhängig (d. h. kann auf unterschiedlichen Maschinen verschieden ausfallen) und lässt sich durch den **sizeof**-Operator ermitteln. So ist zum Beispiel **sizeof(char)** gleich **1**, während **sizeof(int)** oft **4** beträgt. Wenn Sie einen Typ einer bestimmten Größe haben wollen, benutzen Sie einen Typ-Alias der Standardbibliothek, wie etwa **int32_t** (§17.8).

Zahlen können als Gleitkommazahlen oder als Integer-Werte vorliegen.

- Gleitkomma-Literale sind an einem Dezimalpunkt (z. B. **3.14**) oder einem Exponenten (z. B. **314e-2**) erkennbar.
- Integer-Literale sind standardmäßig dezimal (z. B. **42** bedeutet zweiundvierzig). Das Präfix **0b** kennzeichnet ein binäres (Basis 2) Integer-Literal (z. B. **0b10101010**). Das Präfix **0x** kennzeichnet ein hexadezimales (Basis 16) Integer-Literal (z. B. **0xBAD12CE3**). Das Präfix **0** kennzeichnet ein oktales (Basis 8) Integer-Literal (z. B. **0334**).

Damit lange Literale für uns Menschen besser lesbar sind, können Sie ein einfaches Anführungszeichen (') als Trennzeichen benutzen. So beträgt zum Beispiel π ungefähr **3.14159'26535'89793'23846'26433'83279'50288** oder, falls Sie die hexadezimale Notation bevorzugen, **0x1.921F'B544'42D1'8P+1**.

1.4.1 Rechenoperatoren

Die arithmetischen Operatoren können für geeignete Kombinationen der Grundtypen benutzt werden:

```
x+y       // Plus
+x        // unäres Plus
x-y       // Minus
-x        // unäres Minus
x*y       // Multiplizieren
x/y       // Dividieren
x%y       // Rest (Modulo) für Integer
```

Das gilt auch für Vergleichsoperatoren:

```
x==y       // Gleich
x!=y       // Ungleich
x<y        // Kleiner als
x>y        // Größer als
x<=y       // Kleiner als oder gleich
x>=y       // Größer als oder gleich
```

Es gibt darüber hinaus auch Logikoperatoren:

```
x&y       // Bitweises Und
x|y       // Bitweises Oder
x^y       // Bitweises Exklusiv-Oder
~ x       // Bitweises Komplement
x&&y      // Logisches Und
x||y      // Logisches Oder
! x       // Logisches Nicht (Negation)
```

Ein bitweiser logischer Operator liefert als Ergebnis den Operandentyp, für den die Operation auf jedem Bit durchgeführt wurde. Die Logikoperatoren **&&** und **||** geben je nach den Werten ihrer Operanden einfach **`true`** oder **`false`** zurück.

In Zuweisungen und arithmetischen Operationen führt C++ alle sinnvollen Konvertierungen zwischen den Grundtypen durch, sodass diese frei gemischt werden können:

```
void some_function() // Funktion, die keinen Wert zurückliefert
{
     double d = 2.2; // Initialisiert eine Gleitkommazahl
     int i = 7;      // Initialisiert Integer
     d = d+i;        // Weist d eine Summe zu
     i = d*i;        // Weist i ein Produkt zu; Achtung: das double d*i
                     // wird zu einem int abgeschnitten
}
```

Die in Ausdrücken benutzten Konvertierungen werden als *die üblichen arithmetischen Konvertierungen* bezeichnet und sollen sicherstellen, dass die Ausdrücke mit der höchsten Genauigkeit ihrer Operanden verarbeitet werden. So wird zum Beispiel eine Addition eines **`double`** und eines **`int`** mittels Gleitkomma-Arithmetik mit doppelter Genauigkeit ausgeführt.

Beachten Sie, dass **=** der Zuweisungsoperator ist, **==** dagegen auf Gleichheit prüft.

Zusätzlich zu den herkömmlichen arithmetischen und logischen Operatoren bietet C++ speziellere Operationen zum Modifizieren einer Variablen:

```
x+=y      // x = x+y
++x       // Inkrementiert: x = x+1
x?=y      // x = x-y
?? x      // Dekrementiert: x = x-1
x*=y      // Skaliert: x = x*y
x/=y      // Skaliert: x = x/y
x%=y      // x = x%y
```

Diese Operatoren sind kompakt, bequem und werden sehr häufig verwendet.

Die Auswertung erfolgt von links nach rechts für **`x.y`**, **`x->y`**, **`x(y)`**, **`x[y]`**, **`x<<y`**, **`x>>y`**, **`x&&y`** und **`x||y`**. Zuweisungen (z. B. **`x+=y`**) werden von rechts nach links ausgewertet. Aus historischen Gründen, die mit dem Drang nach Optimierung zusammenhängen, ist die Auswertungsreihenfolge von anderen Ausdrücken (z. B. **`f(x) + g(y)`**) sowie von Funktionsargumenten (z. B. **`h(f(x),g(y))`**) leider nicht festgelegt.

1.4.2 Initialisierung

Bevor ein Objekt benutzt werden kann, muss ihm ein Wert übergeben werden. C++ bietet eine Vielzahl an Notationen zum Ausdrücken einer Initialisierung, wie etwa das bereits vorgestellte **=** sowie eine universelle Form, nämlich Listen, die durch ein Paar geschweifte Klammern (**{}**) begrenzt werden:

```
double d1 = 2.3;              // Initialisiert d1 auf 2.3
double d2 {2.3};              // Initialisiert d2 auf 2.3
double d3 = {2.3};            // Initialisiert d3 auf 2.3 (das = ist mit
                              // { ... } optional)

complex<double> z = 1;        // eine komplexe Zahl mit
                              // Gleitkomma-Skalaren doppelter
Genauigkeit
complex<double> z2 {d1, d2};
complex<double> z3 = {d1, d2};    // das = ist mit { ... } optional

vector<int> v {1, 2, 3, 4, 5, 6}; // ein Vektor aus ints
```

Die Art, den Operator **=** zu verwenden, ist traditionell und stammt schon aus C. Falls Sie sich unsicher sind, benutzen Sie die allgemeine **{}**-Listenform. Damit schützen Sie sich zumindest vor Konvertierungen, bei denen Informationen verloren gehen:

```
int i1 = 7.8;         // i1 wird zu 7 (Überraschung?!)
int i2 {7.8};         // Fehler: Gleitkomma- zu Integer-Konvertierung
```

Leider sind Konvertierungen, bei denen Informationen verloren gehen, die sogenannten *verengenden Konvertierungen* (Narrowing Conversions), wie etwa **`double`** nach **`int`** und **`int`** nach **`char`**, erlaubt und kommen implizit zum Einsatz, wenn Sie **=** benutzen (nicht jedoch, wenn Sie **{}** verwenden). Die durch implizite verengende Konvertierungen verursachten Probleme sind der Preis, den man für die Kompatibilität zu C bezahlen muss (§19.3).

Eine Konstante (§1.6) kann nicht uninitialisiert bleiben und eine Variable sollte nur unter ausgesprochen seltenen Umständen uninitialisiert bleiben. Führen Sie einen Namen nur dann ein, wenn Sie einen passenden Wert dafür haben. Benutzerdefinierte Typen (wie **`string`**, **`vector`**, **`Matrix`**, **`Motor_controller`** und **`Orc_warrior`**) können so definiert werden, dass sie implizit initialisiert werden (§5.2.1).

Wenn Sie eine Variable definieren, müssen Sie deren Typ nicht explizit angeben, falls der Typ sich aus der Initialisierung schlussfolgern lässt:

```
auto b = true;       // ein Boolean
auto ch = 'x';       // ein char
auto i = 123;        // ein int
auto d = 1.2;        // ein double
auto z = sqrt(y);    // z hat den Typ, der von sqrt(y) zurückgegeben wird
auto bb {true};      // bb ist ein Boolean
```

Bei **`auto`** neigt man meist dazu, **`=`** zu benutzen, weil keine potenziell lästige Typkonvertierung im Spiel ist. Falls Sie es jedoch vorziehen, konsistent die **`{}`**-Initialisierung zu verwenden, so können Sie das auch hier tun.

Man setzt **`auto`** immer dann ein, wenn es keinen speziellen Grund gibt, den Typ ausdrücklich zu erwähnen. »Spezielle Gründe«, den Typ dennoch anzugeben, sind unter anderem:

- Die Definition ist in einem großen Gültigkeitsbereich, in dem Sie den Leserinnen und Lesern Ihres Codes den Typ ganz eindeutig klarmachen wollen.
- Der Typ des Initialisierers ist nicht offensichtlich.
- Sie wollen den Umfang oder die Genauigkeit einer Variablen ganz ausdrücklich festlegen (z. B. **`double`** anstelle von **`float`**).

Durch Verwendung von **`auto`** werden Redundanz und das Schreiben langer Typnamen vermieden. Das ist vor allem in der generischen Programmierung wichtig, bei der es für die Programmierer schwierig sein kann, den exakten Typ eines Objekts zu kennen, und die Typnamen recht lang sein können (§13.2).

1.5 Gültigkeitsbereich und Lebensdauer

Eine Deklaration führt ihren Namen in einen Gültigkeitsbereich ein:

- *Lokaler Gültigkeitsbereich*: Ein Name, der in einer Funktion (§1.3) oder einem Lambda (§7.3.2) deklariert wurde, wird als *lokaler Name* bezeichnet. Sein Gültigkeitsbereich erstreckt sich vom Punkt der Deklaration bis zum Ende des Blocks, in dem seine Deklaration auftritt. Ein *Block* wird durch ein Paar

geschweifte Klammern (**{}**) begrenzt. Namen von Funktionsargumenten werden als lokale Namen betrachtet.

- *Klassen-Gültigkeitsbereich*: Ein Name wird als *Member-Name* (oder *Class-Member-Name*) bezeichnet, wenn er in einer Klasse (§2.2, §2.3, Kapitel 5), aber außerhalb einer Funktion (§1.3), eines Lambda (§7.3.2) oder eines **enum class** (§2.4) definiert wurde. Sein Gültigkeitsbereich erstreckt sich von der öffnenden geschweiften Klammer (**{**) seiner umschließenden Deklaration bis zur dazugehörenden schließenden geschweiften Klammer (**}**).
- *Namensraum-Gültigkeitsbereich*: Ein Name wird als *Namensraum-Member-Name* bezeichnet, wenn er in einem Namensraum (Namespace) (§3.3), aber außerhalb einer Funktion, eines Lambda (§7.3.2), einer Klasse (§2.2, §2.3, Kapitel 5) oder eines **enum class** (§2.4) definiert wurde. Sein Gültigkeitsbereich erstreckt sich vom Punkt der Deklaration bis zum Ende seines Namensraums.

Ein Name, der nicht innerhalb eines anderen Konstrukts deklariert wurde, wird als *globaler Name* bezeichnet und liegt im globalen Namensraum.

Darüber hinaus gibt es Objekte ohne Namen, wie etwa temporäre Objekte und Objekte, die mithilfe von **new** (§5.2.2) erzeugt wurden. Zum Beispiel:

```
vector<int> vec;  // vec ist global (ein globaler Vektor aus Integern)

void fct(int arg) // fct ist global (benennt eine globale Funktion)
                  // arg ist lokal (benennt ein Integer-Argument)
{
     string motto {"Wer wagt, gewinnt"}; // motto ist lokal
     auto p = new Record{"Hume"};        // p zeigt auf ein unbenanntes
                                         // Record (erzeugt durch new)
    // ...
}
struct Record {
     string name; // name ist ein Member von Record (ein String-Member)
    // ...
};
```

Ein Objekt muss vor seiner Benutzung konstruiert (initialisiert) werden. Am Ende seines Gültigkeitsbereichs wird es zerstört. Für ein Namensraumobjekt ist das Ende des Programms der Punkt der Zerstörung. Für ein Member wird der Punkt der Zerstörung durch den Punkt der Zerstörung des Objekts festgelegt, dessen Member es ist. Ein Objekt, das durch **new** erzeugt wurde, »lebt« hingegen, bis es mittels **delete** zerstört wird (§5.2.2).

1.6 Konstanten

C++ unterstützt zwei Arten von *Unveränderlichkeit* (damit ist ein Objekt mit einem unveränderlichen Zustand gemeint):

- **`const`** bedeutet in etwa: »Ich verspreche, diesen Wert nicht zu verändern«. Dies wird vor allem benutzt, um Schnittstellen zu spezifizieren, damit Daten mithilfe von Zeigern und Referenzen an Funktionen übergeben werden können, ohne dass man befürchten muss, dass sie modifiziert werden. Der Compiler setzt das Versprechen durch, das von **`const`** gegeben wurde. Der Wert eines **`const`** kann zur Laufzeit berechnet werden.
- **`constexpr`** bedeutet in etwa: »wird zum Zeitpunkt des Kompilierens ausgewertet«. Dies wird vor allem dafür verwendet, um Konstanten festzulegen, um die Ablage von Daten in schreibgeschütztem Speicher zu ermöglichen (wo es unwahrscheinlich ist, dass diese beschädigt werden) und zu Performance-Zwecken. Der Wert eines **`constexpr`** muss vom Compiler berechnet werden.

Zum Beispiel:

```
constexpr int dmv = 17;  // dmv ist eine benannte Konstante
int var = 17;            // var ist keine Konstante
const double sqv = sqrt(var);   // sqv ist eine benannte Konstante,
                      // die möglicherweise zur Laufzeit berechnet wird

double sum(const vector<double>&);    // sum wird sein Argument nicht
                                      // modifizieren (§1.7)

vector<double> v {1.2, 3.4, 4.5};    // v ist keine Konstante
const double s1 = sum(v);    // Okay: sum(v) wird zur Laufzeit
                             // ausgewertet
constexpr double s2 = sum(v);    // Fehler: sum(v) ist kein konstanter
                                 // Ausdruck
```

Damit eine Funktion in einem konstanten Ausdruck verwendet werden kann, das heißt in einem Ausdruck, der vom Compiler ausgewertet wird, muss sie mit **`constexpr`** oder **`consteval`** definiert werden. Zum Beispiel:

```
constexpr double square(double x) { return x*x; }

constexpr double max1 = 1.4*square(17);    // Okay: 1.4*square(17) ist
                                // ein konstanter Ausdruck
constexpr double max2 = 1.4*square(var);    // Fehler: var ist keine
              // Konstante, weshalb square(var) auch keine Konstante ist
```

```
const double max3 = 1.4*square(var);    // Okay: kann zur Laufzeit
                                        // ausgewertet werden
```

Eine **constexpr**-Funktion kann durchaus für nichtkonstante Argumente verwendet werden, allerdings ist das Ergebnis dann kein konstanter Ausdruck. Der Aufruf einer **constexpr**-Funktion mit nichtkonstanten Argumenten ist in Kontexten erlaubt, die keine konstanten Ausdrücke verlangen. Auf diese Weise müssen Sie die gleiche Funktion nicht zweimal definieren: einmal für konstante Ausdrücke und einmal für Variablen. Wenn Sie wollen, dass eine Funktion nur für die Auswertung während des Kompilierens benutzt wird, deklarieren Sie sie mit **consteval** statt mit **constexpr**. Zum Beispiel:

```
consteval double square2(double x) { return x*x; }

constexpr double max1 = 1.4*square2(17);    // Okay: 1.4*square(17) ist
                                            // ein konstanter Ausdruck
const double max3 = 1.4*square2(var);       // Fehler: var ist keine
                                            // Konstante
```

Funktionen, die mit **constexpr** oder **consteval** deklariert werden, sind die C++-Version des Prinzips von reinen Funktionen. Sie können keine Nebeneffekte haben und können nur Informationen verwenden, die ihnen als Argumente übergeben wurden. Insbesondere können sie nichtlokale Variablen nicht modifizieren, aber sie können Schleifen haben und ihre eigenen lokalen Variablen benutzen. Zum Beispiel:

```
constexpr double nth(double x, int n)    // angenommen 0<=n
{
     double res = 1;
     int i = 0;
     while (i<n) {        // while-Schleife: macht etwas, solange
                                 // die Bedingung erfüllt ist (§1.7.1)
         res *= x;
         ++i;
     }
     return res;
}
```

An einigen Stellen verlangen die Regeln der Sprache den Einsatz von konstanten Ausdrücken (z. B. bei Array-Grenzen (§1.7), Case-Bezeichnern (§1.8), Template-Wertargumenten (§7.2) und Konstanten, die mit **constexpr** deklariert werden). In

anderen Fällen ist die Auswertung zum Zeitpunkt des Kompilierens aus Performance-Gründen wichtig. Unabhängig von Performance-Fragen ist die Unveränderlichkeit von Objekten eine wichtige Designentscheidung.

1.7 Zeiger, Arrays und Referenzen

Die grundlegendste Form der Sammlung (Collection) von Daten ist ein zusammenhängender Speicherbereich, der mit Elementen desselben Typs belegt ist, ein sogenanntes *Array*. Im Prinzip ist es das, was die Hardware bereitstellt. Ein Array aus Elementen des Typs **char** kann folgendermaßen deklariert werden:

```
char v[6];        // Array aus sechs Zeichen
```

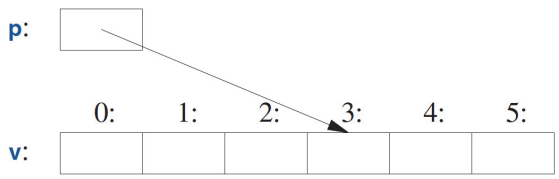
Ein Zeiger (Pointer) auf einen Speicherbereich lässt sich so deklarieren:

```
char* p;          // ein Zeiger auf ein Zeichen
```

In Deklarationen bedeutet **[]** »Array aus« und ***** »Zeiger auf«. Alle Arrays haben **0** als untere Grenze, sodass **v** die sechs Elemente **v[0]** bis **v[5]** besitzt. Die Größe eines Arrays muss ein konstanter Ausdruck sein (§1.6). Eine Zeigervariable kann die Adresse eines Objekts des entsprechenden Typs enthalten:

```
char* p = &v[3];    // p zeigt auf das vierte Element von v
char x = *p;        // *p ist das Objekt, auf das der Zeiger p zeigt
```

In einem Ausdruck bedeutet das unäre Präfix ***** »Inhalt von« und das unäre Präfix **&** »Adresse von«. Das kann grafisch so dargestellt werden:



Stellen Sie sich vor, Sie würden die Elemente eines Arrays ausgeben:

```
void print()
{
     int v1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
    for (auto i=0; i!=10; ++i)     // gibt die Elemente aus
        cout << v[i] << '\n';
    // ...
}
```

Diese **for**-Anweisung kann gelesen werden als: »Setze **i** auf 0; solange **i** noch nicht **10** ist, gib das **i**-te Element aus und erhöhe **i** um 1«. C++ bietet auch eine einfachere **for**-Anweisung, die sogenannte bereichsbasierte **for**-Anweisung, für Schleifen, die eine Sequenz in der einfachsten Weise durchlaufen:

```
void print2()
{
    int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (auto x : v)        // für jedes x in v
        cout << x << '\n';

    for (auto x : {10, 21, 32, 43, 54, 65})    // für jeden Integer in
                                               // der Liste
        cout << x << '\n';
    // ...
}
```

Die erste bereichsbasierte **for**-Anweisung kann gelesen werden als: »Kopiere jedes Element aus **v**, vom ersten bis zum letzten, nach **x** und gib es aus«. Beachten Sie, dass Sie keine Array-Grenze angeben müssen, wenn Sie es mit einer Liste initialisieren. Die bereichsbasierte **for**-Anweisung kann für jede Sequenz von Elementen benutzt werden (§13.1).

Falls Sie die Werte aus **v** nicht in die Variable **x** kopieren wollen, sondern mit **x** nur ein Element referenzieren möchten, könnten Sie schreiben:

```
void increment()
{
    int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (auto& x : v)    // addiert 1 zu jedem x in v
        ++x;
    // ...
}
```

In einer Deklaration bedeutet das unäre Suffix **&** »Referenz auf«. Eine Referenz ist vergleichbar mit einem Zeiger, allerdings müssen Sie nicht das Präfix * benutzen,

um auf den Wert zuzugreifen, auf den durch die Referenz verwiesen wird. Außerdem kann im Gegensatz zu einem Zeiger eine Referenz nicht dazu gebracht werden, nach ihrer Initialisierung auf ein anderes Objekt zu verweisen.

Referenzen sind besonders nützlich beim Festlegen von Funktionsargumenten. Zum Beispiel:

```
void sort(vector<double>& v); // sortiert v (v ist ein Vektor aus doubles)
```

Indem Sie eine Referenz benutzen, stellen Sie sicher, dass Sie bei einem Aufruf von **`sort(my_vec)`** nicht aus Versehen **`my_vec`** kopieren. Es wird daher tatsächlich **`my_vec`** sortiert und nicht eine Kopie davon.

Wenn Sie ein Argument nicht modifizieren möchten, aber auch die Kosten des Kopierens vermeiden wollen, benutzen Sie eine **`const`**-Referenz (§1.6), das heißt eine Referenz auf eine Konstante. Zum Beispiel

```
double sum(const vector<double>&)
```

Funktionen, die **`const`**-Referenzen übernehmen, sind sehr verbreitet.

In Deklarationen werden Operatoren (wie **`&`**, **`*`** und **`[]`**) als *Deklaratoroperatoren* bezeichnet:

```
T a[n]    // T[n]: a ist ein Array aus n Ts
T* p      // T*: p ist ein Zeiger auf T
T& r      // T&: r ist eine Referenz auf T
T f(A)    // T(A): f ist eine Funktion, die ein Argument des Typs A
          // entgegennimmt und ein Ergebnis des Typs T zurückgibt
```

1.7.1 Der Null-Pointer

Sie versuchen, dafür zu sorgen, dass ein Zeiger immer auf ein Objekt zeigt, damit beim Dereferenzieren des Zeigers ein gültiges Ergebnis erzeugt wird. Wenn Sie kein Objekt haben, auf das gezeigt wird, oder Sie die Vorstellung vermitteln müssen, dass »kein Objekt verfügbar« ist (z. B. für das Ende einer Liste), dann geben Sie dem Zeiger den Wert **`nullptr`** (Null-Pointer). Es gibt für alle Typen von Zeigern nur ein **`nullptr`**:

```
double* pd = nullptr;
Link<Record>* lst = nullptr;    // Zeiger auf einen Link auf ein Record
int x = nullptr;         // Fehler: nullptr ist ein Zeiger, kein Integer
```

Oft erweist es sich als klug zu überprüfen, dass ein Zeigerargument tatsächlich auf etwas zeigt:

```
int count_x(const char* p, char x)
  // zählt, wie oft x in p[] auftritt
  // p soll auf ein 0-terminiertes Array aus char (oder auf nichts) zeigen
{
     if (p==nullptr)
         return 0;
     int count = 0;
     for (; *p!=0; ++p)
         if (*p==x)
             ++count;
     return count;
}
```

Man kann mit **++** einen Zeiger weiterrücken, sodass er auf das nächste Element eines Arrays zeigt, und außerdem den Initialisierer in einer **for**-Anweisung weglassen, wenn er nicht gebraucht wird.

Die Definition **count_x()** geht davon aus, dass **char*** ein *String im Stil von C* ist, das heißt, dass der Zeiger auf ein 0-terminiertes Array aus **char** zeigt. Die Zeichen in einem String-Literal sind unveränderlich, um also **count_x("Hello!")** zu verarbeiten, habe ich **count_x()** mit einem **const char***-Argument deklariert.

In älterem Code wird anstelle von **nullptr** üblicherweise **0** oder **NULL** benutzt. Mit **nullptr** vermeidet man jedoch eine mögliche Verwechslung zwischen Integern (wie **0** oder **NULL**) und Zeigern (wie **nullptr**).

In dem **count_x()**-Beispiel benutze ich für die **for**-Anweisung keinen Initialisierer, Sie können also auch die einfachere **while**-Anweisung einsetzen:

```
int count_x(const char* p, char x)
  // zählt, wie oft x in p[] auftritt
  // p soll auf ein 0-terminiertes Array aus char (oder auf nichts) zeigen
{
     if (p==nullptr)
         return 0;
     int count = 0;
     while (*p) {
         if (*p==x)
             ++count;
         ++p;
     }
     return count;
}
```

Die **while**-Anweisung wird ausgeführt, bis ihre Bedingung **false** wird.

Das Prüfen eines numerischen Werts (z. B. **while (*p)** in **count_x()**) ist äquivalent mit dem Vergleich des Werts mit **0** (hier also **while (*p!=0)**). Das Prüfen eines Zeigerwerts (z. B. **if (p)**) ist äquivalent mit dem Vergleich des Werts mit dem **nullptr** (hier also **if (p!=nullptr)**).

Es gibt keine »Nullreferenz«. Eine Referenz muss auf ein gültiges Objekt verweisen (und die Implementierungen gehen davon aus, dass sie es tut). Es gibt obskure und schlaue Möglichkeiten, diese Regel zu verletzen – machen Sie das nicht!

1.8 Bedingungen prüfen

C++ stellt die übliche Menge an Anweisungen zum Ausdrücken von Verzweigungen und Schleifen bereit, wie etwa **if**-Anweisungen, **switch**-Anweisungen, **while**- und **for**-Schleifen. Hier ist zum Beispiel eine einfache Funktion, die eine Benutzereingabe anfordert und einen booleschen Wert zurückgibt, der die Antwort anzeigt:

```
bool accept()
{
     cout << "Wollen Sie weitermachen (j oder n)?\n"; // Frage anzeigen
     char answer = 0;    // auf einen Wert initialisieren, der nicht in
                         // der Eingabe auftaucht
     cin >> answer;      // Antwort lesen

     if (answer == 'j')
         return true;
     return false;
}
```

Passend zum Ausgabeoperator **<<** (»ausgeben an«) gibt es den **>>**-Operator (»einlesen von«) für Eingaben; **cin** ist der Standard-Eingabe-Stream (Kapitel 11). Der Typ des rechten Operanden von **>>** legt fest, welche Eingabe akzeptiert wird; der rechte Operand ist außerdem das Ziel der Eingabeoperation. Das Zeichen **\n** am Ende des Ausgabestrings repräsentiert ein Newline (§1.2.1).

Beachten Sie, dass die Definition von **answer** dort erscheint, wo sie benötigt wird (und nicht schon vorher). Eine Deklaration kann überall dort auftauchen, wo auch eine Anweisung stehen kann.

Das Beispiel lässt sich noch verbessern, indem man auch die Antwort **n** (für »nein«) berücksichtigt:

```
bool accept2()
{
     cout << "Wollen Sie weitermachen (j oder n)?\n";  // Frage anzeigen
     char answer = 0;    // auf einen Wert initialisieren, der nicht in
                         // der Eingabe auftaucht
     cin >> answer;      // Antwort lesen

     switch (answer) {
     case 'j':
         return true;
     case 'n':
         return false;
     default:
         cout << "Das ist dann wohl ein Nein.\n";
         return false;
     }
}
```

Eine **switch**-Anweisung prüft einen Wert gegen eine Menge aus Konstanten. Diese Konstanten, **case**-Bezeichner genannt, müssen eindeutig sein. Falls der geprüfte Wert zu keinem der Bezeichner passt, wird **default** gewählt. Gibt es keinen zu dem Wert passenden **case**-Bezeichner und wurde auch kein **default** angegeben, dann findet gar keine Aktion statt.

Sie müssen ein **case** nicht verlassen, indem Sie aus der Funktion zurückkehren, die seine **switch**-Anweisung enthält. Oft wollen Sie die Ausführung mit der Anweisung fortsetzen, die auf die **switch**-Anweisung folgt. Das können Sie mit einer **break**-Anweisung erreichen. Schauen Sie sich als Beispiel einen überaus geschickten, wenn auch simplen Parser für ein einfaches, befehlsgesteuertes Videospiel an:

```
void action()
{
     while (true) {
         cout << "Aktion eingeben:\n";    // Aktion anfordern
         string act;
         cin >> act;               // Zeichen in einen String einlesen
         Point delta {0,0};        // Point enthält ein {x,y} Paar

         for (char ch : act) {
             switch (ch) {
             case 'u':               // nach oben (up)
```

```
            case 'n':                // nach Norden
                ++delta.y;
                break;
            case 'r':                // nach rechts
            case 'e':                // nach Osten (east)
                ++delta.x;
                break;
           // ... weitere Aktionen ...
            default:
                cout << "Ich hänge fest!\n";
            }
            move(current+delta*scale);
            update_display();
        }
    }
}
```

Genau wie eine **for**-Anweisung (§1.7) kann eine **if**-Anweisung eine Variable einführen und prüfen. Zum Beispiel:

```
void do_something(vector<int>& v)
{
    if (auto n = v.size(); n!=0) {
       // ... wir kommen hierher, falls n!=0 ...
    }
   // ...
}
```

Hier wird der Integer **n** für die Verwendung in der **if**-Anweisung definiert, mit **v.size()** initialisiert und sofort mit der Bedingung **n!=0** hinter dem Semikolon geprüft. Ein Name, der in einer Bedingung deklariert wird, befindet sich im Gültigkeitsbereich beider Zweige der **if**-Anweisung.

Wie bei der **for**-Anweisung deklariert man einen Namen in der Bedingung einer **if**-Anweisung, um den Gültigkeitsbereich der Variablen zu beschränken, womit man die Lesbarkeit verbessert und Fehler minimiert.

Am gebräuchlichsten ist es, eine Variable gegen **0** (oder den **nullptr**) zu prüfen. Dazu verzichten Sie einfach auf die explizite Erwähnung der Bedingung. Zum Beispiel:

```
void do_something(vector<int>& v)
{
```

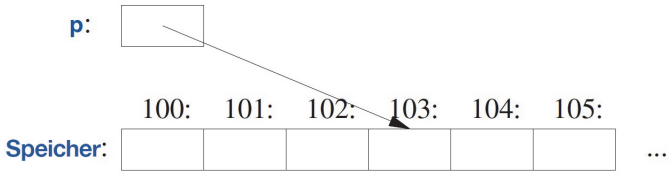
```
    if (auto n = v.size()) {
       // ... wir kommen hierher, falls n!=0 ...
    }
   // ...
}
```

Nutzen Sie nach Möglichkeit immer diese knappere und einfachere Form.

1.9 Auf Hardware abbilden

C++ ermöglicht eine direkte Abbildung auf Hardware. Wenn Sie eine der grundlegenden Operationen benutzen, dann wird Ihre Implementierung die sein, die die Hardware bietet. Zum Beispiel führt das Addieren zweier **`int`**, **`x+y`**, direkt den entsprechenden Maschinenbefehl aus.

Eine C++-Implementierung sieht den Speicher eines Computers als eine Sequenz von Speicherorten, in die sie (typisierte) Objekte legen kann, die sie mithilfe von Zeigern adressiert, also anspricht:



Ein Zeiger wird im Speicher als eine Maschinenadresse dargestellt, der numerische Wert von **p** in dieser Abbildung wäre also **103**. Falls das verdächtig nach einem Array (§1.7) aussieht, dann liegt das daran, dass ein Array für C++ die grundsätzliche Abstraktion einer »fortlaufenden Abfolge von Objekten im Speicher« ist.

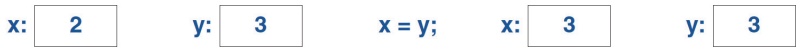
Die einfache Abbildung grundlegender Sprachkonstrukte auf die Hardware ist entscheidend für die sagenhafte, maschinennahe Arbeitsweise, für die C und C++ seit Jahrzehnten berühmt sind. Das C und C++ zugrunde liegende Maschinenmodell basiert tatsächlich auf der Computerhardware und nicht auf irgendwelchen Formen von Mathematik.

1.9.1 Zuweisung

Eine Zuweisung eines integrierten Typs ist ein einfacher Kopierbefehl auf Maschinenebene. Nehmen Sie dies hier an:

```
int x = 2;
int y = 3;
x = y;        // x wird 3; wir erhalten also x==y
```

Das ist eindeutig. Grafisch kann das so dargestellt werden:

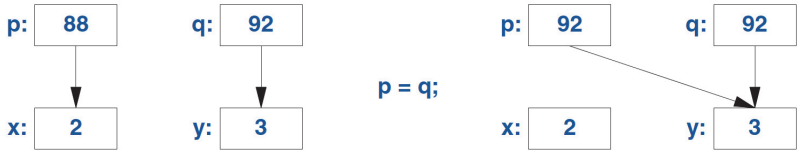


Die zwei Objekte sind unabhängig voneinander. Sie können den Wert von **y** ändern, ohne dass der Wert von **x** beeinträchtigt wird. Zum Beispiel verändert **x=99** den Wert von **y** nicht. Anders als in Java, C# und anderen Sprachen – aber genauso wie in C – gilt das für alle Typen, nicht nur für **int**s.

Falls Sie wollen, dass unterschiedliche Objekte auf den gleichen (gemeinsam genutzten) Wert verweisen, dann müssen Sie das angeben. Zum Beispiel:

```
int x = 2;
int y = 3;
int* p = &x;
int* q = &y;  // p!=q und *p!=*q
p = q;        // p wird &y; jetzt ist p==q, also (offensichtlich) *p==*q
```

Grafisch wird das so ausgedrückt:

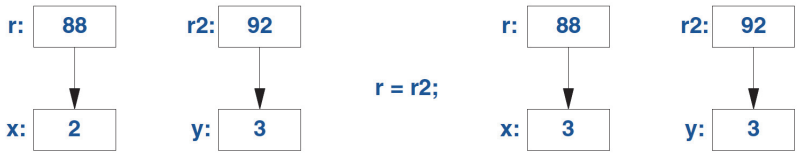


Ich habe willkürlich **88** und **92** als Adressen gewählt. Wieder können Sie sehen, dass das Objekt, auf das die Zuweisung erfolgt, den Wert aus dem zugewiesenen Objekt erhält, wodurch sich zwei unabhängige Objekte (hier: Zeiger) mit demselben Wert ergeben. Das heißt, **p=q** ergibt **p==q**. Nach **p=q** zeigen beide Zeiger auf **y**.

Eine Referenz und ein Zeiger verweisen/zeigen beide auf ein Objekt und beide werden im Speicher als Maschinenadresse dargestellt. Allerdings folgen sie unterschiedlichen Regeln in der Sprache. Die Zuweisung auf eine Referenz verändert nicht, worauf die Referenz verweist, sondern ändert den Inhalt des referenzierten Objekts:

```
int x = 2;
int y = 3;
int& r = x;     // r verweist auf x
int& r2 = y;    // r2 verweist auf y
r = r2;         // liest aus r2, schreibt auf r: x wird zu 3
```

Grafisch kann es so dargestellt werden:



Um auf den Wert zuzugreifen, auf den mit einem Zeiger gezeigt wird, benutzen Sie *; das wird für eine Referenz implizit gemacht.

Neben **x=y** haben Sie **x==y** für jeden integrierten Typ und gut designten benutzerdefinierten Typ (Kapitel 1), der = (Zuweisung) und == (Prüfung auf Gleichheit) anbietet.

1.9.2 Initialisierung

Die Initialisierung unterscheidet sich von der Zuweisung. Damit eine Zuweisung korrekt funktioniert, muss das Objekt, dem etwas zugewiesen wird, schon einen Wert haben. Die Aufgabe der Initialisierung andererseits ist es, ein nichtinitialisiertes Stück Speicher zu einem gültigen Objekt zu machen. Für fast alle Typen ist nicht definiert, welche Wirkung es hat, wenn aus einer nichtinitialisierten Variablen gelesen oder auf sie geschrieben wird. Betrachten Sie die Referenzen:

```
int x = 7;
int& r {x};   // bindet r an x (r verweist auf x)
r = 7;        // Zuweisung auf das, worauf r verweist

int& r2;      // Fehler: nichtinitialisierte Referenz
r2 = 99;      // Zuweisung auf das, worauf r2 verweist
```

Glücklicherweise können Sie keine nichtinitialisierte Referenz haben; wäre es möglich, dann würde dieses **r2 = 99** die **99** zu irgendeinem unspezifizierten Speicherort zuweisen; die Folge wären falsche Ergebnisse oder ein Absturz.

Sie können = verwenden, um eine Referenz zu initialisieren, aber lassen Sie sich davon nicht verwirren. Zum Beispiel:

```
int& r = x;   // bindet r an x (r verweist auf x)
```

Das ist immer noch eine Initialisierung, die **r** an **x** bindet, und nicht irgendeine Form des Kopierens von Werten.

Die Unterscheidung zwischen Initialisierung und Zuweisung ist auch für viele benutzerdefinierte Typen ausgesprochen wichtig, etwa für **string** und **vector**, bei denen das Objekt, dem etwas zugewiesen wurde, eine Ressource besitzt, die irgendwann wieder freigegeben werden muss (§6.3).

Die grundlegende Semantik der Argumentübergabe und Funktionswertrückgabe ist die der Initialisierung (§3.4). Zum Beispiel erhalten Sie auf diese Weise Referenzparameter (§3.4.1).

1.10 Ratschläge

Die Ratschläge, die Sie hier sehen, sind den C++ Core Guidelines [Stroustrup, 2015][1] entnommen. Verweise auf Guidelines sehen so aus: [CG: ES.23], womit die 23. Regel im *Expressions and Statement* gemeint wäre. Im Allgemeinen bietet eine solche Core Guideline weitere Erklärungen und Beispiele.

[1] Keine Panik! Alles wird im Laufe der Zeit klarer; §1.1; [CG: In.0].

[2] Benutzen Sie nicht ausschließlich die integrierten Features. Viele grundlegende (integrierte) Features verwendet man normalerweise am besten indirekt durch Bibliotheken, wie die ISO-C++-Standardbibliothek (Kapitel 9–Kapitel 18); [CG: P.13].

[3] Binden Sie die benötigten Bibliotheken mit **#include** oder (vorzugsweise) **import** ein, um das Programmieren zu vereinfachen; §1.2.1.

[4] Sie müssen nicht jede Einzelheit von C++ kennen, um gute Programme zu schreiben.

[5] Konzentrieren Sie sich auf Programmiertechniken, nicht auf Spracheigenschaften.

[6] Der ISO-C++-Standard ist die entscheidende Instanz bei Fragen und Problemen zur Sprachdefinition; §19.1.3; [CG: P.2].

[7] »Verpacken« Sie sinnvolle Operationen in Funktionen mit sorgfältig ausgewählten Namen; §1.3; [CG: F.1].

[8] Eine Funktion sollte eine einzige logische Operation ausführen; §1.3; [CG: F.2].

[9] Fassen Sie sich bei den Funktionen kurz; §1.3; [CG: F.3].

[10] Verwenden Sie das Überladen, wenn Funktionen konzeptuell die gleiche Aufgabe mit unterschiedlichen Typen ausführen; §1.3.

[11] Falls eine Funktion zum Zeitpunkt des Kompilierens ausgewertet werden darf, deklarieren Sie sie mit **constexpr**; §1.6; [CG: F.4].

[12] Falls eine Funktion zum Zeitpunkt des Kompilierens ausgewertet werden muss, deklarieren Sie sie mit **consteval**; §1.6.

1 https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

[13] Falls eine Funktion keine Nebeneffekte haben darf, deklarieren Sie sie mit **`constexpr`** oder **`consteval`**; §1.6; [CG: F.4].

[14] Entwickeln Sie Verständnis dafür, wie die Grundelemente der Sprache auf die Hardware abgebildet werden; §1.4, §1.7, §1.9, §2.3, §5.2.2, §5.4.

[15] Nutzen Sie Trennzeichen, um große Literale besser lesbar zu machen; §1.4; [CG: NL.11].

[16] Vermeiden Sie komplizierte Ausdrücke; [CG: ES.40].

[17] Vermeiden Sie verengende Konvertierungen; §1.4.2; [CG: ES.46].

[18] Minimieren Sie den Gültigkeitsbereich einer Variablen; §1.5, §1.8.

[19] Halten Sie die Gültigkeitsbereiche klein; §1.5; [CG: ES.5].

[20] Vermeiden Sie »magische Konstanten«, verwenden Sie symbolische Konstanten; §1.6; [CG: ES.45].

[21] Bevorzugen Sie unveränderliche Daten; §1.6; [CG: P.10].

[22] Deklarieren Sie (nur) einen Namen pro Deklaration; [CG: ES.10].

[23] Halten Sie häufig benutzte und lokale Namen (Bezeichner) kurz; halten Sie selten benutzte und nichtlokale Namen länger; [CG: ES.7].

[24] Vermeiden Sie ähnlich aussehende Namen; [CG: ES.8].

[25] Vermeiden Sie **`ALL_CAPS`**-Namen; [CG: ES.9].

[26] Benutzen Sie **`auto`**, um sich wiederholende Typnamen zu vermeiden; §1.4.2; [CG: ES.11].

[27] Vermeiden Sie nichtinitialisierte Variablen; §1.4; [CG: ES.20].

[28] Deklarieren Sie eine Variable erst, wenn Sie einen Wert haben, mit dem Sie sie initialisieren können; §1.7, §1.8; [CG: ES.21].

[29] Wenn Sie eine Variable in der Bedingung einer **`if`**-Anweisung deklarieren, dann bevorzugen Sie die Version mit dem impliziten Prüfen gegen **`0`** oder **`nullptr`**; §1.8.

[30] Bevorzugen Sie bereichsbasierte **`for`**-Schleifen gegenüber **`for`**-Schleifen mit einer expliziten Schleifenvariablen; §1.7.

[31] Benutzen Sie **`unsigned`** nur für Bit-Manipulationen; §1.4; [CG: ES.101][CG: ES.106].

[32] Halten Sie die Verwendung von Zeigern einfach und unkompliziert; §1.7; [CG: ES.42].

[33] Benutzen Sie **`nullptr`** statt **`0`** oder **`NULL`**; §1.7; [CG: ES.47].

[34] Sagen Sie in Kommentaren nichts, was ganz klar in Code ausgedrückt werden kann; [CG: NL.1].

[35] Drücken Sie in Kommentaren Ihre Absichten aus; [CG: NL.2].

[36] Wahren Sie einen konsistenten Stil bei Ihren Einrückungen; [CG: NL.4].