C++ SPEED RUN

SCRIPT ZUR DOZENTURREIHE AN DER SCHOOL4GAMES BERLIN GMBH

LARS KOKEMOHR, SOMMERSEMESTER 2017

INHALT

MEMORY MANAGEMENT

STACK UND HEAP

Die Besonderheit an C++ ist, dass man viel Einfluss darauf hat, wo und wie Daten im Speicher abgelegt werden.

Beim Programmstart wird der Stack (der „Stapel“), ein Speicherblock von fester Größe, für das Programm reserviert. Dieser wird für alle automatisch verwalteten Laufzeitdaten (wie Funktionsparameter, den Call Stack etc.) verwendet.

Außerdem kann man sich Speicher aus dem vom Betriebssystem verwalteten „Heap“ geben lassen und ihn selbst wieder freigeben.

Um eine Variable im Speicher zu erstellen müssen also zwei Dinge passieren:
1. Speicher für die Variable muss reserviert werden.
2. Der Speicher muss auf passende Startwerte gesetzt werden.

Da der Stack-Speicher schon beim Programmstart reserviert wird, muss hier nur der zweite Schritt passieren:

```
int x;      // Benennt Speicher von der Größe eines integer mit x
            // und richtet ihn für die Verwendung als Ganzzahl ein.
```

Bei Variablen auf dem Heap muss man hingegen beide Schritte vornehmen:

```
new int(); // new reserviert Speicher von der Größe eines integer
           // auf dem Heap, int() richtet ihn für die Verwendung
           // als Ganzzahl ein.
```

Da der so reservierte Speicher später auch wieder freigegeben werden muss, muss man sich merken, an welcher Adresse dieser Speicher liegt. Dazu speichert man den Rückgabewert von new in einer Variable, in der Regel auf dem Stack:

```
int* p = new int(); // Benennt Speicher von der Größe einer
                    // Adresse eines integer auf dem Stack mit p.
                    // Setzt p dann auf die Adresse des mit new
                    // int() reservierten Speichers auf dem Heap.
```

Um anzuzeigen, dass es sich um eine Adresse handelt, wird der Datentyp mit einem * versehen.
int* ist also eine Adresse eines (oder auch ein Zeiger auf einen) int.
T* wäre eine Adresse eines Objekts vom Typ T.
char** wäre eine Adresse eines char*, d.h. eines Speicherplatzes in dem die Adresse eines char gespeichert ist.

SYMBOLE IM ZUSAMMENHANG MIT ZEIGERN

Das Zeichen * wird ebenfalls benutzt, um von einer Adresse auf den Ort zu kommen, auf den sie zeigt.

```
int* p = new int(); // Erstelle eine neue Ganzzahl auf dem Heap.
                    // Speichere ihre Adresse in p.
*p = 3;             // Setze die Zahl an der eben gespeicherten
                    // Adresse auf 3.
```

In Verbindung mit einem Datentyp bedeutet * also „Zeiger auf“. In Verbindung mit einer Variable bedeutet es „Ziel von“.

Das Gegenstück zu * („Ziel von“) ist & („Adresse von“).

```
    int i = 3;
    int* p = &i;
    // p zeigt jetzt auf i,
    // denn im int-Zeiger p wird die Adresse von i gespeichert.

    *p = 5;
    // i ist jetzt 5, denn das, worauf p zeigt, wurde auf 5 geändert.
```

In Verbindung mit Zeigern sieht man oft auch die Zeichenkombination ->. Diese ist nur eine Kurzschreibweise für „ruf … auf das Ziel von … auf“.

```
class Test
{
public:
    void demo()
    {
        // ...
    }
};

int main()
{
    Test* t = new Test();

    t->demo();
    // Ruf demo() auf das Ziel von t auf.
    // Das ist gleichbedeutend mit:
    (*t).demo();
    // Hol das Ziel von t, also (*t), und ruf darauf demo() auf,
    // also .demo()
}
```

SPEICHERVERBRAUCH

Die Speichermenge, die ein Datentyp benötigt, ist nicht fest definiert und kann sich in Einzelfällen zwischen 32bit und 64bit Applikationen oder zwischen verschiedenen Compilern unterscheiden. Dieses sind die wichtigsten eingebauten Datentypen und ihre Größen in MSVC:

Datentyp	Größe	Wertebereich
Ganzzahlen		
char	1 Byte	-128 bis 127
short	2 Byte	-32.768 bis 32.767
int	4 Byte	-2.147.483.648 bis 2.147.483.647
long	4 Byte	-2.147.483.648 bis 2.147.483.647
long long	8 Byte	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
Der Wertebereich von Ganzzahlen kann mit dem Zusatz „unsigned“ verschoben werden, so dass er bei 0 beginnt (also z.B. unsigned char = von 0 bis 255).		
Fließkommazahlen		
float	4 Byte	
double	8 Byte	
long double	8 Byte	
Zeiger		
T*	4 Byte bei 32bit, 8 Byte bei 64bit	

OPERATOR NEW() UND CONSTRUCTORS

Beide Vorgänge (das Bereitstellen von Speicher und die Ersteinrichtung dieses Speichers) können verändert werden.

Heap-Speicher wird von der Funktion

```
void* operator new(size_t count);
```

bereitgestellt. Sie bekommt als Parameter die Menge an Bytes, die benötigt werden, übergeben und gibt die Adresse des neu erstellten Speicherblocks zurück. Diese Funktion kann überschrieben werden, wenn man genauer steuern möchte, wie der Speicher vom Betriebssystem bereitgestellt werden soll, oder wenn man den Speicher selbst verwalten möchte, in dem man sich beispielsweise bei Programmstart gleich einen großen Speicherbereich holt und diesen dann selbst verwaltet.
Außerdem verwendet man diese Technik um nachzuvollziehen, ob in einem Programm Speicher angefordert aber nie mehr freigegeben wurde. Durch einen entsprechend überschriebenen operator new() kann man sogar mitprotokollieren, durch welchen Code das Speicherleck verursacht wurde.

Von dieser Funktion gibt es einige Varianten, die wichtigsten sind die für den operator new[], der Speicher für Arrays bereitstellt, sowie operator new() als Memberfunktion, der dann nur dann aufgerufen wird, wenn Speicher für die Klasse erstellt wird, zu der die Memberfunktion gehört.

Passend zu den operator new()-Funktionen gibt es operator delete()-Funktionen, denen die Adresse übergeben wird, die von new() herausgegeben wurde, und die dann den dort bereitgestellten Speicher wieder freigeben.

Die Ersteinrichtung des Speichers passiert durch einen Konstruktor der Klasse, von der eine neue Instanz erstellt werden soll. Wird kein Parameter bei der Erstellung angegeben, wird der parameterlose Default-Konstruktor verwendet.

```
class Test
{
public:
    Test()
    {
        // dieser Code wird aufgerufen, wenn eine neue Instanz
        // von Test ohne Angabe von Parametern erstellt wird.
    }
};

int main()
{
    Test t;    // Verwendet den Default-Konstruktor
}
```

Es ist aber auch möglich, Konstruktoren mit Parameter zu schreiben:

```
class Test
{
public:
    Test(int i)
    {
        // dieser Code wird ausgeführt, wenn eine neue Instanz
        // von Test mit einem Int als Parameter erstellt wird.
    }
};

int main()
{
    // Alle folgenden Zeilen verwenden den Konstruktor Test(int i)
    Test t(3);
    Test t2 = 3;
    Test t3{ 3 };  // C++11
    Test t4 = { 3 }; // C++11
}
```

Sofern ein Konstruktor nicht mit dem Keyword „explicit“ versehen wird, kann er auch für einen Zwischenschritt verwendet werden, um automatisch Daten zu konvertieren.

```
void demo(Test t)
{
    // Macht etwas mit t.
}

int main()
{
    demo(3);   // ruft implizit demo(Test(3)); auf, da der Konstruktor
               // Test(int i) nicht mit „explicit“ deklariert wurde.
}
```

Auch zu den Konstruktoren gibt es ein Gegenstück: der Destruktor wird aufgerufen, wenn ein Objekt gelöscht wird und gibt gegebenenfalls Ressourcen wieder frei.

```
class Test
{
    int* p;
    int* p2;

public:
    // Constructor
    Test()
        : p(new int(3))
        , p2(nullptr)
    {}

    void test()
    {
        p2 = new int(4);
    }

    // Destructor
    ~Test()
    {
        delete p;
        delete p2;
    }
};
```

MEMBER INITIALIZER LISTS

Der Code für einen Konstruktor hat eine Besonderheit: zwischen der Funktionssignatur und dem Funktionskörper kann ein Block aus deklarativem Code eingefügt werden, mit dem Member der Klasse initialisiert werden können. Das ist beispielsweise nötig, um konstante Member auf einen Wert zu setzen, da diese nach der Erstellung ihren Wert nicht mehr ändern können.

```
class Test
{
public:
    int x;
    float y;
    float z;

    Test()
        : x(1)      // deklarativer Code: x wird direkt mit dem
        , y(2.0f)   // Wert 1, y mit dem Wert 2.0f erstellt
    {
        z = 3.0f;   // imperativer Code: z wird nach der Erstellung
                    // auf 3.0 gesetzt.
    }
};
```

MEMORY LAYOUT

Die Reihenfolge, in der Variablen in einer Klasse zusammengefasst werden, bestimmt, in welcher Reihenfolge sie später im Speicher liegen.

```
class Demo
{
    int x;
    char y[4];
    float z;
};

int main()
{
    Demo d;
    Demo* p = new Demo();
}
```

Die Variable d verweist also auf einen Speicherbereich auf dem Stack, der erst aus den Bytes eines integer, dann denen von vier chars und dann denen eines float besteht.
Die Variable p enthält die Adresse eines gleich aufgebauten Speicherblocks im Heap.

ALIGNMENT

Es steht dem Compiler frei, zwischen den Membervariablen einer Klasse Puffer einzufügen um die Performance zu erhöhen. In der Regel wird der Compiler versuchen, Speicherblöcke immer auf Vielfachen ihrer eigenen Größe beginnen zu lassen.

```
class Demo
{
    char c;
    int i;
};
```

Hier wird also voraussichtlich zwischen c und i ein Puffer von drei Bytes eingefügt, damit der vier Byte große integer vier Bytes Abstand vom Beginn des ein Byte großen char hat.

CASTING

Da in C++ Datentypen lediglich angeben, wie groß ein Speicherblock ist, wie er aufgebaut ist, auf welche Werte er bei der Erstellung gesetzt werden soll und welche Funktionen auf ihn aufgerufen werden können, ist es möglich, einen Speicherblock mit unterschiedlichen Typinformationen anzusprechen. Die dabei vorgenommene Umdeutung des Datentyps nennt man „casten“.

```
class Color
{
public:
    unsigned char r;
    unsigned char g;
    unsigned char b;
    unsigned char a;
};

class ColorId
{
public:
    int id;
};

int main()
{
    Color* c = new Color();
    ColorId* i = (ColorId*)c;

    // c und i zeigen nun auf den gleichen Speicher, verwenden ihn
    // aber unterschiedlich.
}
```

Es gibt drei Arten, einen Cast erkenntlich zu machen:

1. C-style casts:

```
    float* f = new float();
    int* i = (int*)f;
```

2. C++-style casts:

```
    // reinterpret_cast ändert den Datentyp
    float* f = new float();
    int* i = reinterpret_cast<int*>(f);

    // dynamic_cast ändert den Datentyp und prüft zur Laufzeit, ob
    // die Daten vom gewünschten Typ sind
    Base* b = new Child();
    Child* c = dynamic_cast<Child*>(b);// Dieser cast ist gültig.
    float* f = dynamic_cast<float*>(b);// Dieser cast gibt 0 zurück.


    // const_cast macht konstante Variablen veränderlich
    const float* f = new float();
    float* g = const_cast<float*>(f);
```

3. Mit unions:

```
union FloatOrInt
{
    float f;   // Dieser Speicher soll ohne ausdrücklichen Cast
    int i;     // sowohl als float als auch als int ansprechbar sein.
};

int main()
{
    FloatOrInt test;
    test.f = 1.23f; // test.i verwendet den gleichen Speicher wie
    test.i = 5;    // test.f, daher ist test.f jetzt überschrieben.
}
```

Dabei muss beachtet werden, dass Casts nur dann zulässig sind, wenn es Code für die Konvertierung gibt (d.h. einen Konstruktor oder einen operator [Typ](), der den neuen Datentyp aus dem alten erzeugt) oder wenn die beiden Typen durch Vererbung voneinander abhängen.

Soll ein Datentyp in einen anderen verwandelt werden ohne dass einer dieser Zusammenhänge besteht, muss der Cast über einen Zeiger passieren:

```
class Test
{
    int x;
};

int main()
{
    int i = 3;

    // Test t = (Test)i;
    // FEHLER, führt zu:
    // error C2440: "Typumwandlung":
    // "int" kann nicht in "Test" konvertiert werden

    Test* t = (Test*)&i;
    // Kein Fehler: Die Adresse von i kann in
    // „Zeiger auf Test“ umgewandelt werden.
}
```

ZEIGER-ARITHMETIK

Da Zeiger lediglich Adressen von Datensätzen enthalten, ist es möglich, diese Werte um Ganzzahlen zu erhöhen oder verkleinern. Dabei ändert sich die Adresse immer um Vielfache der Größe des Datentyps, auf den der Zeiger zeigt.

```
class Test
{
    char w;
    char x;
    char y;
    char z;
    int a;
};

int main()
{
    Test* t = new Test(); // t zeigt auf Speicher, der entsprechend
                          // der Klasse Test dimensioniert ist

    char* c = (char*)t;   // c zeigt auf den Anfang von t, also auf t.w
    char* c2 = t + 1;     // c2 zeigt auf das Byte, das auf t.w folgt,
                          // also auf t.x

    int* i = (int*)i;    // i zeigt auf die ersten vier Bytes in Test
    int* i2 = i + 1;     // i2 zeigt auf die zweiten vier Bytes,
                         // also auf t.a
}
```

ARRAYS

Es ist in C++ möglich, mehrere gleiche Variablen in direkter Folge im Speicher anzulegen. Diese werden als C-Arrays bezeichnet:

```
    char c[4];     // Eine Folge von 4 chars
```

C-Arrays haben in C++ große Ähnlichkeit mit Zeigern. Es gibt Situationen, in denen sie sich nicht genau wie ein Zeiger verhalten, man kann sie aber leicht in einen verwandeln und so weiterverwenden.

```
int main()
{
    int i[5];      // Eine Folge von 5 Ganzzahlen.
    int* p = i;    // Hier ist kein Cast nötig, da i wie ein Zeiger
                // auf den Beginn der 5 Ganzzahlen verwendet werden
                // kann.
}
```

Wenn man ein C-Array auf dem Heap erstellt, muss man den Zeiger darauf nicht mit delete sondern mit delete[] freigeben:

```
int main()
{
    int* p = new int[5]; // Eine Folge von 5 Ganzzahlen auf dem Heap.
    delete[] p;          // Freigeben des Arrays.
}
```

VERWENDUNG VON ZEIGERN

Folgende Zeichen muss man im Zusammenhang mit Zeigern kennen:

```
int main()
{
    int i = 3;
    int* p = &i;   // &i ist die Adresse von i.
    *p = 5;        // *p ist das Ziel von p, i ist jetzt also 5.

    int a[5];      // a ist zu verwenden wie die Adresse einer
                   // Folge von 5 ints.
    *a = 1;        // Das Ziel von a ist die erste Ganzzahl.
    *(a + 1) = 2;  // Das Ziel von a+1 ist die zweite Ganzzahl.
    a[1] = -2;     // a[1] ist das Gleiche wie *(a+1), also auch die
                   // zweite Ganzzahl.

}
```

FUNKTIONSAUFRUFE UND DER STACK

Der Stack wird für automatisch verwaltete Laufzeitdaten verwendet. Dazu gehören lokale Variablen in Funktionen, aber auch Funktionsparameter und sogar Informationen zum Funktionsaufruf selbst. Das hat zur Folge, dass bei einem Funktionsaufruf die Parameter der Funktion ans aktuelle Ende des Stacks kopiert werden, selbst wenn sie vorher bereits auf dem Stack lagen.

Das bringt unter Umständen zwei Probleme mit sich: werden große Objekte als Parameter verwendet müssen viele Daten kopiert werden. Außerdem hat jede Funktion nur auf Kopien der Parameter Zugriff, kann also einen Wert nicht verändern.

Um diese Probleme zu vermeiden, kann man lediglich die Adresse der zu verwendenden Daten übergeben. Diese Adresse ist klein und daher billig zu kopieren, und da auch eine Kopie der Adresse immer noch auf die gleichen Daten zeigt, kann eine Funktion über eine kopierte Adresse auf die Originaldaten zugreifen und sie verändern.

```
void test(int x)
{
    x = 3;
}

void test2(int* p)
{
    *p = 4;
}

int main()
{
    int i = 1;

    test(i);
    // i ist jetzt immer noch 1, da test() lediglich eine
    // Kopie von i verändert hat.

    test2(&i);
    // i ist jetzt 4, da test2() über die Adresse von i auf
    // den Speicher, in dem i gespeichert ist, zugegriffen
```

```
    // hat und ihn verändert hat.
}
```

Für diese Fälle bietet es sich an, mit Referenzen zu arbeiten. Referenzen sind technisch im Grunde das gleiche wie Zeiger, sie lassen sich jedoch wie Variablen verwenden, man erspart sich also die explizite Dereferenzierung.

```
void test(int* p)
{
    // Zeiger: umständlichere Syntax
    *p = 4;
}

void test2(int c)
{
    // Kopie: kann den Originalwert nicht verändern
    c = 6;
}

void test3(int& r)
{
    // Hier kann der Code so geschrieben werden als wäre r eine
    // Kopie, das Ergebnis ist aber das Gleiche wie bei
    // der Verwendung eines Zeigers.
    r = 5;
}

int main()
{
    int i = 1;

    test(&i); // Zeiger: umständlichere Syntax
    test2(i); // Kopie: verändert i nicht
    test3(i); // Referenz: einfache Syntax, verändert i
}
```

Wird ein Parameter kopiert spricht man auch von call-by-value, wird hingegen eine Referenz oder ein Zeiger übergeben spricht man von call-by-reference.

VERERBUNG

Ein zentrales Konzept der Objektorientierung ist die Vererbung, auch „ist-ein-Beziehung“ genannt, da sie ausdrückt, dass ein Objekt eines Typs immer gleichzeitig auch ein Objekt eines anderen Typs ist. Beispielsweise ist jede Eiche ein Baum. Der Objekttyp „Eiche“ erbt also alle Eigenschaften des Typs „Baum“.

Für Klassen bedeutet das, dass alle Membervariablen der Basisklasse in die Kindklasse übernommen werden und dass alle Funktionen, die man auf die Basisklasse aufrufen kann, auch auf die Kindklasse aufgerufen werden können. Das wird in C++ so erreicht, dass die Basisklasse effektiv als erster (unsichtbarer) Member übernommen wird und (unsichtbare) Memberfunktionen erstellt werden, die die öffentlichen Memberfunktionen der Basisklasse aufrufen.

```
class Base
{
public:
    void test()
    {};

private:
    int x;
};

class Child : public Base
{
private:
    float y;
};

int main()
{
    Child c;
    // Die Klasse Child beginnt jetzt mit dem Speicher für den
    // Integer x, das heißt mit den Daten der Basisklasse.
    // Erst dann kommt der Speicher für die Kommazahl y.

    c.test();
    // Auf c kann die Funktion test() aufgerufen werden, die
    // in der Basisklasse definiert ist.
}
```

Dieses Prinzip gilt auch für Mehrfachvererbung, die jedoch schnell zu Mehrdeutigkeiten führen kann und deswegen von der großen Mehrheit der Programmierer vermieden wird.

Neben den Sichtbarkeitsmodifikatoren public (für alle sichtbar) und private (nur aus der aktuellen Klasse sichtbar) gibt es noch den Modifikator protected, der so viel bedeutet wie „private für alle außer Childklassen“.

BESONDERHEITEN BEI KLASSEN

VERDECKUNG UND VIRTUELLE METHODEN

Wenn eine Kindklasse eine Funktion beinhaltet, die die gleiche Signatur hat wie die der Basisklasse, dann verdeckt die neue Funktion die der Basisklasse. Welche Variante der Funktion aufgerufen wird, entscheidet sich nach dem Typ der Variable, über die auf das Objekt zugegriffen wird.

```
class Base
{
public:
    void test()
    {
        printf("Code of the base class.\n");
    }
};

class Child : public Base
{
public:
    void test()
    {
        printf("Code of the child class.\n");
    }
};

int main()
{
    Base* b = new Child();

    b->test();
    // gibt "Code of the base class." aus.

    ((Child*)b)->test();
    // gibt "Code of the child class.“ aus.
}
```

Mit dem Keyword „virtual“ kann man hingegen erreichen, dass von einer Methode immer die Variante aufgerufen wird, die zu dem Datentyp passt, der bei der Erstellung verwendet wurde. Das Keyword „override“ in der Childklasse ist optional, verhindert aber Fehler durch Tippfehler oder spätere Codeänderungen.

```
class Base
{
public:
    virtual void test()
    {
        printf("Code of the base class.\n");
    }
};

class Child : public Base
{
public:
    virtual void test() override
    {
        printf("Code of the child class.\n");
    }
};

int main()
{
    Base* b = new Child();

    b->test();
    // ACHTUNG: gibt "Code of the child class." aus, da
    // test() mit dem Keyword „virtual“ deklariert wurde und
    // b auf ein Objekt vom Typ Child zeigt.

    ((Child*)b)->test();
    // gibt weiterhin "Code of the child class.“ aus.
}
```

BESONDERE METHODEN

Es gibt einige Methoden in jeder Klasse, die anders behandelt werden als normale, selbst geschriebene Methoden. Dazu gehören die Konstruktoren und der Destruktor sowie die Operatoren.

KONSTRUKTOREN UND DESTRUKTOR

Die Konstruktoren und der Destruktor wurden schon unter „Operator new() und Constructors“ beschrieben. Es gibt jedoch einen besonderen Konstruktor, und zwar den, der als Parameter ein Objekt des Typs akzeptiert, das gerade erstellt werden soll. Dieser Konstruktor soll eine Kopie eines anderen Objekts erstellen und wird entsprechend immer dann automatisch aufgerufen, wenn diese Situation auftritt. Daher nennt man diesen Konstruktor auch den „Copy Constructor“.

```
class Test
{
    int i;

public:
    Test(int p_i)
        : i(p_i)
    {}

    Test(const Test& p_other)
        : i(p_other.i)
    {
        // Wird immer aufgerufen, wenn eine Kopie von einer
        // Instanz von Test erzeugt wird.
    }
};

void run(Test t)
{
    // t wird als Parameter auf den Stack kopiert.
}

int main()
{
    Test t(3);
    Test t2 = t;   // Erzeugt eine Kopie von t

    run(t);        // Erzeugt automatisch eine Kopie,
                   // da t „by value“, also als Kopie, übergeben wird

}
```

Wird hingegen ein bereits existierendes Objekt auf den Wert eines anderen gesetzt, wird der Gleichheitsoperator verwendet.

```
class Test
{
    int i;

public:
    Test()
        : i(0)
    {}

    void operator=(const Test& p_other)
    {
        i = p_other.i;
    }
};

int main()
{
    Test t;
    Test t2;
    t2 = t;    // Ruft operator=() auf, da t bereits existiert.

    Test t3 = t;   // Ruft den Copy Constructor auf,
                   // da t3 gerade erst erzeugt wird.
}
```

RULE OF THREE / FIVE

Wenn man keinen Konstruktor selbst schreibt, wird vom Compiler automatisch ein Defaultkonstruktor, ein Copy Constructor, ein Destruktor und ein operator=() geschrieben. Sobald man jedoch einen Konstruktor selbst schreibt, wird keine der Funktionen mehr generiert.

Daher sollte man sobald man etwas überschreibt gleich die „big three“ mit überschreiben, d.h. den Copy Constructor, den operator=() und den Destruktor.

In C++11 ändert sich diese Regel zur „rule of five“, denn dort müssen dann auch der Move Constructor und der operator=() für rvalues überschrieben werden.

OPERATORS

Neben den Konstruktoren und dem Destruktor gibt es noch eine weitere Gruppe von Funktionen, die besonders behandelt werden: Operatoren. Diese Funktionen tragen „operator“ im Namen und können durch besondere Syntax aufgerufen werden. So ist operator+() beispielsweise die Funktion, die dann aufgerufen wird, wenn zwei Werte addiert werden.

```
class Test
{
    int i;

public:
    Test(int p_i)
        : i(p_i)
    {}

    Test operator+(const Test& p_other) const
    {
        return Test(i + p_other.i);
    }
};

int main()
{
    Test t1(1);
    Test t2(2);

    // Diese Zeile
    Test t3 = t1 + t2;
    // ist eine andere Schreibweise für
    Test t4 = t1.operator+(t2);
}
```

Mithilfe dieser Funktionen können Klassen so geschrieben werden, dass sie sehr intuitiv verwendet werden können. Die Gefahr bei intuitivem Code ist allerdings immer, dass man falsch einschätzt, was andere als intuitiv erachten und was nicht. Deswegen sollte man gut überlegen bevor man Operatoren für die eigenen Klassen programmiert.

Eine besondere Art von Operator sind die Operatoren, mit denen man eine Klasse in eine andere Verwandeln kann:

```
class Test
{
    int i;

public:
    Test(int p_i)
        : i(p_i)
    {}

    operator int() const
    {
        return i;
    }

    operator float() const
    {
        return 1.0f * i;
    }
};

int main()
{
    Test t(3);
    int i = t;     // verwendet Test::operator int()
    float f = t;   // verwendet Test::operator float()
}
```

PRIVATE CONSTRUCTORS, DELETED METHODS, PURE VIRTUAL FUNCTIONS, DEFAULTED METHODS

Wenn man verhindern möchte, dass von einer Klasse eine neue Instanz erstellt werden kann, kann man die Konstruktoren der Klasse als private deklarieren. Seit C++11 kann man dazu auch das Keyword delete verwenden. Das Ergebnis ist in beiden Fällen, dass ein Aufruf von new …() schon beim Kompilieren zu einem Fehler führt und somit verdeutlicht wird, dass man von dieser Klasse keine Instanzen selbst erstellen soll.

```
class Test
{
private:
    Test();
};

class Test2
{
    Test2() = delete;
};

int main()
{
    //  new Test();
    //  Test t;
    // error C2248: 'Test::Test':
    // cannot access private member declared in class 'Test'

    //  new Test2();
    //  Test2 t;
    // error C2280: 'Test2::Test2(void)':
    // attempting to reference a deleted function
}
```

Außerdem kann man virtuelle Funktionen mit =0 deklarieren, die dann nur formal vorhanden sind. Das führt dazu, dass von einer Klasse keine Instanz mehr erstellt werden kann, sie kann aber weiterhin als Basisklasse verwendet werden. Solche Klassen nennt man abstrakte Klassen.

```
class Animal
{
public:
    // move kann hier nicht definiert werden,
    // weil sich nicht jedes Tier gleich bewegt
    virtual void move() = 0;
};

class Bird : public Animal
{
    void fly() {};

public:
    virtual void move() override
    {
        fly();
    };
};

class Horse : public Animal
{
    void run() {};

public:
    virtual void move() override
    {
        run();
    };
};

int main()
{
    // Animal a;
    // Fehler: Die Klasse ist abstrakt, wenn man nur von
    // „einem Tier" spricht, ist nicht genug über das Objekt
    // bekannt, um es zu erstellen.

    Animal* b = new Bird();
    b->move();

    Animal* h = new Horse();
    h->move();
    // Kein Problem: Bird und Horse sind Child classes von Animal
    // die move() definieren, daher weiß der Compiler jetzt alles,
    // was er braucht um mit diesem Animal arbeiten zu können.
}
```

Außerdem kann man mit =default dafür sorgen, dass Funktionen, für die es eine Default-Implementation gibt (Default-Konstruktor, Copy Constructor, Zuweisungsoperator etc.), automatisch erstellt werden, wenn sie andernfalls nicht erstellt würden.

```
class Test
{
    int i;

public:
    // Ein selbst geschriebener Konstruktor,
    // ab jetzt wird nichts mehr automatisch erstellt.
    Test(int p_i)
    : i(p_i)
    {}

    // =default sorgt dafür, dass dieser Konstruktor
    // doch wieder automatisch erstellt wird
    Test() = default;
};
```

METAPROGRAMMING

Als Metaprogramming bezeichnet man es, wenn man Code schreibt, der nicht direkt kompiliert wird, sondern der erst den Code generieren soll, der kompiliert wird. Dazu stehen in C++ mehrere mehr oder weniger mächtige Features zur Verfügung.

Dabei handelt es sich um Makros, mit denen man willkürlich ein Token durch anderen Text ersetzen kann, Defines, mit denen man Code ein- und ausblenden kann, Templates, mit denen man Klassen und Funktionen generieren kann indem ein Platzhalter durch einen Datentyp oder eine Ganzzahl ersetzt wird und typedefs, mit denen ein Datentyp umbenannt werden kann.

MAKROS UND DEFINES

Das mächtigste dieser Features sind Makros und Defines. Technisch sind diese beiden Techniken identisch, allerdings werden sie unterschiedlich verwendet.

Ein Makro ersetzt einen beliebigen Token durch einen vorgegebenen Text. Dabei kann es wahlweise auch einen oder mehrere Parameter enthalten. Soll der Inhalt eines Makros über mehrere Zeilen gehen, müssen alle bis auf die letzte mit \ beendet werden.

```
#define MAX 10
#define MUL(x,y) x*y

#define LONG_MAKRO printf(„Dieses Makro"); \
printf(„geht über mehrere Zeilen".);

int main()
{
    int x = MAX;
    // gleichbedeutend mit
    // int x = 10;

    int y = 3;
    int z = MUL(x, y);
    // gleichbedeutend mit
    // int z = x*y;
}
```

Makros sind mit Abstand das mächtigste C++-Feature, aber auch das unsicherste, da hier wirklich nur Text ersetzt wird. So kann das MUL-Makro aus dem Beispiel oben auch auf diesen Weise verwendet werden:

```
#define MUL(x,y) x*y

int main()
{
    int* p = new int(3);
    int i = MUL(, p);

    // i ist jetzt 3
    // denn MUL(,p) wird zu *p, da der erste Parameter leer ist.
    // Aus dem Sternchen für mal wird also
    // das Sternchen für „Ziel von".
}
```

An dem Beispiel sieht man, dass sich Makros nicht für die Bedeutung der Zeichen interessieren sondern sie blind austauschen. Dadurch können im Fehlerfall sehr überraschende Folgen entstehen.
Wer also Makros einsetzen möchte, sollte sich vorher informieren, wie man Makros am sichersten gestaltet. In der Regel ist es aber besser, nach Möglichkeit auf andere Techniken zurückzugreifen.

Wenn etwas mit #define als Makro definiert wurde, kann es mit #undef wieder gelöscht werden. Das ist manchmal nötig um zu verhindern, dass sich Makrodefinitionen überschneiden oder dass etwas als Makro definiert wird, was in anderem Code beispielsweise als Klassenname verwendet wird.

Außerdem lässt sich mit #ifdef abfragen ob etwas definiert ist bzw. mit #ifndef ob es nicht definiert ist. Damit lassen sich ganze Codeblöcke ausschalten, beispielsweise wenn man unterschiedlichen Code für verschiedene Systeme schreiben möchte oder wenn man Teile des Programms nur im Debugmodus aktivieren möchte.

```
#define DEBUG

int main()
{
#ifdef DEBUG
    printf("Dies wird ausgegeben, weil DEBUG definiert ist.\n");
#else
    printf(„Dies wird übersprungen, weil DEBUG definiert ist.\n“);
#endif

#ifndef DEBUG
    printf(„Dies wird übersprungen, weil DEBUG definiert ist.\n“);
#endif


#undef DEBUG

#ifndef DEBUG
    printf("Dies wird ausgegeben, weil DEBUG nicht mehr definiert
ist.\n");
#endif
}
```

Diese Technik wird bei den sogenannten Include Guards verwendet. Da der #include-Befehl den Inhalt einer Headerdatei in die aktuelle Datei hineinkopiert, und die so kopierte Datei ihrerseits weitere #includes enthalten kann, kann es passieren, dass eine Datei doppelt kopiert wird. In dem Fall würden Klassen doppelt deklariert, was zu Fehlern führen würde.

Um das zu vermeiden, wird in jeder Headerdatei geprüft, ob ein willkürliches Token definiert ist. Wenn nicht, wird es definiert und der Code der Headerdatei übernommen.

```
#ifndef __TEST_H__
#define __TEST_H__

class Test
{
    //…
}

#endif /* __TEST_H__ */
```

In manchen Compilern ist es auch zulässig stattdessen das kürzere Makro #pragma once zu verwenden.

TYPEDEFS, USING-ALIASE

Mit dem Keyword typdef (und seit C++11 auch mit dem Keyword using) kann man für einen Datentyp einen anderen Namen angeben. Das kann man dazu benutzen, Code besser lesbar zu machen, in der Regel ist aber der Hintergedanke, dass man auf diese Weise in einer späteren Programmversion einen Datentyp ändern kann, ohne jede Verwendung des Datentyps manuell anpassen zu müssen.

```
typedef int Ganzzahl;
using Kommazahl = float;

int main()
{
    Ganzzahl i = 3;
    Kommazahl f = 2.4f;
}
```

TEMPLATES

Templates sind ähnlich wie Makros eine Möglichkeit, Code aus Bausteinen zu generieren. Anders als Makros ist aber stark eingeschränkt, was als Parameter angegeben werden kann, nämlich Datentypen oder Ganzzahlen. Außerdem werden Templates etwas anders verwendet als Makros. Während Makros erst definiert und dann im Code verwendet werden müssen,

```
#define MUL_FUNC int mul(int x, int y) \
{ \
    return x*y; \
}

// Damit der Code für die Funktion „mul" auch kompiliert wird,
// muss das Makro mindestens einmal aufgerufen werden.
MUL_FUNC

int main()
{
    int x = mul(3, 4);
}
```

werden Templates automatisch ausgeführt, wenn sie benötigt werden.

```
template<class T>
T multiply(T x, T y)
{
    return x*y;
}

int main()
{
    int a = multiply(3, 4);
}
```

Der Compiler versucht dabei selbst zu erkennen, für welche Datentypen das Template erzeugt werden muss.

```
template<class T>
T multiply(T x, T y)
{
    return x*y;
}

int main()
{
    int a = multiply(3, 4);
    float b = multiply(3.0f, 4.0f);

    // float c = multiply(3, 4.0f);
    // FEHLER: Der gewünschte Typ kann nicht automatisch
    // ermittelt werden, da ein Parameter ein int und einer
    // ein float ist.
```

Wenn der benötigte Typ nicht automatisch ermittelt werden kann, muss er explizit angegeben werden.

```
    float c = multiply<float>(3, 4.0f);
    // So geht es, denn der Typ wurde manuell angegeben.
}
```

Statt „class“ kann auch das Keyword „typename“ verwendet werden.

```
template<typename T>
T multiply(T x, T y)
{
    return x*y;
}
```

Statt Datentypen kann auch eine Ganzzahl als Templateparameter angegeben werden.

```
template<typename T, int SIZE>
class Array
{
    T values[SIZE];
};

int main()
{
    Array<int, 5> a;     // a.values ist ein Array von 5 ints.
    Array<float, 10> b; // b.values ist ein Array von 10 floats.
}
```

Es ist möglich, zu einem Template Ausnahmen selbst zu definieren. Dafür lässt man bei den Templateparametern mindestens einen weg und hängt ihn stattdessen am Funktions- bzw. Klassennamen in der gewünschten Variante an. Aus

```
template<class T>
void test();
```

würde also

```
template<>
void test<float>();
```

Hier ein Beispiel für eine Templatefunktion, die in der Regel zwei Zahlen mithilfe des *-Zeichens multipliziert und für eine eigene Klasse eine Ausnahme definiert.

```
class MyVector
{
public:
    MyVector add(const MyVector& p_other)
    {
        // hier würde die Summe der Vektoren errechnet
    }
};

// gibt das generelle Template für add<…> an.
template<class T>
T add(T a, T b)
{
    return a+b;
}

// gibt den Spezialfall für add<MyVector> an, denn für
// die Klasse MyVector ist operator* nicht definiert.
template<>
MyVector add<MyVector>(MyVector a, MyVector b)
{
    return a.add(b);
}

int main()
{
    int a = add(3, 4);
    float b = add(3.0f, 4.0f);

    MyVector v1;
    MyVector v2;
    MyVector v3 = add(v1, v2);
}
```

Im vorigen Beispiel wird auch deutlich, dass auch Templates nur wenig mehr machen, als Code zu generieren. Der Compiler wird nur Fehler anmerken, die aus dem Template-Code selbst erkennbar werden. Wird hingegen eine Funktion aufgerufen, die es nicht gibt, wird dies erst im Linker auffallen.

Wird hingegen eine gleichnamige Funktion mit anderer Bedeutung aufgerufen, wird dies weder vom Compiler noch vom Linker bemängelt und nur lediglich zu unerwarteten Fehlern führen.

```
class Screen
{
public:
    // stellt text auf dem Bildschirm dar
    static void display(std::string text)
    {
        printf("displaying text: %s\n", text.c_str());
    }
};

class GraphicsManager
{
public:
    // soll ein Display mit dem Namen screenName zurückgeben
    static Screen* display(std::string screenName)
    {
        return new Screen();
    }
};

// soll einen Text in ein Log schreiben
template<class T>
void log(std::string text)
{
    // Diese Zeile geht davon aus, dass display() immer einen
    // Text darstellt.
    T::display(text);
}

int main()
{
    log<Screen>("Dieser Text wird sinnvoll dargestellt.");
    log<GraphicsManager>("Diese Zeile sucht nach einem Screen mit dem
Namen dieses Texts...");

    // HINWEIS: Dieses Negativbeispiel verstößt gegen die Regel,
    // Funktionen immer mit Verben zu benennen.
    // Die eine heißt display() im Sinne von „stell etwas dar“,
    // die andere im Sinne von „gib mir ein Display“.
    // Wird die Regel eingehalten, ist der Fehler viel
    // unwahrscheinlicher.
}
```

Tipp: in vielen Fällen werden unnötig Templates verwendet wo „Polymorphie“, das heißt Vererbung und virtuelle Methoden, gereicht hätten, denn auch virtuelle Methoden sorgen dafür, dass abhängig vom tatsächlichen Typ eines Wertes unterschiedliche Methoden aufgerufen werden.

CONST UND CONSTEXPR

Mit dem Keyword const kann deklariert werden, dass eine „Variable“ unveränderlich sein soll. Das lässt sich sowohl auf Stackvariablen als auch auf den Typ, auf den ein Pointer zeigt, anwenden.

```
int main()
{
    // konstante Ganzzahl
    const int i = 3;

    // Fehler: i kann nach der Erstellung nicht verändert werden.
    // i = 5;

    // Zeiger auf konstante Ganzzahl
    const int* p1 = new int(4);

    // Fehler: das Ziel von p1 ist constant
    // *p1 = 5;

    // Erlaubt: p1 ist ein nicht konstanter Zeiger
    p1 = p1 + 1;

    // konstanter Zeiger auf (nicht konstante) Ganzzahl
    int* const p2 = new int(5);

    // Fehler: p2 ist constant
    // p2 = p2 + 1;

    // Erlaubt: p2 zeigt auf nicht konstanten int
    *p2 = 1;
}
```

Als Faustregel hilft es, die Deklarationen von hinten nach vorne zu lesen. So ist „const int*“ ein Zeiger auf const int, während „int* const“ ein konstanter Zeiger auf int ist.

Außer Variablen können auch Memberfunktionen mit const deklariert werden. Das hat zur Folge, dass für die Dauer dieser Funktion alle Member der Klasse konstant sind, also nicht verändert werden dürfen.

Diese Einschränkung ist wichtig, da sie zum einen dem Compiler erlaubt, den Code besser zu optimieren, und da nur konstante Funktionen auf konstante Objekte aufgerufen werden können, da nur diese formal garantieren, dass sie am Objekt nichts ändern.

```
class Test
{
    int i;
    float f;

public:
    void test() const
    {
        // Fehler: i ist konstant.
        // i = 3;

        // Fehler: f ist ebenfalls konstant.
        // f = 2.5f;
    }

    void test2()
    {
        // Erlaubt: test2 ist nicht mit const deklariert
        i = 3;
        f = 2.5f;
    }
};

int main()
{
    const Test t;

    // Erlaubt: test() garantiert, dass t unverändert bleibt
    t.test();

    // Fehler: test2() könnte t ändern, doch t ist konstant.
    // t.test2();
}
```

Außerdem kann man konstante Ausdrücke für Deklarationen verwenden, da sie schon zum Kompilierzeitpunkt feststehen.

```
    const int bufferSize = 10;
    char buffer[bufferSize];
    // Deklariert einen Speicherblock von 10 Byte Länge.

    //int i = 5;
    //char c[i];
    // Deklariert einen Speicherblock von ?? Byte Länge -
    // i kann sich zur Laufzeit ändern und kann deswegen
    // nicht schon vor Programmstart auf einen Wert
    // festgelegt werden.
    // Fehler: i ist nicht constant.
```

Seit C++11 gibt es außer const auch noch das keyword constexpr. Mit diesem kann man Funktionen

kennzeichnen, die zu jedem Zeitpunkt für gleiche Eingabewerte auf gleiche Ergebnisse kommen (sogenannte pure functions).

Ähnlich wie das const-Keyword erlaubt das dem Compiler den Code besser zu optimieren (z.B. wenn eine Funktion immer nur mit einem von drei Parameterwerten aufgerufen wird, können die Ergebnisse schon beim Kompilieren ausgerechnet und abgespeichert werden).
Außerdem ermöglicht es, solche Funktionen in Deklarationen zu verwenden.

```
int bufferSize(int count, size_t elementSize)
{
    return count * elementSize;
}

constexpr int bufferSize2(int count, size_t elementSize)
{
    return count * elementSize;
}

int main()
{
    const int numElements = 10;
    const size_t elementSize = sizeof(int);

    //char buffer[bufferSize(numElements, elementSize)];
    // Fehler: bufferSize ist nicht constexpr,
    // daher kann der Compiler nicht annehmen,
    // dass immer das gleiche Ergebnis herauskommt.

    char buffer[bufferSize2(numElements, elementSize)];
    // bufferSize2 ist constexpr, es wurde also garantiert,
    // dass das Ergebnis bei jedem Aufruf mit bestimmten
    // Werten auf das gleiche Ergebnis kommt, egal ob
    // es vom Kompiler oder erst zur Laufzeit aufgerufen wird.
}
```

ÄNDERUNGEN IN C++11 UND 14

ENUMS

Wenn man eine Konstante festlegen möchte, lässt sich das über eine einfache konstante Variable bewerkstelligen. Oft muss man aber mehrere Konstanten festlegen, die miteinander im Zusammenhang stehen. Für diese Fälle kann man enums verwenden.

```
const int targetFramerate = 60;

enum ActorType
{
    Player,     // Player ist jetzt 0
    Npc,        // Npc ist jetzt 1
    Monster     // Monster ist jetzt 2
}
```

Die enum automatisiert dabei, wie die Konstanten Zahlenwerten zugeordnet werden. Wenn wie im Beispiel oben nichts weiter angegeben wird, wird der ersten Konstante die Zahl 0 zugeordnet und für die folgenden Werte einfach hochgezählt. Alternativ kann man auch selbst Werte angeben, fehlende Werte werden dann weiterhin einfach hochgezählt. Dabei können Werte auch doppelt vergeben werden (wie bei TestB, TestF und TestG im Beispiel).

```
enum TestId
{
    TestA,          // TestA ist jetzt 0
    TestB,          // TestB ist jetzt 1
    TestC = 123,    // TestC ist jetzt 123
    TestD,          // TestD ist jetzt 124
    TestF = 1,      // TestF ist jetzt 1
    TestG = 1       // TestG ist jetzt 1
}
```

Ein übliches Pattern ist, eine enum mit einem Wert für ungültige Einträge und der Zahl -1 zu beginnen und mit einem Wert für die Anzahl der gültigen Konstanten zu beenden. Dadurch kann leicht überprüft werden, ob eine Zahl in den gültigen Wertebereich einer enum fällt.

```
enum MessageId
{
    InvalidMessageId = -1,

    Login,          // Login ist jetzt 0
    Logout,         // Logout ist jetzt 1
    CreatePlayer,   // CreatePlayer ist jetzt 2

    NumMessageIds   // NumMessageIds ist jetzt 3,
                    // die Anzahl der gültigen MessageIds.
};

bool isValidMessageId(int p_id)
{
    return ((p_id > InvalidMessageId)
        && (p_id < NumMessageIds));
}
```

Enums sind nur beschränkt typsicher. So ist es zwar nicht möglich, kommentarlos eine Ganzzahl zu verwenden wo ein Enum-Wert erwartet wird, umgekehrt geht das aber schon.

```
enum TestId
{
    One = 1,
    Two,
    Three
};

void testA(int p_i)
{
}

void testB(TestId p_t)
{
}

int main()
{
    // Diese Zeile wird ohne Probleme kompiliert.
    testA(One);

    // FEHLER:
    // error C2664: 'void testB(Test)':
    // cannot convert argument 1 from 'int' to 'TestId'
    testB(2);

    // Kein Problem
    testB((TestId)2);
}
```

Seit C++11 kann man daher statt einer enum eine „enum class“ verwenden, was zu mehr Typsicherheit führt:

```
enum class TestId
{
    One = 1,
    Two,
    Three
};

void testA(int p_i)
{
}

int main()
{
    testA((int)TestId::One);
}
```

Statt One muss die Konstante jetzt mit TestId::One angegeben werden, und auch die Typumwandlung zum int passiert jetzt nicht mehr automatisch.

TYPE INFERENCE

Mit C++11 ist ein neuer Mechanismus zur Type Inference, also zur automatischen Typerkennung hinzugekommen. In C++98 gab es bereits einen solchen Mechanismus, und zwar bei der automatischen Erkennung von Templateparametern.

```
template<class T>
void test(T p_t)
{
}

int main()
{
    int i = 3;
    // ruft automatisch test<int>(i) auf.
    test(i);

    float f = 2.5f;
    // ruft automatisch test<float>(f) auf.
    test(f);
}
```

Der Aufruf “test(i)” bzw. “test(f)” enthält keine Angabe, mit welchem Typ test<> generiert werden soll, das wird automatisch ermittelt.

Dieses Prinzip lässt sich seit C++11 auch für lokale Variablen verwenden. Wie bei Templates auch muss allerdings gleich aus der ersten Zeile erkennbar sein, um welchen Typ es sich handelt. Eine so weit reichende Type Inference wie man sie aus Sprachen wie Haxe oder F# kennt bietet C++ also nicht.

```
int main()
{
    auto i = 3;
    auto f = 3.0f;
    auto d = 3.0;

    //Fehler: Typ kann nicht sofort ermittelt werden.
    // auto a;
    // a = 3;
}
```

An dieser Stelle muss darauf hingewiesen werden, dass die Type Inference mit auto zu leicht anderen Ergebnissen kommt als die bei den Templates. Dieser Unterschied kommt aber erst bei recht speziellem Code zum Tragen, deswegen würde eine detaillierte Erklärung hier den Rahmen sprengen.

Eine Variable mit auto zu deklarieren sorgt dafür, dass der Datentyp nicht mehr ausdrücklich dasteht, es verringert also die Lesbarkeit. Auf der anderen Seite hat es jedoch die Vorteile, dass im Regelfall der Typ nicht mehr doppelt angegeben werden muss, dass sich der Typ der Variable bei späteren Codeänderungen automatisch mit ändern kann und dass in komplizierten Fällen durch die automatische Erkennung Fehler vermieden werden können.

```
class Test
{};

class Test2
{};

Test* test()
{
    return new Test();
}

Test2* newVersionOfTest()
{
    return new Test2();
}

typedef float Zahl;
Zahl test2()
{
    return 2.2f;
}

int main()
{
    // Test kommt doppelt vor, müsste also bei Änderungen
    // doppelt geändert werden
    Test* t1 = new Test();

    // Mit "auto": Der Klassenname kommt nur einmal vor
    auto t2 = new Test();

    // Diese Zeile müsste nicht angepasst werden, wenn sich
    // der Rückgabetyp von test() ändert.
    auto t3 = test();
    // auto t3 = newVersionOfTest();

    double val = test2();
    // Hier wurde fälschlicherweise angenommen, dass test2
    // einen double zurückgibt.
    auto val2 = test2();
    // Mit auto kann der Fehler nicht passieren,
    // es hätte hier also zu einem effizienteren Ergebnis
    // geführt.
}
```

SCHLEIFEN

Es gibt in C++ zwei Arten von Schleifen: while- und for-Schleifen. Bei beiden wird der Code in der Schleife so lange ausgeführt wie eine Bedingung zutrifft. Der Unterschied ist, dass die for-Schleife zusätzlich Platz bietet für eine Anweisung die einmal zu Beginn ausgeführt werden soll und für eine Anweisung, die nach jedem Schleifendurchlauf einmal aufgerufen wird.

```
    // Beispiel 1: for-Schleife
    for (int i = 0; i<10; ++i)
    {
        // dieser Code wird so lange ausgeführt bis i
        // nicht mehr kleiner als 10 ist.
    }

    // Beispiel 2: while-Schleife
    int i = 0;
    while (i<10)
    {
        // dieser Code wird so lange ausgeführt bis i
        // nicht mehr kleiner als 10 ist.
        i++;
    }
```

Die Startanweisung „int i=0;“ und die Anweisung „i++;“ tauchen in beiden Beispielen auf, haben bei der for-Schleife aber einen fest definierten Platz. For- und while-Schleifen sind also komplett austauschbar, sind aber je nach Situation unterschiedlich sinnvoll. Im vorigen Codebeispiel wäre eine for-Schleife angebrachter, im folgenden ist die while-Schleife sinnvoller.

```
#include <cmath>

void testWhile(int p_value)
{
    while (p_value < 1000)
    {
        if (p_value < 500)
            p_value = rand() % 1000;
        else
            p_value++;
    }
}

void testFor(int p_value)
{
    for (; p_value < 1000; )
    {
    if (p_value < 500)
        p_value = rand() % 1000;
    else
        p_value++;
    }
}
```

In diesem Beispiel ist keine Startanweisung nötig, da p_value bereits definiert ist. Der Code, der in jedem Schleifendurchlauf gleich ist, ist hingegen durch die if-else-Weiche so umfangreich, dass es nicht sinnvoll wäre, ihn in die Kopfzeile der for-Schleife zu schreiben. Daher wurde in dem Beispiel der erste und der dritte Teil der for-Kopfzeile freigelassen.

Ein Sonderfall sind Endlosschleifen, die in der Regel als while-Schleifen, in selteneren Fällen als for-Schleifen geschrieben sind.

```
while (true)
{
    // wird unendlich oft wiederholt
}

for (;;)
{
    // wird unendlich oft wiederholt
}
```

Endlosschleifen machen nur in Verbindung mit den Keywords break und continue Sinn (sofern sie nicht mit return abgebrochen werden).

Break bricht die Schleife ab und fährt hinter dem Ende der Schleife mit der Ausführung fort. Continue hingegen bricht lediglich den aktuellen Schleifendurchlauf ab und fährt mit dem nächsten Durchlauf fort. Ein „continue;“ in der letzten Zeile einer Schleife ändert somit nichts am Ergebnis.

```
for (int i = 0; true; i++)
{
    // Wenn i ungerade ist
    if (i % 2 == 1)
    {
        // wird der folgende Code übersprungen und mit dem
        // nächsten i fortgefahren
        continue;
    }

    // Wenn eine willkürliche Bedingung zutrifft
    if (someRandomCondition(i))
    {
        // soll die Schleife komplett beendet werden.
        break;
    }
}
```

Von beiden Schleifen gibt es je eine Variante.

Passend zur while-Schleife gibt es die do-while-Schleife, bei der die Bedingung nicht vor sondern erst nach dem ersten Schleifendurchlauf geprüft wird.

```
int i = 1000;
while (i<1)
{
    printf("Diese Zeile wird nicht ausgegeben,");
    printf(" da i nie kleiner als 1 ist.\n");
    i++;
}

do
{
    printf("Diese Zeile wird einmal ausgegeben,");
    printf(" da die Bedingung erst nach dem ersten");
    printf(" Durchlauf geprüft wird.\n");
    i++;
} while (i<1);
```

Die Variante für die For-Schleife ist hingegen eine Kurzschreibweise für ein häufiges Pattern, was auftritt, wenn man den Code in der Schleife einmal für jeden Wert in einem Container ausführen lassen möchte.

```
#include <list>

int main()
{
    std::list<int> container = { 1,2,3,4,5 };

    // Klassische for-Schleife
    for (auto it = container.begin(); it != container.end(); it++)
    {
        int value = *it;
        printf("Value: %i", value);
    }

    // Range-based loop
    for (auto value : container)
    {
        printf("Value: %i", value);
    }
}
```

Der „range based loop“ (die For-Schleife der Form “value : container”) lässt sich also immer dann anwenden, wenn es für das Objekt „container“ eine Memberfunktion begin() und eine Memberfunktion end() gibt, die beide einen Forward-Iterator zurückgeben, d.h. ein Objekt, auf das die Operationen ++ (post-inkrement) und * (Dereferenzierung) angewendet werden können.

Oder anders gesagt: der Code, der im letzten Codebeispiel unter „klassische for-Schleife“ steht, muss gültig sein, damit der ranged-based loop angewendet werden kann.

```
class Test
{
public:
    int values[5];

    int* begin()
    {
        return values;
    }

    int* end()
    {
        return values + 5;
    }
};

int main()
{
    Test t = { 1,2,3,4,5 };
    for (auto i : t)
    {
        printf("value: %i\n", i);
    }
}
```

Eine Ausnahme gibt es aber noch: die Funktionen begin() und end() müssen nicht unbedingt als Memberfunktionen implementiert sein, sie können auch globale Funktionen sein, die den Container als Parameter akzeptieren.

```
class Test
{
public:
    int values[5];
};

int* begin(Test& p_t)
{
    return p_t.values;
}

int* end(Test& p_t)
{
    return p_t.values + 5;
}

int main()
{
    Test t = { 1,2,3,4,5 };
    for (auto i : t)
    {
        printf("value: %i\n", i);
    }
}
```

INITIALIZER LISTS

Seit C++11 gibt es außer den in einem früheren Kapitel beschriebenen (member) initializer lists auch noch ein neues Konstrukt unter dem Namen initializer lists, daher wurden erste nachträglich in „member“ initializer list umbenannt.

Eine initializer list ist eine kommaseparierte Liste von Werten zwischen zwei geschweiften Klammern.

```
#include <initializer_list>

auto initializerList = { 1,2,3,4,5 };
// initializerList ist jetzt vom Typ std::initializer_list<int>
```

Wie member initializer lists auch sind initializer lists dafür gedacht, in Verbindung mit Konstruktoren genutzt zu werden. Während member initializer lists allerdings einen festen Platz im Konstruktor haben, können initializer lists benutzt werden, um die Startwerte eines Objekts von außen anzugeben.

```
#include <initializer_list>

class Test
{
public:
    int i;
    float f;
    char c;

    //Test()
    //  : i(1), f(2.2f), c('c')      // <- member initializer list
    //{}
};

int main()
{
    //Test t1;
    Test t2 = { 2, 3.3f, 'd' };     // <- initializer list
}
```

Mit den initializer lists wurde allerdings auch versucht, eine einheitliche Schreibweise für die Initialisierung einzuführen. Je nach Situation muss man in C++98 mal die eine und mal die andere Schreibweise zur Initialisierung verwenden:

```
class Test
{
    const int c = 3;     // hier muss = verwendet werden
    //const int c(3);    // diese Schreibweise ist hier nicht erlaubt

    int i;

public:
    Test(int p_i)
        : i(p_i)         // hier muss () verwendet werden
        //: i = p_i      // diese Schreibweise ist hier nicht erlaubt
    {}
};
```

Mit initializer lists wurde dieses Problem gelöst:

```
class Test
{
    const int c{ 3 };    // hier ist {} erlaubt

    int i;

public:
    Test(int p_i)
        : i{ p_i }       // und hier auch
    {}
};
```

Auf der anderen Seite ist das Prinzip, nach dem entschieden wird, auf welche Weise die initializer list verwendet wird, etwas komplizierter geworden. In C++98 wurden grundsätzlich die Konstruktoren verwendet, sowohl bei der Schreibweise mit () als auch der mit =.

Werden hingegen initializer lists verwendet, gibt es drei Möglichkeiten, aus denen nach einem (ziemlich) eindeutigen Prinzip ausgewählt wird:

1. Wenn eine Klasse einen Konstruktor hat, der einen Parameter vom Typ std::initializer_list<T> akzeptiert, wird dieser verwendet, sofern die angegebenen Werte in den Typ T konvertiert werden können.
2. Nur wenn in Schritt 1 kein passender Konstruktor gefunden wird, wird nach Konstruktoren gesucht, deren Parameter zu den Werten in der initializer list passen.
3. Nur wenn kein selbst geschriebener Konstruktor existiert und alle Variablen der Klasse öffentlich sind werden die Werte der initializer list direkt in die Variablen der Klasse übernommen.

```
#include <initializer_list>

class Test
{
public:
    int i;
    float f;

    Test() {};
    Test(int p_i, float p_f) {};
    Test(std::initializer_list<int>) {};
};

int main()
{
    // Test t{ 1,2.0f };
    // Fehler: es existiert eine Konvertierung von float nach int,
    // daher wird der Konstruktor mit std::initializer_list<int>
    // verwendet.
    // Da aber nicht jeder float in einen int gespeichert werden
    // kann, wird ein Fehler ausgegeben.
    // Achtung: der „passendere" Konstruktor Test(int, float) wird
    // ignoriert, da Regel 1 zutrifft, nur halt zu Fehlern führt.

    Test t2{ 1, 2 };
    // Es wird der Konstruktor mit std::initializer_list<int>
    // verwendet.
}
```

Wird der letzte Konstruktor entfernt, ist das Ergebnis:

```
#include <initializer_list>

class Test
{
public:
    int i;
    float f;

    Test() {};
    Test(int p_i, float p_f) {};
};

int main()
{
    Test t{ 1,2.0f };    // Der Konstruktor Test(int, float)
                         // wird verwendet
}
```

Wird erneut der letzte Konstruktor entfernt, ist das Ergebnis:

```
#include <initializer_list>

class Test
{
public:
    int i;
    float f;

    Test() {};
};

int main()
{
    // Test t{ 1,2.0f };
    // Fehler: es existieren Konstruktoren, aber keiner passt
    // zu den Werten in der initializer list

    // Workaround:
    Test t;
    t.i = 1;
    t.f = 2.0f;
}
```

Entfernt man jetzt noch den letzten Konstruktor, erhält man:

```
class Test
{
public:
    int i;
    float f;
};

int main()
{
    Test t{ 1,2.0f };
    //Da kein Konstruktor angegeben ist, wird die initializer list
    // direkt auf die öffentlichen Member angewendet.
}
```

Besonders praktisch sind initializer lists im Zusammenhang mit STL-Containern wie std::list oder std::map. Außerdem kann man initializer lists auch verschachteln, d.h. eine Liste kann andere enthalten.

```
#include <initializer_list>
#include <list>
#include <map>
#include <string>

class Test
{
public:
    int i;
    float f;
    std::list<char> list;
    std::map<std::string, int> map;
};

int main()
{
    Test t{ 1,2.0f,{ 'a', 'b', 'c' },{ { "one", 1 },{ "two", 2 } } };
}
```

LAMBDA-FUNKTIONEN / CLOSURES

In klassischen objekt-orientierten Sprachen sind Funktionen nichts, was in einer Variable gespeichert werden könnte. Variablen sind in diesen Sprachen für Daten reserviert, Funktionen werden im Code beim Namen genannt und während des Kompiliervorgangs in eine feste Verbindung zwischen dem Ort des Aufrufs und dem aufzurufenden Code verwandelt.

```
class Player
{
    void onDamaged()
    {
        printf("Player was damaged!");
    }

public:
    void receiveDamage()
    {
        onDamaged();
        // Dieser Aufruf wird immer die Methode onDamaged() aufrufen.
```

```
    }
};
```

Oftmals möchte man aber die Möglichkeit haben, von außen festzulegen, welche Funktion in einer bestimmten Situation aufgerufen werden soll. Im Beispiel oben müsste vielleicht eine Gesundheitsanzeige, die nicht Teil der Spielerklasse ist, geupdatet werden.

Um das zu ermöglichen, kann man in klassischen objekt-orientierten Sprachen eine Klasse erstellen, die kaum mehr als die eine aufzurufende Funktion enthält. Man verpackt also die Funktion, die man nicht in einer Variable speichern kann, in einem Objekt, was sich in einer Variable speichern lässt.

```
class PlayerDamageHandler
{
public:
    void onPlayerDamaged()
    {}
};

class Player
{
    PlayerDamageHandler* handler;

public:
    void setPlayerDamageHandler(PlayerDamageHandler* p_handler)
    {
        handler = p_handler;
    }

    void receiveDamage()
    {
        handler->onPlayerDamaged();
        // Diese Zeile ruft die onPlayerDamaged-Methode in dem
        // in setPlayerDamageHandler() übergebenen Objekt auf.
    }
};
```

Die PlayerDamageHandler-Klasse hat also nur den Sinn, eine Funktion in einem ansonsten mehr oder weniger leeren Objekt zu verpacken. Um das am Code besser erkennbar zu machen, kann man in C++ für die Funktion den operator() verwenden, was dazu führt, dass das Objekt wie eine Funktion aussieht. Daher nennt man diese Art von Objekt einen Functor.

```
class OnPlayerDamaged
{
public:
    void operator()()
    {
        printf("Player was damaged!");
    }
};

class Player
{
    OnPlayerDamaged* callback;

public:
    void setOnPlayerDamaged(OnPlayerDamaged* p_callback)
    {
        callback = p_callback;
    }

    void receiveDamage()
    {
        OnPlayerDamaged& onPlayerDamaged = *callback;
        // onPlayerDamaged ist eine Referenz auf das Ziel
        // von callback

        onPlayerDamaged();
        // Diese Zeile ruft die operator()-Methode in dem
        // in setOnPlayerDamaged() übergebenen Objekt auf.
        // Sie ist also gleichbedeutend mit
        // (*callback)();
        // oder
        // callback->operator()();
    }
};

int main()
{
    Player p;

    // Erstelle ein neues OnPlayerDamaged-objekt, das die
    // gewünschte Funktion enthält.
    OnPlayerDamaged* cb = new OnPlayerDamaged();
    p.setOnPlayerDamaged(cb);

    p.receiveDamage();
}
```

Dieses Verfahren kann schnell dazu führen, dass man eine Menge „Boilerplate-Code“ schreibt, also Code, der nur formal wichtig ist, aber wenig neue Informationen beinhaltet.

Seit C++11 kann man sich diesen Umweg sparen, denn mit Lambda-Funktionen gibt es eine kürzere Syntax, um solche Funktorobjekte automatisch erstellen zu lassen. Die so erstellten Objekte nennt man dann Closures.

Im folgenden Beispiel wird ein Closure-Objekt mit einer Lambda-Funktion erstellt. Um es zu speichern wird in der Player-Klasse ein std::function<>-Objekt verwendet. Dieses braucht einen ungewöhnlichen Template-Parameter, nämlich die Signatur (also alles bis auf Funktions- und Parameternamen) der Funktion, die durch das Closure-Objekt verpackt warden soll.

```
#include <functional>

class Player
{
    std::function<void()> callback;

public:
    void setOnPlayerDamaged(const std::function<void()>& p_callback)
    {
        callback = p_callback;
    }

    void receiveDamage()
    {
        callback();
    }
};

int main()
{
    Player p;

    auto cb = []() {printf("Player was damaged!"); };
    p.setOnPlayerDamaged(cb);

    p.receiveDamage();
}
```

In der Zeile „auto db = [](){…};“ wird das Closure-Objekt erstellt. Als Datentyp kann hier nur auto angegeben werden, da das Closure-Objekt automatisch erstellt wird und der Klassenname also gar nicht bekannt ist. Zeigt man mit dem Mauszeiger in Visual Studio auf „auto“, sagt der Tooltip, dass der Typ „class lambda []void ()->void“ erkannt wurde.
Im Call Stack wird eine solche Closure-Klasse dann mit einem generierten Namen wie „main::__l2::<lambda>„ auftauchen.

Eine Lambda-Funktion besteht aus vier Teilen, von denen zwei optional sind:

1	2	3	4
[]	()	->…	{}
	(optional)	(optional)	

1: In den eckigen Klammern zu Beginn werden eventuelle Member zum Closure-Objekt hinzugefügt, indem lokale Variablen kopiert oder als Referenz übernommen werden.

```
int main()
{
    int a = 0;
    float b = 1.1f;

    auto closure = [a, &b] {a += 1; b += 1.0f; };
    // a wird als Kopie übernommen,
    // b als Referenz (aufgrund des &-Zeichens)

    closure();

    // b ist jetzt 2.1f, da closure eine Referenz auf b verändert hat.
    // a ist hingegen immer noch 0, da closure nur eine Kopie
    // verändert hat.
}
```

Sollen alle Werte als Kopie oder alle als Referenz übernommen werden, dann braucht man nur & (alles als Referenz) oder = (alles als Kopie) in die Klammern zu schreiben und es wird automatisch erkannt, welche Variablen benötigt werden.

```
    int a = 0;
    float b = 1.1f;

    // Da nur [=] angegeben wurde, werden alle benötigten Variablen
    // kopiert. Es werden a und b kopiert, weil die in der Lambda-
    // Funktion verwendet werden.
    auto closure = [=] {printf("%i, %f\n", a, b); };

    // Hier wurde nur [&] angegeben, es werden also a und b
    // referenziert.
    auto closure2 = [&] {a += 1; b += 1.0f; };
```

Seit C++14 gibt es noch eine weitere Möglichkeit anzugeben, was in der Closure an Werten übernommen werden soll, die noch flexibler ist:

```
    int a = 2;
    float b = 1.1f;

    // a wird unter dem Namen a als Kopie übernommen
    // b wird unter dem Namen b als Referenz übernommen
    // a*b wird unter dem Namen product gespeichert
    auto closure =[a = a, &b = b, product = a*b]
                  {printf("%i, %f, %f\n", a, b, product); };
    closure();
```

Der []-Teil der Lambdafunktion ist also so etwas wie der Konstruktor des Closure-Objekts.

2: In den folgenden runden Klammern werden die Parameter für die eingeschlossene Funktion angegeben.

```
    int factor = 2;

    auto timesTwo = [=](int p_value) {return factor * p_value; };

    int x = timesTwo(3);
    int y = timesTwo(4);
    // x ist jetzt 6, y ist jetzt 8
```

Bei Funktionen, die keine Parameter haben, sind diese Klammern deswegen optional.

```
    auto print  = [] () {printf("with brackets\n"); };
    auto print2 = [] {printf("without brackets\n"); };
```

3: Hinter den runden Klammern kann angegeben werden, welcher Datentyp zurückgegeben wird. Da dieser in der Regel automatisch erkannt wird, braucht er nur dann angegeben zu werden, wenn die automatische Erkennung nicht funktioniert oder zum falschen Ergebnis käme.

```
    auto returnInt  = [] {return 1; };
    // automatisch erkannt

    auto returnFloat = []() -> float {return 1; };
    // automatisch erkannter Typ wäre int

    auto returnIntPtr = []() -> int* {return nullptr; };
    // nullptr hat keinen Typ, daher kann der Rückgabetyp
    // nicht automatisch erkannt werden.
```

4: Am Schluss kommt der Inhalt der Funktion in geschweiften Klammern.

Eine Besonderheit bei Lambda-Funktionen ist, dass Parameter mit auto angegeben werden können. In diesem Fall verhält sich die Funktion wie ein Template, d.h. die Typen werden bei jeder Verwendung neu erkannt.

```
int main()
{
    auto div = [](auto x, auto y) {return x/y; };

    auto a = div(5, 2);
    // a verwendet int durch int gleich int

    auto b = div(5.0f, 2.0f);
    // b verwendet float durch float gleich float

    printf("a ist %i, b ist %f\n", a, b);
}
```

SMART POINTERS

Eine der wichtigsten Regeln in C++ lautet, dass es zu jedem new ein delete und zu jedem new [] ein delete [] geben muss. Hält man sich nicht daran, riskiert man Speicherlecks, das heißt dass angeforderter Speicher nicht mehr freigegeben wird und das Programm so lange es läuft immer mehr Arbeitsspeicher verbraucht.

```
void test()
{
    int* i = new int();
    delete i;

    int* a = new int[10]();
    delete[] a;

    int* p = new int();
    // Speicherleck: beim Verlassen der Funktion endet der Scope
    // von p, d.h. es ist nicht mehr möglich, auf die Stackvariable p
    // zuzugreifen. Somit ist die Adresse des angeforderten Speichers
    // nicht mehr bekannt und er kann nicht mehr freigegeben werden.
}
```

Um dieses Risiko zu vermeiden wurden smart pointer erfunden. Diese Objekte verpacken einen Pointer und geben ihn automatisch frei bevor ein Speicherleck entstehen kann. Je nach Typ des smart pointers wird dabei eine andere Regel angewendet.

AUTO_PTR

Ein auto_ptr löscht das Objekt auf das er zeigt sobald er aus dem Scope läuft.

```
#include <memory>

void test()
{
    std::auto_ptr<int> p(new int());

    // hier wird p gelöscht und damit auch das Ziel von p
}
```

Wenn man diesen smart pointer kopieren möchte, stellt man allerdings ein ungewöhnliches Verhalten fest: ein auto_ptr verhindert, dass zwei Kopien auf das gleiche Objekt zeigen, damit nicht einer, wenn er aus dem Scope läuft, die Ressource des anderen löscht. Dies wird beim auto_ptr allerdings dadurch erreicht, dass schon im Moment des Kopierens die erste Kopie ungültig gemacht wird.

```
#include <memory>

class Test
{
public:
    void demo() {};
};

int main()
{
    std::auto_ptr<Test> p(new Test());
    p->demo();

    auto q = p;
    // Hier wird der Inhalt von p nach q verschoben,
    // so dass p anschließend ungültig ist.

    // In Ordnung: q ist jetzt gültig.
    q->demo();

    // p->demo();
    // Fehler: q ist nicht mehr gültig,
    // p hat den Inhalt von q übernommen.
}
```

Dieser smart pointer sollte nicht mehr verwendet werden, weil er in der Regel besser durch unique_ptr und shared_ptr ersetzt werden kann. Deswegen ist er seit C++11 als „deprecated“ gekennzeichnet.

UNIQUE_PTR

Ein unique_ptr verhält sich fast genauso wie ein auto_ptr, erlaubt aber gar nicht erst, dass Kopien von ihm angefertigt werden.

```
#include <memory>

void test()
{
    std::unique_ptr<int> p(new int());

    // auto q = p;
    // Fehler: Kopieren von p ist nicht erlaubt.
}
```

Das gilt auch für „indirekte“ Kopien, also wenn ein unique_ptr Teil einer Klasse ist.

```
#include <memory>

class Test
{
    std::unique_ptr<int> p;
};

void test()
{
    Test t;

    // auto t2 = t;
    // Fehler: Kopieren von t ist nicht erlaubt,
    // da t.p nicht kopiert werden darf.
}
```

SHARED_PTR

Der shared_ptr ist das Gegenstück zum unique_ptr. Er darf kopiert werden und stellt sicher, dass die Resource, auf die gezeigt wird, nicht sofort gelöscht wird, wenn die erste Kopie des shared_ptr gelöscht wird, sondern erst, wenn die letzte Kopie aus dem Scope läuft.

```
#include <memory>

void test()
{
    std::shared_ptr<int> p(new int(3));

    {
        // Erlaubt: p ist shared_ptr
        auto q = p;


        // q läuft aus dem Scope und wird gelöscht,
        // das Ziel von q bleibt aber noch bestehen,
        // da p noch darauf zeigt
    }

    // Scope von p endet, das Ziel von p und q
    // wird gelöscht.
}
```

Das ist auch der Grund, warum auto_ptr nicht mehr verwendet werden sollte,

```
#include <memory>

class Test
{
public:
    void demo() {};
};

int main()
{
    std::auto_ptr<Test> p(new Test());
    p->demo();

    // Erlaubt: p ist auto_ptr
    // Hier wird allerdings der Inhalt von p
    // nach q verschoben, so dass q anschließend
    // ungültig ist.
    auto q = p;
    // In Ordnung: q ist jetzt gültig.
    q->demo();

    // p->demo();
    // Fehler: q ist nicht mehr gültig,
    // p hat den Inhalt von q übernommen.
}
```

WEAK_PTR

Die letzte Art von smart pointern sind weak pointer. Diese verhalten sich sehr ähnlich wie shared pointer, halten aber das Objekt, auf das sie zeigen, nicht am Leben. Dadurch kann man auf eine Resource Bezug nehmen solange sie existiert, ohne ihre Lebensdauer zu beeinflussen.

```
#include <memory>

class Quest
{
};

class Player
{
public:
    std::shared_ptr<Quest> currentQuest;
};

class QuestUI
{
public:
    std::weak_ptr<Quest> displayedQuest;

    void display()
    {
        // Erstelle einen shared_ptr aus dem weak_ptr.
        // Wenn das Ziel von displayedQuest gelöscht wurde,
        // entsteht hierbei ein leerer shared_ptr.
        std::shared_ptr<Quest> quest = displayedQuest.lock();

        // Prüfe, ob quest leer ist.
        if (quest)
        {
            printf("displayedQuest ist noch gültig.\n");
            // Es gibt noch mindestens einen shared_ptr
            // der auf die Quest zeigt.
        }
        else
        {
            printf("Die Quest wurde gelöscht.\n");
            // Kein shared_ptr hat mehr auf die Quest gezeigt,
            // der letzte hat sie gelöscht,
            // denn der weak_ptr displayedQuest erhält sie nicht
            // am Leben.
        }
    }
};

int main()
{
    Player p1;
    Player p2;
    QuestUI questWindow;

    // erstelle neue Quest
    std::shared_ptr<Quest> newQuest(new Quest());

    // übergib die Quest an die beiden Spieler und das UI
    p1.currentQuest = newQuest;
```

```
    p2.currentQuest = newQuest;
    questWindow.displayedQuest = newQuest;

    // Lösch den shared_ptr, in dem die Quest erstellt wurde
    newQuest = nullptr;

    // Stellt die Quest noch dar, da beide Spieler sie
    // noch referenzieren
    questWindow.display();

    p1.currentQuest = nullptr;

    // Stellt die Quest noch dar, da der zweite Spieler sie
    // noch referenziert
    questWindow.display();

    p2.currentQuest = nullptr;

    // Stellt keine Quest mehr dar, weil beide Spieler sie
    // nicht mehr referenzieren
    questWindow.display();
}
```

Im letzten Beispiel wurde displayedQuest mithilfe der Funktion „lock()“ in einen shared_ptr verwandelt um zu prüfen, ob das Ziel des weak_ptr noch gültig ist.

Alternativ kann man auch displayedQuest.expired() verwenden. Dadurch erfährt man allerdings nur, ob das Ziel noch gültig ist, und erhält nicht gleich einen shared pointer auf das Ziel. Das ist deswegen ein Problem, weil es so passieren kann, dass man in einer Zeile nachfragt, ob der Zeiger noch gültig ist, und noch bevor man diesen angeblich gültigen Zeiger benutzen kann, wird das Ziel von einem anderen Thread gelöscht. Daher sollte man zumindest beim Multithreading vorzugsweise die lock()-Funktion verwenden.

RVALUE-REFERENZEN

Wenn man in C++ mit Daten arbeiten möchte, benennt man sie als erstes. In den allerersten Beispielen in diesem Dokument ging es gleich um int x oder int* p, in denen eine Ganzzahl oder ein Zeiger auf eine Ganzzahl gespeichert werden können. Über x und p kann man nun Befehle formulieren, die auf diesen Speicher Bezug nehmen. Diese Werte nennt man auch lvalues, denn eine besonders offensichtliche Eigenschaft ist, dass sie auf der linken Seite eines Gleichheitszeichens stehen können, dass man ihnen also selbst Werte zuweisen kann.

Das gilt jedoch nicht für alle Werte. Manche Datensätze erscheinen nicht namentlich im Code sondern entstehen beispielsweise als Folge einer Funktion. Diese Werte nennt man rvalues, denn sie können nur auf der rechten Seite eines Gleichheitszeichens stehen.

```
int test()
{
    return 3;
}

int main()
{
    int i; // lvalue
    i = 3; // i kann links vom Gleichheitszeichen stehen

    const int ci = 5;
    // ebenfalls ein lvalue, auch wenn es nach dieser Zeile nicht mehr
    // links vom Gleichheitszeichen stehen darf

    test();// rvalue
    i = test();    // test() kann nur rechts vom Gleichheitszeichen
                   // stehen.

    // test() = 5;
    // Fehler: test() erzeugt einen rvalue, kann also nicht
    // links vom Gleichheitszeichen stehen.
}
```

Das wirkt sich darauf aus, wie man auf Werte Referenzen erstellen kann, da eine Referenz wie ein lvalue verwendet werden kann. Würde mal also eine Referenz auf einen rvalue erstellen, könnte man diesen also wie einen lvalue verändern.

```
int test()
{
    return 3;
}

int main()
{
    int i = 3;
    int& r = i;

    //int& r2 = test();
    // Fehler: Referenz darf nur auf lvalues erstellt werden,
    // es sei denn der referenzierte Wert ist const:
    const int& r3 = test();
    // Durch const wird verspochen, dass der Wert, der von r3
    // referenziert wird, nicht verändert wird, also nie
    // als lvalue verwendet wird.
}
```

Daher werden diese Referenzen als lvalue-Referenzen bezeichnet, um sie von den neuen rvalue-Referenzen zu unterscheiden.

Das Problem mit rvalues ist, dass die Lebensdauer dieser Werte nicht vom Programmierer gesteuert ist sondern vom Compiler festgelegt wird. Sie sind in der Regel temporäre Werte, die nur bestehen, solange ein Wert von einem Ort zu einem anderen übergeben wird. Beispielsweise muss der Wert, der in dieser Zeile

```
int i = 3+4;
```

von 3+4 zurückgegeben wird, so lange im Speicher liegen, bis er in die Variable i kopiert werden konnte. Danach kann er jedoch aus dem Speicher entfernt werden. Um Kontrolle darüber zu haben, wie lange der Wert im Speicher bleibt, wurde er in die Stackvariable i kopiert.

Nun ist das Problem, dass nicht jeder Wert so billig kopiert werden kann wie eine Ganzzahl. Wenn eine Klasse beispielsweise einen großen Speicherblock auf dem Heap referenziert, muss dieser unter Umständen in solchen Situationen mit kopiert werden.

```
#define SIZE 1024*1024*100

class HugeThing
{
    char* data;

public:
    // Im Konstruktor werden 100MB belegt.
    HugeThing()
    : data(new char[SIZE])
    {}

    // Im Copy Constructor werden neue 100MB angefordert
    HugeThing(const HugeThing& p_other)
    : data(new char[SIZE])
    {
        // und dann die Daten aus dem Original in die Kopie kopiert.
        memcpy(data, p_other.data, SIZE);
    }
};

HugeThing test()
{
    HugeThing retVal;
    return retVal;
}

int main()
{
    // Hier wird der temporäre Wert retVal aus der test()-Funktion
    // in die lokale Variable thing kopiert.
    HugeThing thing = test();
}
```

Es werden also 100MB an Speicher angelegt wenn retVal erstellt wird und dann werden erneut 100MB angefordert, damit der Rückgabewert von test() in thing kopiert werden kann.

Stattdessen hätte man aber auch einfach den Heapspeicher, den der Rückgabewert verwendet hat, an thing übergeben können, denn der Rückgabewert ist ja ohnehin nur temporär. Er wurde nur erstellt, um kopiert und dann wieder gelöscht zu werden.

Das Verhalten der Klasse so zu ändern, dass das Problem nicht auftritt, ist einfach: anstatt im Copy Constructor den Datenblock zu kopieren muss er nur von einem Objekt an das andere übergeben werden.

```
    // Diesmal wird der Zeiger data einfach kopiert
    HugeThing(HugeThing& p_other)
        : data(p_other.data)
    {
        // und dann im Original auf null gesetzt.
        p_other.data = nullptr;
    }
```

Das hat allerdings zum Ergebnis, dass jetzt nie mehr eine richtige Kopie erstellt wird, auch dann nicht, wenn wir es ausdrücklich verlangen:

```
int main()
{
    HugeThing thing1;
    HugeThing thing2 = thing1;
    // Durch den geänderten Copy Constructor ist thing2 jetzt
    // keine Kopie von thing1 mehr, sondern thing1 musste seine
    // Daten an thing2 übergeben und ist jetzt nicht mehr gültig.
}
```

Um das Problem zu lösen müssen wir also unterschiedlichen Code angeben, je nachdem ob wir einen lvalue kopieren sollen (der nach dem Kopiervorgang weiterexistiert) oder einen rvalue (der nur temporär ist und nach dem Kopiervorgang gelöscht wird).

Genau das wurde in C++11 neu eingebaut. Die bisherigen Referenzen heißen jetzt lvalue-Referenzen, funktionieren aber wie gehabt. Neu hinzugekommen sind rvalue-Referenzen, die mit dem Zeichen && gekennzeichnet werden.

```
#define SIZE 1024*1024*100

class HugeThing
{
    char* data;

public:
    // Im Konstruktor werden 100MB belegt.
    HugeThing()
    : data(new char[SIZE])
    {}

    // Im Copy Constructor werden neue 100MB angefordert
    HugeThing(const HugeThing& p_other)
        : data(new char[SIZE])
    {
        // und dann die Daten aus dem Original in die Kopie kopiert.
        memcpy(data, p_other.data, SIZE);
    }

    // Im Move Constructor wird der Zeiger auf data einfach kopiert
    HugeThing(HugeThing&& p_other)
        : data(p_other.data)
    {
        // und dann im Original auf null gesetzt.
        p_other.data = nullptr;
    }
};

HugeThing createHugeThing()
{
    HugeThing thing;
}

int main()
{
    // Diese Zeile benutzt jetzt nicht mehr den copy constructor
    // sondern den move constructor, da der Rückgabewert von
    // createHugeThing() ein rvalue ist und die Klasse HugeThing
    // einen move constructor hat.
    HugeThing thing = createHugeThing();

    // Diese Zeile benutzt den copy constructor, da thing nach der
    // Zeile weiterexistiert und gültig bleiben soll.
    HugeThing copyOfThing = thing;
}
```

STD::MOVE() UND STD::FORWARD<>()

In der Regel geht es bei dieser „move semantics“ genannten Technik nur darum, rvalues und lvalues automatisch zu erkennen und unterschiedlich zu behandeln. Es gibt aber auch Situationen, in denen es gewünscht ist, einen lvalue als temporären Wert zu behandeln, oder in denen die automatische Unterscheidung zwischen r- und lvalues nicht zum gewünschten Ergebnis kommt. In diesen Fällen kann man mit std::move() einen Wert von einem lvalue in einen rvalue verwandeln.

```
#include <string>
#include <utility>

class Player
{
    // enthält eine Resource, die im Copy Constructor
    // kopiert und im Move Constructor verschoben wird.
};

Player loadPlayerTemplate(const std::string& p_filename)
{
    // lädt die Startwerte für Spieler
}

int main()
{
    Player prototype = loadPlayerTemplate("player.dat");

    // Jetzt sollen genau drei Spieler erzeugt werden.
    // Die ersten beiden werden kopiert.
    Player warrior = prototype;
    Player archer = prototype;

    // Der dritte wird mit dem move constructor erzeugt,
    // weil das Template anschließend ohnehin nicht mehr
    // verwendet werden soll.
    Player mage = std::move(prototype);
}
```

Es ist dabei wichtig zu verstehen, dass nach einem solchen Verschiebungsvorgang das Objekt, auf das std::move() angewendet wurde, (in der Regel) ungültig ist. Im folgenden Beispiel sieht man außerdem den move-operator=(), der dann angewendet wird, wenn ein Objekt bereits erstellt ist und ihm nur per = ein neuer Wert zugewiesen wird.

```
#include <string>

class Player
{
public:
    std::string name;

    Player(const std::string& p_name)
        : name(p_name)
    {}

    Player(Player&& p_other)
        : name(p_other.name)
    {
        p_other.name = "";
    }

    void operator=(Player&& p_other)
    {
        name = p_other.name;
        p_other.name = "";
    }
};

#include <utility>

int main()
{
    Player p1("Highlander");

    // Diese Zeile ruft den move constructor auf.
    Player p2 = std::move(p1);
    printf("p1 is called '%s', p2 is called '%s'.\n",
               p1.name.c_str(), p2.name.c_str());
    // p2 hat jetzt einen Namen, p1 nicht mehr.

    // Diese Zeile ruft den move operator=() auf.
    p1 = std::move(p2);
    printf("p1 is called '%s', p2 is called '%s'.\n",
               p1.name.c_str(), p2.name.c_str());
    // p1 hat jetzt wieder einen Namen, p2 nicht mehr.
}
```

RVALUE REFERENCES

Im eben beschriebenen Fall ist offensichtlich, dass das Objekt, dessen Daten verschoben werden, ein lvalue ist und es eine willkürliche Entscheidung ist, es als rvalue zu behandeln. Es kommt aber auch vor, dass ein Wert, den man für einen rvalue halten würde, tatsächlich ein lvalue ist.

Dazu wiederhole ich etwas Code ganz vom Anfang dieses Dokuments:

```
Player* p = new Player();
```

In dieser Zeile wird durch new Player() Speicher im Heap erstellt. Damit man später noch auf diesen Speicher zugreifen kann, wird seine Adresse auf dem Stack gespeichert. Die Frage, ob „p“ jetzt auf dem Stack oder im Heap liegt ist, ist also doppeldeutig. „p“ ist eigentlich ein Zeiger, und dieser liegt auf dem Stack. Er zeigt aber auf einen Player, und dieser liegt im Heap.
Ganz ähnlich sieht es aus, wenn wir eine rvalue-Referenz an eine Funktion übergeben.

Im folgenden Code wird ein Player von der Funktion createPlayer() erzeugt. Dieser wird nicht erst in eine Variable, also einen lvalue, kopiert, sondern direkt als Parameter in die Funktion duplicatePlayer() übergeben. Diese ruft dann den Konstruktor von Player mit dem übergebenen Parameter auf („return Player(p_original);“ in duplicatePlayer()).

Da der Parameter p_original vom Typ Player&& ist, würden wir erwarten, dass der Konstruktor verwendet wird, der ebenfalls einen Parameter vom Typ Player&& akzeptiert, also der move constructor. Es wird aber tatsächlich der copy constructor aufgerufen.

```
class Player
{
public:
    Player() = default;

    Player(const Player& p_other)
    {
        printf("copy c-tor\n");
    }

    Player(Player&& p_other)
    {
        printf("move c-tor\n");
    }
};

Player duplicatePlayer(Player&& p_original)
{
    return Player(p_original);
}

Player createPlayer()
{
    return Player();
}

int main()
{
    Player p = duplicatePlayer(createPlayer());
}
```

Der Grund dafür ist, dass der Wert p_original in duplicatePlayer() eine Kopie des übergebenen rvalue ist. p_original ist also selbst ein lvalue, auch wenn er eine Referenz auf einen rvalue darstellt. Daher der Vergleich mit Stack- und Heapvariablen: ein Zeiger ist eine Stackvariable, die auf Heapspeicher zeigt. Genau so ist ein Parameter vom Typ T&& ein lvalue, der einen rvalue referenziert.

Daher wird in diesem Fall der Konstruktor aufgerufen, der für lvalues vorgesehen ist. Dieses Verhalten macht auch Sinn, wenn man Bedenkt, dass eine Variable ungültig wird, wenn ihr Inhalt verschoben wird. Weil Parameter grundsätzlich lvalues sind, können wir diesen Code schreiben:

```
Player duplicatePlayer(Player&& p_original)
{
    Player p1(p_original);
    Player p2(p_original);
    return Player(p_original);
}
```

Wäre p_original ein rvalue, dann würde er bereits in der ersten Zeile verschoben und wäre in der zweiten und dritten Zeile ungültig.
Außerdem würde gegen die Faustregel verstoßen, dass Variablen, die einen Namen haben, immer lvalues sind.

UNIVERSAL REFERENCES UND STD::FORWARD<>()

Damit der Code also das macht, was er machen soll (nämlich den rvalue an den move constructor weiterreichen), muss er mit std::move() wieder in einen rvalue verwandelt werden.

```
Player duplicatePlayer(Player&& p_original)
{
    return Player(std::move(p_original));
}
```

Dieser Code bietet eine duplicatePlayer()-Funktion an, die mit rvalues funktioniert. Wenn aber die Vorlage vorher in einer Variable gespeichert wird, kann die Funktion nicht mehr aufgerufen werden, da sie nur mit rvalues funktioniert. Also müsste man eine zweite Funktion für lvalues hinzufügen.

```
#include <utility>

// eine Version für rvalues
Player duplicatePlayer(Player&& p_original)
{
    // p_original wird in rvalue verwandelt
    return Player(std::move(p_original));
}

// eine Version für lvalues
Player duplicatePlayer(const Player& p_original)
{
    // p_original bleibt lvalue
    return Player(p_original);
}
```

Das funktioniert zwar, ist aber umständlich und führt zu Code Duplication, denn bis auf die Anwendung von std::move() hat sich zwischen den beiden Funktionen nichts geändert.

Um diese Code Duplication zu vermeiden kann sogenannte universal references verwenden, also Referenzen, die sowohl auf rvalues als auch auf lvalues angewendet werden können. Dafür gibt es kein neues

Sonderzeichen, was universal references von rvalue- oder lvalue-Referenzen unterscheiden würde, sondern nur eine Ausnahmeregelung:

Wird das rvalue-Zeichen && zusammen mit Type Inference verwendet, ist es eine universelle Referenz, die eine lvalue- oder eine rvalue-Referenz sein kann.

```
template<class T>
T duplicatePlayer(T&& p_original)
{
    return T(p_original);
    // Noch nicht optimal:
    // ohne std::move wird der Copy constructor aufgerufen.
}
```

In diesem Beispiel ist T&& eine universelle Referenz, da T per type inference automatisch erkannt wird. Somit braucht nur noch eine Funktion geschrieben zu werden, allerdings unterscheidet sich der Code für rvalues von dem für lvalues durch die Verwendung von std::move(). Um das Problem zu lösen gibt es die Funktion std::forward<>(), die selbst erkennt, ob der Wert, auf den sie angewendet wird, ein lvalue ist, der einen rvalue beinhaltet oder einer, der einen lvalue beinhaltet. Mit anderen Worten std::forward<>() wird zu std::move() wenn es auf rvalue-Referenz-Parameter angewendet wird und macht nichts, wenn es auf lvalue-Referenz-Parameter angewendet wird.

```
#include <utility>

template<class T>
T duplicatePlayer(T&& p_original)
{
    return T(std::forward<T>(p_original));
    // Jetzt wird automatisch ein rvalue oder ein lvalue erzeugt,
    // je nachdem was die universelle Referenz
    // p_original referenziert.
}

int main()
{
    Player p = duplicatePlayer(createPlayer());
    // Ausgabe: move c-tor

    Player p2 = duplicatePlayer(p);
    // Ausgabe: copy c-tor
}
```

Universelle Referenzen lassen sich mit beiden type-inference-Mechanismen erstellen, also nicht nur mit Templates sondern auch mit auto.

```
int main()
{
    auto duplicatePlayer = [](auto&& p_original) {
        //return Player(std::forward<??>(p_original));
    };
}
```

Der Parameter „auto&& p_original“ in der Lambda-Funktion ist jetzt ebenfalls eine universelle Referenz, allerdings fehlt der Typ T, der in der std::forward<>-Funktion angegeben werden muss. Um den Code zu korrigieren, wird das Keyword decltype benötigt, was im Folgenden erklärt wird.

DECLTYPE

Das Keyword “decltype” lässt sich gut am vorhergehenden Codebeispiel erklären.

Da die std::forward<>()-Funktion eine Template-Funktion ist, benötigt sie eine Angabe, welchen Datentyp sie forwarden soll. Wenn ein Template verwendet wird, ist das kein Problem, denn dort kann man den Datentyp mit einem Platzhalter benennen (in den Beispielen war das immer T) und ihn so beim forward-Template angeben (also „std::forward<T>()“).

Wird hingegen auto&& verwendet, wurde der automatisch erkannte Typ nicht mit einem Platzhalter benannt. Also braucht man eine Möglichkeit zu sagen: „der gleiche Datentyp wie …“, und das kann man mit dem keyword „decltype“ erreichen.

```
    Child c;
    decltype(c) x;
    // x ist jetzt auch ein Child,
    // da decltype(c) das gleiche ist wie Child

    Base* b = new Child();
    decltype(b) y;
    // y ist jetzt auch ein Base*,
    // da decltype(b) das gleiche ist wie Base*
```

Mit dem etwas umständlichen Wort „decltype“ soll ausgedrückt werden, dass es sich um den deklarierten Typ handelt, nicht den Laufzeittyp. Im vorigen Beispiel ist y ein Zeiger auf Base, da es den gleichen Typ hat, mit dem b deklariert wurde. Dass b wenn das Programm läuft und an dieser Zeile ankommt auf ein Objekt vom Typ Child zeigt, ist für decltype irrelevant.

Das Codebeispiel für universal references mit auto lässt sich also so vervollständigen:

```
#include <utility>

int main()
{
    auto duplicatePlayer = [](auto&& p_original) {
        return Player(std::forward<decltype(p_original)>(p_original));
    };

    Player p = duplicatePlayer(createPlayer());
    Player p2 = duplicatePlayer(p);
}
```