Klassen in C++ verhalten sich völlig anders als solche in Java und vielen anderen objekt-orientierten Programmiersprachen:

- Wir haben keine implizite Zeigersemantik.
- Objekte einer Klasse können (wenn nichts anderes bestimmt ist) kopiert und als Wert einer Funktion oder Methode übergeben werden (*call by value*).
- Objekte können nicht nur auf dem Heap leben.
- Objekte werden immer in wohldefinierter Weise abgebaut.

Eine triviale Klasse

vector2d.cpp

```
#include <iostream>

class Vector2D {
   public:
      double x, y;
};

void print_point(Vector2D v) {
   std::cout << "(" << v.x << ", " << v.y << ")";
}

Vector2D add_vectors(Vector2D v1, Vector2D v2) {
   Vector2D result;
   result.x = v1.x + v2.y; result.y = v1.y + v2.y;
   return result;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2; Vector2D b; b.x = 10; b.y = 20;
   Vector2D c = add_vectors(a, b);
   print_point(c); std::cout << std::endl;
}
```

Parameterübergabe mit *call by value*

vector2d.cpp

```
Vector2D add_vectors(Vector2D v1, Vector2D v2) {
   Vector2D result;
   result.x = v1.x + v2.y; result.y = v1.y + v2.y;
   return result;
}
```

- Wenn ein Objekt per Parameter übergeben wird, dann wird per Voreinstellung jede einzelne Variablenkomponente (hier x und y) kopiert. Die Parameter *v1* und *v2* leben dann lokal auf dem Stack.
- Objekte können auch zurückgegeben werden. In diesem Fall wird *result* komponentenweise bei der Rückgabe in c kopiert.

vector2d-scale.cpp

```
void scale_vector(Vector2D v, double factor) {
   v.x *= factor; v.y *= factor;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2;
   scale_vector(a, 10);
   print_point(a); std::cout << std::endl;
}
```

- Diese Variante von *scale_vector* ist sinnlos, da nur die lokale Kopie *v* verändert wird, jedoch nicht der Vektor *a* in *main*.

Es gibt hierzu zwei Möglichkeiten:

- Unter expliziter Verwendung von Zeigern (so wie es auch in C üblich war).
- Unter Verwendung von Referenzen.

Dann lässt sich das auch unnötiges Kopieren vermeiden, das bei größeren Objekten (man denke an sehr große Matrizen) recht teuer werden kann.

Parameterübergabe mit einem Zeiger

vector2d-scale2.cpp

```
void scale_vector(Vector2D* vp, double factor) {
   vp->x *= factor; vp->y *= factor;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2;
   scale_vector(&a, 10);
   print_point(a); std::cout << std::endl;
}
```

- Der Adress-Operator „&“ liefert die Adresse eines Objekts – hier mit dem Datentyp *Vector2D∗*.
- Die Funktion *scale_vector* erwartet hier einen Zeiger und ist entsprechend gezwungen, diesen immer zu dereferenzieren.

Parameterübergabe mit einer Referenz

vector2d-scale3.cpp

```
void scale_vector(Vector2D& vp, double factor) {
   vp.x *= factor; vp.y *= factor;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2;
   scale_vector(a, 10);
   print_point(a); std::cout << std::endl;
}
```

- Referenzen sind implizite Zeiger.
- Entsprechend fällt das explizite Dereferenzieren und die Verwendung des Adress-Operators bei der Übergabe weg.
- Die Umsetzung ist äquivalent zur vorherigen Variante mit expliziten Zeigern.

vector2d-scale3.cpp

```
void print_point(const Vector2D& v) {
   std::cout << "(" << v.x << ", " << v.y << ")";
}
```

- Die Verwendung von Referenzen ist auch dann sinnvoll, wenn das Objekt nicht zu verändern ist, da dann der Kopieraufwand entfällt.
- In diesem Fall ist es sinnvoll, **const** hinzuzufügen. Das sichert zu, dass die so per Referenz übergebene Variable nicht verändert wird.

Eine einfache Klasse

counter.hpp

```
class Counter {
   public:
      // constructors
      Counter() : counter{0} {
      }
      Counter(int counter) : counter{counter} {
      }
      // accessors
      int get() const {
         return counter;
      }
      // mutators
      int increment() {
         assert(counter < INT_MAX);
         return ++counter;
      }
      int decrement() {
         assert(counter > INT_MIN);
         return --counter;
      }
   private:
      int counter;
};
```

counter.hpp

```
private:
   int counter;
```

- Datenfelder sollten normalerweise privat gehalten werden, um den direkten Zugriff darauf zu verhindern. Stattdessen ist es üblich, entsprechende Zugriffsmethoden (*accessors*, *mutators*) zu definieren.

counter.hpp

```
  Counter() : counter{0} {
  }
  Counter(int counter) : counter{counter} {
  }
```

- Eine Klasse kann beliebig viele Konstruktoren anbieten, solange sie sich in ihrer Signatur unterscheiden.
- Wenn kein Konstruktor angegeben ist, generiert der Übersetzer automatisch einen parameterlosen Konstruktor, der, sofern möglich, sämtliche Datenfelder analog ohne Parameter konstruiert.
- Es ist zulässig, die Parameter der Konstruktoren genauso zu nennen wie die entsprechenden Objektvariablen.

⟨ctor-initializer⟩ ⟶ „:“ ⟨mem-initializer-list⟩

⟨mem-initializer-list⟩ ⟶ ⟨mem-initializer⟩ [„...“]

⟶ ⟨mem-initializer-list⟩ „,“ ⟨mem-initializer⟩ [„...“]

⟨mem-initializer⟩ ⟶ ⟨mem-initializer-id⟩ „(“ [⟨expression-list⟩] „)“

⟶ ⟨mem-initializer-id⟩ ⟨braced-init-list⟩

⟨mem-initializer-id⟩ ⟶ ⟨class-or-decltype⟩

⟶ ⟨identifier⟩

⟨braced-init-list⟩ ⟶ „{“ ⟨initializer-list⟩ [„,“] „}“

⟶ „{“ „}“

- Seit C++11 wird die {...}-Notation (⟨braced-init-list⟩) bevorzugt, da sie einen unschönen grammatikalischen Konflikt vermeidet und mehr Möglichkeiten erlaubt.

- Es ist in C++ sehr wichtig, zwischen einer Initialisierung mit Hilfe eines ⟨ctor-initializer⟩ und einer regulären Zuweisung innerhalb des ⟨compound-statement⟩ des Konstruktors zu unterscheiden.
- Bevor das ⟨compound-statement⟩ des Konstruktors ausgeführt wird, müssen alle Teilobjekte konstruiert sein. Wenn die Initialisierung innerhalb des ⟨ctor-initializer⟩ fehlt, dann wird jeweils der parameterlose Konstruktor verwendet.
- Wenn der parameterlose Konstruktor für eines der Teilobjekte nicht zur Verfügung stehen sollte, dann geht es überhaupt nicht ohne die Konstruktion innerhalb des ⟨ctor-initializer⟩.
- Eine implizite automatische Konstruktion in Kombination mit einer späteren Zuweisung kann zu ineffizienteren Code führen. Daher wird grundsätzlich empfohlen, soweit wie möglich alle Teilobjekte innerhalb des ⟨ctor-initializer⟩ zu initialisieren.
- Die Reihenfolge im ⟨ctor-initializer⟩ sollte der der Deklaration der Teilobjekte entsprechen. In letzterer Reihenfolge werden sie ausgeführt.

counter.hpp

```
  Counter() : counter{0} {
  }
  Counter(int counter) : counter{counter} {
  }
```

- Es ist zulässig, die Parameter der Konstruktoren genauso zu nennen wie die entsprechenden Objektvariablen.
- Bei der ⟨mem-initializer-id⟩ wird nur unter den zu initialisierenden Namen der Objektvariablen gesucht bzw. dem Namen der eigenen Klasse oder den Namen der Basisklassen.
- In der ⟨expression-list⟩ bzw. der ⟨initializer-list⟩ werden zuerst die Parameternamen durchsucht, bevor die Namen der Objektvariablen in Betracht gezogen werden.

intinit.cpp

```
int main() {
   cout << "Testing..." << endl;
   int i; // undefined
   cout << "i = " << i << endl;
   int j{17}; // well defined
   cout << "j = " << j << endl;
   int k{}; // well defined: 0
   cout << "k = " << k << endl;
}
```

- Die elementare Datentypen bieten ebenfalls parameterlose Konstruktoren an und einen Konstruktor mit einem Parameter des entsprechenden Typs.
- Wenn jedoch kein Konstruktor explizit aufgerufen wird, erfolgt keine Initialisierung.

```
clonmel$ intinit
Testing...
i = 4927
j = 17
k = 0
clonmel$
```

counter.hpp

```
int get() const {
   return counter;
}
```

- Zugriffsmethoden, die den abstrakten Zustand des Objekts nicht verändern dürfen, werden mit **const** ausgezeichnet.
- Nur diese Methoden dürfen aufgerufen werden, wenn das Objekt in einem Kontext nur lesenderweise zur Verfügung steht.
- Mit **mutable** deklarierte Variablen dürfen auch von **const**-Methoden verändert werden. Dies ist sinnvoll etwa zur Vermeidung sich sonst wiederholender Berechnungen und sollte nicht zu außen sichtbaren Veränderungen führen. (Abstrakter vs. konkreter Zustand eines Objekts.)

```
Counter(const Counter& orig) : counter{orig.counter} {
}
```

- Wenn einer der Konstruktoren genau einen Parameter mit einem Referenztyp der eigenen Klasse hat, dann handelt es sich dabei um einen expliziten Kopierkonstruktor.
- Normalerweise wird dieser mit **const** versehen.
- Der Kopierkonstruktor wird dann ggf. implizit verwendet bei der Parameterübergabe, bei **return** und einer Zuweisung.
- Wenn der Kopierkonstruktor nicht explizit deklariert und nicht ausdrücklich unterbunden wird, dann erzeugt der Übersetzer einen, wobei jedes Teilobjekt entsprechend kopierkonstruiert wird. Das funktioniert nur, wenn dies für alle Teilobjekte geht.

- Klassen, die selbst Ressourcen verwalten wie etwa dynamisch angelegte Speicherbereiche, benötigen sehr viel Sorgfalt in C++, damit mit der impliziten Verwendung von Konstruktoren und Methoden keine Probleme entstehen.
- Es ist dabei insbesondere sicherzustellen, dass dynamische Datenstrukturen nur ein einziges Mal freigegeben werden. D.h. das unbemerkte Kopieren von Zeigern ist ein Problem.
- Das folgende Beispiel illustriert dies an einer sehr einfachen Klasse, die ein **int**-Array verwaltet.

Klassen mit Zeigern

array.hpp

```
class Array {
   public:
      Array() : nof_elements(0), ip(nullptr) {}
      Array(unsigned int nof_elements) : nof_elements{nof_elements},
            ip{new int[nof_elements] {}} {
      }
      Array(const Array& other) : nof_elements{other.nof_elements},
            ip{new int[nof_elements]} {
         for (unsigned int i = 0; i < nof_elements; ++i) {
            ip[i] = other.ip[i];
         }
      }
      Array(Array&& other) : nof_elements{other.nof_elements},
            ip{other.ip} {
         other.nof_elements = 0; other.ip = nullptr;
      }
      ~Array() { delete[] ip; }
      Array& operator=(const Array& other) = delete;
      Array& operator=(Array&& other) = delete;
      unsigned int size() const { return nof_elements; }
      int& operator()(unsigned int i) {
         assert(i < nof_elements); return ip[i];
      }
      const int& operator()(unsigned int i) const {
         assert(i < nof_elements); return ip[i];
      }
   private:
      unsigned int nof_elements; int* ip;
};
```

array.hpp

```
Array(unsigned int nof_elements) : nof_elements{nof_elements},
      ip{new int[nof_elements] {}} {
}
```

- Mit dem **new**-Operator kann Speicher dynamisch belegt werden. Neben einem Typnamen kann auch in Array-Notation eine Dimensionierung mit angegeben werden.
- Hier wird ein **int**-Array mit *nof_elements* Elementen angelegt.
- Der **new**-Operator lässt hier auch sogleich die Initialisierung der neuen Speicherfläche hinzu. Da {} hier angegeben ist, wird das Array mit Nullen initialisiert. (Ohne diesen expliziten Hinweis würden die **int** uninitialisiert bleiben.)

array.hpp

```
~Array() {
   delete[] ip;
}
```

- Mit dem Operator **delete**[] kann ein Array wieder freigegeben werden.
- Wenn der Zeiger ein **nullptr** sein sollte, stört das nicht.
- Anders als in Java wird diese Funktion garantiert aufgerufen, wenn die Lebenszeit des Objekts beendet ist.

Der Begriff *Resource Acquisition Is Initialization* (RAII) geht auf Bjarne Stroustrup und Andrew Koenig zurück. Folgende Prinzipien sind damit verknüpft:

- Ressourcen werden mit Objekten fest verknüpft.
- Bei der Initialisierung des Objekts wird die Ressource akquiriert.
- Beim Dekonstruieren des Objekts wird die Ressource freigegeben.

Diese Technik vermeidet Fehler und stellt insbesondere sicher, dass auch im Falle einer Ausnahmenbehandlung (*exception handling*) alles sauber abgebaut und damit freigegeben wird.

Da bei C++ Objekte kopiert und nicht ohne weiteres nur Zeiger einander zugewiesen werden, ist bei Klassen, die mit Ressourcen in Verbindung stehen, Vorsicht geboten:

Wenn immer eine Klasse eine Ressource verwaltet, sind folgende spezielle Methoden immer explizit zu definieren bzw. zu deaktivieren, um die unerwünschte implizite Definition zu unterbinden:

- Kopier-Konstruktor
- Zuweisungs-Operator
- Dekonstruktor

array.hpp

```
Array(const Array& other) : nof_elements{other.nof_elements},
      ip{new int[nof_elements]} {
   for (unsigned int i = 0; i < nof_elements; ++i) {
      ip[i] = other.ip[i];
   }
}
```

- Der Kopierkonstruktor hat die Aufgabe, die gesamte Datenstruktur zu duplizieren.
- Hierfür ist das Kopieren nur des Zeigers unzureichend. Denn dann würde die Klon-Semantik verlorengehen und wir hätten das Problem, dass das Array beim Abbau mehrfach freigegeben würde. Die vom Übersetzer zur Verfügung stehende voreingestellte Implementierung dieses Konstruktors wäre somit fatal.
- Hier wird wiederum dynamisch Speicher von der gegebenen Größe angelegt und dann mit Hilfe der **for**-Schleife die Elemente einzeln kopiert. (Wie das effizienter geht, kommt später.)

Verschiebekonstruktor für das Array

array.hpp

```
Array(Array&& other) : nof_elements{other.nof_elements},
      ip{other.ip} {
   other.nof_elements = 0;
   other.ip = nullptr;
}
```

- Der Verschiebekonstruktor (*move constructor*) wird dann verwendet, wenn das Quellobjekt unmittelbar nach dem Aufruf abgebaut wird. Das ist insbesondere bei temporären Objekten der Fall.
- In diesem Fall können wir die dynamische Datenstruktur einfach übernehmen und müssen dann nur sicherstellen, dass beim Abbau des Quellobjekts keine versehentliche Freigabe der Datenstruktur erfolgt.

array.hpp

```
Array& operator=(const Array& other) = delete;
Array& operator=(Array&& other) = delete;
```

- Der Übersetzer unterstützt auch implizit Zuweisungen. Bei Objekten mit Zeigern ist hier ebenfalls die voreingestellte Implementierung unzureichend.
- Hier wird gezeigt, wie die voreingestellte Implementierung mit Hilfe von = **delete** unterdrückt werden kann, ohne sie durch eine eigene Implementierung zu ersetzen.

Zugriff auf ein Array-Element

array.hpp

```
int& operator()(unsigned int i) {
   assert(i < nof_elements); return ip[i];
}
const int& operator()(unsigned int i) const {
   assert(i < nof_elements); return ip[i];
}
```

- Statt traditioneller *set*- und *get*-Methoden kann auch der direkte Zugriff auf ein ansonsten privates Element gegeben werden, indem eine Referenz zurückgegeben wird.
- Das Resultat kann dann sowohl als *lvalue* (links von einer Zuweisung) als auch als *rvalue* (rechts der Zuweisung) verwendet werden.
- Methoden können auch nach einem Operator benannt werden mit Hilfe des Schlüsselworts **operator**. (Hier ist es der Funktionsoperator ().)

```
Array a{10};
a(1) = 77;
a(2) = a(1) + 10;
```

array.hpp

```
friend void swap(Array& a1, Array& a2) {
   std::swap(a1.nof_elements, a2.nof_elements);
   std::swap(a1.ip, a2.ip);
}
Array(const Array& other) :
      nof_elements{other.nof_elements},
      ip{new int[nof_elements]} {
   for (unsigned int i = 0; i < nof_elements; ++i) {
      ip[i] = other.ip[i];
   }
}
Array(Array&& other) : Array() {
   swap(*this, other);
}
Array& operator=(Array other) {
   swap(*this, other);
   return *this;
}
```

- Wenn die Zuweisung unterstützt werden soll, dann dient ein *swap*-Operator der Vereinfachung.