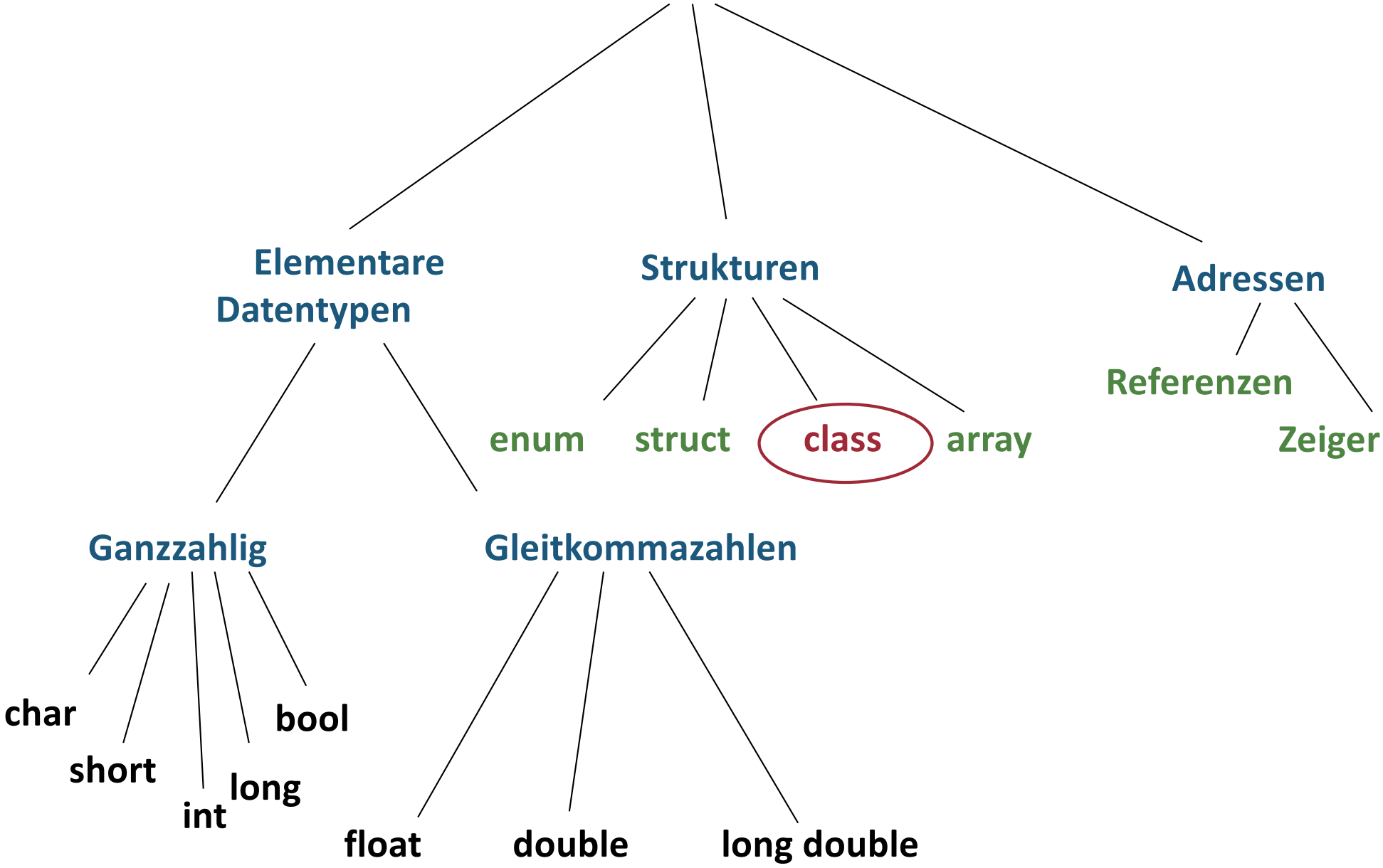
K05

Klassen & Methoden in C++

1. **Definition**
2. Konstruktoren und Destruktor
3. Statische Klassenelemente
4. Überladen von Operatoren

Wiederholung: Das Typsystem in C++



Definition von Klassen (i)

Definitionsschema:

```
class Demo {
  private: // optional
     // hier private Attribute/Methoden
  protected:
     // hier geschützte Attribute/Methoden
  public:
     // hier öffentliche Attribute/Methoden
};
```

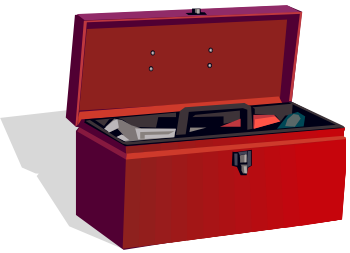
- Die Attribute und Methoden – welche allgemein auch Member genannt werden – einer Klasse gehören in eine der drei Sichtbarkeitskategorien
- Die Labels **`private: protected: public:`** können beliebig oft in beliebiger Reihenfolge verwendet werden.

Definition von Klassen (ii)

- `private`: Member von aussen nicht zugänglich; kann nur innerhalb der Klasse verwendet werden
- `protected`: wie `private`, jedoch kann Member innerhalb abgeleiteter Klassen verwendet werden (die Sichtbarkeit ist auf die Klassenhierarchie beschränkt)
- `public`: Member ist öffentlich verfügbar

- Die Voreinstellung bei Klassen (**`class`**) ist **`private`**
- Die Voreinstellung bei Strukturen (**`struct`**) ist **`public`**

Memberzugriff



- Im wesentlichen nichts neues
  - Vorausgesetzt Regeln der Sichtbarkeit sind erfüllt
- Zugriff auf Member erfolgt analog zu **`struct`**

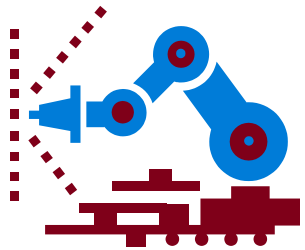
```
class Student { /* … */ };

Student s;
Student* sp = &s;
Student& sr = s;

cout << s.name   << "," << s.age()   << endl;
cout << sp->name << "," << sp->age() << endl;
cout << sr.name  << "," << sr.age()  << endl;
```

K05

Klassen & Methoden in C++

1. Definition
2. **Konstruktoren und Destruktor**
3. Statische Klassenelemente
4. Überladen von Operatoren

Konstruktoren & Destruktor



- Was passiert genau bei der Instanziierung eines Objekts?
  1. Es wird Speicher reserviert.
  2. Die Instanzvariablen werden initialisiert.

- Dazu gibt es spezielle Methoden: Konstruktoren („Ctor”)
  - Der Konstruktor wird als erste Methode aufgerufen, sobald Speicher allokiert ist
  - Innerhalb eines Konstruktors ist beliebiger Code möglich
  - Konstruktoren werden dazu verwendet, das Objekt in einen definierten Zustand zu bringen (Instanzvariablen initialisieren)

- Analog dazu führt der Destruktor („Dtor“) entsprechende „Aufräumarbeiten“ durch um Ressourcen (z.B. Speicher) wieder freizugeben.

Konstruktoren & Destruktor

- Jede C++ Klasse sollte ein Default-Konstruktor, ein Kopierkonstruktur, ein Zuweisungsoperator und ein Destruktor definieren sein.

Beispiel: Deklaration von Konstruktoren und Destruktor

```
class IntArray {
private:
   int *data, len;
public:
   IntArray();                                  // Default-Konstruktor
   IntArray(int);                               // ctor mit Arraygrösse
   IntArray(const IntArray&);                   // Kopierkonstruktor
   IntArray& operator=(const IntArray&); // Zuweisungsoperator
   ~IntArray();                                 // Destruktor
};
```

derselbe Name

Konstruktoren sind überladen (unterschiedliche Parameter)

Konstruktoren besitzen keinen explizit angegebenen Rückgabetyp.

Der Default-Konstruktor

```
MyClass()
```

- Ist eine Methode, die den Namen der Klasse trägt, ohne Parameter.
- Rückgabewert ist neue Instanz der Klasse.
- Wird aufgerufen, wenn nichts anderes bei der Instanziierung gesagt wird, d.h. wenn keine Parameter angegeben werden (damit werden andere Konstruktoren aufgerufen).

- Falls vom Programmierer nicht angegeben, dann generiert der Compiler automatisch einen Default-Konstruktor, welcher:
  - Default-Konstruktor der Instanzvariablen aufruft,
  - Ansonsten einen Leeren Rumpf hat.

Der Kopierkonstruktur

- Intention: Erzeuge neue Instanz aus einer bereits existierenden Instanz
  - Erzeugung nicht notwendigerweise identischer Kopien (Clone)
- Definition:

```
IntArray(const IntArray& src) { // copy ctor
    . . .
}
```

- Der Kopierkonstruktor wird oft aufgerufen, ohne dass man es "sieht", z.B.:
  - Bei call-by-value Parameterübergabe

```
func(ia2);   // copy ctor called
```

  - Bei Variablendefinition durch Zuweisung

```
IntArray ia3 = ia2;   // copy ctor called
```

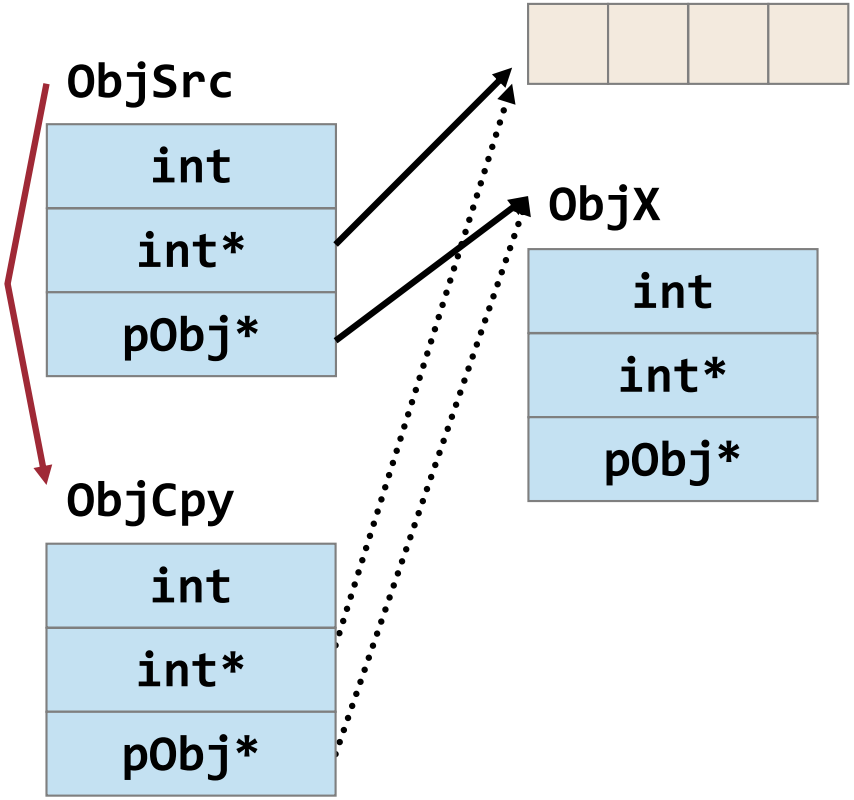
- Der Kopierkonstruktor wird vom Compiler selbst erzeugt, falls er nicht explizit im Programm deklariert wird. Achtung: shallow copy!

Shallow versus Deep Copy (i)

Betrifft das Kopieren von Zeigern bzw. Arrays

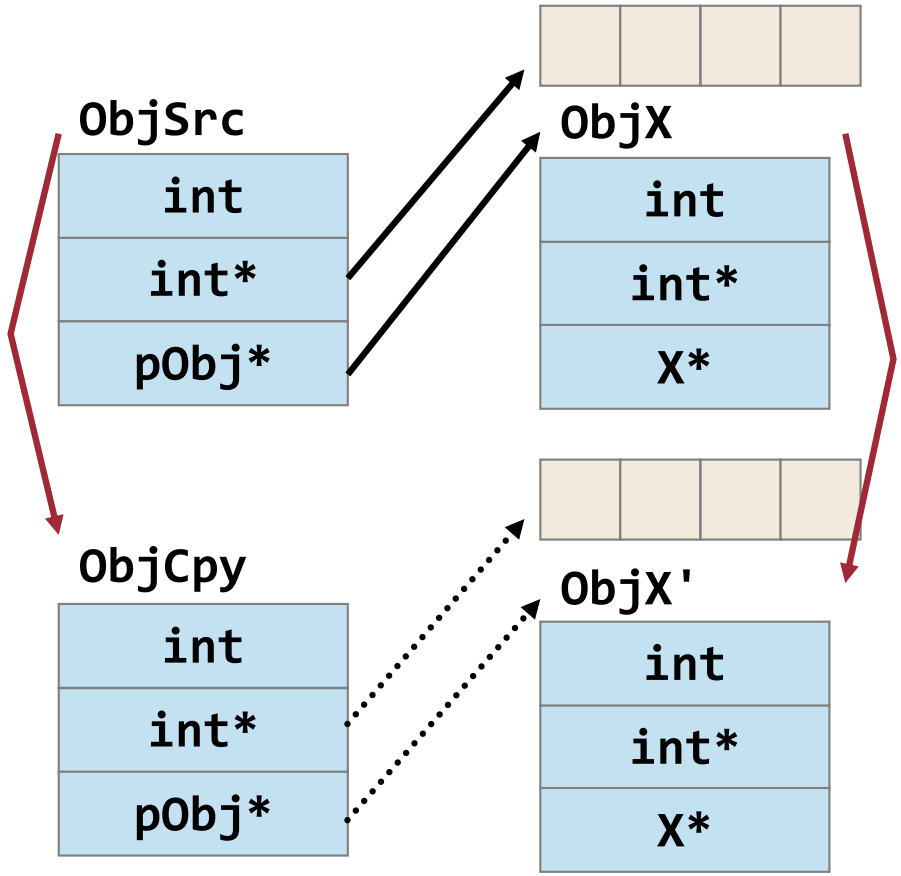
Shallow Copy (Flache Kopie)

Nur Kopie der Instanzvariablen



Deep Copy (Tiefe Kopie)

Kopie wird auch von referenzierten Objekten erzeugt



Shallow versus Deep Copy (ii)

Vereinfachung bei zusammenhängenden Speicherbereichen:

```
// shallow copy
IntArray::IntArray(const IntArray& ref) {
   len   = ref.len;
   data  = ref.data;
}
```

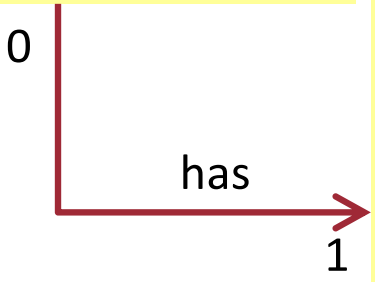
```
// deep copy
IntArray::IntArray(const IntArray& ref) {
   len = ref.len;
   data = new int[len];
   // memcpy Bestandteil von ANSI C
   memcpy(data, ref.data, len*sizeof(int));
}
```

```
for (int i=0; i<len; i++) {
   data[i] = ref.data[i];
   *(data+i)  = *(ref.data+i);
}
```

Initialisierungsreihenfolge (i)

- Die Attribute einer Klassen können natürlich vom Typ einer anderen Klasse sein

```
class Cube {
   Rectangle rec;
   int height;
};
```




```
class Rectangle {
   int length, width;
public:
    Rectangle (int l=0, int w=0) {
        length = l;
        width = w;
    }
};
```

- In diesen Fällen ist folgende Initialisierungsreihenfolge definiert:
  1. Attribute in der Reihenfolge ihrer Deklaration (von oben nach unten)
  2. Danach erfolgt Aufruf des Konstruktors der zu instanziierenden Klasse

Initialisierungsreihenfolge (ii)

... dass heisst:

- zuerst werden Instanzen der Attribute angelegt und initialisiert, bevor das ganze Objekt aufgebaut wird
- ohne weitere Angaben wird für jedes nichtprimitive Attribut der Default-Konstruktor ausgeführt

Beispiel: Mögliche Implementierung eines Cube-Konstruktors:

```
#include "cube.h"
Cube::Cube( int h=0, int w=0, int l=0 ) {
  height = h;
  rec.length = l; // Wenn Konstruktor aufgerufen
  rec.width = w;  // wird ist rec schon existent.
}
```

Initialisierungsreihenfolge (iii)

- Nachteile:
  - Attribute werden zunächst mit Default-Werten versehen und dann mit richtigen Werten initialisiert → nicht effizient!
  - Attribute können nicht konstant (**`const`**) sein und gleichzeitig erst zur Konstruktionszeit parametrisiert initialisiert werden (da notwendige Initialisierung innerhalb Deklaration noch nicht möglich ist).
  - Klassen, für die kein Default-Konstruktor definiert ist, können für Attribute nicht von „aussen“ parametrisiert verwendet werden.

- Erweiterung der Konstruktorsyntax: **Attributinitialisierer**

```
#include "cube.h"
Cube::Cube(int h=0, int l=0, int w=0) : height(h), rec(l, w)
{ /* hier was sonst noch zu tun ist */ }
```

  - Der Aufruf der Default-Konstruktoren entfällt dadurch da...
  - ...der Attributinitialisierer in der Definition angegeben wird (nicht in der Deklaration)

Initialisierung konstanter Attribute

```
class IntArray {
   const int MAX_LEN;
   int* data, len;
public:
   IntArray(int l, int ML=100);
};

IntArray::IntArray(int l, int ML) : len(l), MAX_LEN(ML)
{
   assert(len > 0);
   assert(len <= MAX_LEN);
   data = new int[len];
}
```

Dateninitialisierung

Einziger Ort, an dem man konstantes Attribut parametrisiert initialisieren kann

```
// Verwendung:
IntArray ia(5, 50);        // alle Member initialisiert
```

Der Destruktor

- Eine C++-Klasse kann (muss aber nicht) genau einen Destruktor definieren
- Dieser wird automatisch aufgerufen, wenn ...
  - der Gültigkeitsbereich einer automatisch (auf dem Stack) allokierten Instanz verlassen wird,
  - eine dynamisch allokierte Instanz explizit mit **`delete`** gelöscht wird.
- Name entspricht dem der Klasse, keine Parameter, kein Rückgabewert, identifiziert durch eine vorangestellte Tilde **`~`**
- Destruktor kann virtuell sein (dazu später mehr)
- Vergleichbar mit **`finalize()`**-Methode in Java; Unterschied:
  - C++: Destruktor wird genau dann aufgerufen, wenn der Gültigkeitsbereich verlassen oder **`delete`** aufgerufen wird.
  - Java: Finalizer wird aufgerufen bevor der Garbage Collector die Instanz löscht.
- Der Destruktor ist der Ort, an dem
  - dynamisch allokierter Speicher, der innerhalb einer Instanz allokiert wird, oder
  - Ressourcen (z.B. offene Dateien, Streams, Netzwerkverbindungen)

  wieder freigegeben werden sollte(n).

Richtlinien im Zshg. mit Heap-Speicher

- Wenn ein Konstruktor dynamisch reservierten Speicher allokiert (**`new`**), dann …
  - … muss spätestens der Destruktor den Speicher freigegeben (**`delete`**),
  - … sollte man immer einen Copy-Konstruktor definieren,
  - … sollte man immer einen Zuweisungsoperator definieren.

- „Gang of Four“ :=
  - Ctor + Copy-Ctor + Dtor + Zuweisungsoperator

- Sonst können zur Laufzeit Probleme auftreten:
  - Memory Leaks (siehe Kapitel 3),
  - Shallow Copy realisiert obwohl eine Deep Copy gefordert ist.

Beispiel: Konstruktoren

IntArray.h

```
class IntArray {
private:
    int *data; int len;
public:
    IntArray();                 // Default-Konstruktor
    IntArray(int);              // ctor mit Parameter (Arraygrösse)
    IntArray(const IntArray&); // Kopierkonstruktor
    IntArray& operator=(const IntArray&); // Zuweisungsoperator
    ~IntArray();                          // Destruktor
};
```

IntArray.cpp

```
#include <cassert>
IntArray::IntArray(int l) {
   assert(l > 0);
   len = l;
   data = new int[len];
}
IntArray::IntArray() { // Default-Konstructor
   len = 0 ;      // Werte initialisieren um sicherzustellen
   data = NULL;  // dass sie nicht beliebig sind.
}
```

Beispiel zusammengefasst: Ctor & Dtor

```
#include <cassert> // -DNDEBUG
IntArray::IntArray(int l) {
   assert(l > 0);
   len = l;
   data = new int[len];
}
```

```
IntArray::IntArray() {
   len = 0 ;     // Werte initialisieren um sicherzustellen
   data = NULL;  // dass sie nicht beliebig sind.
}
```

```
IntArray::IntArray(const IntArray& ref) {
   len = ref.len;
   data = new int[len];
   // memcpy Bestandteil von ANSI C
   memcpy(data, ref.data, len*sizeof(int));
}
```

```
IntArray::~IntArray() {  // Destruktor
  delete[] data;
}
```

Zusammenfassendes Beispiel

Wann werden Konstruktoren, Destruktoren und Operatoren aufgerufen?

```
void f1(const IntArray& ia) {…}
void f2(IntArray ia) {…}
IntArray f3() {…}

int main() {
    IntArray ia1;           // default ctor
    IntArray ia2(10);       // IntArray(int)
    IntArray ia3(ia2);      // copy ctor
    IntArray ia4 = ia2;     // copy ctor
    IntArray a_of_ia[20];   // default ctor für 20 Elemente
    ia1 = ia2;              // Zuweisungsoperator
    f1(ia1);                // keine neue Instanz erzeugt (Referenz)!
    f2(ia1);                // copy ctor
    ia1 = f3();             // copy ctor Aufruf bei return
    return 0;               // dtor von ia1, ia2, ia3, ia4 und für
}                           // jedes Element von a_of_ia
```

Und noch ein Beispiel

```
#include <iostream>
using namespace std;
class Body {
public:
    static int count; // Shall be an instance counter.
    Body()  { cout << "in Ctor " << ++count << endl;}
    ~Body() { cout << "in Dtor " << --count << endl;}
    Body( const Body& body ) {
       cout << "in CopyTor " << ++count << endl;
    }
};
void f(Body body) {  };
int Body::count=0;

int main() {
   // create objects
   Body body;
   f(body);
}
```

Wie ist die Ausgabe?

```
in Ctor 1
in Dtor 0
in Dtor -1???
```

Wie ist die Ausgabe jetzt?

```
in Ctor 1
in CopyTor 2
in Dtor 1
in Dtor 0
```

Der `this`-Zeiger

- Erlaubt den Zugriff auf das aktuelle Objekt
  - Eine Methode kann auf jeden Member zugreifen, ohne dabei konkret das Objekt anzugeben.
  - Die Adresse des Objektes steht implizit in der Methode mit dem konstanten Zeiger **`this`** zur Verfügung:

```
Klasse * const this = &aktObj;
```

Beispiel: Verwendung innerhalb einer Methode

```
len = 5;        // Zuweisung an Membervariable len
func();         // Methode func aufrufen
// implizit erzeugt der Compiler diesen Ausdruck
this->len = 5;
this->func();
```

Typisches Beispiel für `this`

- Die Verwendung des **`this`**-Zeigers ist dann notwendig, wenn das aktuelle Objekt (also **`this`**) als Ganzes angesprochen werden muss
- Rückgabe des **`this`**-Zeigers bei der Return-Anweisung des **Zuweisungsoperators** (welcher kein Konstruktor ist)

```
IntArray& IntArray::operator=(const IntArray& v) {
   len = v.len;   // this->len = v.len;
   for (int i=0; i<len; ++i)
      data[i] = v.data[i];

   return *this;
}
```

```
IntArray a(10), b(10);
a.operator=(b);  // a = b;
                 // Operator überladen
```

K05

Klassen & Methoden in C++

1. Definition
2. Konstruktoren und Destruktor
3. **Statische Klassenelemente**
4. Überladen von Operatoren

Statische Klassenmember in C++

Header-Datei: **Employee.h**

```
class Employee {
public:
    Employee();
    ...
    static int count;
};
```

Implementierung: **Employee.cpp**

```
// Initialisierung (Definition)
// muss ausserhalb der
// Klassendeklaration erfolgen
int Employee::count = 0;
Employee::Employee() {
    ...
    ++count;
}
```

- Statische Klassenattribute werden nur einmal gespeichert (unabhängig von der Anzahl an Instanzen der Klasse)
- Existieren auch wenn keine Instanz existiert
- Nicht zu verwechseln mit globalen Variablen; sie sind unter der „Kontrolle“ bzw. im Namensbereich der Klasse
- Schlüsselwort: **`static`**
- **`private`**, **`protected`** oder **`public`**

Statische Member – Zugriff

- Für statische Attribute/Methoden gelten die üblichen Regeln der Datenkapselung (Sichtbarkeit).
- Auf einen als **`public`** deklarierten, statischen Member ist daher der Zugriff über eine Instanz der Klasse möglich:

```
Employee person;
cout << "Number of employees: " << person.count << endl;
```

Diese Variante wird nicht empfohlen!

Verwendung des Bereichsoperators **`::`**

```
cout << "Number of employees: " << Employee::count << endl;
```

Statisch versus Instanzgebunden

Unterschiede (die klar sein sollten):

- Statische Methoden sind beim Aufruf an keine Instanz der Klasse gebunden.
- Im Unterschied zu einer Instanzmethode steht ihnen deshalb kein `this`-Zeiger zur Verfügung ...
- ... was wiederum zur Folge hat, dass statische Methoden keinen Zugriff auf Attribute und Methoden haben, die selbst nicht statisch sind.

K05

Klassen & Methoden in C++

1. Definition
2. Konstruktoren und Destruktor
3. Statische Klassenelemente
4. **Überladen von Operatoren**

Operatoren überladen

- Das Überladen von Operatoren ermöglicht es, vorhandene Operatoren (**+**, **-**, *****, **/**, usw.) auch für Klassen zu definieren und damit auf Instanzen anzuwenden.
- Die meisten der C++ Operatoren die für elementare Datentypen definiert sind lassen sich nicht sofort auf Klassenobjekte anwenden.
  - Ausnahme: z.B. der Zuweisungsoperator (=) ist ein Operator, der automatisch auch bei Klassen definiert/nutzbar ist.
- Dabei wird der Definitionsbereich des Operators erweitert.
- Arithmetische Operatoren können nur verwendet werden, wenn sie auch für Klassenobjekte überladen werden.
- Das Überladen eines Operators findet immer in Zusammenhang mit mindestens einer Klasse statt, d.h. bei binären Operatoren muss mindestens einer der beiden Typen ein benutzerdefinierter Typ (Klasse) sein.

Operatoren-Überladen für Klassen

- Man kann die Funktionalität einer Klasse nicht nur durch Methoden, sondern auch durch Operatoren festlegen.

- Beispiel:

```
Complex c1, c2, c3;
c3 = c1.mult(c2);      ← schwer zu lesen
c3 = c1 * c2;          ← klar
```

- Vorteil: Ausdrücke mit Operatoren sind oft intuitiver und schneller zu erfassen als Methodenaufrufe.

Überladen von Operatoren – Regeln

- Es können keine neuen Operatoren eingeführt werden:
  - `***`   oder `;-)` funktioniert nicht!
- Die Bedeutung der Operatoren auf elementaren Datentypen lässt sich nicht umdefinieren (z.B. Addition **+** für **`int`**).
- Die Anzahl der Operanden eines Operators kann nicht geändert werden:
  - Binärer Operator bleibt stets binär (**`*`**, **`-`**, …)
  - Unärer Operator bleibt stets unär (**`++`**, **`--`**, **`!`**).
- Präzedenz und Assoziativität bleibt unverändert.
  - z.B. **`a * b + c == (a * b) + c`** bzw.
    `cout << "x" << "y" == ((cout << "x") << "y")`
- Operatoren müssen alle explizit überladen werden.
  - z.B. Überladen von **`+`** und **`+=`**

(Nicht-)Überladbare Operatoren

Operatoren, die man nicht überladen kann	Bedeutung
`.` `.*` `::`	Zugriffsoperatoren
`?:` `sizeof` `typeid`	Auswahloperator, Speicherplatz
Überladbare Operatoren	**Bedeutung**
`+` `-` `*` `/` `%` `++` `--`	arithmetische Operatoren
`==` `!=` `<` `<=` `>` `>=`	Vergleichsoperatoren
`&&` `||` `!`	Logische Operatoren
`=`	Zuweisungsoperator
`&` `|` `^` `~` `<<` `>>`	Bitoperationen
`()` `[]`	Funktionsaufruf, Indexoperator
`&` `*` `->` `,`	Sonstige Operatoren
`new` `new[]` `delete` `delete[]`	

Vordefinierte Operatoren

- Für jede Klasse werden automatisch folgende Operatoren überladen wenn sie nicht selbst überladen wurden:
  - `=` (Zuweisungsoperator): erzeugt Shallow Copy.
  - `,` (Verkettungsoperator): verkettet Abfolgen von Anweisungen.
  - `&` (Adressoperator): ermittelt Adresse des Operanden im Speicher.

Beispiel – Operator überladen (i)

Binärer Operator:

```
class X {
   X operator * (const X& right); // Deklaration
}
X X::operator * (const X& right) { // Definition
   ...
}
```

- Definiert den binären Operator * für Instanzen der Klasse X (Semantik von * bleibt davon unabhängig für elementare Datentypen unverändert)

Verwendung:

```
X x, y, z;
x = y.operator*(z);
x = y * z;
```

Beispiel – Operator überladen (ii)

Unärer Operator:

```
class X {
   X operator ! (); // Deklaration
}
X X::operator ! () { // Definition
  ...
}
```

- Definiert den unären Operator **!** für Instanzen der Klasse X, wobei **!** ein Operator ist, der für elementare Datentypen existiert

Verwendung:

```
X x, z;
x = z.operator!();
x = ! z;
```

Beispiel – Operator überladen (iii)

Beispiele:

```
x = y;          // äquivalent zu x.operator=(y);
i += 1;         // i.operator+=(1);
if( q == r )    // q.operator==(r)      oder   operator==(q, r)
cout << x;      // cout.operator<<(x);  oder   operator<<(cout, x);

a = b + c;      // a = b.operator+(c);  oder   a = operator+(b, c);
```

Es gibt zwei Möglichkeiten, Operatoren zu überladen:

Als Operatormethode innerhalb der Klasse **X** z.B.

```
// Addition, binärer Operator
X operator+(const X& y) const;
```

Als globale Operatorfunktion ausserhalb aller Klassen z.B.

```
// Addition, binärer Operator
X X::operator+(const X& y, const X& z);
```

Beispiel: Klasse **IntArray** (ii)

Operator überladen: neue Definition des Verhaltens für diese Klasse

```
class IntArray {
public:
   // Zuweisungsoperator:              a = b
   IntArray& operator=(const IntArray& b);
   // Addition, binärer Operator:     a += b
   IntArray& operator+=(const IntArray& b);
   // Addition mit int, binärer op:   a += i
   IntArray& operator+=(int i);
   // Addition, binärer Operator:     c = a+b
   IntArray operator+(const IntArray& b) const;
   // Gleichheitstest, binärer Op.:   a == b
   bool operator==(const IntArray& b) const;
   // Präfix increment, unärer op.:   ++a
   IntArray& operator++();
   // Postfix increment, unärer Op.: a++
   IntArray operator++(int);
   ...
private:
   int*  data, len;
};
```

Klasse **IntArray**: Zuweisungsoper.

```
IntArray& IntArray::operator=(const IntArray& rhs)
{
   if (&rhs == this)
     return *this;

   if (len != rhs.len) {
      delete [] data;
      data = new int[rhs.len];
   }

   len = rhs.len;
   memcpy(data, rhs.data, len * sizeof(int));
   /* for (int i=0; i<len; ++i) data[i] = rhs.data[i]; */
   return *this;
}
```

Verhindert Selbstzuweisung **obj = obj**

Grösse der dynamischen Arrays ändern, falls diese unterschiedlich sind

Kopiere die Daten

Objekt zurückgeben

```
b = c;   <=>  b.operator=( c );
```

Klasse **IntArray**: Additionsoperat.

```
// addition, binary operator: a += b
IntArray& IntArray::operator+=(const IntArray& v) {
   assert(len == v.len);

   for (int i=0; i<len; ++i)
      data[i] += v.data[i];

   return *this;
}
// addition, binary operator: c = a + b
IntArray IntArray::operator+(const IntArray& v) const
{
   IntArray retval(*this);

   retval += v;

   return retval;
}
```

Ändere das Objekt

(**this**) Objekt zurückgeben

Erzeuge neues Objekt

Neues Objekt zurückgeben

Klasse **IntArray**: Vergleichsoperator

```
bool IntArray::operator==(const IntArray& v) const
{
   if (len != v.len)
      return false;

   for (int i=0; i<len; ++i)
      if (data[i] != v.data[i])
         return false;

   return true;
}
```

Klasse **IntArray**: Präfix-/Postfixoper.

- Der unäre Inkrement-Operator (**++**) für Klassen ist speziell, weil es eine Präfix- und eine Postfix-Notation gibt:
  - Präfix-Fall (**++a**): gibt den inkrementierten Wert zurück.
  - Postfix-Fall (**a++**): gibt den alten Wert als Kopie zurück und inkrementiert a

- Definition der Präfix-Version:

```
// präfix increment
IntArray& IntArray::operator++() {
   for (int i=0; i<len; ++i)
      ++data[i];
   return *this;
}
```

Aufruf:

```
IntArray a;
++a;
```

Klasse **`IntArray`**: Präfix-/Postfixoper.

- Postfix-Version: etwas aufwändiger, da der alte Wert zurückgegeben wird.
- Konvention: Der Postfix-Operator wird durch einen Dummy-Parameter (**`int`**) vom Präfix-Operator unterschieden:

```
// postfix increment
IntArray IntArray::operator++(int) {
   IntArray tmp(*this); // create copy of this
   // increment *this object
   for (int i=0; i<len; ++i)
      data[i]++;
   return tmp; // return copy
}
```

Aufruf:

```
IntArray a;
a++;
```

Klasse `IntArray`: Index-Operator (i)

Zur Erinnerung: bei Arrays ...

- ... wird der Index-Operator zu Zeigerarithmetik:
  - `v[i]` ist gleichbedeutend mit `*(v+i)`
- D.h. es gelten für den Index-Operator bei Arrays Einschränkungen:
  - Linker Operand muss ein Zeiger sein.
  - Der andere Operand muss ein ganzzahliger Ausdruck sein.
  - Der Ergebnistyp ist festgelegt.

- Diese Einschränkungen gelten bei Klassen nicht:
  - Der linke Operand muss ein Objekt der Klasse sein.
  - Der rechte Operand darf ein beliebiger Datentyp sein.
  - Der Ergebnistyp ist nicht festgelegt.

Klasse **IntArray**: Index-Operator (ii)

```
class IntArray {
   int *data;
   int len;
public:
   IntArray(int l) {
      len = l;
      data = new int[len];
      for (int i=0; i< len; i++)
         data[i] = i;
   }
   int& operator[](int i) {
      if (i<0 || i>=len) {
          cerr << "Out of Range!"
               << endl;
          exit(1);
      }
      //Referenz auf i-tes Element
      return *(data+i);
   }
};
```

```
int main()
{
   // ctor für 5 Elemente
   IntArray a(5);
   for (int i=0; i<6; i++)
      cout << a[i] << endl;
}
```

```
~/test/a> ./int_test
0
1
2
3
4
Out of Range!
```

Shift-Operator für die Ausgabe (i)

- Will man eine Instanz **`c`** einer Klasse (z.B. **`Complex`**) auf dem Standardausgabestrom ausgegeben (welcher standardmässig auf die Konsole geleitet ist), mittels **`cout << c`**, so erhält man eine Fehlermeldung des Compilers

```
int i;
cout << i; // operator<<(cout, i) – OK

Complex a;
cout << a; // Fehler!
```

Shift-Operator für die Ausgabe (ii)

... Ergo: den Operator **<<** überladen für **Complex**

```
class Complex {
  ...
  // Deklaration global operators (friend)
  friend ostream& operator<< (ostream&, const Complex&);
}
ostream& operator<<(ostream& o, const Complex& c) {
   if (c.imag==0) o << c.real;                  // als Realzahl
   else if (c.real==0) o << c.imag << "i";      // rein imaginär
   else {                                       // (a+bi) oder (a-bi)
      o << "(" << c.real;
      if (c.imag >= 0) o << '+';
      if (c.imag  < 0) o << '-';
      o << c.imag << "i)";
   }
   return o;
} // output <<
```

Die Rückgabe von **ostream** bewirkt, dass man mehrere Komponenten hintereinander mit << verknüpfen kann

```
cout << a << "  " << b << endl;
```