C++

Grundlagen und Objektorientierte Programmierung

Matthias Pospiech

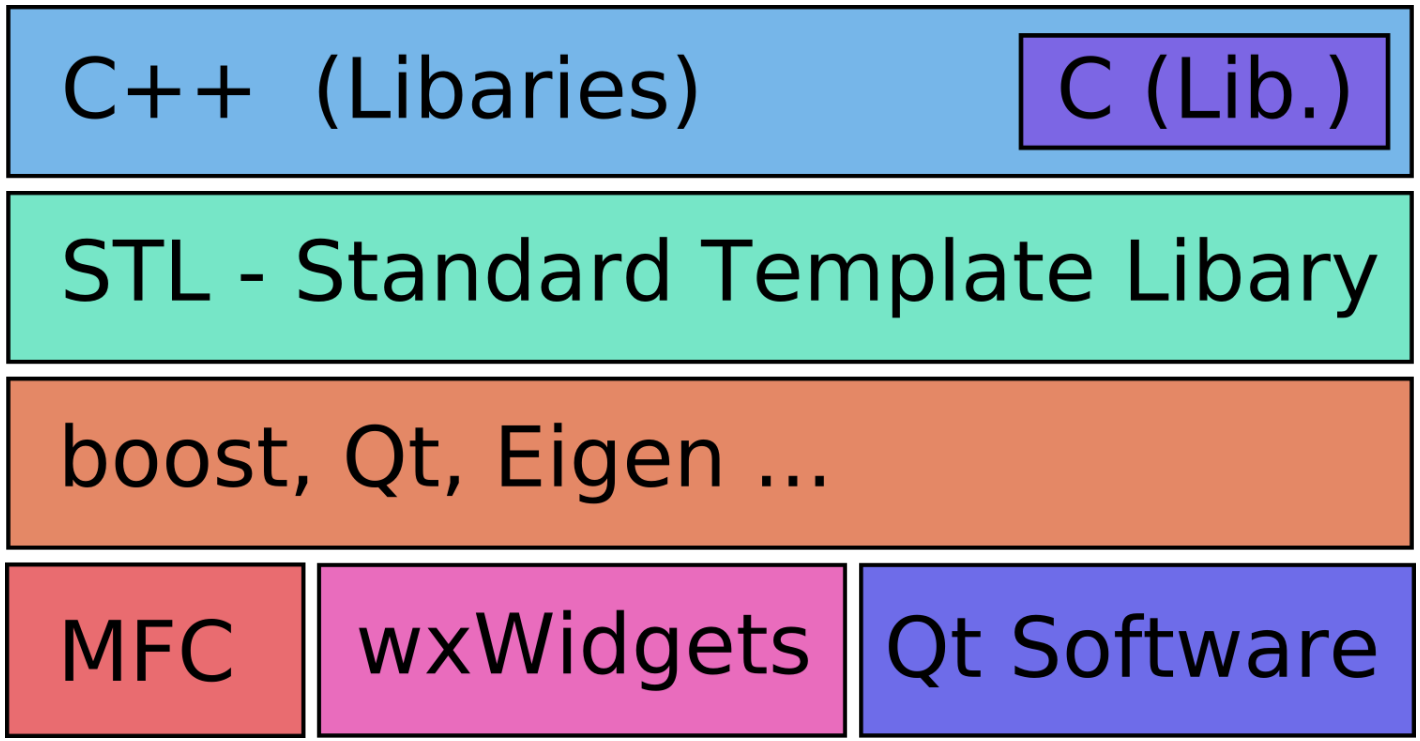
IQO

Stufen der C++ Programmierung

Es gibt nicht ein C++, sondern verschiedene Art-und-Weisen C++ zu nutzen:

1. C (C++ basiert of C)
   C-Arrays, globale Variablen und Funktionen
2. Objektorientiertes C++
   Klassen, Konstruktoren, Destruktoren, Kapselung, Vererbung (inheritance), Polymorphismus, Virtuelle Funktionen (dynamic binding)
3. Template C++ (Funktionen unabhängig vom Datentyp)
4. STL (Standard Template Libary)
   vector, complex, string, ...

C++ und Erweiterungen



Datentypen

Eingebaute Typen

- `bool` (true, false)
- `char` (einzelnes 8 bit ASCII Zeichen)
- `int` ($-2.147.483.648$ - $2.147.483.647$)
- `long` ($-9.223.372.036.854.775.808$ - $9.223.372.036.854.775.807$)
- `float` ($1,5 \cdot 10^{-45}$ - $3,4 \cdot 10^{38}$)
- `double` ($5 \cdot 10^{-324}$ - $1,7 \cdot 10^{308}$)

STL

- `std::string`
- `std::vector<double>`
- `std::complex<double>`
- ...

sowie beliebige weitere selbst- und automatisch definierte Typen

Kontrollstrukturen

if Anweisungen

```
1  if ( a < b ) {
2      a = b;
3  }
4  else if ( a > b ) {
5      b = a;
6  }
7  else if ( a == b ) {
8      ...
9  }
10 else {  // wenn alles andre fehlschlaegt ..
11     cout << "What else could I test ?" << endl;
12 }
```

Kontrollstrukturen

Logische Operatoren

```
1  int i = 10, j = 20;
2  if ((i == 10) && (j == 10)) // und
3  {
4    cout << "Beide Werte sind gleich zehn" << endl;
5  }
6  if ((i == 10) || (j == 10)) // oder
7  {
8    cout << "Ein Wert oder beide Werte
9              sind gleich zehn" << endl;
10 }
```

Kontrollstrukturen

for Schleifen

```
1 //  (start; bedingung; weiter)
2 for (int x=1; x<=10; x++)
3 {
4     cout << x << endl;
5 }
```

Kontrollstrukturen

while Schleifen (Testen am Anfang)

```
1 int x=1;
2 while (x!=0)
3 {
4   cout << "Geben sie 0 ein, um das
5            Programm zu beenden" << endl;
6   cin >> x;
7 }
```

Kontrollstrukturen

do Schleifen (Testen am Ende)

```
1 int x;
2 do
3 {
4   cout << "Geben sie 0 ein, um das
5            Programm zu beenden" << endl;
6   cin >> x;
7 } while (x!=0);
```

Kontrollstrukturen

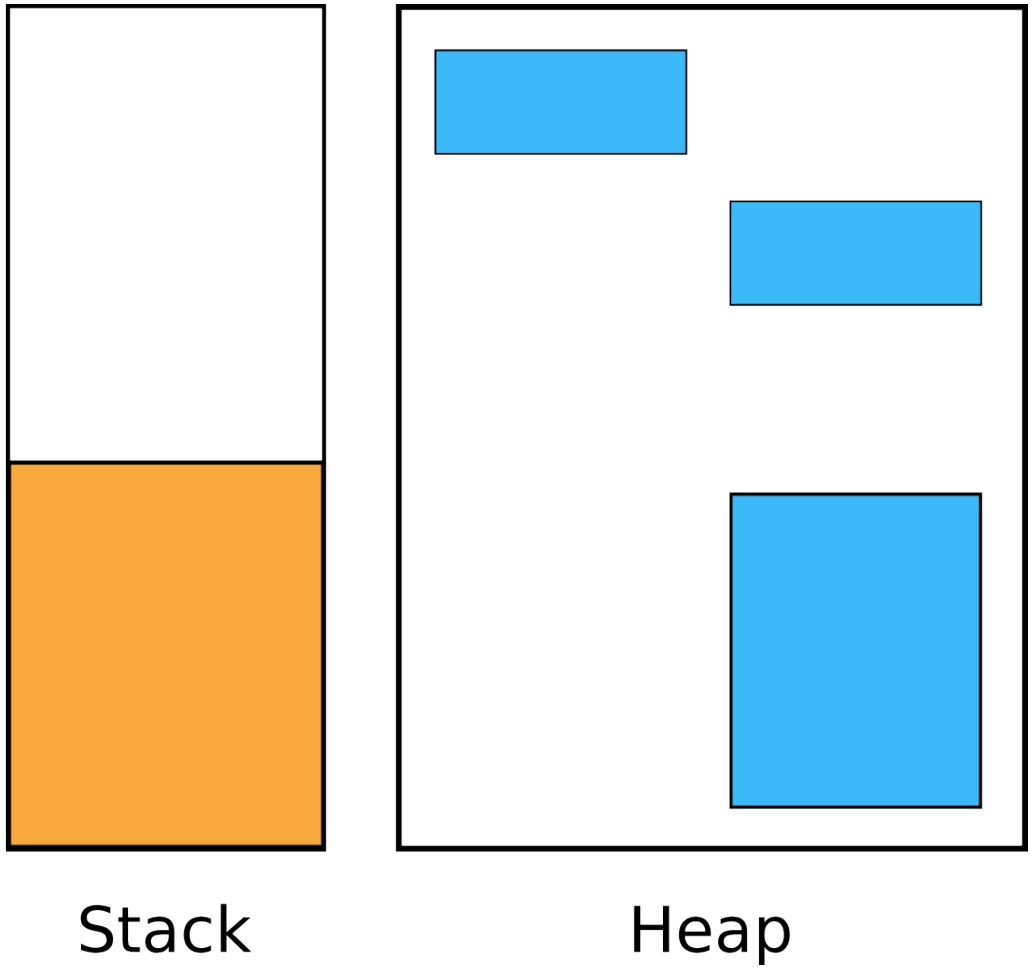
switch Fallunterscheidung

```
1  char c;
2  cin >> c;
3  switch(c)
4  {
5  case 'a':
6      cout << "Ausgabe: a" << endl;
7      break;
8  case 'b':
9      cout << "Ausgabe: b" << endl;
10     break;
11 default:
12     cout << "Weder noch" << endl;
13     break;
14 }
```

Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

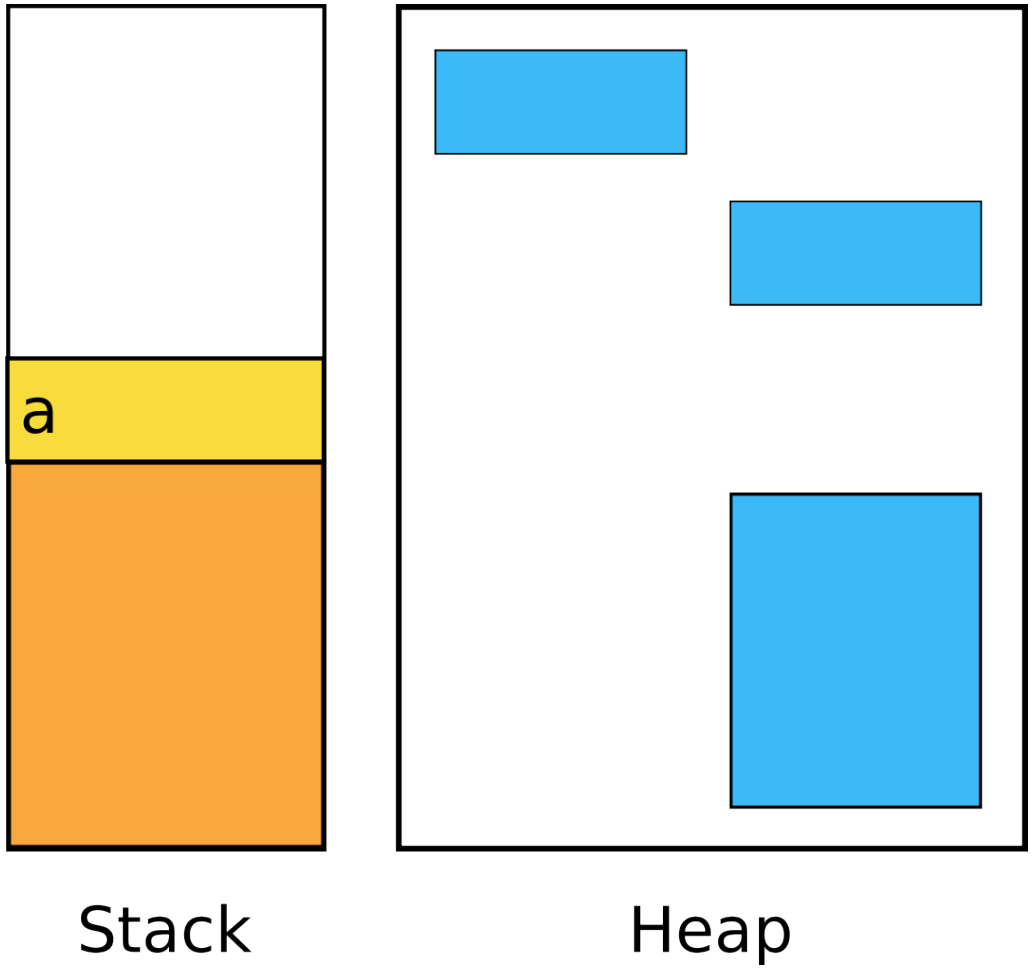
```
void function()
{




}
```



Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

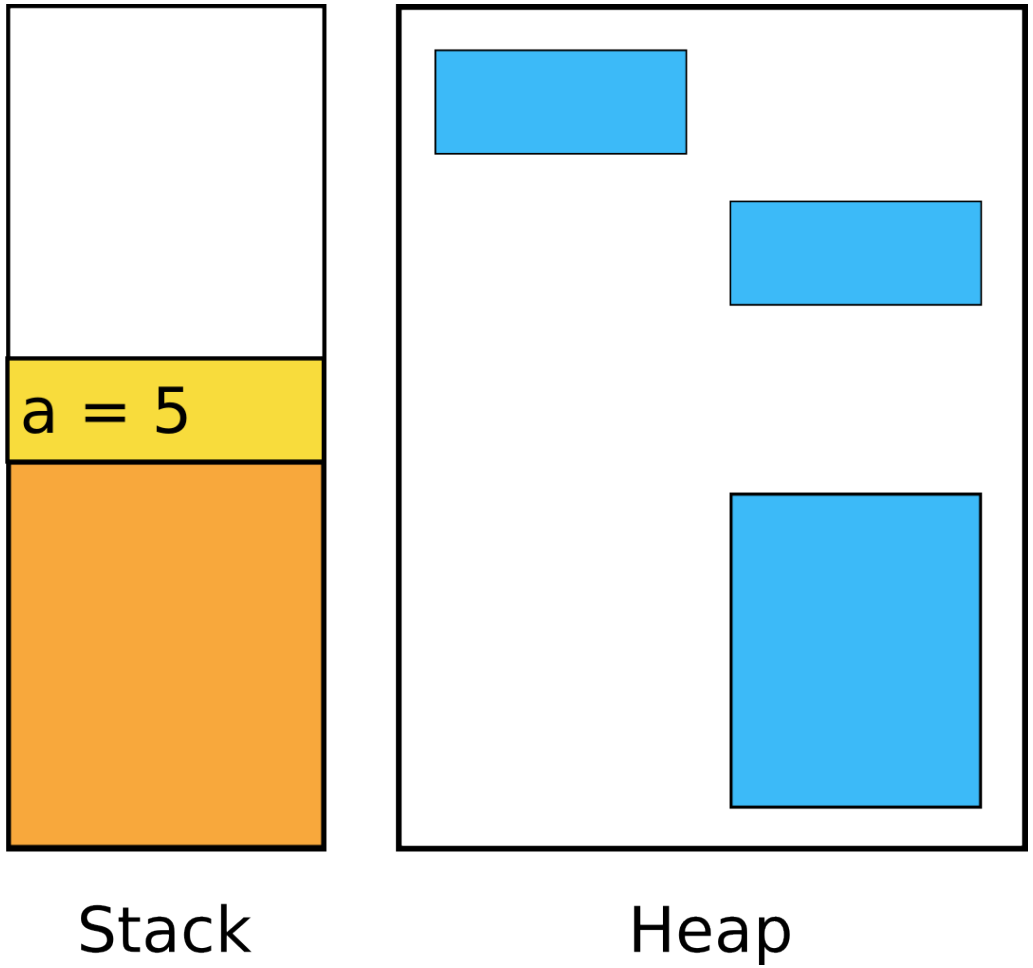
```
void function()
{
 int a;



}
```



Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

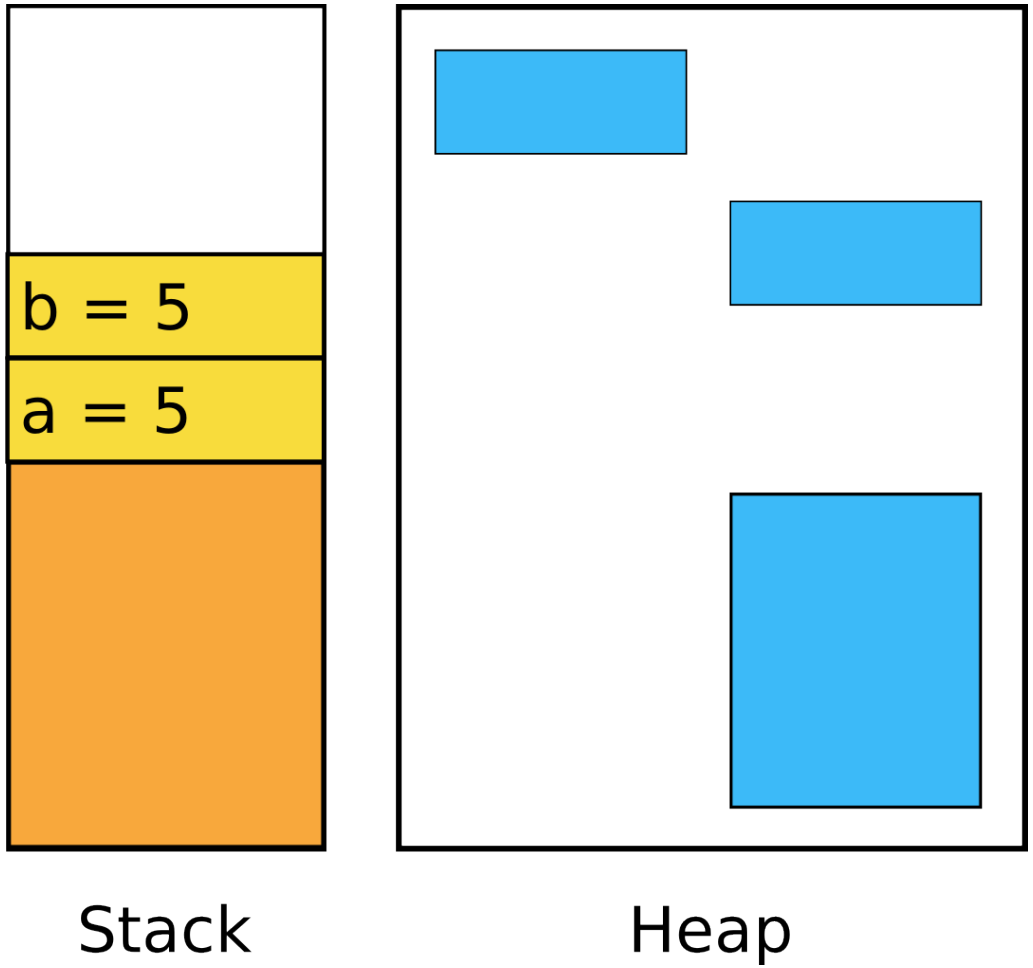
```
void function()
{
 int a;
 a = 5;




}
```



Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

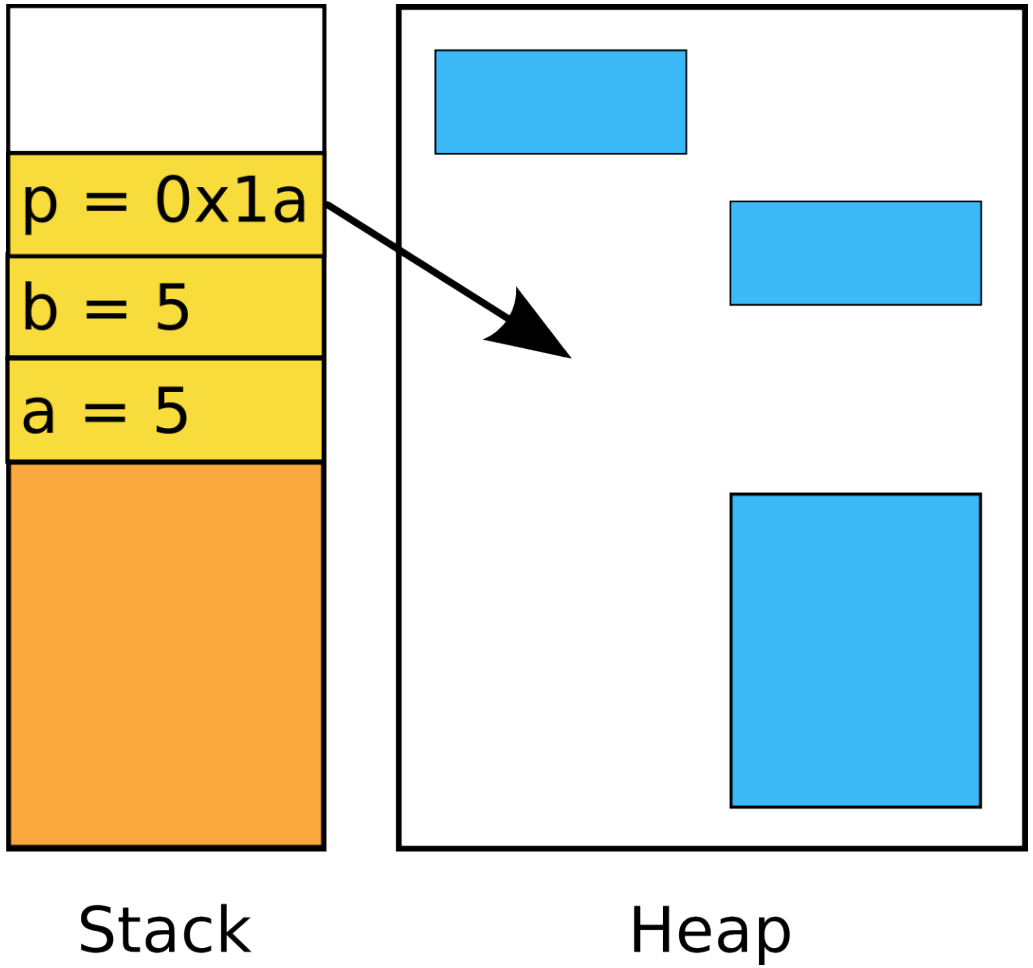
```
void function()
{
 int a;
 a = 5;
 int b=a;



}
```



Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

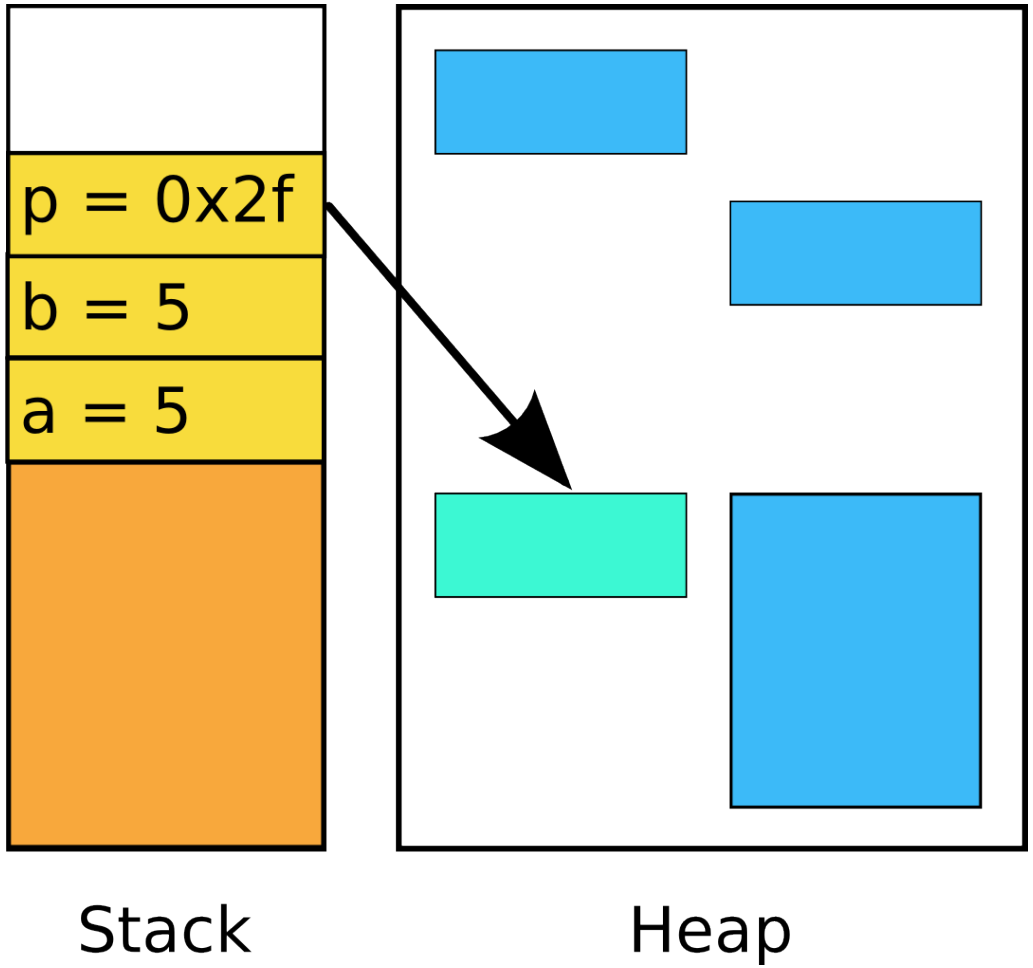
```
void function()
{
 int a;
 a = 5;
 int b=a;
 int * p;



}
```



Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

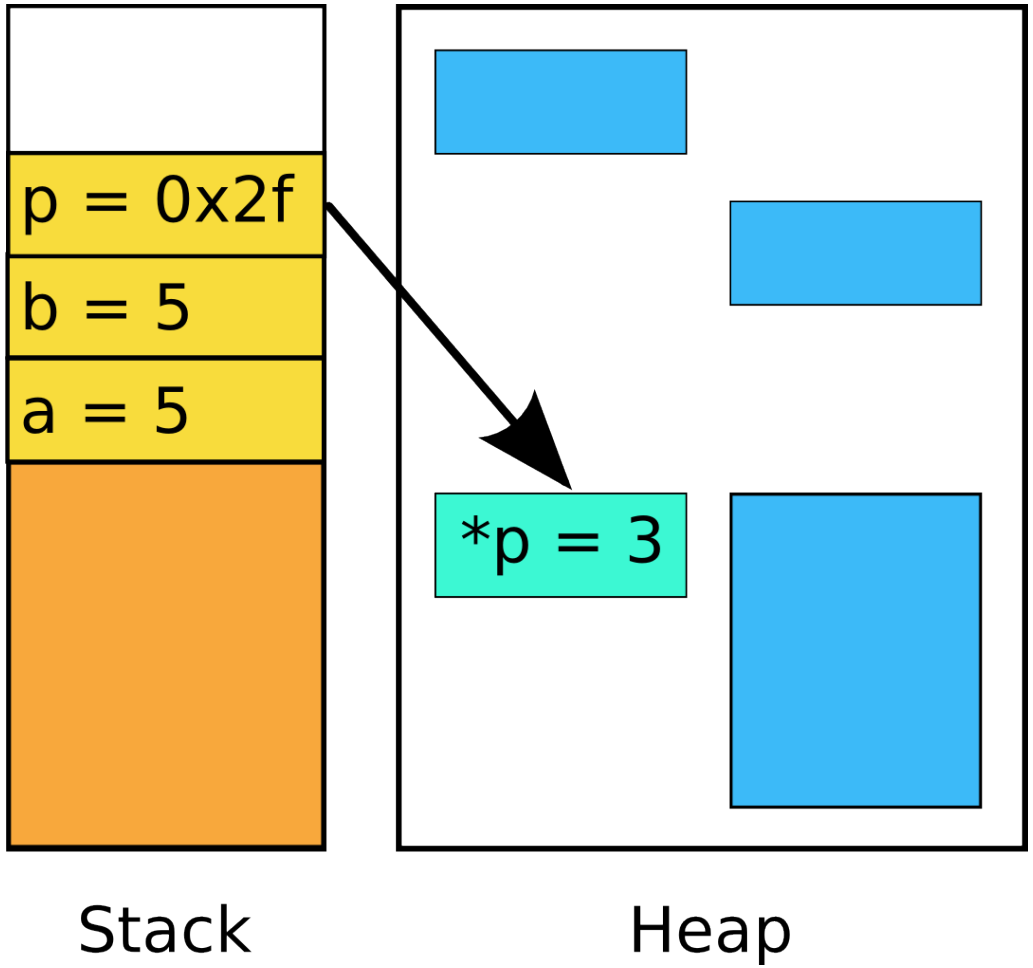
```
void function()
{
 int a;
 a = 5;
 int b=a;
 int * p;
 p = new int;


}
```



Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

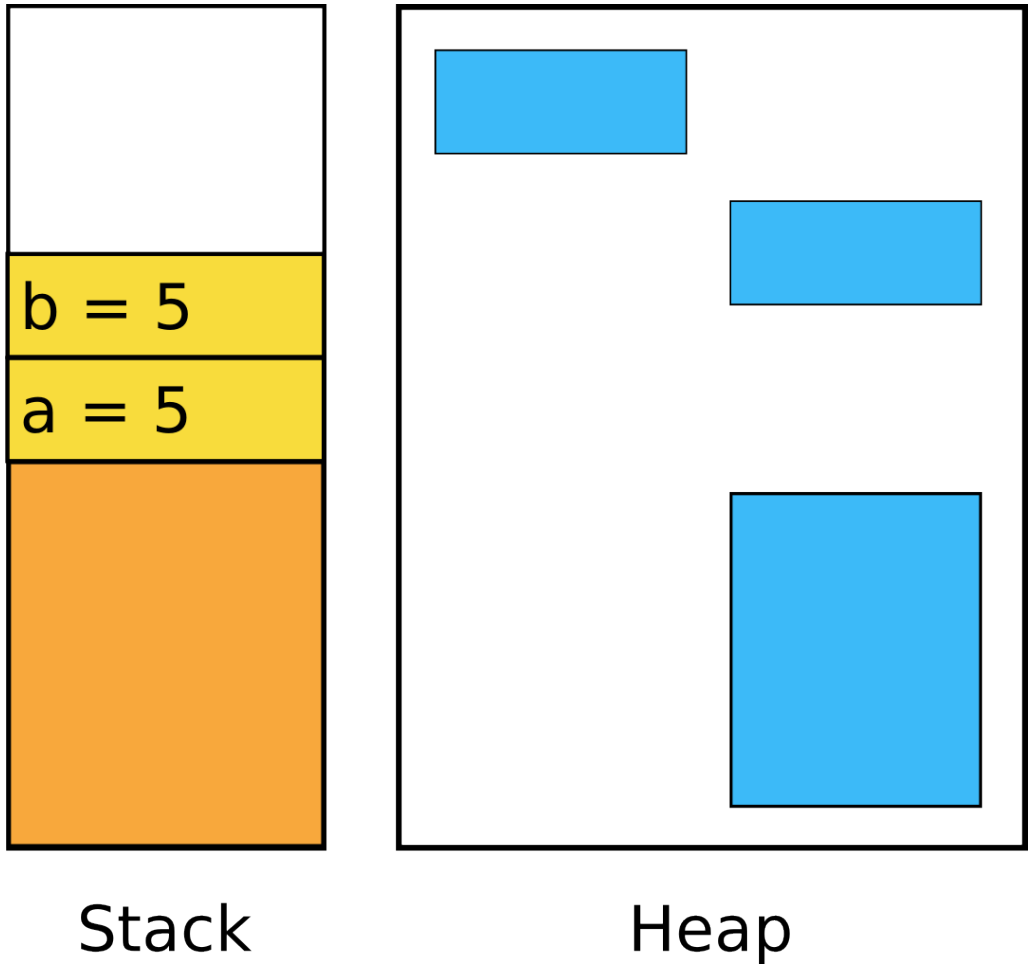
```
void function()
{
 int a;
 a = 5;
 int b=a;
 int * p;
 p = new int;
 *p = 3;

}
```



Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

```
void function()
{
 int a;
 a = 5;
 int b=a;
 int * p;
 p = new int;
 *p = 3;
 delete p;
}
```



Heap & Stack

Stack

Vorteile:

- automatisch gelöscht

Nachteile:

- sehr begrenzter Speicherplatz

Heap & Stack

Vorteile Heap

Vorteile:

- *unbegrenzter* Speicherplatz

Nachteile:

- Speichermanagement notwendig (Freigeben des Speichers)

Deklaration erfolgt über Pointer.

Nutzung bei großen Arrays und großen Klassen.

Variablen, Referenzen und Pointer

Erzeugung einer Variable (Erstellen einer Instanz)

```
1 int a;
2 int b = 5;
3 laser UltraFastLaserImpl;
4 QSize size(5,6); // Erzeugung mit Konstruktor
5                  // size.width() = 5, size.height() = 6
```

Variablen, Referenzen und Pointer

Referenzen

(Alias auf bestehende Objekte)

```
1 int a;
2 int & ref = a; // Referenzen müssen auf etwas zeigen
3 a = 5;         // ref = a = 5
```

Variablen, Referenzen und Pointer

Pointer

```
1 int * a;  // Pointer of Variable vom Type 'int',
2           // zeigt auf keine existente Variable!
3 a = NULL; // Speicherort als ungültig markieren
```

Variablen, Referenzen und Pointer

Pointer

```
1 int * a;  // Pointer of Variable vom Type 'int',
2           // zeigt auf keine existente Variable!
3 a = NULL; // Speicherort als ungültig markieren
4
5 int * b = new int; // Erzeugung eines Speicherortes
6                    // und Objektes
7 *b = 5;   // Inhalt von Pointer b = 5
```

Variablen, Referenzen und Pointer

Pointer

```
1  int * a;  // Pointer of Variable vom Type 'int',
2            // zeigt auf keine existente Variable!
3  a = NULL; // Speicherort als ungültig markieren
4
5  int * b = new int; // Erzeugung eines Speicherortes
6                     // und Objektes
7  *b = 5;   // Inhalt von Pointer b = 5
8
9  a = b;    // Speicherort von a identisch mit b
10 *a = 2;   // *a = *b = 2
```

Variablen, Referenzen und Pointer

Pointer

```
1  int * a;  // Pointer of Variable vom Type 'int',
2            // zeigt auf keine existente Variable!
3  a = NULL; // Speicherort als ungültig markieren
4
5  int * b = new int; // Erzeugung eines Speicherortes
6                     // und Objektes
7  *b = 5;   // Inhalt von Pointer b = 5
8
9  a = b;    // Speicherort von a identisch mit b
10 *a = 2;   // *a = *b = 2
11
12 delete b; // Alle mit new erzeugten Variablen müssen
13           // mit delete gelöscht werden.
```

Variablen, Referenzen und Pointer

*, & Grammatik

```
1 int * a;
2 int b = 5;
3 a = &b;      // & ermittelt Speicherort
4 int c = *a; // * ermittelt Wert an Speicherort
```

Die Operatoren * und & wechselt zwischen Deklaration und Zuweisung ihre Bedeutung!

Variablen, Referenzen und Pointer

Beispiel

```
int a;
int& b = a;
int& c = b;
int *d = &c;

a = 6;
*b = 7;
*d = 8;
```

welcher Fehler ist hier eingebaut?

Variablen, Referenzen und Pointer

Beispiel

```
int a;
int& b = a;
int& c = b;
int *d = &c;

a = 6;
*b = 7; // Ungültige Dereferenzierung
*d = 8;
```

welcher Fehler ist hier eingebaut?

Variablen, Referenzen und Pointer

Beispiel

```
int a;
int& b = a;
int& c = b;
int *d = &c;

a = 6;

*d = 8;
```

welche Werte haben a,b,c,d am Ende ?

Variablen, Referenzen und Pointer

Beispiel

```
int a;
int& b = a;
int& c = b;
int *d = &c;

a = 6;

*d = 8; // a=b=c=d=8
```

welche Werte haben a,b,c,d am Ende ?

Variablen, Referenzen und Pointer

Beispiel2

```
1 string s1("foo");
2 string s2("bar");
3
4 string & rs = s1;
5 string *ps  = & s1;
6
7 rs = s2;
8 ps = &s2;
```

welche Werte haben `s1, s2, *ps, rs` ?

Variablen, Referenzen und Pointer

Beispiel2

```
1  string s1("foo");
2  string s2("bar");
3
4  string & rs = s1;   // rs = s1 = "foo"
5  string *ps  = & s1; // *ps = s1 = "foo"
6
7  rs = s2;            // rs = s2 = "bar"
8  ps = &s2;           // *ps = s2 = "bar"
9
10                     // rs = *ps = s2 = "bar"
11                     // s1 = "foo"
```

welche Werte haben `s1, s2, *ps, rs` ?

Konstante Variablen

```
1 int   x = 4;          // normal Variable
2 x = 10;               // ok
3
4 const int x = 2;      // konstante Variable
5 x = 10;               // error
```

Konstanten in Funktionsaufrufen

```
1 void print(const string& s)
2 {
3     cout << s << endl;
4 }
```

Konstanten in Funktionsaufrufen

```
1 void g1(string& s);
2 void f1(const string& s)
3 {
4    g1(s); // Compile-time Error
5           // g1, garantiert nicht das s
6           // unverändert bleibt.
7 }
```

Konstanten in Funktionsaufrufen

Lösung:

```
1 void g1(string& s) const;
2 void f1(const string& s)
3 {
4    g1(s);
5 }
```

Konstanten in Funktionsrückgaben

```
1 const double ValueInArray(int i)
2 {
3     return Array[i];
4 }
```

Variablenübergabe an Funktionen

call by value

```
1 void addiere(int a, int b)
2 {
3     int temp = a + b;
4     cout << temp;
5 }
```

Sinnvoll (schnell) bei allen eingbauten Datenypen.

Niemals bei großen Klassenobjekten verwenden, da sonst bei der Übergabe eine Kopie erstellt wird

Variablenübergabe an Funktionen

call by value

Änderungen an a und b werden nicht zurückübergeben

```
1  void addiere(int a, int b)
2  {
3      int temp = a + b;
4      cout << temp;
5      a = -1;           // siehe unten
6  }
7  ...
8  int wert = 5;
9  addiere(wert, 3);  // Ausgabe: 8;
10 cout << wert;      // Ausgabe: 5;
```

Variablenübergabe an Funktionen

call by reference

Änderungen an a und b werden zurückübergeben!

```
1 void addiere(int & a, int & b)
2 {
3    a = a + b;
4 }
5 ...
6 int wert = 5;
7 addiere(wert, 3);
8 cout << wert;       // Ausgabe: 8;
```

Sinnvoll nur wenn man Werte zurückgeben möchte.

Variablenübergabe an Funktionen

call by reference const

```
1  complex<double> addiere(
2     complex<double> const & a,
3     complex<double> const & b)
4  {
5     return a + b;
6  }
7  ...
8  complex<double> c1(5,3);  // 5+3i
9  complex<double> c2(2,-1); // 2-i
10 complex<double> c3 = addiere(c2, c3) // c3 = 7+2i
```

Werte werden schnell übergeben, dürfen aber nicht verändert werden. Sinnvoll immer, außer bei eingebauten Datentypen.

Variablenübergabe an Funktionen

Pointer Übergabe

```
1 void addiere(double * result,
2               double * A, double * B, int size)
3 {
4    for (int i=0; i<size, ++i)
5    {
6       result[i] = A[i] + B[i];
7    }
8 }
```

Pointer Übergabe notwendig bei C-Arrays, und allgemein sinnvoll bei großen Klassenobjekten.

Kompiliern, Linken, Dateiaufbau

Deklaration/Definition

Für jede Funktione/Klasse wird vor der Definition (was zu tun ist) eine Deklaration (wie aufzurufen) benötigt:

```
1 void doSomething(double input, int size); //Deklaration
2 ...
3 void doSomething(double input, int size)  //Definition
4 {
5 ...
6 }
```

Kompiliern, Linken, Dateiaufbau

Aufteilung in .h und .cpp Datei

somefile.h

```
1 void doSomething(double input, int size); //Deklaration
2 ...
```

somefile.cpp

```
1 #include "somefile.h"
2 void doSomething(double input, int size)  //Definition
3 {
4 ...
5 }
```

Kompiliern, Linken, Dateiaufbau

Beispiel:

mathfunctions.h

```
1 double sqr(double const x);
```

mathfunctions.cpp

```
1 double sqr(double const x) { return x*x; }
```

Kompiliern, Linken, Dateiaufbau

Beispiel:

mathfunctions.h

```
1 double sqr(double const x);
```

mathfunctions.cpp

```
1 double sqr(double const x) { return x*x; }
```

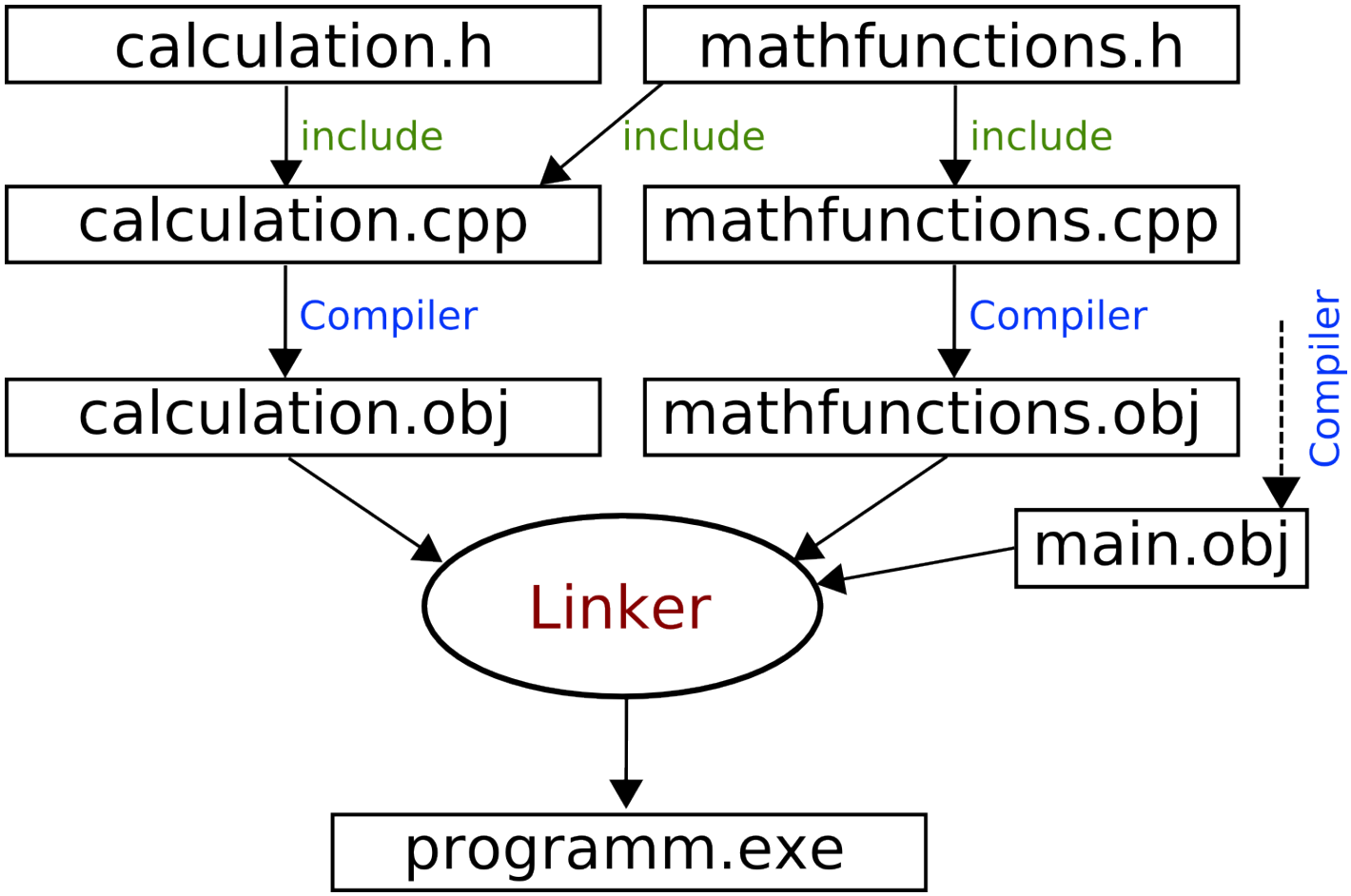
calculation.h

```
1 double calc(double const x);
```

calculation.cpp

```
1 #include "calculation.h"
2 #include "mathfunctions.h"
3 double calc(double const x) { return sqr(3.1414 * x); }
```

Kompiliern, Linken, Dateiaufbau



Kompiliern, Linken, Dateiaufbau

doppeltes Laden verhindern → Anpassen der .h Dateien

mathfunctions.h

```
1 #ifndef MATHFUNCTIONS_H
2 #define MATHFUNCTIONS_H
3
4 double sqr(double const x);
5
6 #endif
```

calculation.h

```
1 #ifndef CALCULATION_H
2 #define CALCULATION_H
3
4 #include "mathfunctions.h"
5 double calc(double const x);
6
7 #endif
```

Inlcude Dateien

System-Dateien

```
1 #include <string>  // C++ Datei
2 #include <stdio.h> // C Datei!
3 #include <cstdio>  // C++ Equivalent
```

Projekt-Dateien

```
1 #include "mathfunctions.h"
2 #include "DialogShowResults.h"
```

Guter Programmierstil

- Einheitlich - eine Struktur wählen und beibehalten
- Lesbar - Variablen eindeutige Namen geben

vergleiche:

```
N = T/dt;
Nd = N/SI;
```

mit

```
MaxIterationNumber = TotalTime/TimeIntervall;
DataSize = MaxIterationNumber / SaveIntervall;
```

Guter Programmierstil

- Einheitlich - eine Struktur wählen und beibehalten
- Lesbar - Variablen eindeutige Namen geben

selbst kleine Funktionen machen den Code lesbarer

```
sqrt((a.x - b.x)^2 + (a.y - b.y)^2);
```

ersetzen durch:

```
Distance(a,b); // Functionsaufruf
```

Guter Programmierstil

- Einheitlich - eine Struktur wählen und beibehalten
- Lesbar - Variablen eindeutige Namen geben
- nicht zu clever sein zu wollen

```
for (int i = 0; i < length; i++)  // ok
...
```

Guter Programmierstil

- Einheitlich - eine Struktur wählen und beibehalten
- Lesbar - Variablen eindeutige Namen geben
- nicht zu clever sein zu wollen

```
for (int i = 0; i < length; i++)  // ok
...
while (--length)                  // clever
...
```

Guter Programmierstil

- Einheitlich - eine Struktur wählen und beibehalten
- Lesbar - Variablen eindeutige Namen geben
- nicht zu clever sein zu wollen

```
for (int i = 0; i < length; i++)  // ok
...
while (--length)                  // clever
...
while (*t++ = *s++)               // ick!
...
```

Code Standards

hier nur kleiner Ausschnitt, siehe auch

- `http://www.possibility.com/Cpp/CppCodingStandard.html` (65 Seiten)
- `http://geosoft.no/development/cppstyle.html` (19 Seiten)

Code Standards

Variablen

`i, j, k`	Integer Schleifen Zähler (for Schleifen)
`n, len, length`	Integer Zahlen von einer Auflistung
`x, y`	Kartesische Koordinaten.
`average, height, numCrystals, pixelCount`	selbsterklärende Namensgebung
`isValid, hasData`	boolsche Variablen
`m_VariableName`	Member Variabeln
`variableName`	lokale Variabeln (Stack)

keine Abkürzungen verwenden!

Code Standards

Functionen

```
1 double CurrentTemperature();
2 bool isDataValid();
3 void perfomComputation();
4 void saveData();
5 void setMaximum(double max);
6 double Maximum();
```

Code Standards

Klassen Großbuchstaben zur Trennung:

```
1 class Matrix
2 class ModelockingLaser
3 class DialogSaveData
```

Code Standards

Klammern

```
1 if (a > 5) {
2   // This is K&R style
3 }
4
5 if (a > 5)
6 {
7   // This is ANSI C++ style
8 }
```

Kommentare

nutzlos:

```
1 int counter;                      /* declare a counter variable */
2 i = i + 1;                        /* add 1 to i */
3 while (index<length)...           /* while the index
4                                      is less than the length */
5 num = num + 3 - (num % 3); /* add 3 to num and
6                                      subtract num mod 3 */
```

Einfach Regel:

Erkläre was der Code macht, nicht was der Code selbst sagt.

Vermeide offensichtliche Aussagen.

```
1 num = num + 3 - (num % 3); /* increment num to
2                               the next multiple of 3 */
```

Kommentare

Formatierung (Doxygen)

```
1  /*!
2   * Class for storage of Complex Matrices
3   */
4  class ComplexMatrix
5  {
6   ...
7  private:
8     QSize m_Size; //!< Size of Matrix
9     complex<double> * m_ComplexMatrix; //!< Storage Array
10    ...
```

Kommentare

Formatierung (Doxygen)

```
1       ...
2       /*!
3        * Tests if Size is set.
4        * @param width
5        * @param height
6        * @see bool isSize(QSize size);
7        * @return test result
8        */
9       bool isSize(int width, int height);
10  ...
```

statische C-Arrays

können zur Laufzeit nicht verändert werden!

```
1  int Numbers[5]; // Array mit 5 Werten
2  Numbers[0] = 16;
3  Numbers[1] = 2;
4  ...
5  int Numbers2[] = { 16, 2, 77, 40, 12071 }; // einfacher
6
7  int Matrix[3][4];  // 2D-Array
8  Matrix[2][2] = 25;
9
10 Matrix[4][5] = 25; // possible CRASH
11                    // keine Überprüfung der Grenzen!
```

statische C-Arrays

```
1 int Numbers[2]; // Array mit 2 Werten
2 Numbers[0] = 16;
3 Numbers[1] = 2;
```

Numbers ist Pointer auf Start vom Array

```
1 int * Array = Numbers; // Kopie von Pointer
2                        // KEINE Kopie des Arrays!
3                        // Numbers = & Numbers[0]
4
5 cout << Array[1];      // Ausgabe: 2
```

dynamische C-Arrays

Deklaration

```
1  int size = width * height;
2  // Anfordern eines Arrays mit Größe size
3  // dynamische 2D C-Arrays gibt es nicht
4  double * Matrix = new Matrix[size];
5
6  // interpretieren als 2D Array
7  // entspricht Matrix[x][y]
8  Matrix[y + height * x] = 3.1;
9
10 // löschen, [] zum Löschen des Arrays!
11 delete [] Matrix
```

dynamische C++-Arrays

Deklaration

```
1  #include <vector>
2  using std::vector;
3
4  vector<double> Matrix; // vector Klasse
5  int size = width * height;
6  Matrix.resize(size);   // Größe zuweisen
7
8  if (!Matrix.empty())
9  {
10   Matrix[y + height * x] = 3.1;
11 }
12 //Am Ende hinzufügen
13 Matrix.push_back(5.4);
14 Matrix.push_back(6.6);
```

dynamische C++-Arrays

Deklaration

```
1  #include <vector>
2  using std::vector;
3
4  vector<vector<double> > Matrix; // 2D Array
5  int size = width * height;
6  Matrix.resize(width);   // Größe der 1. Dimension zuweisen
7
8  for (int i = 0; i < width; ++i)
9      Matrix[i].resize(height);
10
11 Matrix[x][y] = 3.1;
12
13 // löschen nicht notwendig
```

Funktionsübergabe von dynamischen Arrays

Deklaration

```
1 // C-Array, übergibt Pointer
2 void setMatrix(double * matrix);
3 // C++-Array, übergibt Kopie!
4 void setMatrix(vector<double> matrix);
5 // C++-Array, übergibt Referenz
6 void setMatrix(vector<double> & matrix);
7
8 // 2D C++-Array
9 void setMatrix(vector<vector<double> > & matrix);
```

dynamischen Arrays Kopieren

Deklaration

```
1  // C-Array -> C-Array
2  void setMatrix(double * matrix, int size) {
3     // unsicheres kopieren des vollständigen Speichers
4     memcpy(m_Matrix, matrix, size * sizeof(double));
5  }
6  // C++-Array -> C++-Array
7  void setMatrix(vector<double> & matrix)  {
8     m_Matrix = matrix; // vollständig
9  }
10 // C++-Array -> C-Array
11 void setMatrix(vector<double> & matrix)  {
12    memcpy(m_Matrix, &matrix[0], size * sizeof(double));
13 }
```

Klassen in C++

Zusätzlich zu den **vordefinierten Datentypen** (`int`, `double` usw.) kann man mit Klassen eigene Typen definieren.

Diese bezeichnet man als **Objekte**, da Klassen nicht nur **Daten** enthalten sondern auch **Methoden**, um diese zu verarbeiten und weiterzugeben.

Klassen in C++

Zusätzlich zu den **vordefinierten Datentypen** (`int`, `double` usw.) kann man mit Klassen eigene Typen definieren.

Diese bezeichnet man als **Objekte**, da Klassen nicht nur **Daten** enthalten sondern auch **Methoden**, um diese zu verarbeiten und weiterzugeben.

Kommende Beispiele:

- Matrix
- Auto
- MyArray
- Shapes (Rechteck und Kreis)
- Laser ?

Vorteile von Klassen

```
1  // C Beispiel zur Matrix Rechnung
2  double a[4]; // speichert 2x2 Matrix
3  double b[4];
4  double c[4];
5
6  c = MatrixMult(a,b); // c = a * b
7
8  // c = (a*b + c*a) * b
9  c = MatrixMult(MatrixAdd(MatrixMult(a,b),
10                 MatrixMult(c,a)), b);
```

Vorteile von Klassen

übersichtlicherer, einfacherer, lesbarer Code

```
1  // C++ Beispiel zur Matrix Rechnung
2  matrix a(2,2); // speichert 2x2 Matrix
3  matrix b(2,2);
4  matrix c(2,2);
5
6  c = a * b;
7  c = (a*b + c*a) * b;
8
9  // zum Vergleich:
10 // c = MatrixMult(MatrixAdd(MatrixMult(a,b),
11 //                  MatrixMult(c,a)), b);
```

Vorteile von Klassen

Darstellung von Hierarchien

```
1 double index = Laser.ActiveCrystal.RefractiveIndex();
```

Vorteile von Klassen

komplexe Objekte strukturiert darstellen

Beispiel eines Auto (Eigenschaften und Funktionen)

```
1  Auto sampleAuto;
2  sampleAuto.setMaxVelocity(220); // 200 km/h
3  sampleAuto.setLength(4.45);  // 4,45 m
4  sampleAuto.move(50);         // 50 m vorwärts
5
6  double velocity = sampleAuto.currentVelocity();
7
8  // 100 Autos in einem Array, alle ein exakte
9  // Kopie von 'sampleAuto'
10 vector<Auto> vieleAutos(100, sampleAuto);
```

Aufbau von Klassen

```
1  class Auto
2  {
3     Auto();  // Konstruktor
4     ~Auto(); // Destruktor
5     ...
6     double Velocity() // inline Member Function
7     {
8        return m_Velocity;
9     }
10    void setVelocity(double velocity);
11    void move();         // Member Functionen
12    ...
13    double m_Velocity;  // Member Variable
14 };
15
16 void Auto::move()
17 {
18    ...
19 }
```

Aufbau von Klassen

Eine Klassendefinition enthält den Bauplan und die Funktionsdefinitionen für eine Klasse.
Benutzen kann man diese erst nach eine Erstellung einer Instanz.

```
1 class Auto  // Klassendefinition
2 {
3     ...
4 };
5
6 Auto sampleAuto; // Erstellung einer Instanz
```

Aufbau von Klassen

Eine Klassendefinition enthält den Bauplan und die Funktionsdefinitionen für eine Klasse.
Benutzen kann man diese erst nach eine Erstellung einer Instanz.

```
1 class Auto  // Klassendefinition
2 {
3     ...
4 };
5
6 Auto sampleAuto; // Erstellung einer Instanz
```

Analog zu

1. Bauplan eines Hauses (Klassendefinition)
2. Betreten eines gebauten Hauses nach dem Bauplan (Nutzung einer Klasseninstanz)

Klassen mit Pointerzugriff

Klassen als normale Variabeln

```
1 Auto sampleAuto;
2 sampleAuto.setVelocity(100);
```

Zugriff auf Member über 'Punkt'.

Klassen mit Pointerzugriff

Klassen als normale Variabeln

```
1 Auto sampleAuto;
2 sampleAuto.setVelocity(100);
```

Zugriff auf Member über 'Punkt'.

Klassen als Pointer

```
1 Auto * sampleAuto = new Auto;
2 Auto->setVelocity(100);
```

Zugriff auf Member über 'Pfeil'.

Äquivalent zu

```
1 (*sampleAuto).setVelocity(100);
```

Zugriffseinschränkungen (public, private)

```
1  class Auto
2  {
3     MotorEngine m_Engine;
4     void startEngine()
5     {
6        if (!m_isMoving)
7           m_Engine.start();
8     }
9     void stopEngine()
10    {
11       if (!m_isStopped)
12          m_Engine.stop();
13    }
14    void move() { ... }
15 };
```

Zugriffseinschränkungen (public, private)

```
1  class Auto
2  {
3     MotorEngine m_Engine;
4     void startEngine()
5     {
6        if (!m_isMoving)
7           m_Engine.start();
8     }
9     void stopEngine()
10    {
11       if (!m_isStopped)
12          m_Engine.stop();
13    }
14    void move() { ... }
15 };
```

```
1 Auto sampleAuto;
2 sampleAuto.startEngine();
3 sampleAuto.move();
4 sampleAuto.m_Engine.stop(); // ups
```

Zugriffseinschränkungen (public, private)

```
1  class Auto
2  {
3  private:
4     MotorEngine m_Engine;
5     ...
6  public:
7     void move() { ... }
8     void startEngine() {
9        if (!m_isMoving)
10          m_Engine.start();
11    }
12    void stopEngine()  {
13       if (!m_isStopped)
14          m_Engine.stop();
15    }
16 };
```

Zugriffseinschränkungen (public, private)

```
1  class Auto
2  {
3  private:
4     MotorEngine m_Engine;
5     ...
6  public:
7     void move() { ... }
8     void startEngine() {
9        if (!m_isMoving)
10          m_Engine.start();
11    }
12    void stopEngine()  {
13       if (!m_isStopped)
14          m_Engine.stop();
15    }
16 };
```

```
1 Auto sampleAuto;
2 sampleAuto.startEngine();
3 sampleAuto.move();
4 // Compiliert nicht!
5 sampleAuto.m_Engine.stop();
```

Private Variablen und Methoden können nur von Methoden aus der Klasse genutzt werden.

Zugriffseinschränkungen (public, private)

Interne Variablen mit `private` schützen.

Öffentlich zugänglich machen mit set und get Funktionen.
Zuweisung dabei auf Gülitgkeit überprüfen

```
1  class Auto
2  {
3  private:
4     double m_Velocity;
5  public:
6     void setVelocity(double velocity)
7     {
8        if ((velocity > 0) && (velocity < m_MaxVelocity))
9        {
10          m_Velocity = velocity;
11       }
12    }
13    double Velocity()
14    {
15       return m_Velocity;
16    }
17 };
```

Konstruktor, Destruktor

```
1  class MyArray
2  {
3      // Konstruktor
4      MyArray(int size){
5          m_size = size;
6          m_Array = new double[m_size];
7      }
8      // Destruktor
9      ~MyArray(){
10         delete [] m_Array;
11     }
12     private:
13         int m_size;
14         double * m_Array;
15     ...
16 }
```

Zuweisung und Kopie

Zuweisung

```
1 double dValue;
2 dValue = 5.6; // Zuweisung zu einem 'existierenden' Objekt
3
4 LaserCrystal crystalNew;
5 crystalNew = crystalOld;
```

Zuweisung und Kopie

Zuweisung

```
1 double dValue;
2 dValue = 5.6; // Zuweisung zu einem 'existierenden' Objekt
3
4 LaserCrystal crystalNew;
5 crystalNew = crystalOld;
```

Kopie erstellen

```
1 double dValue(5.6);
2 //dasselbe wie
3 double dValue = 5.6; // Zuweisung zu einem 'neuen' Objekt
4
5 LaserCrystal crystalNew(crystalOld);
6 //dasselbe wie
7 LaserCrystal crystalNew = crystalOld;
```

Copy-Konstruktor, Zuweisungs-Operator

Wie kopiert oder weist man eine Klasse einem neuen Objekt zu?

Automatisch generierte Funktionen:

```
1  class MyArray
2  {
3      MyArray(int size);    // Konstruktor
4      ~MyArray();           // Destruktor
5      // Copy Constructor
6      MyArray(const MyArray& c);
7      // Assignment Operator
8      MyArray& operator=(const MyArray& c);
9      ...
10 };
```

kopiert alle Member Variablen, die einen Zuweisungsoperator kennen. Das gilt insbesondere für Pointer nicht!

Copy-Konstruktor, Zuweisungs-Operator

Beispielimplementierung (MyArray)

```
1  // MyArray arrayNew(arrayOld);
2  MyArray::MyArray(const MyArray& c){ // Copy Constructor
3     this->m_size = c.m_size;
4     m_Array = new double[m_size];
5     for(int i=0; i< m_size; ++i)
6        this->m_Array[i] = c.m_Array[i];
7  }
8  // arrayNew = arrayOld; // arrayNew.operator=(arrayOld);
9  MyArray::MyArray& operator=(const MyClass& c){ // Assignment Op.
10    if(this != &c) { // never assign to yourself
11       this->m_size = c.m_size;
12       m_Array = new double[m_size];
13       for(int i=0; i< m_size; ++i)
14          this->m_Array[i] = c.m_Array[i];
15    }
16    return *this; // return reference (allows a = b = 3)
17 }
```

Initialisierungs Listen

```
1  class SomeDataSaver
2  {
3      // Konstruktor
4      SomeDataSaver( HugeClass & huge)
5      {
6          m_Huge = huge;
7          // ist intern identisch mit
8          // HugeClass m_Huge; // sehr aufwendig und unnötig
9          // m_Huge = huge;
10     }
11 }
```

```
1  class SomeDataSaver
2  {
3      // Konstruktor
4      SomeDataSaver( HugeClass & huge) : m_Huge(huge)
5      {
6      }
7  }
```

Member Functionen

Welche Functionen müssen zur Klasse gehören, welche nicht?

```
1 complex c(1,-3);    // c = 1 - 3i
2 // Member Function
3 double r = c.real() // r = 1
4
5 // Keine Member Function
6 complex c2 = sqr(c);  // nicht c2 = c.sqr();
```

➡ so wenig wie möglich!

Shape Klassen

```
1  class CircShape
2  {
3  public:
4      CircShape();
5      ~CircShape();
6      CircShape(const CircShape & s);
7      const CircShape& operator=(const CircShape& rhs);
8
9      void setRadius(double radius);
10     double radius();
11
12 private:
13     double m_radius;
14 };
```

Shape Klassen

```
1  class RectShape
2  {
3  public:
4      RectShape();
5      ~RectShape();
6      RectShape(const RectShape & s);
7      const RectShape& operator=(const RectShape& rhs);
8
9      void setSize(double width, double height);
10
11     void setWidth(double width);
12     double width() const;
13
14     void setHeight(double height);
15     double height() const;
16
17 private:
18     double m_width;
19     double m_height;
20 };
```

Ableiten von Klassen

Man kann Klassen erweitern (Werte/Methoden), indem man sie 'ableitet'

Eine neue Klasse wird immer von einer allgemeineren Klasse abgeleitet.

Angestellter

vorname
nachname

Ableiten von Klassen

Man kann Klassen erweitern (Werte/Methoden), indem man sie 'ableitet'

Eine neue Klasse wird immer von einer allgemeineren Klasse abgeleitet.

Angestellter

vorname
nachname

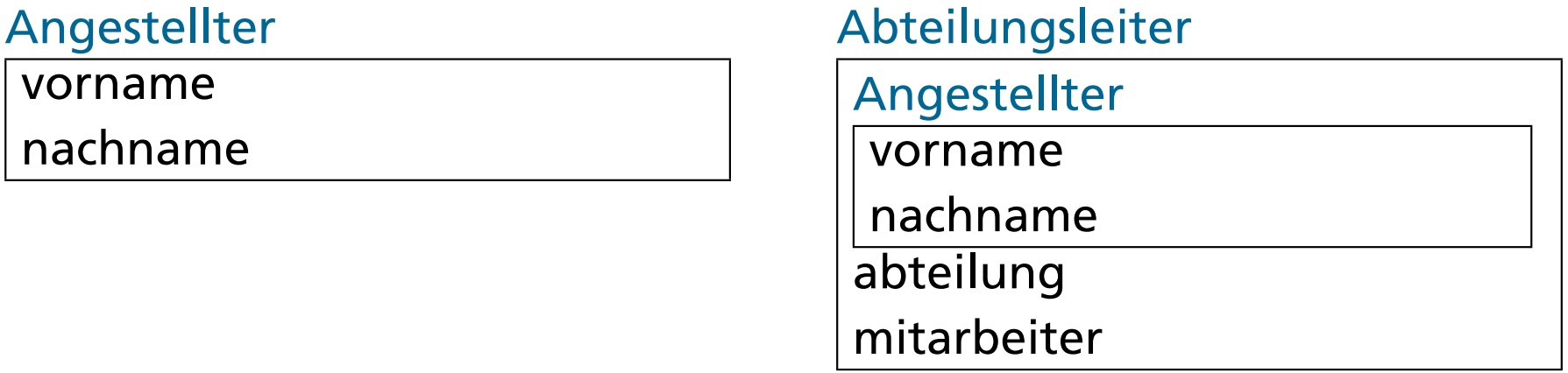
Abteilungsleiter

vorname
nachname
abteilung
mitarbeiter

Ableiten von Klassen

Man kann Klassen erweitern (Werte/Methoden), indem man sie 'ableitet'

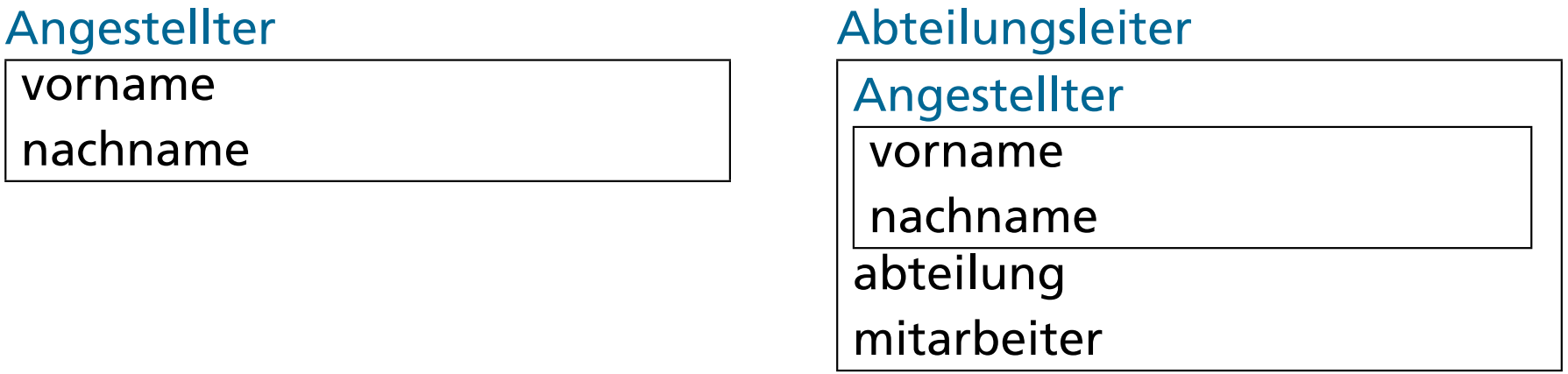
Eine neue Klasse wird immer von einer allgemeineren Klasse abgeleitet.



Ableiten von Klassen

Man kann Klassen erweitern (Werte/Methoden), indem man sie 'ableitet'

Eine neue Klasse wird immer von einer allgemeineren Klasse abgeleitet.



Abteilungsleiter 'ist-ein' Angestellter.

➡ Abteilungsleiter von Angestellter ableiten.

Ableiten von Klassen

```
1  class Angestellter
2  {
3     string vorname, nachname;
4  };
5
6  class Abteilungsleiter : public Angestellter
7  {
8     string abteilung;
9     vector<Angestellter> mitarbeiter;
10 };
```

Mehrfaches Ableiten

Polizeiauto → Auto → Fahrzeug
Krankenwagen → Auto → Fahrzeug

Laster → Fahrzeug

AbstractShape - Abstrakte Klassen

```
1  class AbstractShape
2  {
3  public:
4      // pure virtual function: muss von Ableitender
5      // Klasse implementiert werden
6      virtual double area () const = 0;
7
8      // Vergleichsoperator
9      bool operator<(const AbstractShape& rhs)  {
10         return (this->area() < rhs.area());
11     }
12 };
```

AbstractShape - Abstrakte Klassen

```
1  class AbstractShape
2  {
3  public:
4     AbstractShape(){}
5     // virtual function: kann von Ableitender Klasse
6     // überschrieben werden
7     virtual ~AbstractShape() {}
8     virtual double area () const = 0;
9     bool operator<(const AbstractShape& rhs)  {
10        return (this->area() < rhs.area());
11    }
12    // zusätzliche Beschreibung
13    void setDescription(std::string const & str)  {
14       m_description = str;
15    }
16 private:
17    std::string m_description;
18 };
```

CircShape

Ableiten von 'AbstractShape'

```
1  class CircShape : public AbstractShape
2  {
3  public:
4      CircShape();
5      ~CircShape();
6      CircShape(const CircShape & s);
7      const CircShape& operator=(const CircShape& rhs);
8
9      void setRadius(double radius);
10     double radius();
11
12     // Implementierung der virtuellen Funktion
13     // aus AbstractShape
14     double area () const;
15
16 private:
17     double m_radius;
18 };
```

Operatoren

Beispiele: +, -, *, /, <, >, !, (), ++, –, -=, +=, *=, usw.

```
1 matrix a(2,2);
2 matrix b(2,2);
3 matrix c(2,2);
4
5 c = a + b;
6 // identisch mit
7 c.operator+(a,b);
```

Operatoren sind normale Funktionen, die man für eine Klasse definieren kann.

Matrix Klasse (unvollständig)

```
1  class matrix
2  {
3  public:
4     matrix(int row_count, int col_count);
5     matrix(const matrix &s);
6     ~matrix();
7
8     matrix & operator =(const matrix &s);
9     double & operator()(int row, int col);
10    matrix & operator+=(const matrix& rhs);
11    friend const matrix operator*(const double & x, const matrix& m);
12
13 private:
14    double * m_Matrix;
15    int m_rows, m_cols;
16
17    int ArrPos(int row, int col) const;
18 };
19
20 const matrix operator+(const matrix & lhs, const matrix & rhs);
```

Matrix Element Operator

```
1  // matrix M(4,4);
2  // double element = M(2,3);
3  double & matrix::operator()(int row, int col)
4  {
5     return m_Matrix[ArrPos(row, col)];
6  }
7
8  int matrix::ArrPos(int row, int col) const
9  {
10 // y + height * x (y=row, x=col)
11 return (row + m_rows*col);
12 }
```

Matrix Operator +=

```
1  // matrix a,b;
2  // b+=a
3  // b.operator+=(a);
4  matrix & matrix::operator+=(const matrix& rhs)
5  {
6     for(int i=0; i<rhs.m_rows; ++i)
7     {
8        for(int j=0; j<rhs.m_cols; ++j)
9        {
10          m_Matrix[ArrPos(i,j)] += rhs(i,j);
11       }
12    }
13    // return reference
14    return *this; // allows c+=(b+=a)
15 }
```

Memberfunction der Klasse 'matrix' - greift auf Member Variable 'm_Matrix' zurück

Matrix Operator +

```
1 // matrix a,b;
2 // b = b + a;
3 // b = operator+(b,a);
4 const matrix operator+(const matrix & lhs, const matrix & rhs)
5 {
6    matrix ans(lhs);
7    ans+=rhs;
8    return ans;
9 }
```

keine Memberfunction der Klasse 'matrix'!

Matrix Scalar Multiplication

```
1  // matrix a,b;
2  // b = 3.0 * a;
3  // b = operator*(3.0,a);
4  const matrix operator*(const double & x, const matrix& m)
5  {
6      matrix ans(m.m_rows, m.m_cols);
7      for(int i=0; i<m.m_rows; ++i)
8      {
9          for(int j=0; j<m.m_cols; ++j)
10         {
11             ans(i,j) = x * m(i,j);
12         }
13     }
14     return ans;
15 }
```

Keine Memberfunction - benötigt aber trotzdem Zugriff auf Membervariablen (m_rows, m_cols).

➡ Function als friend der Klasse definieren.

Matrix Scalar Multiplication - friend Funktion

```
1  class matrix
2  {
3  public:
4     friend const matrix operator*(const double & x, const matrix&
           m);
5     ...
6  };
7
8  const matrix operator*(const double & x, const matrix& m)
9  {
10    ...
11 }
```

Deklaration als friend in der Klasse.