# ProgCPP_ZF FS18

N. Kaelin

23. März 2018

## Inhaltsverzeichnis

# 1 Kapitel 1

## 1.1 Einführung

### 1.1.1 Charakteristiken von C++

- C++ erlaubt sowohl prozedurale, objektorientierte als auch generische Programmierung Achtung: nicht jedes C++-Programm ist objektorientiert
- C++ ist sehr mächtig
- C++ ist eine Obermenge von C
- Syntaktisch ist C++ sehr ähnlich oder identisch zu C
- C++ ist sicherer als C

### 1.1.2 Entstehung von C++

- C Ritchie 1971, typisiert
- ANSI C seit 1983
- C++ Stroustrup 1986, Klassen und OO
- Final Standard ISO/ANSI ab 1990

  **ISO/IEC 14882:2003 Programming Languages - C++, aka C++03**

  **ISO/IEC 14882:2011 Programming Languages - C++, aka C++11**

  **ISO/IEC 14882:2014 Programming Languages - C++, aka C++14**

### 1.1.3 Welches C++?

- **Im Modul ProgCPP setzen wir den Standard 14882:2003, d.h. C++03 ein**
- Wieso nicht C++11 oder C++14?
  - Der neue Standard hat einige interessante Erneuerungen zu bieten, die C++ noch näher an C# kommen lassen
  - Durch diese neuen Features wird die Sprache leider nicht einfacher sondern umfangreicher und komplizierter
  - Der Nutzen von einzelnen Neuerungen ist m.E. fraglich
  - Bei Embedded Systems wird heute noch mehrheitlich C verwendet. Diejenigen, die C++ einsetzen, nehmen C++03
  - Ich bin gespannt, ob und wann C++11 im Embedded-Bereich den Durchbruch schafft

### 1.1.4 C++-Unterstützung von Texas Instruments (TI)

- TI als Repräsentant für einen Anbieter von Embedded-Entwicklungsumgebungen
- The TI compilers for all devices support
  - C++98 (ISO/IEC 14882:1998)
  - C++03 (this is a bug fix update to C++98)
- The TI compiler does not support
  - C++ TR1
  - C++11 (ISO/IEC 14882:2011)
- Noch neuere Versionen erst recht nicht

### 1.1.5 Hello World!

./listings/hello.cpp

```
// Datei: hello.cpp
// Hello World
// N. Kaelin, 02.03.2018

#include <iostream>
using namespace std;

int main(void)
{
        cout << "Hello World!" << endl;
        return 0;
}
```

### 1.1.6 C++-Compiler (noch nicht Eclipse)

Statt **gcc** (für C-Programme) muss **g++** (für C++-Programme) oder clang++ verwendet werden

**g++ -o hello hello.cpp**
**clang++ -o hello hello.cpp**

## 1.2 Lexikalische Elemente von C++

### 1.2.1 Lexikalische Elemente

- C++ ist hier völlig identisch zu C
  - Bezeichner
  - Schlüsselwörter (werden ergänzt durch zusätzliche **(**REFERENZ)
  - Literale
  - Operatoren (werden ergänzt durch zusätzliche)
  - Kommentare
- Codierstil beachten
  Die Ellemtel-Richtlinien haben sich etabliert **(**siehe Anhang)

### 1.2.2 Styleguide: Bezeichner (-Namen)

- Variablen, Konstanten und Objekte
  - mit Kleinbuchstaben beginnen
  - erster Buchstaben von zusammengesetzten Wörtern ist gross (mixed case)
  - keine Underscores

  Beispiele: counter, maxSpeed

- Funktionen
  - mit Kleinbuchstaben beginnen
  - erster Buchstaben von zusammengesetzten Wörtern ist gross (mixed case)
  - Namen beschreiben Tätigkeiten
  - keine Underscores

  Beispiele: getCount(), init(), setMaxSpeed()

- Klassen, Strukturen, Enums
  - mit Grossbuchstaben beginnen
  - erster Buchstaben von zusammengesetzten Wörtern ist gross (mixed case)

- keine Underscores

Beispiele: MotorController, Queue, Color

## 1.3 Typkonzept

### 1.3.1 Datentypen

- In C++ gibt es die gleichen Basisdatentypen (plain old data types, POD types) wie in C
- Zusätzlich: es gibt einen Typ für boole'sche Werte: **bool**
  (Kap. 3.4.5, 4.3.1)
- Der Typ **bool** hat die beiden Werte **true** und **false**

### 1.3.2 #define (Kap. 4.5)

- In C oft noch geduldet, in C++ verpönt
- *#define* bewirkt eine reine Textersetzung durch den Präprozessor, umgeht dadurch Syntax- und Typprüfung
- Für die Definition von symbolischen Konstanten soll statt *#define* das Schlüsselwort **const** oder **enum** verwendet werden
- schlecht:
  #define PI 3.14159
  #define VERSUCHE_MAX 4
- gut:
  const double pi = 3.14159;
  const int versucheMax = 4;
  enum {versucheMax = 4}; // noch besser als const int

## 1.4 Ausdrücke und Operatoren

### 1.4.1 Ausdrücke und Operatoren

- C++ ist hier völlig identisch zu C
- Es gibt noch ein paar zusätzliche Operatoren in C++ (z.B. für type casting)

## 1.5 Anweisungen

### 1.5.1 Anweisungen: ebenfalls nichts neues

- Die Syntax der Blockanweisung ist identisch zu C
- Die Syntax der Steuerstrukturen ist identisch zu C
  - Sequenz
  - Iteration (for, while, do ... while)
  - Selektion (if, if ... else, switch)
- Sprunganweisungen sind auch identisch zu C
  - break
  - continue
  - return
  - goto → nicht verwenden!!!

## 1.6 Streams

### 1.6.1 Streamkonzept

- Ein Stream repräsentiert einen sequentiellen Datenstrom
- Die Operatoren auf dem Stream sind « und »
  Für vordefinierte Datentypen sind diese Operatoren schon definiert, für eigene selbstdefinierte Klassen können diese Operatoren überladen werden **(KAPITEL ANGEBEN)**
- C++ stellt 4 Standardströme zur Verfügung
  - cin: Standard-Eingabestrom, normalerweise die Tastatur
  - cout: Standard-Ausgabestrom, normalerweise der Bildschirm
  - cerr: Standard-Fehlerausgabestrom, normalerweise der Bildschirm
  - clog: mit *cerr* gekoppelt

- Alle diese Ströme können auch mit einer Datei verbunden werden

### 1.6.2 Einsatz von Streams

- In C werden printf() und scanf() mit stdin, stdout, stderr verwendet
- printf() und scanf() könnten in C++ immer noch verwendet werden. Dies soll jedoch im Normalfall vermieden werden.
- C++ bietet analog die Streams cin, cout, cerr an
- Der Zugriff auf cin und cout ist einfacher und komfortabler als die Verwendung von printf() und scanf()
- cin und cout müssen immer ganz links in einer Befehlszeile stehen. Die Daten kommen von cin (Tastatur, Operator ››) und gehen zu cout (Konsole, Operator ‹‹)

### 1.6.3 Ausgabe: Klasse ostream

- Methoden für die Ausgabe von vordefinierten Datentypen, z.B.:
  ostream& operator ‹‹(int n);
  ostream& operator ‹‹(double d);
  ostream& operator ‹‹(char c);
- Weitere Klassen können diesen Operator überschreiben, z.B.:
  ostream& operator ‹‹(std::string str);
- Nutzung mit cout (vordefiniertes Objekt der Klasse ostream):
  int i = 45;
  cout ‹‹“Hallo “‹‹i ‹‹endl;

### 1.6.4 Eingabe: Klasse istream

- Methoden für die Eingabe von vordefinierten Datentypen, z.B.:
  istream& operator ››(int& n);
  istream& operator ››(double& d);
  istream& operator ››(char& c);
- Weitere Klassen können diesen Operator überschreiben z.B.:
  istream& operator ››(std::string& str);
- Nutzung mit cin (vordefiniertes Objekt der Klasse istream):
  double d;
  string str;
  cin ››d ››str;

### 1.6.5 Formatierte Ein- und Ausgabe

ios, eine Basisklasse von iostream, stellt verschiedene Möglichkeiten (Format Flags) vor, um die Ein- und Ausgabe zu beeinflussen.

Beispiel:

cout ‹‹showbase ‹‹hex ‹‹27; // Ausgabe: 0x1b

### 1.6.6 Format-Flags um Überblick (unvollständig)

| Flag | Wirkung |
|---|---|
| boolalpha | bool-Werte werden textuell ausgegeben |
| dec | Ausgabe erfolgt dezimal |
| fixed | Gleitkommazahlen im Fixpunktformat |
| hex | Ausgabe erfolgt hexadezimal |
| internal | Ausgabe innerhalb Feld |
| left | linksbündig |
| oct | Ausgabe erfolgt oktal |
| right | rechtsbündig |
| scientific | Gleitkommazahl wissenschaftlich (Mantisse und Exponent) |
| showbase | Zahlenbasis wird angezeigt |
| showpoint | Dezimalpunkt wird immer ausgegeben |
| showpos | Vorzeichen bei positiven Zahlen anzeigen |
| skipws | Führende Whitespaces nicht anzeigen |
| unitbuf | Leert Buffer des Outputstreams nach Schreiben |
| uppercase | Alle Kleinbuchstaben in Grossbuchstaben wandeln |

# 2 Kapitel 2: Funktionen

## 2.1 Grundlegendes

### 2.1.1 Synonyme für Funktionen

- Unterprogramm
- Subroutine
- Prozedur (Funktion ohne Rückgabewert)
- Methode (in der Objektorientierten Programmierung)

### 2.1.2 Aufgabe einer Funktion

- Gleichartige, funktional zusammengehörende Programmteile unter einem eigenene Namen zusammenfassen. Der Programmteil kann mit diesem Namen aufgerufen werden.
- Einige Funktionen (im speziellen mathematische) sollen parametrisiert werden können, z.B. die Cosinusfunktion macht nur Sinn, wenn sie mit unterschiedlichen Argumenten aufgerufen werden kann.
- Divide et impera (divide and conquer, teile und herrsche):
  Ein grosses Problem ist einfacher zu lösen, wenn es in mehrere einfachere Teilprobleme aufgeteilt wird.

### 2.1.3 Funktionen (Vergleich zu C)

- Alles was in C möglich ist, gibt es auch in C++
- Einige Punkte sind in C++ zusätzlich eingeführt worden:
  - Operatorfunktion (Spezialität von C++, folgt später)
  - inline-Funktion
  - Vorbelegung von Parametern (default-Argumente)
  - Überladen von Funktionen (overloading)

### 2.1.4 Definition von Funktionen

- Funktionskopf
  - legt die Aufrufschnittstelle (Signatur) der Funktion fest
  - besteht aus:
    - Rückgabetyp
    - Funktionsname (fast beliebig wählbar)
    - Parameterliste
- Funktionsrumpf
  - Lokale Vereinbarungen und Anweisungen innerhalb eines Blocks

### 2.1.5 Deklaration von Funktionen (Funktionsprototypen)

./listings/fktproto.cpp

```
// Datei: fktproto.cpp
// Deklaration von Funktionen
// N. Kaelin, 02.03.2018

#include <iostream>
using namespace std;

void init(int* alpha);  // das ist der Funktionsprototyp (Deklaration)

int main(void)
{
        int a;
        init(&a);
        cout << "Der Wert von a ist " << a << endl;
        return 0;
}
```

```
void init(int* alpha)
{
        *alpha = 10;
}
```

#### 2.1.6 Kosten einer Funktion

- Der Code einer Funktion ist nur einmal im Speicher vorhanden
  - Vorteil: spart Speicher
- Der Aufruf einer Funktion bewirkt eine zeitliche Einbusseim Vergleich zu einer direkten Befehlsausführung
  - Nachteil: Zeitverlust, Overhead

### 2.2 C-Makro

#### 2.2.1 C-Makro mit #define

- C-Makros bewirken eine reine Textersetzung ohne jegliche Typenprüfung
- Bei Nebeneffekten (welche zwar vermieden werden sollten) verhalten sich Makros oft nicht wie beabsichtigt

---

**Achtung**: C-Makros lösen zwar das Problem mit dem Overhead, sind aber sehr unsicher. Bitte nicht einsetzen!

---

#### 2.2.2 Beispiel mit C-Makro: Maximum zweier int-Werte

```
\#define MAX(a,b)        ((a)>(b) ? (a) : (b))

int z1 = 4;
int z2 = 6;
int m = MAX(z1,z2);
 wird expandiert zu: m = ((z1)‹(z2) ? (z1) : (z2)); // m=6, z1=4, z2=6
m = MAX(++z1,++z2);
 erwartet wird: m=7, z1=5, z2=7
```

#### 2.2.3 Beispiel mit C-Makro: Was passiert wirklich?

```
m = MAX(++z1,++z2);
wird expandiert zu:
 m = ((++z1)‹(++z2) ? (++z1) : (++z2)); m = ((5)‹(7) ? (++z1) : (8));
// z2 wird zweimal inkrementiert!
// m=8, z1=5 ,z2=8
erwartet wird:
m=7, z1=5, z2=7
```

### 2.3 inline-Funktionen

#### 2.3.1 inline-Funktionen: Grundlegendes

- Lösen das Overhead-Problem
  - Code wird direkt eingefügt, kein Funktionsaufruf
- Typenprüfung findet statt
- Einsetzen wenn der Codeumfang der Funktion sehr klein ist und die Funktion häufig aufgerufen wird (z.B. in Schleifen)

- Achtung: Rekursive Funktionen und Funktionen, auf die mit einem Funktionspointer gezeigt wird, werden nicht inlined.

### 2.3.2 Beispiel mit inline-Code: Maximum zweier int-Werte

```
inline int max(int a, int b)
{
        return a \flq b ? a : b;
}

int main()
{
        int z1 = 4;
        int z2 = 6;
        int m = max(z1, z2);
        // m=6, z1=4, z2=6
        m = max(++z1, ++z2);
        // m=7, z1=5, z2=7
}
```

## 2.4 Grundsätze für Optimierungen

#1: Optimize: don't do it
#2: If you have to do it: do it later

## 2.5 default-Argumente

### 2.5.1 Vorbelegte Parameter (default-Argumente)

```
void prtDate(int day=1, int month=3, int year=2009);
```

- Parametern können im Funktionsprototypen (bitte nur dort!) Defaultwerte zugewiesen werden.
- Beim Funktionsaufruf können (aber müssen nicht) die Parameter mit default-Werten weggelassen werden

---

**Achtung**: Hinter (rechts von) einem default-Argument darf kein nicht vorbelegter Parameter mehr folgen, d.h. wenn bei einem Parameter ein default definiert wird, dann müssen bei allen weiteren Parametern dieser Funktion ebenfalls defaults definiert werden.

---

- Grund: Die Parameterübergabe erfolgt in C++ von links nach rechts

### 2.5.2 Beispiel: default-Argumente

```
void prtDate(int day=1, int month=3, int year=2009);

// erlaubt sind z.B. die folgenden Aufrufe:
prtDate();                          // 1-3-2009
prtDate(23);                // 23-3-2009
prtDate(15,6);              // 15-6-2009
prtDate(24,7,2012);         // 24-7-2012



// nicht erlaubt sind z.B. diese Deklarationen:
void prtDate2(int day=7, int month, int year=2009);
void prtDate3(int day, int month=3, int year);
```

### 2.5.3 Nutzen von default-Argumenten

- Wenn in einer bereits existierenden Funktion neue Argumente aufgenommen werden müssen, dann:
  - Neue Argumente hinten als default-Argumente anfügen
  - Die bereits bestehenden alten Aufrufe (mit weniger Argumenten) können unverändert beibehalten werden
  - Die Implementation der Funktion muss angepasst werden
- Sehr nützlich z.B. bei Konstruktoren in der objektorientierten Programmierung

## 2.6 Overloading

### 2.6.1 Überladen von Funktionen (overloading)

- Zweck:
  Eine Funktion sollte allenfalls mit unterschiedlichen Parametern aufgerufen werden können

  void print(char ch);
  void print(int i);
  void print(double d);
- Alternative (in C) wäre:
  void printChar(char ch);
  void printInt(int i);
  void printDouble(double d);
  Ist umständlicher und unverständlicher

### 2.6.2 Overloading in C++

- Die Identifikation einer Funktion erfolgt über die Signatur, nicht nur über den Namen
  - Die Signatur besteht aus:
    Name der Funktion **plus** die Parameterliste (Reihenfolge, Anzahl, Typ)
    (Der Returntyp wird nicht berücksichtigt)
- Der Name der Funktion ist identisch
- Die Implementation muss für jede überladene Funktion separat erfolgen

---

**Hinweis**: Overloading sollte zurückhaltend eingesetzt werden. Wenn möglich sind default-Argumente vorzuziehen.

---

### 2.6.3 Deklaration von überladenen Funktionen: Regeln

- Entsprechen Rückgabetyp und Parameterliste der zweiten Deklaration denen der ersten, so wird die zweite als gültige Re-Deklaration der ersten aufgefasst.
- Unterscheiden sich die beiden Deklarationen nur bezüglich ihrer Rückgabetypen, so behandelt der Compiler die zweite Deklaration als fehlerhafte Re-Deklaration der ersten.
  Der Rückgabetyp von Funktionen kann nicht als Unterscheidungskriterium verwendet werden.
- Nur wenn beide Deklarationen sich in Anzahl oder Typ ihrer Parameter unterscheiden, werden sie als zwei verschiedene Deklarationen mit demselben Funktionsnamen betrachtet (überladene Funktionen).

### 2.6.4 Funktionen sollen nur dann überladen werden, wenn ...

- **die Funktionen eine vergleichbare Operation bezeichnen, die jeweils mit anderen Parametertypen ausgeführt wird**
- **dieselbe Wirkung nicht durch default-Parameter erreicht werden kann**

## 2.7 default-Parameter vs. Overloading

```
// Variante mit Overloading
// 3 unterschiedliche Funktionen belegen Speicher
// 3 unterschiedliche Funktionenmüssen gewartet werden
void print(int i);
void print(int i, int width);
void print(int i, char fillchar, int width):


// Variante mit default-Parametern
// Eine einzige Funktion belegt Speicher
// Nur eine Funktion muss gewartet werden
void print(int i, int width=0, char fillchar=0);
```

---

**Achtung**: keinesfalls default-Parameter in überladenen Funktionen verwenden

---

# 3 Kapitel 3: Pointer und Referenzen

## 3.1 Höhere und strukturierte Datentypen

### 3.1.1 Höhere Datentypen

- Pointer
- Referenzen
- Vektoren

### 3.1.2 Strukturierte Datentypen

- Strukturen
- Klassen

## 3.2 Pointer

### 3.2.1 Adresse

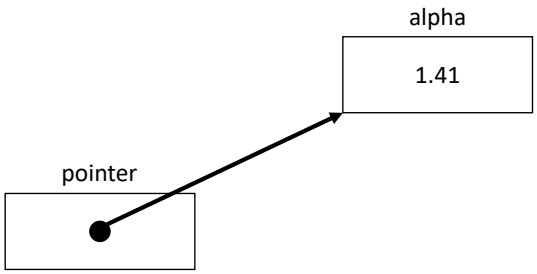
- Die Nummereiner Speicherzelle wird als **Adresse** bezeichnet
- Bei einem byteweise adressierbaren Speicher(ist üblich) liegt an jeder Adresse genau 1 Byte

### 3.2.2 Pointer

- Synonym: Zeiger
- Ein Pointer ist eine Variable, welche die Adresse einer im Speicher befindlichen Variablen oder Funktion aufnehmen kann
- Man sagt, der Pointer zeige (to point) auf diese Speicherzelle
- Pointer in C++ sind zu 99.99% identisch zu Pointern in C

### 3.2.3 Standarddarstellung von Pointern

```
float alpha;
float* pointer;
alpha = 1.4f;
pointer = \&alpha;
```




### 3.2.4 Pointer und Datentyp

- Pointer in C++ sind typisiert (wie in C), sie zeigen auf eine Variable des definierten Typs
- Oder anders ausgedrückt:
  Der Speicherbereich, auf den ein bestimmter Pointer zeigt, wird entsprechend des definierten Pointer-Typs interpretiert
- Der Speicherbedarf einer Pointervariablen ist unabhängig vom Pointer-Typ. Er ist so gross, dass die maximale Adresse Platz findet
  (z.B. 32 Bits für $2^{32}$ Adressen)

### 3.2.5 Definition einer Pointervariablen

```
// Datentyp des Pointers Kennzeichnung des Pointers durch '*'
Typname* pointerName;

// Konkrete Beispiele:

int* ptr1;              // ptr1 ist ein Pointer auf int
double* ptr2;   // ptr2 ist ein Pointer auf double
```


### 3.2.6 Initialisierung mit Null-Pointer

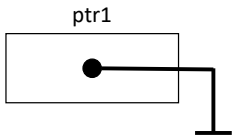
Mit dem Null-Pointer wird angezeigt, dass der Pointer auf **kein** Objekt zeigt. Dem Pointer wird ein definierter Nullwert zugewiesen.

---

**Hinweis**: Der Pointer zeigt nicht auf die Adresse 0!

---

```
int* ptr = 0;   // bitte nicht NULL verwenden!
```



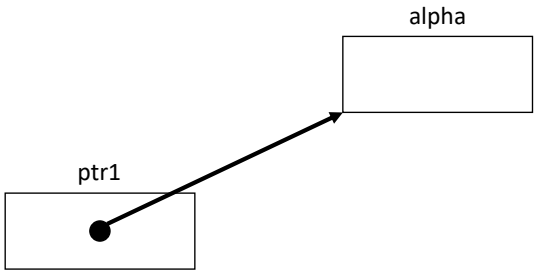

### 3.2.7 Der Adressoperator & (Referenzierung)

Ist x eine Variable vom Typ Typname, so liefert der Ausdruck &x einen Pointer auf die Variable x, d.h. er liefert die Adresse der Variablen x.

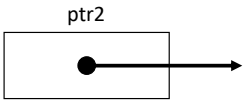
```
int wert;               // Variable wert vom Typ int wird definiert
int* ptr;               // Pointer ptr auf den Typ int wird definiert
                        // ptr zeigt auf eine nicht definierte Adresse

ptr = &wert;            // ptr zeigt nun auf die Variable wert, d.h.
                        // ptr enthaelt die Adresse der Variablen wert
```

### 3.2.8 Kopieren von Adressen

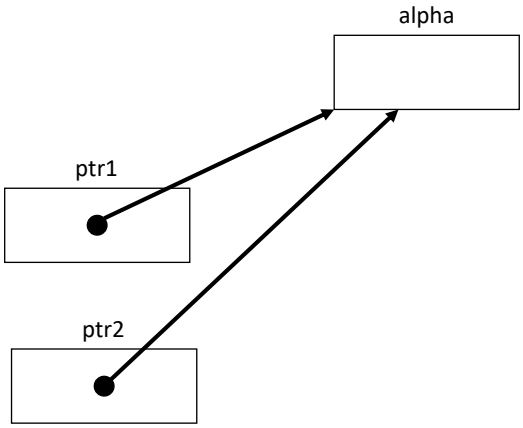
```
float alpha;
float* ptr1 = &alpha;
```




```
float* ptr2;
```




```
ptr2 = ptr1;
```



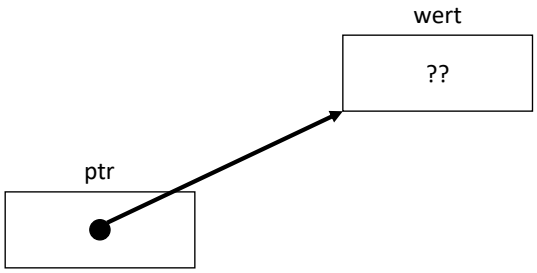

### 3.2.9 Der Inhaltsoperator * (Dereferenzierung)

Ist ptr ein Pointer vom Typ Typname, so liefert der Ausdruck *ptr den Inhalt der Speicherzelle, auf welche ptr zeigt.

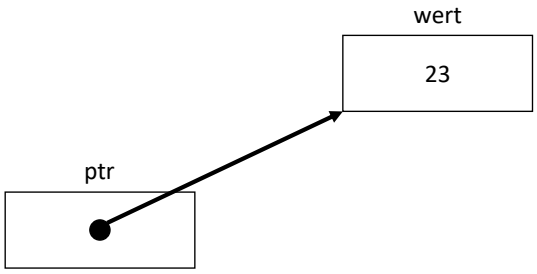
```
int wert;          // Variable wert vom Typ int wird definiert
int* ptr;          // Pointer ptr auf den Typ int wird definiert
                   // ptr zeigt auf eine nicht definierte Adresse
ptr = &wert;       // ptr zeigt nun auf die Variable wert, d.h.
                   // ptr enthaelt die Adresse der Variablen wert
*ptr = 23;         // in die Speicherzelle, auf welche ptr zeigt
                   // (hier: auf die Variable wert), wird 23 geschrieben.
                   // Aequivalent: wert = 23;
```

### 3.2.10 Darstellung in graphischer Pointernotation

```
int wert;
int* ptr;
ptr = &wert;
```




```
*ptr = 23;
```




### 3.2.11 const bei Pointern: Vorsicht

1. Variante: konstanter String

```
char str[] = ''Ein String'';
const char* text = str:
```

Dies bedeutet nicht, dass der Pointer text konstant ist, sondern dass text auf einen konstanten String zeigt.
Von rechts nach links lesen:
"text ist ein Pointer auf eine char-Konstante"

```
char ch = text[1];              // erlaubt (== 'i')
text[1] = 's';                  // nicht erlaubt
text = ''Ein anderer String'';  // erlaubt
```

2. Variante: konstanter Pointer

```
char str[]  = ''Ein String'';
char* const text = strL;
```

Hier ist nun der Pointer text konstant. Die Position von const ist sehr relevant!
Von rechts nach links lesen:
"text ist ein konstanter Pointer auf ein char"

```
char ch = test[1];              // erlaubt (== 'i')
text[1] = 's';                  // erlaubt
text = ''Ein anderer String'';  // nicht erlaubt
```

3. Variante: konstanter Pointer, konstanter String

```
char str[] = ''Ein String'';
const char* const text = str;
```

Hier ist nun der Pointer text konstant und der Text, wohin er zeigt.
Von rechts nach links lesen:
"text ist ein konstanter Pointer auf eine char-Konstante"

```
char ch = text[1];              // erlaubt (== 'i')
text[1] = 's':                  // nicht erlaubt
text = ''Ein anderer String'';  // nicht erlaubt
```

const bei Pointern in Funktionsköpfen

```
void foo (const int* ptr)
{
        *ptr = 14;              // nicht erlaubt
}
```

ptr ist ein Pointer auf eine int-Konstante.

### 3.2.12 void-Pointer

- void-Pointer sind Objekte, die eine gültige Adresse darstellen
- einem void-Pointer kann jeder Pointer zugewiesen werden
- ein void-Pointer kann ohne Typecast nur anderen void-Pointern zugewiesen werden (anders als in C)
- ein void-Pointer kann nicht dereferenziert werden

---

**Hinweis**: in C++ sollten void-Pointer kaum noch angewendet werden

---

void-Pointer: Beispiele

```
int a;
int* pi = &a;
void* pv = pi;                    // ok
double* pd = pv                   // Error (in C erlaubt)
pd = static_cast<double*>pv;      // ok
```

### 3.2.13 Pointer auf Funktionen

- Jede Funktion befindet sich an einer definierten Adresse im Codespeicher
- Diese Adresse kann ebenfalls ermittelt werden
- Interessant wäre, dynamisch zur Laufzeit in Abhängigkeit des Programmablaufs eine unterschiedliche Funktion über einen Funktionspointer aufzurufen

---

**Hinweis**: In C++ gibt es für viele Situationen bessere Alternativen zu Funktionspointern (Polymorphismus)

---

### 3.2.14 Interruptvektortabelle: Tabelle von Funktionspointern

| |
|---|
| Pointer auf ISR n |
| ... |
| ... |
| Pointer auf ISR 2 |
| Pointer auf ISR 1 |

ISR = Interrupt Service Routine

### 3.2.15 Umsetzung von Funktionspointern in C/C++

Der Name der Funktion kann als Adresse auf den ersten Befehl der Funktion verwendet werden (analog Array).

### 3.2.16 Beispiel für Funktionspointer

./listings/ftptr.cpp

```
// Datei: ftptr.cpp
// Funktionspointer
// N. Kaelin, 03.03.2018

#include <iostream>
using namespace std;

int foo(char ch)
```

```
{
        int i;          // muss hier definiert werden
        for(i=1;i<=10;i++)
                cout << ch << "␣";
        cout << endl;
        return i;
}

int main()
{
        int (*p) (char);        // Deklaration des Funktionspointers
        char c;
        int ret;
        cout << "Buchstabe␣eingeben:␣";
        cin >> c;
        p = foo;        // ermittle Adresse der Funktion foo()
        ret = p(c);     // Aufruf von foo() ueber Funktionspointer
        return 0;
}
```

## 3.3 Referenzen

### 3.3.1 Was ist eine Referenz?

- Eine Referenz ist ein Alternativname (Alias) für ein Objekt
- Referenzen ähneln Pointern, sind aber nicht dasselbe. Bei einem Pointer wird immer eine Adresse ermittelt, d.h. dieses Datenobjekt muss sich im adressierbaren Bereich befinden. Eine Referenz kann aber auch auf ein Register verweisen. Grundsätzlich sind Referenzen effizienter als Pointer.
- Syntaktisch sind Referenzen einfacher als Pointer, da ein expliziter Referenzierungs- und Dereferenzierungsoperator entfällt
- Referenzen sind für den Programmierer sicherer anzuwenden als Pointer
- In gewissen Fällen braucht es Pointer. Wenn nicht, dann sollen Referenzen bevorzugt werden.

### 3.3.2 Syntax von Referenzen

```
int x = 24;
int& r1 = x;    // Definition der Referenz r1

x = 55; // x == 55, r1 == 5 (dasselbe Objekt)
r1 = 7; // x == 7, r1 == 7 (dasselbe Objekt)
r1++;   // x == 8, r1 == 8 (dasselbe Objekt)
```

**Hinweis**: Referenzen können nach der Definition nicht ”umgehängt” werden, d.h. eine Referenz kann und muss nur bei der Definition initialisiert werden und kann nicht später auf etwas anders ”zeigen”.

### 3.3.3 Einsatz von Referenzen

- In folgenden zwei Fällen einsetzen:
  - Bei Parameterübergabe (call by reference) anstatt Pointer (entspricht var-Parameter in der Programmiersprache Pascal)
  - Bei Referenz-Rückgabetyp anstatt Pointertyp, d.h. als Returntyp
- Generell:
  Objekte einer Klasse und Strukturvariablen sollen immer by reference übergeben

werden (niemals by value)
- Sonst: zurückhaltend einsetzen

#### 3.3.4 Pointer und Referenzen auf lokale Variablen

---

**Achtung**: Sie dürfen niemals einen Pointer oder eine Referenz auf eine lokale Variable oder ein lokales Objekt mittels return zurückgeben

---

Grund:
Nach Beendigung der Funktion sind die lokalen Variablen ungültig.

### 3.4 Zeiger und Referenzen als Parameter und Rückgabewerte

#### 3.4.1 Call by Value vs. Call by Reference

- Parameter, die by value übergeben werden (Wertparameter) werden kopiert, in der Funktion wird mit Kopien gearbeitet.
- Bei Referenzparametern (call by reference) wird nur eine Referenz (Alias) des Originals übergeben.
- Nur Parameter, welche by reference übergeben werden, könne in der Funktion (bleibend) verändert werden.

#### 3.4.2 3 Beispiele

Versuch 1: Call by value

```
void swap(int a, int b)
{
        int tmp = 1;
        a = b;
        b = tmp;
}

int main()
{
        int x = 4;
        int y = 3;
        swap(x, y);       // nur Kopien werden vertauscht!
        return 0;
}
```

Versuche 2: Call by reference mit Referenzen

```
void swap(int& a, int& b)
{
        int tmp = 1;
        a = b;
        b = tmp;
}

int main()
{
        int x = 4;
        int y = 3;
        swap(x, y);      // OK!
        return 0;
}
```

Versuch 3: Call by reference mit Pointer

```
void swp(int* a, int* b)
{
        int tmp = *a;
        *a = *b;
        *b = tmp;
}

int main()
{
        int x = 4;
        int y = 3;
        swap(&x, &y);    // OK, jedoch muehsame Syntax und evtl. ineffizient
        return 0;
}
```

### 3.4.3 Call by reference: wann einsetzen?

#1: wenn Parameter in der Funktion verändert werden sollen
#2: wenn "grosse" Parameter übergeben werden sollen (struct, class)
zu #2: wenn verhindert werden soll, dass der Parameter verändert wird, so kann dieser mit const deklariert werden

```
int foo(const BigType& b);
```

**Achtung**: Parameterübergabe und Rückgabe von Objekten by value ist ein Hauptgrund für langsame C++-Programme!

### 3.4.4 Merke

**Achtung**: Variablen einer Struktur und Variablen einer Klasse (Objekte) müssen immer by reference übergeben werden, niemals by value.
Read-only Parameter werden zusätzlich mit const spezifiziert.

# 4 Kapitel 4: Arrays, Dynamische Speicherverwaltung

## 4.1 Arrays: Vektoren

### 4.1.1 Problemstellung

Sie müssen 10 Messwerte (z.B. Temperaturwerte) vom Typ int speichern.

```
int data1;
int data2;
int data3;
int data4;
int data5;
int data6;
int data7;
int data8;
int data9;
int data10;
```

Diese Darstellung ist sehr unhandlich.
Wie würden sie 1000 Messwerte abspeichern?

### 4.1.2 Der Array (Feld, Vektor)

Ein Array bietet eine kompakte Zusammenfassung von mehreren Variablen des gleichen Typs.

```
int data[10];    // ein Array von 10 int-Werten
int data[1000]; // ein Array von 1000 int-Werten
double zahl[5]; // ein Array von 5 double-Werten
```

### 4.1.3 Zugriff auf ein Arrayelement

---

**Hinweis**: Der Zugriff auf ein Element eines Arrays erfolgt über den Array-Index. Ist ein Array mit n Elementen definiert, so ist darauf zu achten, dass in C++ (wie in C) der Index mit 0 beginnt und mit n-1 endet.

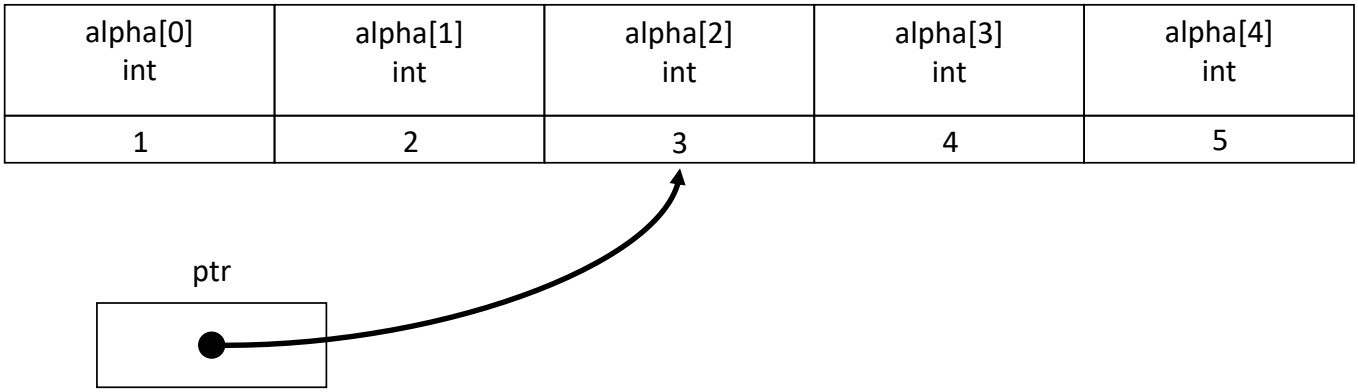
---

```
int alpha[5];    // der Array 'alpha' mit 5 Elementen vom Typ
        // int wird definiert
alpha[0] = 14;  // 1. Element (Index 0) wird auf 14 gesetzt
alpha[4] = 3;    // das letzte Element (Index 4)

alpha[5] = 4;    \color{red}// Bereichsueberschreitung (geht in C++!)\color{black}
```

## 4.2 Arrays und Pointer

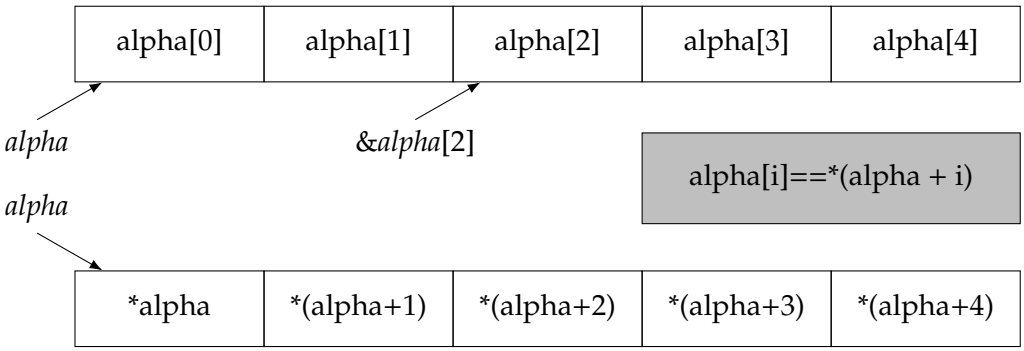
### 4.2.1 Pro Memoria: Eindimensionales Array (Vektor)

```
int alpha[5];
int* ptr;
ptr = &alpha[2];
*ptr = 3452;
```




### 4.2.2 Äquivalenz von Array- und Pointernotation

Der Name des Array kann als konstante Adresse des ersten (Index 0) Elementes des Arrays betrachtet werden.




### 4.2.3 Vergleichen von Arrays

- In C++ gibt es keinen Operator ==, der zwei Arrays miteinander vergleicht
- Arrayvergleiche müssen explizit Element um Element durchgeführt werden
- oder: der Inhalt der beiden Speicherbereiche wird mit Hilfe der Funktion memcmp() byteweise verglichen
- Beispiel:
  Seien arr1 und arr2 zwei Arrays
  Der Vergleich arr1 == arr2 prüft, ob die Anfangsadressen der beiden Arrays identisch sind (wird kaum der Fall sein), nicht aber, ob deren Inhalte identisch sind.

### 4.2.4 Arrayname ist ein nicht modifizierbarer L-Wert

- Der Arrayname ist die konstante Adresse des ersten Elementes des Arrays und kann nicht verändert werden
- Auf den Arraynamen können nur die beiden Operatoren sizeof und & angewandt werden
- Der Arrayname (z.B. arr), als auch der Adressoperator angewandt auf den Arraynamen (&arr) ergeben einen konstanten Pointer auf das erste Element des Arrays, der Typ ist jedoch verschieden
- Einem Arraynamen kann kein Wert zugewiesen werden (einer Pointervariablen schon)

### 4.2.5 Automatische Initialisierung von Arrays

- Globale Arrays werden automatisch mit 0 initialisiert
  - globale Arrays sollten aber nur ausnahmsweise verwendet werden
- Lokale Arrays werden nicht automatisch initialisiert
  - der Inhalt eines lokalen Arrays ist bei der Definition undefiniert

### 4.2.6 Explizite Initialisierung von Arrays

- Bei der Definition eines Arrays kann ein Array explizit (”manuell”) initialisiert werden

- Der Definition folgt ein Zuweisungsoperator und eine Liste von Initialisierungswerten
- Die Liste ist mit geschweiften Klammer begrenzt
- Als Werte können nur Konstanten oder Ausdrücke mit Konstanten angegeben werden, Variablen sind nicht möglich
- Die Werte werden mit Kommata getrennt
- Nach der Initialisierung können die Elemente nur noch einzeln geändert werden

### 4.2.7 Beispiel: Explizite Initialisierung von Arrays

```
in alpha[3] = {1, 2*5, 3};
```

ist "äquivalent" zu:

```
int alpha [3];

alpha[0] = 1;
alpha[1] = 2*5;
alpha[2] = 3;
```

### 4.2.8 Goodies für die explizite Initialisierung

- Werden bei der Initialisierung weniger Werte angegeben als der Array Elemente hat, so werden die restlichen Elemente mit 0 belegt.

```
int alpha[200] = {3, 105, 17};
// alpha[3] bis alpha[199] werden gleich 0 gesetzt
```

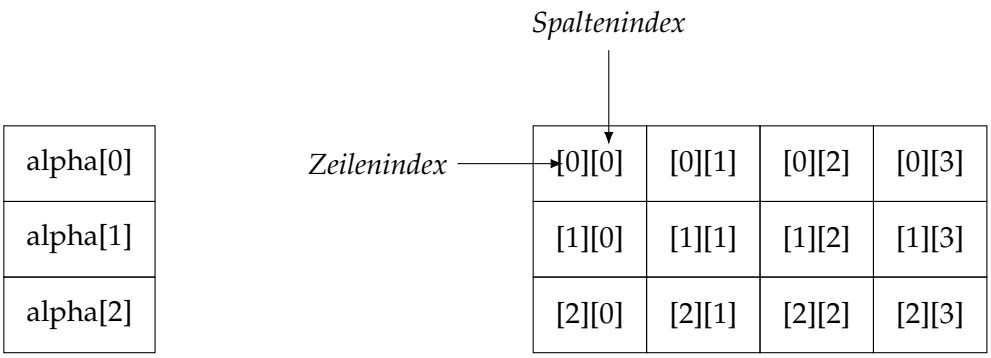
- wird bei der Definition keine Arraygrösse angegeben, so zählt der Compiler die Anzahl Elemente automatisch (offenes Array ohne Längenangabe)

```
int alpha[] = {1, 2, 3, 4};
```

## 4.3 Mehrdimensionale Arrays

```
int alpha[3][4];
```

**Matrix**
Kann betrachtet werden als Vektor alpha[0] bis alpha[2], wobei jedes Vektorelement wiederum einen Vektor mit 4 Elementen enthält



### 4.3.1 Initialisierung eines mehrdimensionalen Arrays

```
int alpha[3][4] = {
        {1, 3, 5, 7},
        {2, 4, 6, 8},
        {3, 5, 7, 9}
        };
// aequivalent dazu ist die folgende Definition:
int alpha[3][4] = {1, 3, 5, 7, 2, 4, 6, 8, 3, 5, 7, 9};
```

| | | | |
|---|---|---|---|
| 1 | 3 | 5 | 7 |
| 2 | 4 | 6 | 8 |
| 3 | 5 | 7 | 9 |

**Hinweis**: Das erste Element kann offen sein, das zweite muss angegeben werden.

## 4.4 Übergabe von Arrays und Zeichenketten

- Bei der Übergabe eines Arrays an eine Funktion wird als Argument der Arrayname übergeben. (i.e. Pointer auf erstes Element des Arrays)
- Der formale Parameter für die Übergabe eines eindimensionalen Arrays kann ein offenes Array sein oder ein Pointer auf den Komponententyp des Arrays.
- Zeichenketten sind char-Arrays und werden deshalb gemäss der oben erwähnten Punkte gehandhabt.

### 4.4.1 Beispiel: Array (Vektor) als Parameter

```
enum {groesse = 3};

void init(int* alpha, int size);        // int* alpha == Pointer auf Arrayelement
void ausgabe(int alpha[], int size);    // int alpha[] == offener Array

int main()
{
        int arr[groesse];
        init(arr, sizeof(arr)/sizeof(arr[0]));  // Argument ist Name des Arrays
        ausgabe(arr, sizeof(arr)/sizeof(arr[0]));// Argument ist Name des Arrays
        return 0;
}
```

### 4.4.2 Übergabe einer Matrix mittels offenem Array

```
void printMat(const double* const mat[],        // Matrix
        int m,                                  // Anzahl Zeilen
        int n);                                 // Anzahl Spalten
```

- Der Aufruf erfolgt mit printMat(matA, rows, cols); // bevorzugt
  oder printMat(&matA[0], rows, cols);
  oder printMat(&(&matA[0])[0], rows, cols);
- matA ist vom Typ double**

matA[0] ist vom Typ double*
matA[0][0] ist vom Typ double

#### 4.4.3 Zeichenketten (Strings)

- Strings sind char-Arrays, abgeschlossen mit dem Zeichen '\0', bzw. 0 (alles analog C)

### 4.5 Dynamische Speicherverwaltung

#### 4.5.1 Pro Memoria: Variablen

- Variablen erleichtern u.a. den Zugriff auf Speicherstellen
  - statt über eine Adresse kann auf die Speicherstelle mit Hilfe des Variablennamens zugegriffen werden
- Die Variablen müssen zur Entwicklungszeit im Code definiert werden
  - zur Laufzeit können keine neuen Variablen erzeugt werden
  - der gesamte Speicherbedarf muss zur Entwicklungszeit statisch mittels Variablen festgelegt werden
  - wie gross wähle ich z.B. einen Array, welcher Messwerte enthält?
  - weiss ich zur Entwicklungszeit bereits die zahl der Messwerte?
- Der Speicher einer Variablen wird automatisch freigegeben, sobald die Variable nicht mehr gültig ist

#### 4.5.2 Dynamische Speicherverwaltung

- Speicher kann zur Laufzeit (dynamisch) vom System angefordert werden
  - Operator: new (in C: Funktion malloc())
- Dynamisch allozierter Speicher muss wieder explizit freigegeben werden
  - Operator: delete (in C: Funktion free())
- Dynamischer Speicher wird nicht auf dem Stack angelegt, sondern auf dem **Heap**
- Auf dynamischen Speicher kann **nur über Pointer** zugegriffen werden

---

**Achtung**: Zugriff auf dynamischen Speicher nie verlieren!

---

#### 4.5.3 Dynamische Speicherverwaltung: Syntax

```
int* pint = new int;     // Speicher fuer int alloziert
char* pCh1 = new char;   // Speicher fuer char alloziert
char* pCh2 = new char;   // Speicher fuer char alloziert

*pInt = 23;
std::cin >> *pCh1;
pCh2 = pCh1;
// pCh2 zeigt nun auch auf die Speicherstelle, auf welche pCh1 zeigt.
// Damit geht aber der Zugriff auf die Speicherstelle verloren, auf die pCh2 gezeigt hat
   (Memory Leak!)

delete pInt;      // Speicher wieder freigeben
delete pCh1;
delete pCh2;      // ergibt Fehler, bereits über pCh1 freigegeben
```

#### 4.5.4 Dynamische Speicherverwaltung: Vorsichtsmassnahmen

- der delete-Operator kann auch auf den Nullpointer angewendet werden. Es passiert dadurch (definiert) nichts.
- Die Anwendung des delete-Operators auf einen bereits freigegebenen Speicherbereich kann Probleme verursachen
- Oft wird deshalb eine Pointer nach der delete-Operation auf 0 gesetzt (defensiver Programmierstil)

```
delete pInt;     // Speicher wieder freigeben
delete pInt;     // Speicher ist bereits freigegeben
pInt = 0;
delete pInt;     // ist problemlos
```

### 4.5.5 Memory Leak, Garbage Collection

- Dynamisch allozierter Speicher, welcher nicht freigegeben wurde oder auf welchen der Zugriff verloren ging, belegt weiterhin Platz im Speicher.
- Der faktisch nutzbare Speicher wird somit immer kleiner. Es ist, als ob der Speicher ein Leck hätte. Dieses Fehlverhalten wird deshalb als **Memory Leak** bezeichnet.
- In einigen Programmiersprachen (z.B. Java) gibt es einen **Garbage Collector** (Abfalleimer), welcher nicht mehr benötigten Speicher automatisch freigibt.
- C++ besitzt keinen Garbage Collector. Der C++-Programmierer ist verantwortlich, dass allozierter Speicher wieder freigegeben wird.

### 4.5.6 Dynamische Allozierung von Arrays

- In C++ kann Speicher für einen Array auch erst zur Laufzeit (dynamisch) vom System angefordert (alloziert) werden
  - Operator: **new[]**
- Der Zugriff auf den Array erfolgt wie bei einem statischen Array
- Dynamisch allozierte Arrays müssen wieder explizit freigegeben werden
  - Operator: **delete[]**
  - Achtung: delete[], nicht nur delete

```
int* pInt = new int[100];        // statt einer Konstanten kann hier auch eine
   Variable verwendet werden (Normalfall)
delete pInt;     // Fehler: nur pInt[0] wird freigegeben
delete[] pInt;   // korrekter Befehl
```

### 4.5.7 Dynamische Allozierung von Matrizen

- Oft wird eine m x n - Matrix als ein eindimensionaler Array der Grösse (m*n) implementiert. Der Zugriff geht dann jedoch nur noch über Pointer:

```
        *(matrix+2*n+3) = 23.44;
```

- Mit der im folgenden gezeigten Variante kann auf ein Matrixelement über die Arrayindizes zugegriffen werden:

```
        matrix[2][3] = 23.44;
```

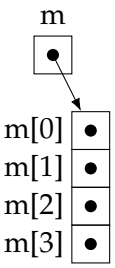
### 4.5.8 Dynamische Matrix mit 4 Zeilen und 3 Spalten

```
double** m = 0;
```

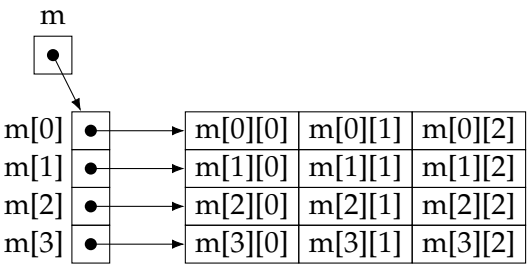
;


```
double** m = 0;
m = new double*[4];                 // Array mit 4 Elementen vom Typ double* (Pointer
    auf double)
```

;



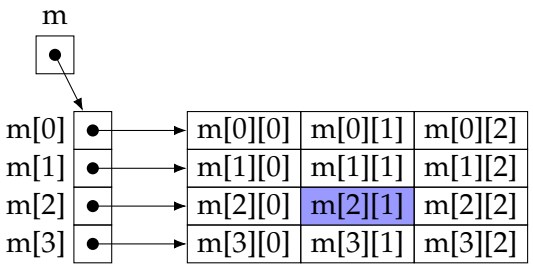

```
double** m = 0;
m = new double*[4];                 // Array mit 4 Elementen vom Typ double* (Pointer
    auf double)
for (int i=0; i<4; ++i)
        m[i] = new double[3];       // Jedes m[i] ist ein Pointer auf ein Array mit 3
            Elementen vom Typ double
                                    // m[i] selbst ist vom Typ double*
```




Für die Konstanten 3 und 4 könnten auch Variablen verwendet werden (im Gegensatz zu einer statisch definierten Matrix).

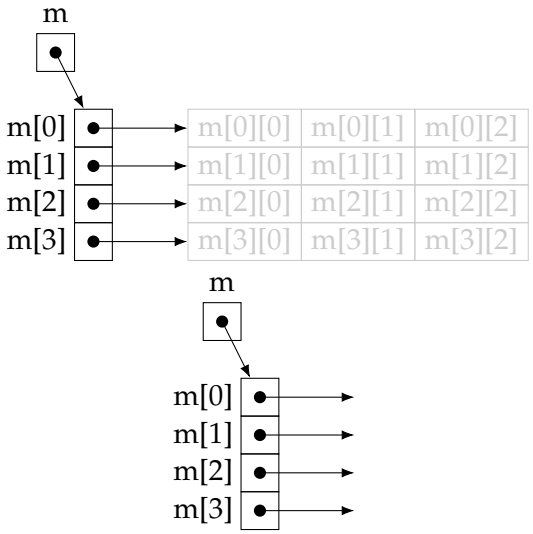
### 4.5.9 Zugriff auf dynamisch erzeugte Matrix

```
double** m = 0;
m = new double*[4];
for (int i=0; i<4; ++i)
        m[i] = new double[3];
m[2][1] = 34.675;           // Der Zugriff erfolgt einfach ueber die Arrayindizes
```

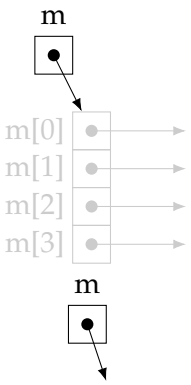



### 4.5.10 Dynamische Matrix freigeben

```
for(int i=0; i<4; ++i)
        delete[] m[i]    // Zuerst jede Zeile freigeben
```




```
for(int i=0; i<4; ++i)
        delete[] m[i]

delete[] m;       // Nun noch den Array mit den double* freigeben
```



**Achtung**: m zeigt immer noch auf die alte Adresse, an welcher sich aber keine gültigen Daten mehr befinden. (Allenfalls m wieder auf 0 setzen)

### 4.5.11 Effizienz der Matriximplementationen

- Nur mit dieser gezeigten Variante kann auf ein Matrixelement über die Arrayindizes zugegriffen werden:

```
matrix[2][3] = 23.44;
```

- Der Nachteil dieser Variante ist, dass es pro Zeile einen zusätzlichen Pointer braucht. Die einzelnen Zeilen liegen u.U. nicht auf aufeinanderfolgenden Speicherstellen.
- Wenn eine m x n - Matrix als ein eindimensionaler Array der Grösse (m*n) implementiert wird, erspart man sich die Zeilenpointer, der Zugriff ist jedoch langsamer und mühsamer. Die einzelnen Elemente der Matrix liegen auf aufeinanderfolgenden Speicherstellen.

# 5 Kapitel 5: Scope, Deklarationen, Type Casts

## 5.1 Strukturen

### 5.1.1 Strukturen in C++

- Grundsätzlich sind Strukturen in C++ identisch zu Strukturen in C
- In C++ haben Strukturen noch zusätzliche Möglichkeiten (folgt im Zusammenhang mit Klassen)
- Die Definition und Nutzung von Strukturen ist in C++ einfacher, typedef braucht es nicht

```
struct Point
{
        double x;
        double y;
};

Point p1;
```

## 5.2 Gültigkeitsbereiche, Namensräume und Sichtbarkeit

### 5.2.1 Gültigkeitsbereiche von Namen (Scope)

- Prinzipiell identisch wie in C
- Der Compiler arbeitet immer Dateiweise
- Namen in einer anderen Datei sind dem Compiler nicht bekannt
- (Globale) Variablen, welche in einer anderen Datei definiert werden, können mit Hilfe des extern-Statements bekannt gemacht werden
- Durch das extern-Statement wird kein Speicherplatz reserviert

```
extern int Foo_globalVariable;
```

- Funktionsprototypen und Definitionen, die von anderen Modulen genutzt werden können (Schnittstellen), werden in einer Headerdatei definiert
- Durch #include der Headerdatei wird der Header geladen und die Namen bekannt gemacht

### 5.2.2 Gültigkeitsbereiche in C++

- Lokaler Gültigkeitsbereich (local scope)
  Alle in einem Block deklarierten Bezeichner gelten von ihrer Deklaration an bis zum Ende des aktuellen Blocks
- Gültigkeitsbereich Funktionsprototyp, Funktion
  Alle in einem Funktionskopf deklarierten Bezeichner (Parameter) gelten in der gesamten Funktion
- Gültigkeitsbereich Namensraum (namespace)
  Alle im Namensraum deklarierten Bezeichner gelten von ihrer Deklaration an bis zum Ende des Namensraums
- Gültigkeitsbereich Klasse
  Alle in einer Klasse deklarierten Bezeichner gelten von ihrer Deklaration an in der gesamten Klasse

### 5.2.3 Gültigkeit (Scope) von Variablen

- Eine Variable ist an einer bestimmten Stelle gültig, wenn ihr Name an dieser Stelle dem Compiler durch eine Vereinbarung bekannt ist
- Gültige Variablen können für den Programmierer unsichtbar sein, wenn sie durch eine andere Variable desselben Namens verdeckt werden

### 5.2.4 Lebensdauer von Variablen

- Die Lebensdauer ist die Zeitspanne, in der das Laufzeitsystem des Compilers der Variablen einen Platz im Speicher zur Verfügung stellt
- Mit anderen Worten, während ihrer Lebensdauer besitzt eine Variable einen Speicherplatz
- Globale Variablen leben solange wie das Programm
- Lokale Variablen werden beim Aufruf des Blocks angelegt und beim Verlassen des Blocks wieder (automatisch!) ungültig

### 5.2.5 Sichtbarkeit von Variablen

- Variablen von inneren Blöcken sind nach aussen nicht sichtbar
- Globale Variablen und Variablen in äusseren Blöcken sind in inneren Blöcken sichtbar
- Werden lokale Variablen mit demselben Namen wie eine globale Variable oder wie eine Variable in einem umfassenden (äusseren) Block definiert, so ist innerhalb des Blockes nur die lokale Variable sichtbar. Die globale Variable bzw. die Variable in dem umfassenden Block wird durch die Namensgleichheit verdeckt.

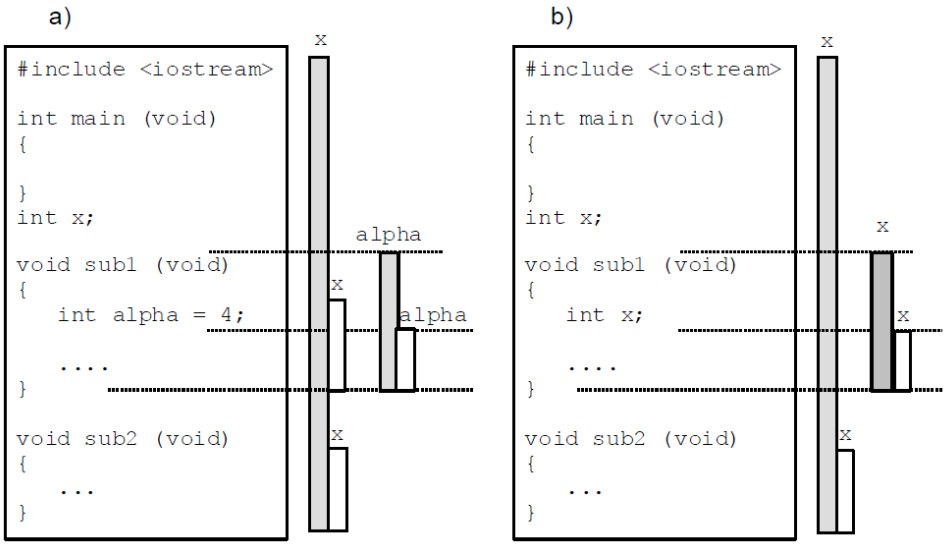
### 5.2.6 Schlussfolgerung (naheliegend aber falsch)

- Alle Variablen global definieren, dann muss ich mir keine Sorgen um die Sichtbarkeit machen

→ Stimmt schon, aber:

- Weil die Variablen in demselben Namensraum sind, müsste ich die Variablennamen im gesamten Projekt abstimmen
  → ist nicht praktikabel
- Globale Variablen haben gewichtige Nachteile: Wer hat den Variablenwert wo wie geändert?
  → schwer nachzuvollziehen

### 5.2.7 Lebensdauer (grau) und Sichtbarkeit (weiss)




### 5.2.8 Codierstil

- Variablen so lokal wie möglich definieren, d.h. im innersten möglichen Block (Tipp: nur am Anfang eines Blocks)
- Globale Variablen wenn immer möglich vermeiden. Sie müssen speziell gekennzeichnet werden. Sie sollen (in C) mit einem Prefix (Modulkürzel) gefolgt von einem underscore character (_) beginnen. Dadurch werden die Namen eindeutig.
  Beispiel:
  Die globale Variable counter im Modul Foo muss wie folgt definiert werden:

```
int Foo_counter;
```

- Globale Variablen am Anfang der Datei definieren, d.h. auf jeden Fall vor der ersten Funktion

---

**Hinweis**: Besser (in C++): Namespace definieren

---

## 5.3 Namensräume (Namespaces)

### 5.3.1 Namensräume

**Namen, die zu verschiedenen Namensräumen gehören, dürfen auch innerhalb desselben Gültigkeitsbereichs gleich sein.** In C gibt es die folgenden Namensräume (gilt auch für C++):

- Marken
- Namen von Strukturen, Unions und enums
- Jede Struktur und Union für ihre Feldnamen
- Bezeichner von Variablen, Funktionen, typedef-Namen, enum-Konstanten

In C++ können zusätzlich explizit definierte Namensräume verwendet werden.

---

**Hinweis**: Dadurch können (und sollen unbedingt!) die in C üblichen Modulkürzel vermieden werden.

---

### 5.3.2 Explizite Namensräume in C++

- Nebst den vordefinierten (impliziten) Namensräumen des vorherigen Abschnitts können in C++ explizit eigene Namensräume (namespaces) definiert werden.
- Bezeichner müssen nur innerhalb ihres Namensraums eindeutig sein
- Für jedes Modul in C (mit Modulkürzel) soll in C++ ein Namensraum definiert werden
- Sie haben bisher in allen C++-Übungen bereits den Namensraum std verwendet

### 5.3.3 C++-Mechanismen für Namespaces

- Deklaration von Namespaces
- Namespace-Alias
  Einem bestehenden Namespace einen anderen Namen zuweisen (eher selten)

```
namespace fbssLib = financial_branch_and_system_service_library;
```

- using-Deklaration
- using-Direktive

### 5.3.4 Deklaration von Namespaces

- Alle in einem Namespace deklarierten Bezeichner werden diesem Namespace zugeordnet
- Auf die Bezeichner des Namespaces kann mit dem Scope-Operator :: zugegriffen werden
- Syntax:
  Hinter dem Schlüsselwort namespace folgt der Name des Namespaces gefolgt von einem Block
- Innerhalb einer Datei kann mehr als ein Namespace deklariert werden (eher unüblich) und ein Namespace kann in mehreren Dateien deklariert sein (häufig, d.h. die Elemente eines Namespaces werden in mehreren Dateien implementiert)

### 5.3.5 Deklaration von Namespaces: Beispiel

```
namespace myLib1
{
        int i;
        void foo();
}

namespace myLib2
{
        int i;
        void foo();
        int go();
}

...
{
        myLib1::foo();  // vollstaendiger Name von foo()
        myLib2::i = 17; // vollstaendiger Name von i
}
```

### 5.3.6 using-Deklaration

- Eine using-Deklaration "importiert" Namen aus einem Namensraum und macht ihn ohne explizite Namensraumangabe verwendbar
- Sie kann lokal in einem Block oder global ausserhalb eines Blocks verwendet werden
- Deklariert nur einzelne Namen

```
int main()
{
        using myLib1::foo;      // lokales Synonym
        foo();                  // ruft myLiby::foo() auf
}
```

### 5.3.7 using-Direktive

- Eine using-Direktive macht alle Namen aus einem Namensraum ohne explizite Namensraumangabe verwendbar

```
usting namespace myLib1;        // ''importiert'' alle Namen aus myLib1

int main()
{
        foo();                  // ruft myLib1::foo() auf
}
```

### 5.3.8 using namespace kann zu Konflikten führen

- Wenn bei mehreren using namespace-Deklarationen und/oder -Direktiven die Namen (ohne Namespace-Angabe) nicht eindeutig sind, müssen die Namen voll qualifiziert verwendet werden (mit Namespace-Angabe)

```
namespace myLib1
{
        int i;
        void foo();
}

namespace myLib2
{
        int i;
        void foo();
        int go();
}

...
{
        myLib1::foo();   //ist nicht eindeutig, Compiler reklamiert
        myLib2::i = 17; // vollstaendiger Name von i
}
```

#### 5.3.9 Namenlose Namespaces

- Ein namenloser Namespace wird wie ein spezieller Namensraum mit einem systemweit eindeutigen Namen behandelt

---

**Hinweis**: Es ist guter Programmierstil, den Gültigkeitsbereich aller nur intern verwendeten Funktionen und Daten mit Hilfe von namenlosen Namespaces auf den Bereich eingrenzen, in dem die Objekte verwendet werden. (In C hat man dafür die Funktionen mit static gekennzeichnet

---

```
namespace
{


}
```

#### 5.3.10 Zugriff auf globale Variable mit Scope-Operator

```
int zahl = 11;  // globale Variable

int main()
{
        int zahl = 22;          // lokale Variable
        zahl = zahl + 4;        // lokale Variable
        ::zahl = 23;            // Zugriff auf globale Variable
}
```

### 5.4 Speicherklassen

#### 5.4.1 Speicherklassen in C++

- auto
  Ist default, wenn nichts geschrieben wird. Eine mit auto deklarierte Variable wird nach Beendigung des

Scopes automatisch entfernt.
Achtung: hat ab C++11 eine andere Bedeutung!!

- register
  Ist dasselbe wie auto mit zusätzlichem Hinweis an Compiler: wenn es geht in ein Register legen (sehr zurückhaltend einsetzen, besser gar nicht)
- static
- extern
- mutable
  (später im Zusammenhang von Klassen)

#### 5.4.2 Speicherklasse static: Variablen

- static-Variablen sind im Datenbereich, nicht auf dem Stack
- Sie werden automatisch auf 0 initialisiert, wenn nichts anderes steht
- Gültigkeitsbereich ist der Block, in dem die Variable definiert ist
- static-Variablen, welche ausserhalb einer Funktion definiert sind (globale Variablen). sind nur in der Datei gültig, in der sie definiert werden
- static-Variablen sind nur einmal vorhanden (auch in multi-threading-Umgebungen), d.h. ihr Wert wird erhalten, auch wenn die Funktion beendet ist. Beim nächsten Aufruf der Funktion geht es mit dem alten Wert weiter.
- Nur einsetzen, wenn man das will!

#### 5.4.3 Speicherklasse static: Funktionen

- static-Funktionen sind nur in der Datei, in welcher sie definiert sind, sichtbar
- Alle Funktionen, welche nicht aussen (für andere Units) sichtbar sein sollen, sollten deshalb **in C** als static definiert werden
- In C++ können dafür namenlose Namespaces verwendet werden (bevorzugt)

#### 5.4.4 Speicherklasse extern: Externe Variablen

- Eine externe Variable kann nur in einer einzigen Datei definiert werden (ohne Speicherklasse extern)
- In den anderen Dateien wird sie mit extern deklariert (bekannt gemacht)
- Eine manuell Initialisierung ist nur bei der Definition möglich
- Globale Variablen, welche nicht manuell initialisiert werden, werden automatisch mit 0 initialisiert
- extern-Deklarationen werden üblicherweise in einer Headerdatei deklariert und am Beginn der Datei mit #include eingefügt

#### 5.4.5 Typqualifikationen (Kap. 9.2.2)

- const
  const-Objekte können nicht verändert werden (read-only)
- volatile
  Der Compiler wird angewiesen, keine Optimierungen (soweit sie die Variable betreffen) vorzunehmen
- volatile wird oft bei Embedded Systems angewandt, wenn z.B. ”hinter” einer Variable ein Register liegt.
- const und volatile können auch kombiniert werden, z.B. bei read-only-Hardwareregistern

#### 5.4.6 Funktionsattribute

- inline
  bereits bekannt
- virtual
  später im Zusammenhang mit Klassen
- explicit
  später im Zusammenhang mit Klassen

### 5.5 Typdefinitionen

#### 5.5.1 typedef zur Vereinbarung eigener Datentypen

- analog C

- In C++ kann aber z.B. bei structs das typedef weggelassen werden
- In C:

```
typedef struct {int x;
                iny;} Point;
```

- In C++:

```
struct Point {
        int x;
        int y;
};
```

- Stil: eigene Typen werden mit einem Grossbuchstaben begonnen

### 5.5.2 Beispiel

```
struct Point {
        int x;
        int y;
};

struct Line {
        Point p1;
        Point p2;
};

int main(void)
{
        line myLine = {12, -34,          // p1
                       783, 12};         // p2

        std::cout << ''Startpunkt: ('' << myLine.p1.x << '', ''
                                       << myLine.p1.y << '')\n'';
        std::cout << ''Endpunkt: ('' << myLine.p2.x << '', ''
                                     << myLine.p2.y << ''\n'';
        return 0;
}
```

### 5.5.3 Gewährleistung von Portabilität

- Oft muss z.B. ein Register ein 16 Bit breiter Wert geschrieben werden. Welcher Typ ist nun 16 Bit breit?
- Das ist implementationsabhängig (vielleicht unsigned short, unsigned int, ...)
- Um die Portabilität (Umschrieben auf ein anderes System) zu vereinfachen, wird ein 16 Bit breiter Datentyp (Word) definiert und dann ausschliesslich verwendet (in stddef.h sind diese Typen üblicherweise bereits definiert). Auf einem anderen System ist dann nur noch dieser typedef zu ändern.

```
typedef unsigned short uint16_t;
```

### 5.5.4 Wie setzt der Compiler ein typedef um?

- Ein typedef ist mehr oder weniger eine reine Textersetzung. Erklärung anhand des folgenden Beispiels:

```
typedef struct {int x;
                int y;} Point;
```

- Überall im Code, wo nun das Wort Point gefunden wird, ersetzt der Compiler dieses in einem ersten Durchgang mit dem Text

```
typedef struct {int x;
                int y;}
```

## 5.6 Initialisierung

- analog C

## 5.7 Type-Case (Typumwandlungen)

### 5.7.1 Typumwandlungen im Allgemeinen

- Unsafe conversion
  Wenn bei der Typumwandlung signifikante Stellen verloren gehen können (typischerweise bei einer Umwandlung von einem "grösseren" in einen "kleineren" Typ, z.B. von double nach int)
  Bei int ist siwihl die Genauigkeit als auch die maximal darstellbare Zahl
- Safe conversion
  Wenn bei der Typumwandlung keine signifikanten Stellen verloren gehen können (typischerweise bei einer Umwandlung von einem "kleineren" in einen "grösseren" Typ, z.B. von in nach double

### 5.7.2 Implizite Typumwandlung

- Die implizite (automatische) Typumwandlung wird auch als Standard-Typumwandlung bezeichnet
- Sie erfolgt analog zur Programmiersprache C (siehe dort)

### 5.7.3 Explizite Typumwandlung

- Nebst den impliziten (automatischen) Typumwandlungen kann in C++ mit Hilfe von 6 verschiedenen cast-Operatoren eine explizite Typumwandlung bewirkt werden.
- Bei der expliziten Typumwandlung gibt der Programmierer explizit an, was er will.

---

**Achtung**: Bei der expliziten Typumwandlung übernimmt der Programmierer die Verantwortung, dass die Umwandlung keine Probleme ergibt.
(z.B. Umwandlung von grosser Zahl in kleineren Typ)

---

### 5.7.4 Explizite Typumwandlung #1, 2: C-Stil und Funktionsstil

- Stroustrup: "The C and C+ cast is a sledgehammer..."
- Syntax für C-Stil (einzige Variante in C):
  (Zieltyp)expression

```
int a = (int)4.6;        // a == 4
```

- Syntax für Funktionsstil:
  Zieltyp(expression)

```
int a = int(4.6);        // a == 4
```

### 5.7.5 Typumwandlung mit C-Stil und Funktionsstil

**Typumwandlung ist...**

- einfache Reinterpretation der bitweisen Darstellung des Ausdrucks
- einfache arithmetische Grössenanpassung
- ein const- oder volatile-Attribut zu einem Ausdruck hinzufügen oder entfernen
- andere (eventuell implementierungsabhängige) Umwandlung

---

**Achtung**: Aus dem Sourcecode geht nicht hervor, welche der aufgeführten Typumwandlungen der Programmierer wollte.
Diese beiden Casts sollten in C++ nicht verwendet werden!

---

### 5.7.6 Explizite Typumwandlung #3: const_cast

**Anwendung:**
Ausschliesslich die (vorübergehende) Entfernung des const-Qualifikators, d.h. die Umwandlung eines Ausdrucks vom Typ T mit den optionalen Qualifikatoren const und volatile in einen Ausdruck desselben Typs ohne den Qualifikator const
**Syntax:**

```
const_cast<Zieltyp>expression

const char* findSubString(const char* str, const char* subStr)
{
        return strstr(const_cast<char*<str,
                      const_cast<char*>subStr);
}
```

Die Funktion strstr() akzeptiert nur Parameter com Typ char* (ohne const)

### 5.7.7 Explizite Typumwandlung #4: static_cast

**Anwendung:**
Umwandlung von Objekten einer Klasse auf Objekte einer Basisklasse oder die Umwandlung mittels einer Umwandlungsfunktion.
Wenn schon Type cast, dann ist static_cast die häufigste.

### 5.7.8 Explizite Typumwandlung #5: dynamic_cast

**Anwendung:**
Umwandlung von polymorphen Objekten im Zusammenhang mit dem Typsystem von C++ eingesetzt (Stichwort RTTI = Runtime Type Information System)
Näheres folgt später im Zusammenhang mit Klassen und Polymorphismus.

### 5.7.9 Explizite Typumwandlung #6: reinterpret_cast

**Anwendung:**
reinterpret_cast ist eine neue Interpretation der zugrunde liegenden Bitkette.
**Syntax:**

```
reinterpret_cast<Zieltyp>expression

char* p = new char[20];
...
int* pi = reinterpret_cast<int*>p;
```

# 6 Kapitel 6: Module und Datenkapseln

## 6.1 Modul (Unit)

### 6.1.1 Motivation

- **Arbeitsteilung**
  Grosse Programme werden von mehreren Personen entwickelt. Praktikabel ist, wenn nur eine Person an einer bestimmten Datei arbeitet.
- **Effizienz**
  Eine Übersetzungseinheit (Datei) muss bei jeder Änderung neu übersetzt werden (je grösser die Datei desto langsamer die Übersetzung)
- **Strukturierung**
  Ein grosses Programm in mehrere vernünftige Teile (Baugruppen, Units) aufteilen (Divide and conquer)

### 6.1.2 Nomenklatur: Modul vs. Unit

- Ein Programmbaustein wird traditionell mit Modul (der oder das Modul) bezeichnet
- Der Test eines Moduls heisst folglich Modultest
- Das Vorgehen, welches Module generiert, heisst Modularisierung
- Heute üblicher wird Modul mit Unit, der Test mit Unittest bezeichnet, das Vorgehen heisst weiterhin Modularisierung
- Prinzipiell spreche ich künftig meist von Unit und Unittest

### 6.1.3 Ziele der Modularisierung

- Klare, möglichst schlanke Schnittstellen definieren
- Units so bilden, dass Zusammengehörendes in einer Unit isoliert wird (Kohäsion) soll hoch sein
- Schnittstellen zwischen den Units sollen klein sein (Kopplung soll klein sein)
- Abhängigkeiten unter den Units sollen eine Hierarchie bilden, zirkuläre (gegenseitige) Abhängigkeiten müssen vermieden werden

### 6.1.4 Eigenschaften einer Unit (eines Moduls)

- realisiert eine in sich abgeschlossene Aufgabe
- kommuniziert über ihre Schnittstelle mit der Umgebung
- kann ohne Kenntnisse ihres inneren Verhaltens in ein Gesamtsystem integriert werden
- ihre Korrektheit kann ohne Kenntnis ihrer Einbettung in einem Gesamtsystem nachgewiesen werden (mittels Unittest)

### 6.1.5 Bestandteile eine C++-Programms

- **Eine** Hauptfunktion main()
- Eine Reihe von unabhängigen Programmbausteinen (Units)

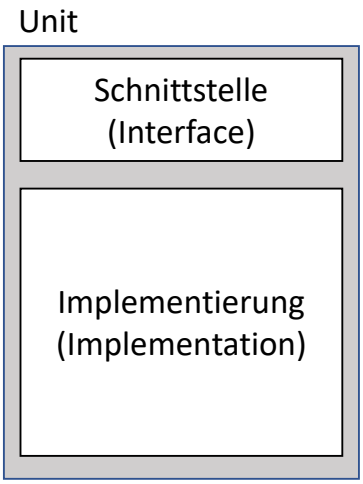
### 6.1.6 Unitkonzept

- Interface definiert die Schnittstelle, d.h. die Deklarationen wie Funktionsprototypen, etc. (Schaufenster)
- Implementation: In diesem Teil sind die Unterprogramme definiert, d.h. auscodiert (Werkstatt)
- Das Interface wird in einer Headerdatei (*.h) beschrieben, die Implementation liegt in einer *.cpp-Datei

## 6.2 Geheimnisprinpzip (Information Hiding)

### 6.2.1 Information Hiding

- In der Schnittstelle (Headerdatei) wird alles beschrieben, was ein Nutzer dieser Unit wissen muss
- Der innere Aufbau der Unit (*.cpp) muss (darf) dem Nutzer der Unit nicht bekannt sein, er benötigt diese Informationen auch gar nicht
- Der Nutzer der Unit darf keine Annahmen bezüglich des inneren Aufbaus der Unit treffen
- Der Entwickler der Unit darf den inneren Aufbau der Unit ändern, solange dadurch die Schnittstelle nicht geändert werden muss




### 6.2.2 Konzept der Datenkapsel

- Eine Unit besteht aus Funktionen und Daten
- In der Schnittstelle wird definiert, was für den Nutzer zur Verfügung steht. Dies können Funktionen und Daten sein
- Die Datenkapsel fordert nun zusätzlich, dass auf die Daten nicht direkt zugegriffen werden darf, sondern nur über Zugriffsfunktionen

### 6.2.3 Beispiel für Datenzugriff bei Datenkapsel

```
// interne Daten
int counter;     // gehoert zur Unit, nicht in Interface
                 // wie kann das in C bewerkstelligt werden?

// Schnittstelle (Interface)
voi setCounter(int c)
{
        counter = c;
}

int getCounter(void)
{
        return counter;
}
```

### 6.2.4 Beispiel für Unit Rechteck (ohne Datenkapsel)

```
        // interne Daten
        double a;       // 1. Seite
        double b;       // 2. Seite
        double area;    // Rechtecksflaeche

        // Schnittstelle (Interface)

        // direkter Zugriff auf a, b, area
        // Annahme: area hat immer den aktuellen Wert, d.h. es muss
        // bei jeder Aenderung von a und b (durch den Client!) berechnet werden
```

**Achtung**: Sehr gefährlich! (kann kaum sichergestellt werden)

### 6.2.5 Beispiel für Unit Rechteck: Verbesserung #1

```
        // interne Daten
        double a;       // 1. Seite
        double b;       // 2. Seite
        double area;    // Rechtecksflaeche

        // Schnittstelle (Interface)
        // kein direkter Zugriff mehr auf a, b, area
        // Funktionen setA(), setB(), getA(), getB(), getArea()
        void setA(double newA)
        {
                a = newA;
                area = a * b;
        }

        double getArea(void)
        {
                return area;
        }
```

**Hinweis**: Evtl. gefährlich (Berechnung von area könnte vergessen werden). Und: soll die Multiplikation wirklich bei jeder Aenderung durchgefuehrt werden?

### 6.2.6 Beispiel für Unit Rechteck: Verbesserung #2

```
// interne Daten
double a;       // 1. Seite
double b;       // 2. Seite
// double area; // Rechtecksflaeche, Attribut wird entfernt

// Schnittstelle (Interface)
// Attribut area wird entfernt
// Funktionen setA(), setB(), getA(), getB(), getArea()
void setA(double newA)
{
a = newA;
}

double getArea(void)
{
        return a * b;
}
```

**Hinweis**: Dank Datenkapsel darf das Attribut area entfernt werden. Die Schnittstelle ändert sich dadurch nicht.

### 6.2.7 Unit nutzen

```
#include "foo.h"
// dadurch wird die Schnittstelle der Unit foo bekanntgemacht
```

### 6.2.8 Unit-Schnittstelle definieren (in Headerdatei)

```
// Datei: foo.h
#ifndef FOO_H_
#define FOO_H_

// Deklarationen


#endif /* FOO_H_ */
```

**Hinweis**: include-Guard: verhindert das mehrfache include derselben Datei

### 6.2.9 Deklarationsreihenfolge in der Headerdatei (*.h)

**Achtung**: kein using namespace ... in Headerdateien!

1. Dateikommentar
2. #include der verwendeten System-Header (iostream, etc.)

#include <...>
3. #include der projektbezogenen Header (#include "...")
4. Konstantendefinitionen
5. typedefs und Definitionen von Strukturen
6. Allenfalls extern-Deklarationen von globalen Variablen
7. Funktionsprototypen, inkl. Kommentare der Schnittstelle, bzw. Klassendeklarationen

---

**Hinweis**: Punkte 2-7 sind innerhalb des inlcude-Guards.

---

#### 6.2.10 Reihenfolge in der Implementierungsdatei (*.cpp)

1. Dateikommentar
2. #include der verwendeten System-Header (iostream, etc.)
3. #include der projektbezogenen Header
4. allenfalls globale Variablen und statische Variablen
5. Präprozessor-Direktiven
6. Funktionsprototypen von lokalen, internen Funktionen
7. Definition von Funktionen und Klassen
   (Kommentare aus Headerdatei nicht wiederholen!)

#### 6.2.11 #include-Konzept

- Mit den #includes wird oft ein Riesenchaos veranstaltet
- Der Ëinfachheit halber"werden ab und zu einfach alle oder fast alle Headerdateien inkludiert
- Das muss unbedingt verhindert werden

**Regel: In jeder Datei (*.h, *.cpp, *.c) werden genau die Headerdateien inkludiert, welche diese Datei selbst benötigt!**

#### 6.2.12 Unit compilieren

**g++ -cfoo.cpp**

- Dadurch entsteht noch kein ausführbares Programm, sondern nur die Datei foo.o, der Objectcode
- Dies muss mit allen *.cpp-Dateien gemacht werden

#### 6.2.13 Units linken

g++ -o foo foo.o goo.o hoo.o

- Alle Objectdateien müssen gelinkt werden
- Dadurch werden allenfalls noch offene Verbindungen (Links) zu aufgerufenen Funktionen aufgelöst

#### 6.2.14 Buildprozess

- Der Buildprozess beinhaltet alle Schritte, um ein ausführbares Programm zu erhalten, bzw. aufzubauen (englisch to build)
  g++ -c foo.cpp
  g++ -c goo.cpp
  g++ -c hoo.cpp
  g++ -o foo foo.o goo.o hoo.o

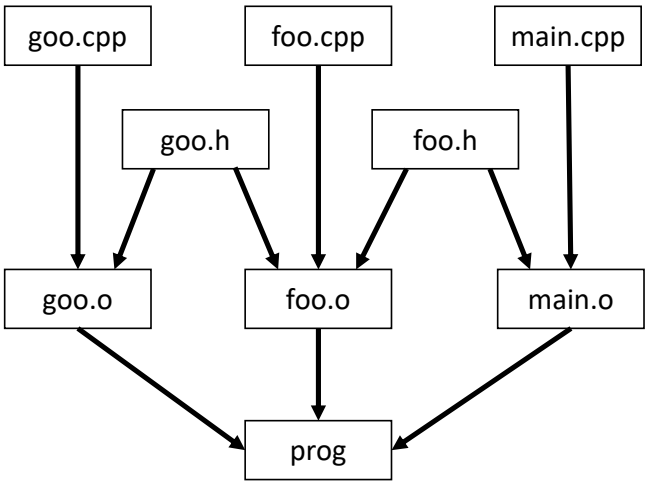
---

**Hinweis**: Es wäre mühsam, wenn diese Befehle jedesmal neu eingetippt werden müssten. Deshalb wird in der Praxis oft ein Buildtool eingesetzt, z.B. make.

---

## 6.3 Make-Tool

### 6.3.1 Abhängigkeiten zwischen Dateien




### 6.3.2 make-File

- In einem make-File können Abhängigkeiten definiert werden
- Wenn eine Datei geändert wurde, dann werden alle Operationen ausgeführt mit den Dateien, welche von dieser geänderten Datei abhängen
- Der Befehl (g++) wird z.B. nur dann ausgeführt, wenn sich an den Dateien, zu denen eine Abhängigkeit besteht, etwas geändert hat

### 6.3.3 Beispiel: makefile

```
# makefile
cc=g++
CFLAGS= -c -Wall
LFLAGS= -Wall
OBJ=foo.o goo.o main.o
EXE=prog

${EXE}:  ${OBJ}
         ${cc} ${LFLAGS} -o ${@} ${OBJ}

foo.o:   foo.h goo.h foo.cpp
         ${cc} ${CFLAGS} -o ${@} foo.cpp

goo.o:   goo.h goo.cpp
         ${cc} ${CFLAGS} -0 ${@} goo.cpp

main.o:  foo.h main.cpp
         ${cc} ${CFLAGS} -o ${@} main.cpp

clean:
         rm -f ${OBJ} ${EXE}
```

**Targets: Diese können mit make angesprungen werden, z.b. make foo.o**
**Wenn make ohne Parameter aufgerufen wird, dann wird das erste Target angesprungen.**

Abhänigkeitsliste, d.h. von diesen Dateien ist das Target abhängig

Befehl, der ausgeführt werden muss, falls etwas geändert hat.
Wichtig: Erstes Zeichen muss ein TAB sein.

# 7 Kapitel 7: Eclipse IDE

## 7.1 Eclipse

- Integrated Development Environment
  (IDE, Integrierte Entwicklungsumgebung)
- Open-Source Software (OSS)
- Offene, erweiterbare Architektur basierend auf Plug-Ins
- Implementiert in Java
- Auf verschiedenen Plattformen lauffähig (multi-platform)
- Für unterschiedliche Programmiersprachen (multi-language)

## 7.2 Workspace

- Der Workspace enthält vom Benutzer definierte Daten (Projekte und Ressourcen wie Ordner und Files)
- Er enthält alle User-Metadaten (Code, Scripts, Database objects, Konfigurationsdaten)
- Ein Benutzer arbeitet zu einer bestimmten Zeit genau in einem einzigen Workspace

### 7.2.1 Ressourcen (Resources)

- Oberbegriff für
  - Projekte
  - Ordner
  - Files
- Üblicherweise in einer hierarchischen Struktur betrachtet
- Können editiert werden

### 7.2.2 Project

- Ein logisches Speicherkonzept für die Speicherung von Programmen
- Gehört einem Workspace an
- Ist implementiert als Verzeichnis in einem Workspace

## 7.3 Debugger

### 7.3.1 Testen und Debugging

- Testen und Debugging sind zwei unterschiedliche Prozesse
- Das Ziel eines Tests ist, Fehler zu finden
- Das Ziel des Debuggings ist, diese Fehler zu lokalisieren und zu korrigieren

### 7.3.2 Funktionen eines Debuggers

- Betrachten von Variablenwerten
- Unterbrechung des Programmablaufs mit Breakpoints
- Schrittweise Ausführung von Programmen (Step into, Step out)

### 7.3.3 Assertions (Zusicherungen)

```
#include <cassert>
assert(i>0);

// in C:
#include <assert.h>
assert(i>0);
```

- Zweck:
  Überprüfung von logischen Annahmen während der Entwicklungsphase, speziell für die Überprüfung von Anfangs- und Endbedingungen in einer Funktion

- Das Programm bricht mit einer Fehlermeldung ab, falls das Argument den bool'schen Wert false besitzt. Im Beispiel oben: Abbrich, falls i <= 0

#### 7.3.4 Zu beachten bei Assertions

- assert() ist wirkungslos, wenn ohne Debugschalter compiliert wird. Dies ist in der (Release-)Version der Fall, die ausgeliefert wird
- Bei den assert()-Anweisungen darf deshalb kein Nebeneffekt programmiert werden, da dieser in der ausgelieferten Version fehlen würde
- Beispiel:
  assert(openFile(filename) == ok);
  Wenn ohne Debugschalter compiliert wird, würde das File nicht mehr geöffnet