Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

# Übung:
# Datenvisualisierung und GPU-Computing

## Programmieren in C und C++

**Michael Vetter**
michael.vetter@uni-hamburg.de

# Agenda

- Ablauf der Übung
- Der C++-Compiler
- Datentypen und Operatoren
- Zeiger und Referenzen
- Kontrollfluss
- Eingabe- und Ausgabeoperationen
- Überladen von Funktionen
- Sichtbarkeit und Lebensdauer von Variablen
- Komplexe Datentypen
- Pause

# Vorstellung der Runde

Michael Vetter
Klimacampus der Universität Hamburg

michael.vetter@uni-hamburg.de

Grindelberg 5, D-20142 Hamburg
Raum 309

Sprechstunde: nach Vereinbarung

# Links

http://openbook.galileocomputing.de/c_von_a_bis_z/

http://www.cplusplus.com/

www.google.de

# Kurze Einführung in die Programmiersprachen C und C++

Was ist C:

- Prozedurale Programmiersprache
- Ursprünglich für Unix Systemprogrammierung entwickelt
- Compiler für sehr viele Plattformen verfügbar

Was ist C++:

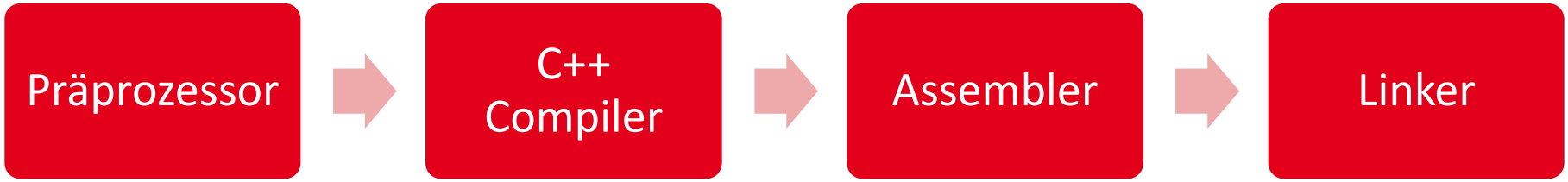
- Objektorientierte Programmiersprache
- Entwicklung aus der Programmiersprache C

# Kurze Einführung in die Programmiersprachen C und C++

C++-Compiler:

- GNU Compiler Collection (unter Windows MinGW)
- Microsoft Visual C++
- Intel C++ Compiler
- Borland C++ Builder

# Ablauf des C++-Compilers



# Einstiegspunkt

Grundgerüst eines C-Programmes:

```
1  int main()
2  {
3      // Ein Kommentar
4      /* noch ein Kommentar */
5      return 0;
6  }
```

# Präprozessordirektiven (1)

Erweitertes Grundgerüst eines C-Programmes:

```
1  #include <eine_Headerdatei>
2  #include <noch_eine_Headerdatei.h>
3
4  int main()
5  {
6      …
7  }
```

Fügt den Inhalt der angegebenen Datei in dieses Dokument ein.

# Datentypen und Operatoren (1)

Datentypen bestehen aus der Menge der möglichen Werte sowie die mit diesen Werten möglichen Operationen.

Grunddatentypen: int, char, bool, float, double

Spezifisierung: short, long, signed, unsigned

Zuweisungs-Operatoren: =

Arithmetrische-Operatoren: +, -, /, *, %

Vergleichs-Operatoren: ==, <, >, <=, >=, !=

# Datentypen und Operatoren (2)

Deklaration:

```
int a;
int b = 0;
```

Operationen:

```
a = 12;
a = 5 * b;
```

Inkrementieren einer Variable:

```
a = a + 1;
a++;
a += 1;
```

# Bitoperationen

| | | |
|---|---|---|
| unsigned int bit_value = 0x1257; | 0001 0010 0101 0111 | |
| unsigned int mask = 0x3122; | 0011 0001 0010 0010 | |
| unsigned int b = bit_value \| mask; | 0011 0011 0111 0111 | OR |
| unsigned int c = bit_value & mask; | 0001 0000 0000 0010 | AND |
| unsigned int d = ~mask; | 1100 1110 1101 1101 | NOT |
| unsigned int e = bit_value & ~mask; | 0000 0010 0101 0101 | AND NOT |
| unsigned int f = bit_value ^ mask; | 0010 0011 0111 0101 | XOR |

# Verzweigung

- IF-Verzweigung

```
if (condition)
  statement
```

- IF-ELSE-Verzweigung

```
if (condition)
  statement
else
  statement
```

- SWITCH-Verzweigung

```
switch (selector)
{
    case label:
        statement
    default:
        statement
}
```

# Schleifen

- WHILE-Schleife

```
while (condition)
  statement
```

- FOR-Schleife

```
for (initializer; condition; update)
  statement
```

- Vorzeitiges verlassen einer Schleife oder einer SWITCH-Verzweigung mittels

```
break;
```

# IO

- C++: IO-Stream

```
#include <iostream>
std::cout << „Text“ << variable << std::endl;
std::cin >> variable;
```

- C: Std.-IO

```
#include <stdio.h>
printf(format_description,variables);
scanf(format_description,variables);
```

# IO - Formatbeschreiber

Integer:

| | |
|---|---|
| %d | int dezimal |
| %hd | short dezimal |
| %ld | long dezimal |
| %o | int oktal |
| %ho | short oktal |
| %lo | long oktal |
| %x | int hexadezimal |
| %hx | short hexadezimal |
| %lx | long hexadezimal |
| %u | unsigned int dezimal |
| %hu | unsigned short dezimal |
| %lu | unsigned long dezimal |
| %i | int dezimal 12, oktal 012, hexadezimal 0x12 |
| %hi | short dezimal 12, oktal 012, hexadezimal 0x12 |
| %li | long dezimal 12, oktal 012, hexadezimal 0x12 |

# IO - Formatbeschreiber

Float:

| | |
|---|---|
| %f | float fixed |
| %lf | double short fixed |
| %LF | long double fixed |
| %e | float scientific |
| %le | double short scientific |
| %Le | long double scientific |
| %g | float fixed und scientific |
| %lg | double short fixed und scientific |
| %Lg | long double fixed und scientific |

Character:

| | |
|---|---|
| %c | Zeichen |

C-String:

| | |
|---|---|
| %s | Zeichenkette |

# Arrays

- Die Deklaration

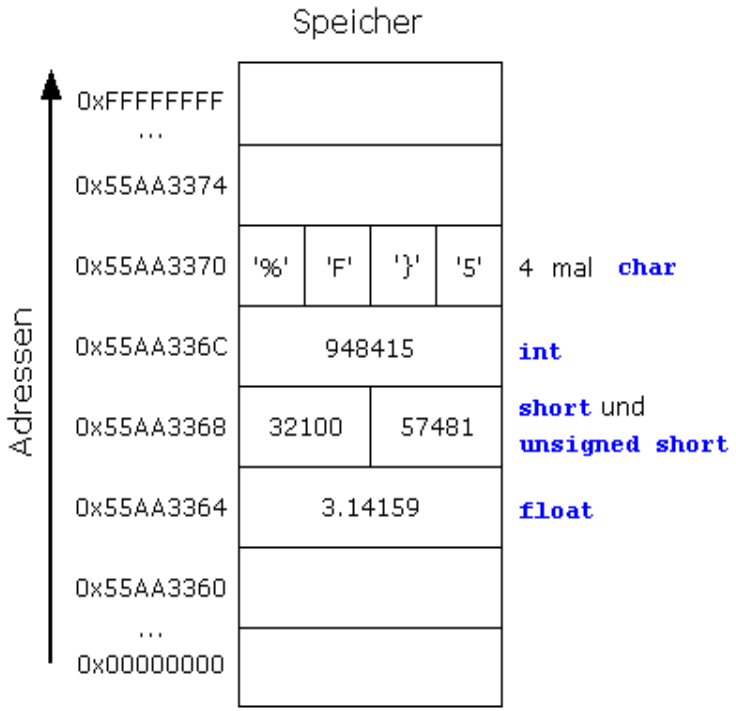
```
int a[10];
```

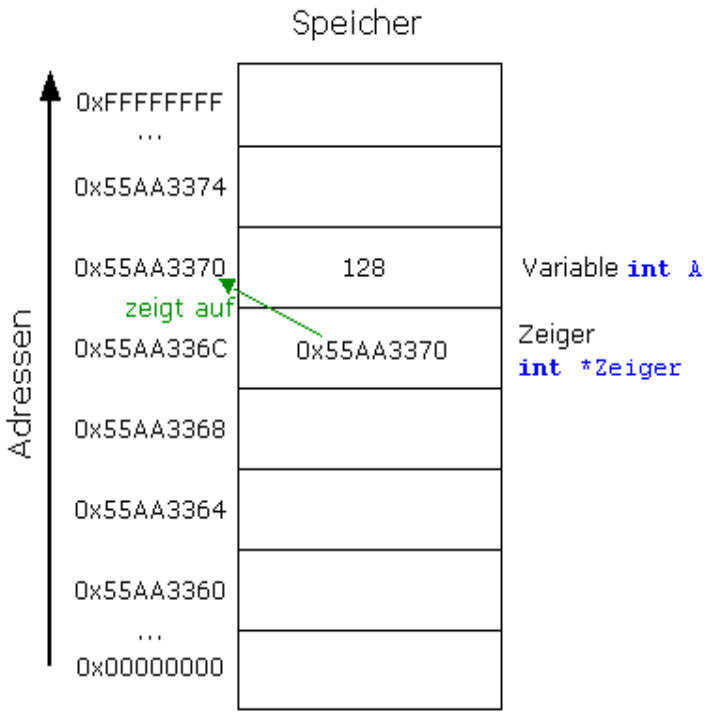
definiert ein Array `a` mit 10 Elementen vom Typ `int`. Auf die einzelnen Elemente kann mit `a[0]` bis `a[9]` zugegriffen werden.

- Auf diese Art lassen sich nur Arrays definieren, deren Größe zur Compilezeit bekannt ist (variable Größe → später)
- Mehrdimensionale Arrays definiert man analog:

```
int a[10][20];
a[5][2] = 19;
```

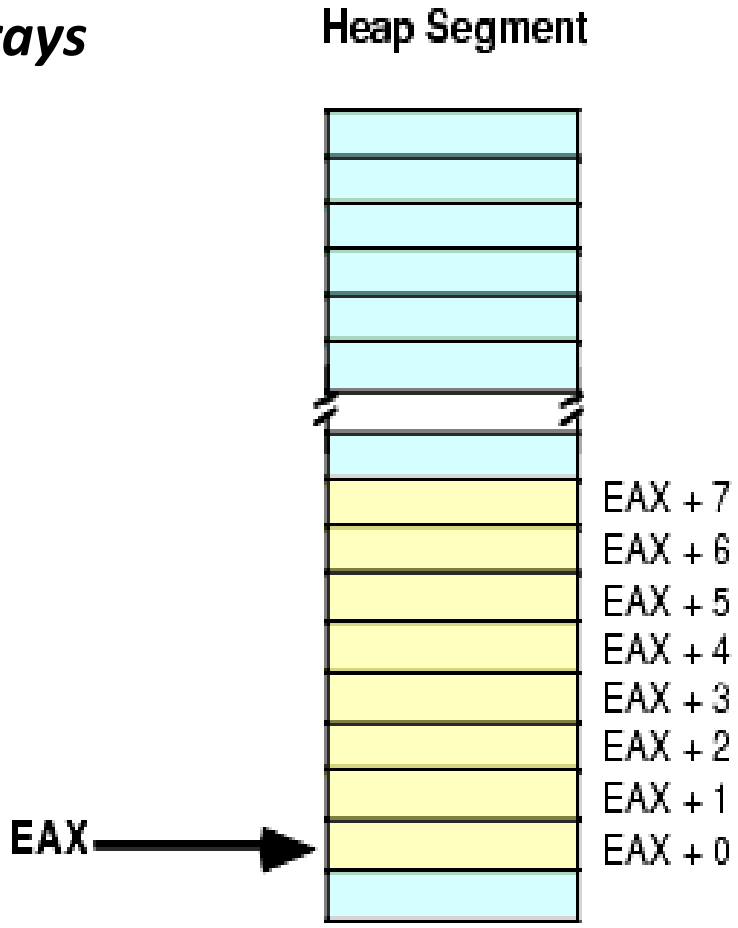
# Speicher , Adressen , Zeiger (Pointer)



# *Zeiger: Speicheranforderung eines Arrays*

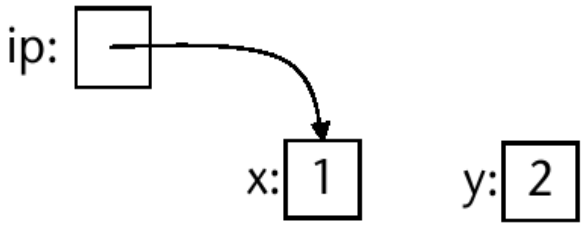
```
int EAX[8];
```



- Für einen beliebigen Datentyp `type` bezeichnet `type*` den Typ "Zeiger auf `type`", d.h. eine Variable vom Typ `type*` kann die *Adresse* eines "Objektes" vom Typ `type` aufnehmen
- Wichtige Operatoren:
  - Adressoperator `&`
  - Dereferenzierungsoperator `*` (Inhaltsoperator)
- Beispiel:

```
int  x = 1;
int  y = 2;
int* ip;        // "Zeiger auf int"

ip  = &x;       // ip enthält Adresse von x
y   = *ip;      // y ist jetzt 1
*ip = 0;        // x ist jetzt 0
ip  = &y;       // ip zeigt jetzt auf y
```

- Illustration:




- Zeigern kann der Wert 0 zugewiesen werden (Null-Pointer). So ist prüfbar, ob ein Zeiger belegt ist oder nicht:

```
int  x  = 1;
int  y  = 2;
int* ip = 0;

if (ip) y = *ip;    // Keine Zuweisung
ip = &x;            // ip enthält Adresse von x
if (ip) y = *ip;    // y = 1
```

# Referenzen

- “Eine Referenz ist ein alternativer Name für ein Objekt”
- Vorstellung: Eine Referenz ist ein (nicht veränderbarer) Zeiger auf ein Objekt, der bei jeder Benutzung dereferenziert wird
- Da eine Referenz immer an ein Objekt gebunden ist, muss man sie zwingend initialisieren
- Beispiel:

```
int  x  = 17;
int& xr = x;

int y = x;      // y = 17
int z = xr;     // z = 17, da xr Synonym für x
```

- Anwendung: Call-by-reference

# Funktionen

- Funktionen erlauben es Code auszulagern und in kleine, zusammengehörenden Teile zu zerschneiden.

```
rückgabe_type
funktions_name(0_bis_beliebige_anzahl_an_parametern);
```

- Beispiel

```
int Add(int lhs, int rhs) //Definition einer Funktion
{
      return lhs + rhs;
}
```

# Programmaufbau

```
#include <stdlib.h>

void print();

int main() {
    print();
    return 0;
}

void print() {
    cout << "Hello world!" << endl;
}
```

# Parameterübergabe (1)

- In Java erfolgt die Parameterübergabe durch übergeben von Werten
- In C++ kann die Parameterübergabe erfolgen durch Übergabe von:
  - Einem Wert
  - Einer Referenz
  - Einer konstanten Referenz

```
void f( int iA, int &riB, const int &criC );
```

# Parameterübergabe (2)

- Erfolg die Übergabe durch einen Wert, wird eine Kopie des Parameters angelegt

```
void f(int iN) { iN++; }
int main() {
    int i_x = 2;
    f(i_x);
    printf("Wert der Variable i_x ist: %d"
,i_x);
}
```

- `f(.)` arbeitet mit einer Kopie (nicht mit original Variable), somit ist die Ausgabe für `x_i` „2“

# Parameterübergabe (3)

```
void f(int *piP) {
    *piP = 5;
    piP = NULL;
}
int main() {
    int i_x=2;
    int *pi_q = &i_x;
    f(pi_q); // hier, i_x == 5, aber pi_q != NULL
}
```

- Der Zeiger wird als ein Wert übergeben, aber das Objekt auf das er verweist kann sich ändern
- Wird auch „call by reference“ genannt

# Parameterübergabe (4)

```
void f(int &riN) { riN++; }
int main() {
    int i_x = 2;
    f(i_x);
    cout << i_x;
}
```

- Der Parameter wurde geändert (wie auch bei der Übergabe von Zeigern)
- Eigentlich wurde hier ein Zeiger übergeben (keine Kopie!!!)

# C/C++-Exkurs: Parameterübergabe (5)

- Problem: einer Referenz sieht man nicht an, daß sie in der Methode verändert wird!
- Guideline:
  - Referenzparameter immer nur mit **const** verwenden, z.B.

```
void doIt (const int& x);
```

  - Falls (Call-by-reference) Parameter verändert werden soll, dann Pointer verwenden z.B.

```
void doIt (const int* x);
```

# Überladen von Funktionen (1)

- In C++ ist es möglich Funktionen zu definieren, die sich nur in der Anzahl bzw. den Typen der Parameter unterscheiden (*overloading*) → Unterscheidung nur durch den Rückgabetyp reicht jedoch nicht!

# Überladen von Funktionen (2)

- Beispiel:

```
float  sqrt(float  value);
double sqrt(double value);
```

- Bei einem Aufruf wird anhand der realen Parameter entschieden, welche Variante auszuführen ist:

```
float  f = 3.14159f;
double d = 2.71828;

float  x = sqrt(f);   // Ruft float-Variante auf
double y = sqrt(d);   // Ruft double-Variante auf
```

# Sichtbarkeit von Variablen

- Variablen sind grundsätzlich nur innerhalb des Blockes sichtbar, in dem sie definiert wurden.
- Variablen die in keinem Block definiert wurden sind global sichtbar (innerhalb der Quellcodedatei).
- Variablen, die über Dateigrenzen hinweg global sichtbar seien sollen werden mit dem Schlüsselwort **extern** deklariert und in nur einer Quellcodedatei ohne das Schlüsselwort **extern** definiert (Speicherzuweisung). Externe Variablen können auch innerhalb eines Blockes definiert werden.

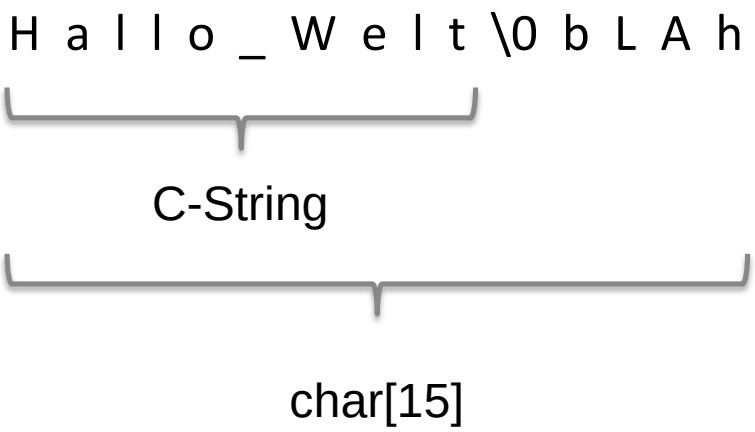
# Lebensdauer von Variablen

- Die Lebensdauer von Variablen, die innerhalb einer Funktion definiert wurden, endet in der Regel mit dem Ende der Funktion.
- Soll eine lokale Variable über mehrere Aufrufe einer Funktion hinweg existieren, so ist sie mit dem Schlüsselwort **static** zu definieren.

# C-Strings

Definition:

Ein C-String ist eine Zeichenkette in einem char-Array, welche mit dem Zeichen ‘\0‘ terminiert wird.



# Structs

- Ein C-Struct
  - Gruppieren von semantisch zusammenhängenden Daten
  - Ähnlich einer Klasse

```
struct Student {
    int m_iId;
    bool m_IsGrad;
}; // the declaration must end with ';'
```

# Aufzählungen

- Neue Datentypen mit festen Werten können wie folgt definiert werden:

```
enum Color { RED, BLUE, YELLOW };

Color col;
col = BLUE;
```

- Der Wert der Variablen vom Typ Color kann einen der Folgenden Werte annehmen { RED, BLUE, YELLOW }

- Zugriff auf Variablen einer Struktur:

```
t_student1.m_iId = 123;
t_student2.m_iId = t_student1.m_iId + 1;
```

- Verschachtelte Strukturen:

```
struct Address {
        string m_City;
        int m_Zip;
};
struct Student {
        int m_iId;
        bool m_bIsGrad;
        Address m_Address;
};
```

- Neue Datentypen können durch bereits existierende Datentypen definiert werden:

```
typedef double EuroType;
EuroType hourSalary = 10.50;
```

# PAUSE

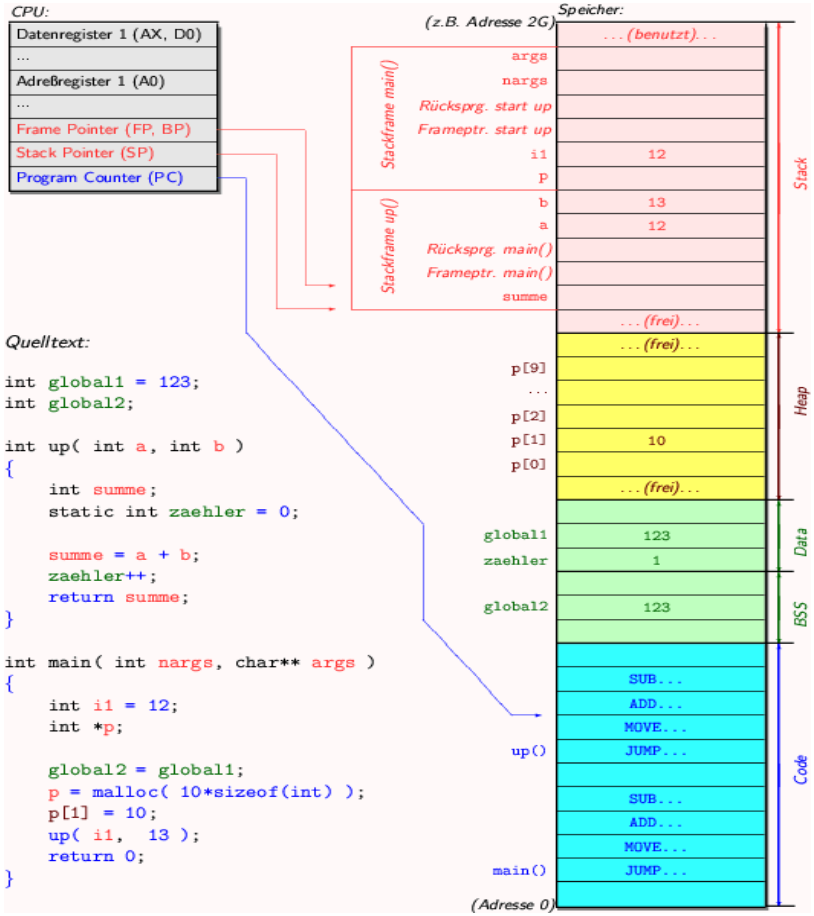
Michael Vetter
michael.vetter@uni-hamburg.de

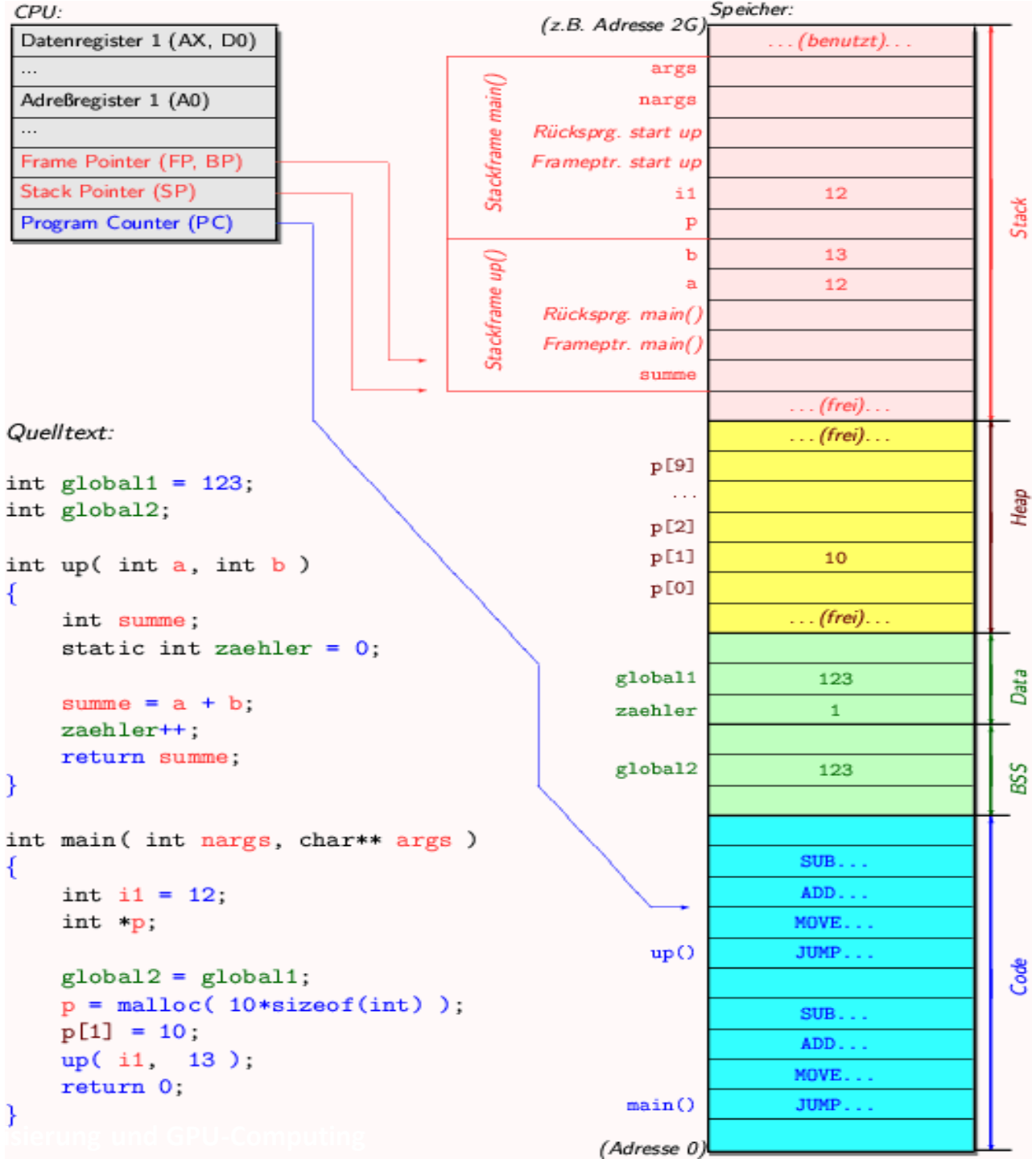
# Agenda

- Dynamische Speicherverwaltung
- File-I/O (2)
- Codestrukturierung
- Klassen in C++
- Konstruktoren, Destruktoren, Copy-Konstruktoren
- Zuweisungs-Operator
- Vererbung
- Kapselung
- Templates
- STL

# *Speicher eines Prozesses:*

- *Stack*
- *Heap*
- *...*




CPU:
Datenregister 1 (AX, D0)
...
Adreßregister 1 (A0)
...
Frame Pointer (FP, BP)
Stack Pointer (SP)
Program Counter (PC)
Speicher:
(z.B. Adresse 2G)
... (benutzt) ...
Stackframe main()
args
nargs
Rücksprg. start up
Frameptr. start up
i1
12
p
Stackframe up()
b
13
a
12
Rücksprg. main()
Frameptr. main()
summe
... (frei) ...
Stack
Quelltext:
int global1 = 123;
int global2;
int up( int a, int b )
{
int summe;
static int zaehler = 0;
summe = a + b;
zaehler++;
return summe;
}
int main( int nargs, char** args )
{
int i1 = 12;
int *p;
global2 = global1;
p = malloc( 10*sizeof(int) );
p[1] = 10;
up( i1, 13 );
return 0;
}
... (frei) ...
p[9]
...
p[2]
p[1]
10
p[0]
... (frei) ...
Heap
global1
123
zaehler
1
Data
global2
123
BSS
SUB...
ADD...
MOVE...
up()
JUMP...
SUB...
ADD...
MOVE...
main()
JUMP...
Code
(Adresse 0)

# Dynamische Speicherverwaltung (C++-Variante)

- In C++ gibt es keine automatische Speicherbereinigung

```
int *pi_p = new int;          // pi_p zeigt auf
neuen                         // Speicherplatz
...
delete pi_p; // Speicher MUSS wieder
             // freigegeben werden
```

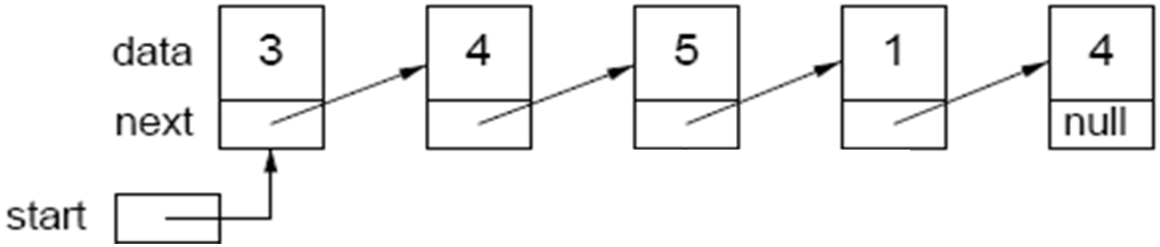
- Aber niemals

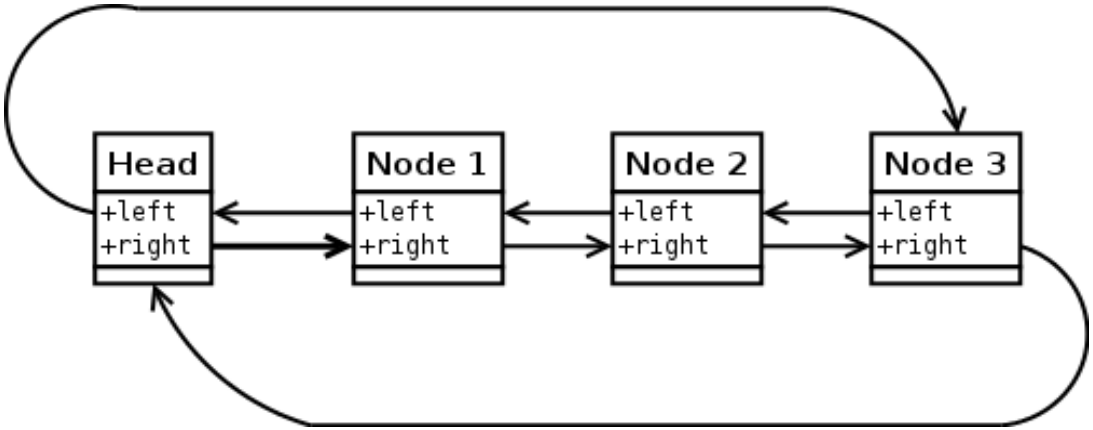
```
int *pi_p; // pi_p ist nicht initialisiert!!
*pi_p = 3;

int *pi_p=NULL; // pi_p ist ein null-Zeiger
                // (ungültige Adresse!!!
*pi_p = 3;
```

- Mache niemals:

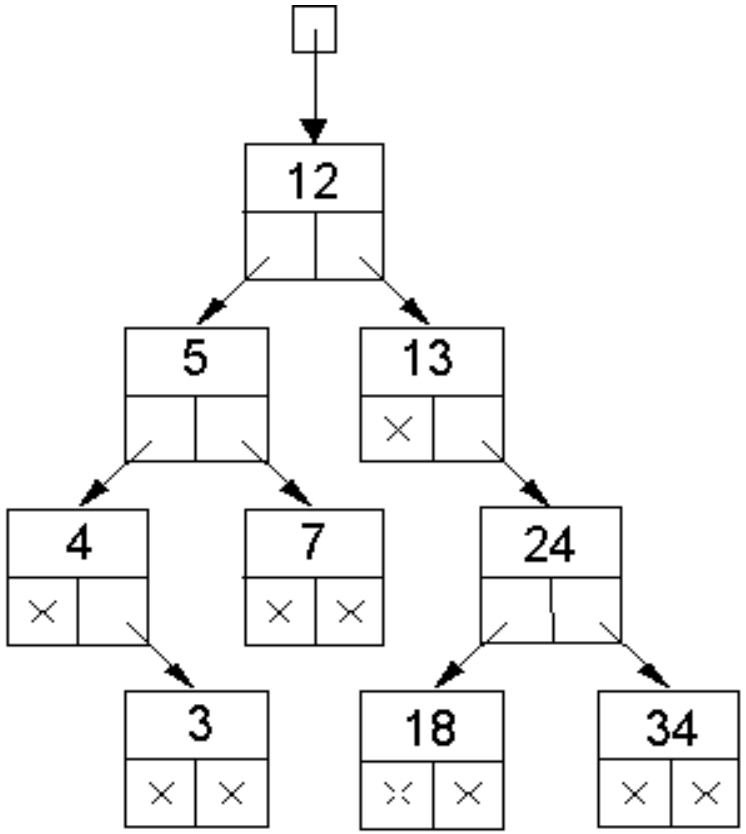
```
int *pi_p = new int;
delete pi_p;
*pi_p = 5; // pi_p ist nicht mehr gültig!!

int *pi_p, *pi_q;
pi_p = new int;
pi_q = pi_p; // pi_p zeigt auf selbe Adresse wie pi_q
delete pi_q;
*pi_p = 3; // Adresse pi_q = pi_p is nicht mehr
           // reserviert!! (abschüssiger Zeiger)

int *pi_p = new int;
pi_p = NULL; // ändere keinen Zeiger ohne den
             // Speicherplatz freizugeben auf den er
             // vorher gezeigt hat (Speicherüberlauf)
```

# *Dynamische Datenstrukturen: Listen*



# *Dynamische Datenstrukturen: Bäume*



# File-I/O – C-Variante (1)

Einbinden des Header-Files:

```
#include <stdio.h>
```

Öffnen der Datei:

```
FILE *file = fopen(filename, parameter);
```

„r“, „w“, „rw“, „a“

Schließen der Datei:

```
fclose(file);
```

# File-I/O – C-Variante (2)

Formatierte Ausgabe:

**fprintf(***file*, *formatbeschreiber*, *liste_von_variablen***);**

Formatierte Eingabe:

**fscanf(***file*, *formatbeschreiber*, *liste_von_variablen***);**

# File-I/O – C++-Variante (1)

Einbinden des Header-Files:

```
#include <fstream>
```

Öffnen der Datei:

```
std::ifstream   infile (filename, parameter);
std::ofstream   outfile (filename, parameter);
std::fstream    file (filename, parameter);
```

Parameter: std::ios::in, std::ios::out, std::ios::app, std::ios::trunc, …

Schließen der Datei:

```
file.close();
```

# File-I/O – C++-Variante (2)

Formatierte Ausgabe:

```
file << variable;
```

Formatierte Eingabe:

```
file >> variable;
```

# Codestrukturierung

- Der Code besteht aus Direktiven, Deklarationen und Definitionen
- Übliche Aufteilung des Codes:
  - Header-Dateien
  - Quellcode-Dateien

**Präprozessor-Direktiven**
- Compilerschalter
- Includes
  - Systemincludes
  - Individuelle Includes
- Symbolische Konstanten
- Makros

**Deklarative Teile**
- Datendeklaration
  - Deklaration von Datenstrukturen
  - Deklaration von Datentypen
- Variablendeklarationen
  - Externverweise auf globale Variablen
- Funktionsprototypen
  - Externverweise auf globale Funktionen
  - Vorwärtsverweise auf lokale Funktionen

**Code**
- Variablen außerhalb von Funktionen
  - lokale Variablen
  - globale Variablen
- Funktionen
  - lokale Funktionen
  - globale Funktionen

# Präprozessordirektiven

**#define** *name*

**#define** *name wert*

**#define** *name(platzhalter) makro*

**#ifdef** *name*

**...**

**#endif**

**#ifndef** *name*

**...**

**#endif**

**#ifdef** *name*

**...**

**#else**

**...**

**#endif**


Michael Vetter
michael.vetter@uni-hamburg.de

# Klassen, Definition

- Allgemein:

```
class Name
{
   public:
      // Öffentliche Komponenten
      // (Konstruktoren, Methoden usw.)

   protected:
      // Geschützte Komponenten

   private:
      // Private Komponenten
};
```

Nicht vergessen!

# Klassen, Beispiel

Zeigt dem Compiler an, dass innerhalb der Methode keine Membervariablen verändert werden dürfen.
⇒ Zusätzliche Optimierung durch den Compiler möglich.

```
class Vector2D
{
   public:
      float x() const;
      float y() const;

      void  setX(float value);
      void  setY(float value);

   protected:
      float mElement[2];
};
```

# Klassen, Programmaufbau (1)

- Klassen werden normalerweise in den *Header Dateien* deklariert
  - Klassenname.h
- Die Implementierung von Funktionen kommt in die *source Dateien*
  - Klassenname.cpp
  - Klassenname.C, Klassenname.cc, nicht verwenden

# Klassen, Programmaufbau (2)

- Da sich die Implementierung nicht innerhalb der Definition befindet, muss man irgendwie die Verbindung zur jeweiligen Klasse herstellen
- Dazu dient folgende Syntax:

```
Rückgabetyp Klasse::Methode(Parameter) {
    // Implementierung
}
```

- Die Implementierung einer Klasse kann ohne weiteres auf mehrere Quelldateien verteilt werden

# Klassen, Programmaufbau (3)

```
#include "Vector2D.h"

float Vector2D::x() const
{
    return mElement[0];
}

[...]

void Vector2D::setX(float value)
{
    mElement[0] = value;
}
```

# Konstruktoren (1)

- Ein *Konstruktor* ist eine spezielle Methode ohne Rückgabewert, deren Namen mit dem der Klasse übereinstimmt
- Bei der Erzeugung einer Klasseninstanz wird nach der Speicherreservierung automatisch der Konstruktor aufgerufen
- Definiert der Programmierer keinen eigenen Konstruktor, dann wird vom Compiler automatisch ein parameterloser *Default-Konstruktor* erzeugt, der aber keine Funktionalität besitzt

# Konstruktoren (2)

- Guideline: immer eigenen Konstruktor definieren!
- Man kann auch mehrere Konstruktoren mit unterschiedlichen Parameterlisten für eine Klasse definieren. Es ist jedoch nicht möglich, aus einem Konstruktor heraus einen anderen der gleichen Klasse aufzurufen
  - Leider ...
- Workaround: Private Initialisierungsmethode, die von allen Konstruktoren genutzt wird

# Konstruktoren (3)

- Definition:

```
class Vector2D {
public:
    Vector2D();
    Vector2D(float x, float y);

private:
    void init(float x, float y);
};
```

# Konstruktoren (4)

- Implementierung:

```
Vector2D::Vector2D() {
    init(0, 0);
}

Vector2D::Vector2D(float x, float y) {
    init(x, y);
}

void Vector2D::init(float x, float y) {
    mElement[0] = x;
    mElement[1] = y;
}
```

# Destruktoren (1)

- Ein *Destruktor* ist das Gegenstück zu einem Konstruktor
- Wird automatisch für jedes Objekt am Ende seiner Lebensdauer aufgerufen
- Ein Destruktor ist immer parameterlos und besitzt ebenfalls keinen Rückgabewert
- Name setzt sich aus dem Klassen-Namen und einer vorangestellten Tilde zusammen
- Destruktoren sollten immer dann verwendet werden, wenn in einer Klasse Member-Variablen dynamisch angelegt werden! Anderenfalls wird der reservierte Speicher nie freigegeben!

# Destruktoren (2)

- Beispiel:

```
Vector2D::~Vector2D()
{
    // Bei Vector2D ist nichts freizugeben!
}

class Vector
{
protected:
     float * a;
public:
     Vector( int n ) { a = new float[n]; }
     ~Vector() { delete [] a; }
}
```

# Copy-Konstruktoren (1)

- Der Copy-Konstruktor wird aufgerufen, wenn ein Objekt:
  - als Wert übergeben wird

```
CIntList f( CIntList clL );
```

  - Bei der Initialisierung mit einem bereits existierenden Objekt

```
int main() {
   CIntList cl_l1, cl_l2;
    ...
   cl_l2 = f( cl_l1 );       // Kopie von cl_l1
   CIntList cl_l3 = cl_l1;
}
```

# Copy-Konstruktoren (2)

- von einer Funktion zurückgegeben wird

```
CIntList f( CIntList clL ) {
   CIntList cl_tmp1 = clL;    // Kopie von clL
   CIntList cl_tmp2(clL);     // Kopie von  clL
    ...
   return cl_tmp1;            // Kopie von cl_tmp1
}
```

# Copy-Konstruktoren (3)

- Deklaration

```
class CIntList {
public:
   CIntList();                              // „default“ ctor
   CIntList( const CIntList &cclL ) // copy ctor
     ...
};
```

# Copy-Konstruktoren (4)

- Definition

```
CIntList::CIntList(const CIntList &crclL):
    m_piItems( new int[crclL.m_iArraySize] ),
    m_iNumItems( crclL.m_iNumItems ),
    m_iArraySize( crclL.m_iArraySize )
{
   for ( int i_k = 0; i_k < m_iNumItems; i_k ++ ) {
      m_piItems[i_k] = crclL.m_piItems[i_k];
   }
}
```

# Der Zuweisungsoperator (1)

- `operator =`
  - In C++ kann so ein Objekt einem anderem zugewiesen werden

```
CIntList cl_l1, cl_l2;
...
cl_l1 = cl_l2;
```

  - Ohne ein eindeutig Zuweisungsoperator (`operator =`), müssen die Objekte Byte-weise kopiert werden (auch *flat copy / shallow copy* genannt)

# Der Zuweisungsoperator (2)

- Ohne `operator =`
  - Wenn das Objekt einen Zeiger beinhaltet, dann würde der Zeiger des neuen Objektes auf dieselbe Adresse zeigen wie das Ausgangsobjekt
  - Wird der Zeiger von `cl_11` gelöscht, dann verweist der Zeiger von `cl_12` auf eine ungültige Adresse

# Der Zuweisungsoperator (3)

- Unterschied zwischen Zuweisungsoperator und Copy-Konstruktor (am Bsp. `cl_11 = cl_12`)
  - `cl_11` ist ein bereits initialisiertes Objekt; enthält dieses einen Zeiger, so muss dieser gelöscht werden, bevor dem Objekt etwas neu zugewiesen werden kann
  - Eine Variable kann sich nicht selber zugewiesen werden → so etwas sollte man niemals machen
  - Der Code des `operator =` muss einen Rückgabewert besitzen

# Der Zuweisungsoperator (4)

- operator =

```
CIntList & CIntList::operator = ( const CIntList &crclL )
{
   // überprüfe ob Eigenzuweisung (self assignemnt)
   if ( this == &crclL )
       return *this;
   else
   {
      delete [] m_piItems;                        // Speicher freigeben
      m_piItems = new int[crclL.m_iArraySize]; // neuen Speicher holen
      m_iArraySize = crclL.m_iArraySize;        // Inst.var. kopieren

      // Kopiere crclL in das neue Array
      // zuweisen m_iNumItems
      for ( m_iNumItems=0; m_iNumItems < crclL.m_iNumItems;
            m_iNumItems ++ )
          m_piItems[m_iNumItems]= crclL.m_piItems[m_iNumItems];
   }
   return *this; // Rüchgabe CIntList
}
```

# Vererbung (1)

- Eine Klasse kann Eigenschaften einer anderen Klasse durch die Ableitung übernehmen
- Basisklasse: Eine Klasse kann als Basis zur Entwicklung einer neuen Klasse dienen, ohne dass ihr Code geändert werden muss. Dazu wird die neue Klasse definiert und dabei angegeben, dass sie eine abgeleitete Klasse der Basisklasse ist.
- Alle öffentlichen Elemente der Basisklasse gehören auch zur neuen Klasse, ohne dass sie erneut deklariert werden müssen.
- Wiederverwendung des Codes
- Spezialisierung

# Vererbung (2)

```
class Person {
public:
    string Name, Adresse, Telefon;
};

class Partner : public Person {
public:
    string Kto, BLZ;
};

class Mitarbeiter : public Partner {
public:
    string Krankenkasse;
};

class Kunde : public Partner {
public:
    string Lieferadresse;
};

class Lieferant : public Partner {
public:
    tOffenePosten *Rechnungen;
};
```

# Vererbung (3)

- Subtyp kann an die Stelle eines Basistyps treten:

```
Person person;
Mitarbeiter mitarbeiter;

    person = mitarbeiter;     // ok
    mitarbeiter = person;     // das mag der Compiler nicht
```

- Subklassen-Konstruktor 'called' zunächst Basisklassen-Konstuktor
  - Prinzip der kaskadierenden Konstruktoren
  - Bei den Destruktoren genau umgekehrt
  - Copy-Konstruktor wird nicht automatisch vererbt

# Methoden überschreiben

```
class tBasis
{
public:
    int TuWas(int a);
};

class tSpezialfall : public tBasis
{
public:
    int TuWas(int a);
};

int tSpezialfall::TuWas(int a)
{
    int altWert = tBasis::TuWas(a);
    ...
    return altWert;
}
```

# Kapselung

```
class Basis {
private:
   int privat;
protected:
   int protect;
public:
   int publik;
};

class Abgelitten : public Basis {
    void zugriff()
    {
        a = privat;  // Das gibt Ärger!
        a = protect; // Das funktioniert.
        a = publik;  // Das funktioniert sowieso.
    }
};

int main()
{
    Basis myVar;
    a = myVar.privat;  // Das läuft natürlich nicht.
    a = myVar.protect; // Das geht auch nicht.
    a = myVar.publik;  // Das funktioniert.
}
```

# Templates (1)

- Templates unterstützen direkt *generic programming*
  - *Generic programming* = Datentypen sind Parameter in Deklarationen
  - So ähnlich wie formale Argumente in "normalen" Deklarationen später tatsächliche Daten (= Werte) aufnehmen
  - Definition der Parameter von Funktionen und Klassen erfolgt durch Datentypen
- Verwende Template Funktionen um gleiche Operationen für zu unterschiedliche Typen definieren
- Beispiel:

```
// gibt größten Parameterwert zurück
template <class T> T max(T a, T b)
{
   return a > b ? a : b ;
}
```

# Templates (2)

```
void main()
{
   // max(int,int) is instantiated
   cout << "max(10, 15) = " << max(10, 15) << endl ;
   // max(char,char) is instantiated
   cout << "max('k', 's') = " << max('k', 's') << endl;
   cout << "max(10.1, 15.2) = " << max(10.1, 15.2) <<
endl;
}
```

- Compiler erkennt Type der Eingangsparameter
- Eine Instanz einer Funktion wird dementsprechend generiert

# Template-Klassen (1)

- Eine typische Template-Klasse:

```
template <class T>
class CStack
{

public:
   CStack(int = 10) ;
   ~CStack() { delete [] m_pStackPtr ; }
   bool Push(const T& crItem);
   bool Pop(T& rResult) ;
   bool IsEmpty() const { return m_iTop == -1 ; }
   bool IsFull() const { return m_iTop == m_iSize - 1; }

private:
   int m_iSize ; // Zähler für Anzahl Elemente auf Stack
   int m_iTop ;
   T* m_pStackPtr ;
} ;
```

# Template-Klassen (2)

```
// Konstruktor; vordefinierte Größe (m_iSize) ist 10
template <class T>
CStack<T>::CStack(int iS)
{
   m_iSize = iS > 0 && iS < 1000 ? iS : 10 ;
   m_iTop = -1 ; // initialisiere Stack
   m_pStackPtr = new T[m_iSize] ;
}

// speichere einen Wert auf Stack
template <class T>
int CStack<T>::Push(const T& crItem)
{
   if ( !IsFull() )
   {
      m_pStackPtr[++m_iTop] = crItem ;
      return true ; // erfolgreich
   }
   return false ; // fehlgeschlagen
}
```

# Template-Klassen (3)

```
#include <iostream>
#include "stack.h"
using namespace std ;

void main()
{
   typedef CStack<float> FloatStackType ;
   typedef CStack<int> IntStackType ;

   FloatStackType cl_fs(5) ;
   float f_f = 1.1 ;
   while ( cl_fs.push(f_f) )    // neues Elements bis
   {                            // Stack voll ist
      cout << f_f << ' ' ;
      f_f += 1.1 ;
   }
```

# Template-Klassen (4)

```
    // schreibe alle Elemente von cl_fs nach stdout
    while ( cl_fs.pop(f_f) )
        cout << f_f << ' ';

    IntStackType cl_is;
    int i_i = 1;
    while ( cl_is.push(i_i) )
    {
       cout << i_i << ' ';
       i_i += 1;
    }

    // schreibe alle Elemente von cl_is nach stdout
    while ( cl_is.pop(i_i) )
        cout << i_i << ' ';
}
```

# Template-Klassen (5)

- Die Deklaration und Definition von *generic classes/functions* (d.h. Templates) gehört in *eine* Datei (nicht zwei)
- Organisiere Deklaration und Definition zweckmäßigerweise so:
  - Deklaration in einem Header-File (`.h`), Implementierung in einem Source-File (`.cpp`, `.hh` oder `.inl`) und binde die Source Dateien am Ende des Header-Files ein.
  - Achtung: Kompiliere nicht den `.cpp`-File !!!

# Standard-Template-Library (STL) (1)

- Die Standard Template Library enthält viele effiziente Container, Algorithmen u.v.m., die für eigene Zwecke verwendet werden können
- Beispiel: dynamische Arrays

```
#include <vector>
…
vector<float> a;  // default-Größe (meist 0)
vector<float> b(10);  // 10 Elemente
```

# Standard-Template-Library (STL) (2)

- Verwendung wie bei herkömmlichen Arrays, zusätzlich z.B.:
  - Hinzufügen weiterer Elemente:

```
a.push_back(2.87f);   // hinten
```

  - Abfragen der Größe:

```
a.size();
```

# Standard-Template-Library (STL) (3)

- Eigene Datentypen können ebenso verwendet werden, z.B.:

```
#include "Vector2D.h"
#include <vector>

using namespace std;
...
vector<Vector2D> points;
points.push_back( Vector2D(1, 5) );  // anonyme Instanz
points.push_back( Vector2D(-3, 0) );

printf("%d …", points.size());
```

# Standard-Template-Library (STL) (4)

- Weitere nützliche STL-Komponenten
  - Container wie `map`, `list`, `stack`
  - Algorithmen wie `find()`, `sort()`, `min()`
  - Zeichenketten `string`