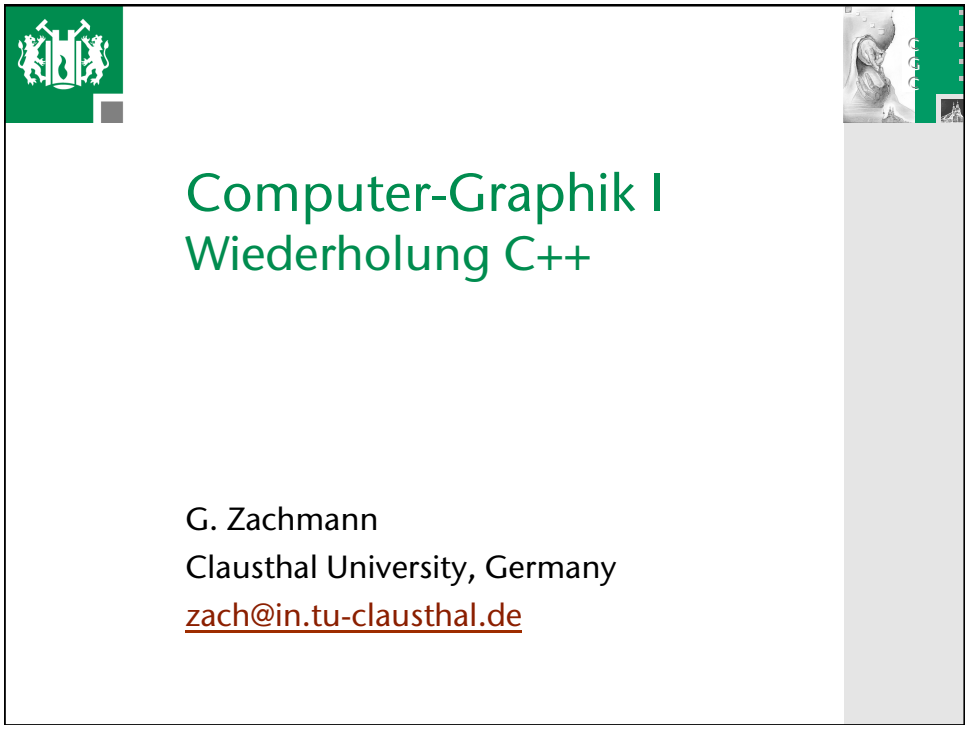
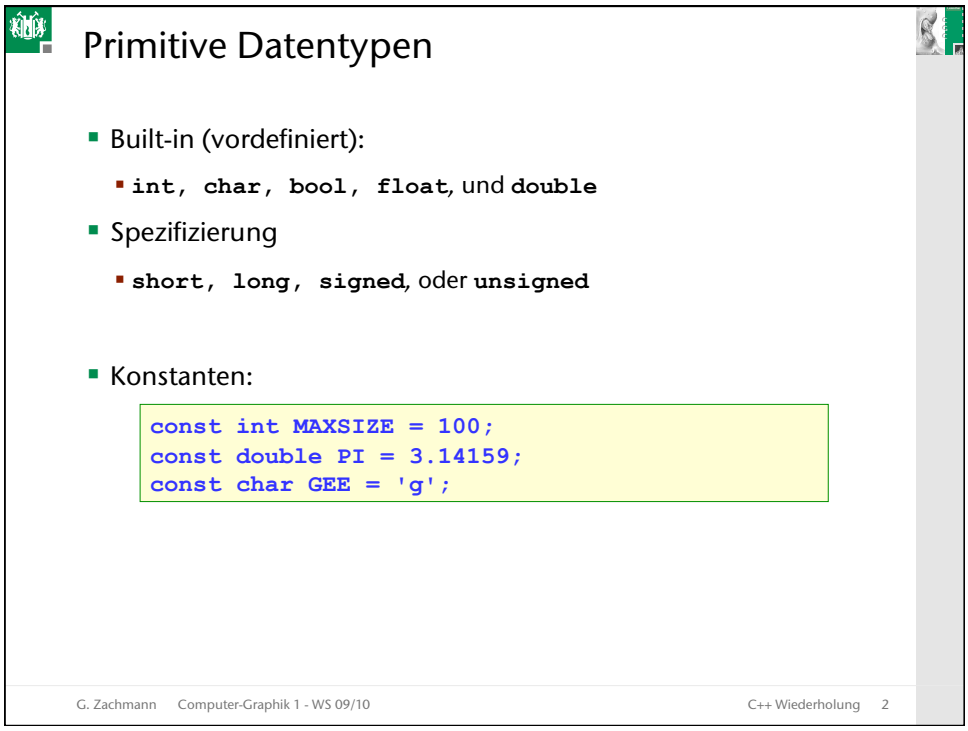


Computer-Graphik I
Wiederholung C++

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de



Primitive Datentypen

- Built-in (vordefiniert):
  - **`int`**, **`char`**, **`bool`**, **`float`**, und **`double`**
- Spezifizierung
  - **`short`**, **`long`**, **`signed`**, oder **`unsigned`**
- Konstanten:

```
const int MAXSIZE = 100;
const double PI = 3.14159;
const char GEE = 'g';
```

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;
```

- Es gibt zur Laufzeit keine Möglichkeit, die Größe eines Arrays festzustellen ...

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 }

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 ';'
```

- Der Name einer Struktur definiert einen neuen Datentyp

```
Student Student1, Student2;
```


- 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;
};
```

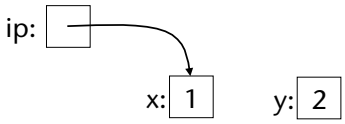
Zeiger

- 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 = NULL;

if ( ip != NULL )
   y = *ip;         // wird nicht ausgefuehrt
ip = &x;            // ip enthält Adresse von x
if ( ip != NULL )
   y = *ip;         // y = 1
```

```
int *pi_p; // Zeiger auf ein int, noch nicht initialisiert
double *pd_d;
int *pi_q, i_x; // i_x ist kein Zeiger

pi_p = NULL
pi_p = &i_x;     // pi_p zeigt auf i_x, "&" ist
                 // der "address of"-Operator


int *pi_p, * pi_q;
pi_q = ...
pi_p = pi_q;     // pi_p und pi_q zeigen auf selbe Adresse
i_x = 3;
pi_p = &i_x;
cout << *pi_p;
*pi_p = 5;       // mit dem *-Operator kann man auf den in
                 // pi_p gespeicherten Wert zugegriffen
```


- 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)
```


- Zugriff auf Variablen aus dynamischen Strukturen

```
struct ListNode
{
   int m_Data;
   ListNode *m_Next;       // Zeigt auf nächsten Element
                           // einer Liste
};
ListNode *pt_head = NULL; // Zeiger auf Liste;
                           // anfangs leer
int i_k;
while (cin >> i_k)
{  // neuer Knoten mit gelesenem Wert
   ListNode *p_tmp = new ListNode;
   p_tmp->m_iData = i_k;  // einfügen eines Knoten am
                           // Anfang der Liste
   p_tmp->m_ptNext = pt_head;
   p_head = pt_tmp;
}
```

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

Definieren neuer Datentypen

- Neue Datentypen können durch bereits existierende Datentypen definiert werden:

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

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;
}
```


Überladen von Funktionen

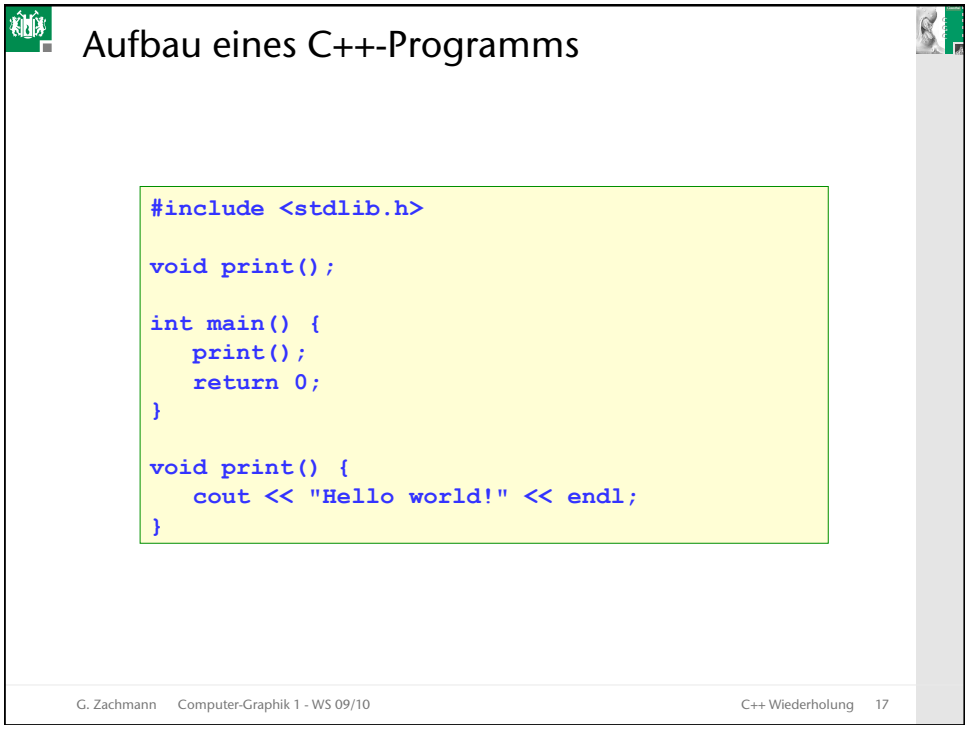
- In C++ ist es möglich Funktionen zu definieren, die sich nur in der Anzahl bzw. den Typen der Parameter unterscheiden (*overloading*) → Unterscheidung nur durch den Rückgabetyp reicht jedoch nicht!
- Beispiel:

```
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
```



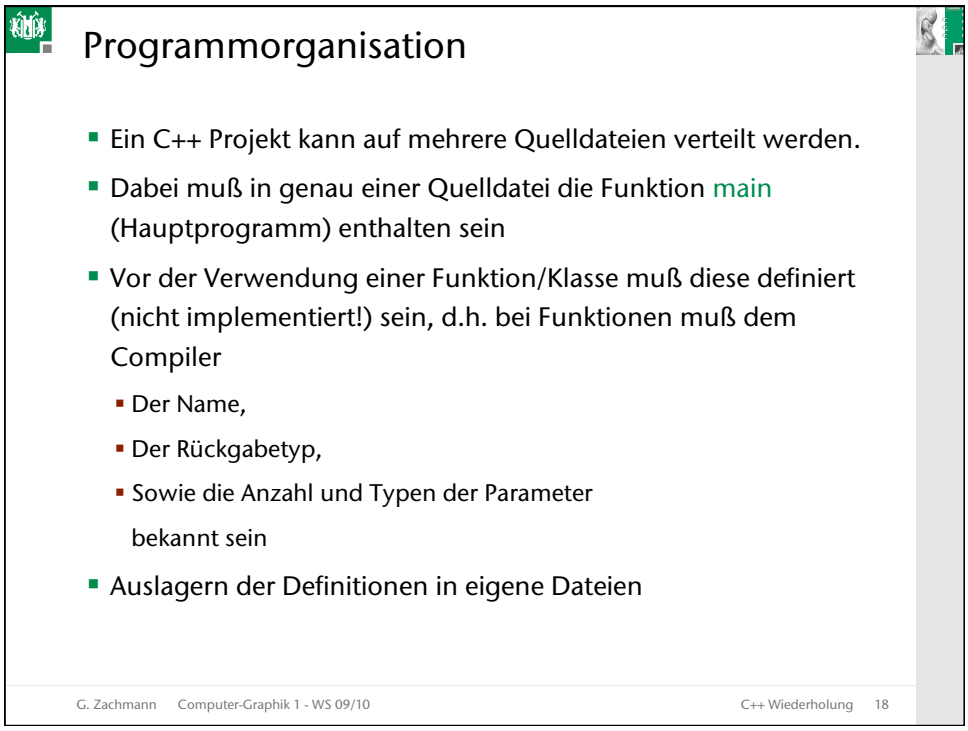
Aufbau eines C++-Programms

```
#include <stdlib.h>

void print();

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

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

Programmorganisation

- Ein C++ Projekt kann auf mehrere Quelldateien verteilt werden.
- Dabei muß in genau einer Quelldatei die Funktion main (Hauptprogramm) enthalten sein
- Vor der Verwendung einer Funktion/Klasse muß diese definiert (nicht implementiert!) sein, d.h. bei Funktionen muß dem Compiler
  - Der Name,
  - Der Rückgabetyp,
  - Sowie die Anzahl und Typen der Parameter

  bekannt sein
- Auslagern der Definitionen in eigene Dateien

- Für die Definition legt man eine ".h" – Header Datei an, für die Implementierung eine ".cpp" – Datei
- Möchte man eine Funktion oder Klasse verwenden, dann muß man die Header – Datei mit der Definition einbinden:

```
#inculde "name"
```

- Zum Einbinden von System-Bibliotheken benutzt man die Variante

```
#inculde <name>
```

- Search Path:
  - Der Compiler schaut in einigen Standart Pfaden nach z.B. **/usr/include**
  - Weitere Verzeichnisse kann man mit der Option **–I dir** hinzufügen
  - **# include** "..." sucht zuerst im aktuellen Verzeichnis, wo die aktuelle Datei steht, dann im restlichen Suchpfad
  - **# include** <...> sucht einfach nur im Suchpfad
  - ... (es gibt noch viele weitere Optionen)

Mehrfach-Includes

- Wird eine Header – Datei mehrfach eingebunden (weil sie z.B. von mehreren anderen Headern benötigt wird), dann erhält man eine Fehlermeldung ("Symbol already defined")
- Lösung: Verwendung von Preprozessor – Befehlen zur bedingten Compilierung

```
#ifndef EINDEUTIGER_NAME
#define EINDEUTIGER_NAME
// Inhalt der Header-Datei
[...]
#endif
```

- Gleicht *„wrapper ifndef"* oder *„ifndef guard"*
- Bemerkung: gcc/g++ optimiert das Preprocessing solcher „once – only headers" – es scannt sie beim erweiterten Mal gar nicht mehr (nur noch bis `#ifndef`)

Parameterübergabe

- 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 );
```


- 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“

Übergeben von Zeigern

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

Übergeben von Referenzen

```
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!!!)

- 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( int * x );
```

oder sogar

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


C++ Klassen

- Klassen werden normalerweise in den *Header Dateien* deklariert
  - Klassenname.h
- Die Implementierung von Funktionen kommt in die *source Dateien*
  - Klassenname.cpp (Windows)
  - Klassenname.C, Klassenname.cc, etc. (Unix/Linux) werden wir nicht verwenden

Klassendefinition

- Allgemein:

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

   protected:
      // Geschützte Komponenten

   private:
      // Private Komponenten
};
```

Nicht vergessen!

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];
};
```

Klassenimplementierung

- 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

Beispiel

```
#include "Vector2D.h"

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

[...]

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

Konstruktoren

- 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
- 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

Beispiel

- Definition:

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

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

- 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

- 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

- Beispiel:

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

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

- Destruktoren sollten immer dann verwendet werden, wenn in einer Klasse Member-Variablen dynamisch angelegt werden! Anderenfalls wird der reservierte Speicher nie freigegeben!

- 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;
}
```

  - 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
}
```


- Der Copy-Konstruktor:
  - Deklaration

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

  - 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];
   }
}
```

- **`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)

- Ohne **`operator =`**
  - Wenn das Objekt einen Zeiger beinhaltet, dann würde der Zeiger des neuen Objektes auf die selbe Adresse zeigen wie das Ausgangsobjekt
  - Wird der Zeiger von **`cl_l1`** gelöscht, dann verweist der Zeiger von **`cl_l2`** auf eine ungültige Adresse
- Unterschied zwischen Zuweisungsoperator und Copy-Konstruktor (am Bsp. **`cl_l1 = cl_l2`**)
  - **`cl_l1`** 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

- **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
}
```


Überladen von Operatoren

- In C++ kann man fast alle Operatoren überladen
- Es ist auch möglich, die Operatoren für eigene Datentypen zu überladen. Man kann jedoch weder neue Operatoren definieren, noch die Funktionsweise für elementare Typen abändern
- Beispiel:

```
Vector2D operator + ( const Vector2D& a,
                      const Vector2D& b )
{
   Vector2D result;

   result.mElement[X] = a.x() + b.x();
   result.mElement[Y] = a.y() + b.y();

   return result;
}
```

- Wichtig: “+=“ ist ein eigener Operator
  - Durch Überladen von “+“ wird er *nicht* automatisch definiert
- Was kann man tun, um umständliche Implementierungen von Operator + und Operator += (die ja fast das selbe machen) zu vermeiden?

Eine gute Lösung...

Zuweisungen haben in C++ auch ein Ergebnis!

rhs wird nicht verändert, ist also konstant.

Nur die Referenz übergeben und keine Kopie erzeugen.

```
Vector2D& Vector2D::operator += ( const Vector2D& rhs )
{
   mElement[X] += rhs.mElement[X];
   mElement[Y] += rhs.mElement[Y];

   return *this;
}
```

Innerhalb der Klasse hat man direkten Zugriff auf alle Attribute. Auch auf die als Parameter übergebener Objekte.

- Ich definiere Zuweisungsoperatoren gern so

```
void Class::operator += ( ... )
```

- Da Ausdrücke der Form

```
a = b += c;
```

oder

```
foo(a+=b)
```

nicht sinnvoll sind

rhs und lhs werden nicht verändert, sind also konstant.

Nur die Referenzen übergeben und keine Kopien erzeugen.

```
const Vector2D operator +(const Vector2D& lhs,
                          const Vector2D& rhs)
{
   return Vector2D(lhs) += rhs;
}
```

Erzeuge eine temporäre Instanz, initialisiere sie mit dem Inhalt von lhs und addiere darauf rhs.

- Beobachtung: dies ist ein globaler Operator
- Besser wegen Modularisierung
- Frage: Warum ist der Ergebnistyp ein `const Vector2D` ?

  Sonst wäre auch `a+b = c;` möglich (grober Unfug)!

Beispiel

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

using namespace std;

int main()
{
   Vector2D a(1, 2);
   Vector2D b(7, 5);

   Vector2D c = a + b;

   cout << "Ergebnis: (" << c.x() << ","
                         << c.y() << ")" << endl;

   return 0;
}
```


Anmerkungen

- Durch die Implementierung von **`operator +`** auf der Basis von **`operator +=`** muß nur noch eine Operator–Implementierung gepflegt werden
- Guideline: Operatoren sollten 'erwartungskonform' sein:

  „*principle of least surprise*“
- Beispiel:
  - Die Multiplikation eines Vektors mit einem Skalar ist kommutativ.
  - Daher sollte man hierfür, den Operator 2x überladen!

```
const Vector2D operator *(float s, const Vector2D& v);
const Vector2D operator *(const Vector2D& v, float s);
```

Vererbung

- 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

Beispiel

```
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;
};
```

```
Person person;
Mitarbeiter mitarbeiter;

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



```
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.
}
```

```
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;
}
```


- Prinzip der kaskadierenden Konstruktoren
- Bei den Destruktoren genau umgekehrt
- Copy-Konstruktor wird nicht automatisch vererbt

Templates

- 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

Template Funktionen

- 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 ;
}
```

Verwendung

```
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

- 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 ;
} ;
```

Definition

```
// 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
}
```


Verwendung

```
#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 ;
   }
```

```
   // 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-Files

- 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 !!!

Test.h

```
// Declaration of template class Ctest
// This class does . . .

template <class T> class Ctest
{
    . . .
}

#include "Test.inl"
```

Test.inl

```
// Implementation of template class Ctest

template <class T>
Ctest<T>::Ctest( )
{
. . .
}
```


Exkurs: STL

- 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
```

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

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

  - Abfragen der Größe:

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

- 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());
```

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

Namensräume *(Namespaces)*

- Wie Pakete in Java

```
namespace SpaceOne {
   Class  CExampleClass1 { ... };
   Class  CExampleClass2 { ... };
   bool   func( int ) { ...}
}
```

- Verwendung von Namespaces (hier am Beispiel **`std`**)

```
#include <set>      // Einbinden der Header Dateien

std::set set_temp1; // std:: scope resolution
             // muß hier angewandt werden

using std;          // verwende Namensraum 'std'

set set_temp2;      // 'set' wird jetzt auch autom.
                    // im Namespace 'std' gesucht
```