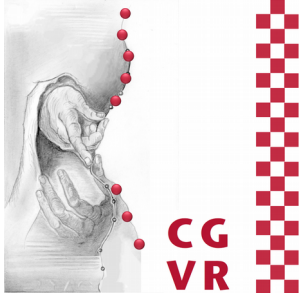


Computer-Graphik I
C++ Crash Course

Philipp Dittmann und Jörn Teuber
University of Bremen, Germany
cgvr.informatik.uni-bremen.de

Ablauf

- C
  - Grundlegendes aus C
- C++
  - Klassen, Vererbung und alles was C++ zu C++ macht
- Mittagspause
- C++11/14
  - Wichtige Dinge, die seit „kurzem“ im C++ Standard sind
- Tools
  - Ein paar Werkzeuge die euer Leben leichter machen können
- Tipps für weitergehende Literatur

C

Primitive Datentypen

- *Built-in* (vordefiniert):
  - **int** (4Byte Integer)
  - **char** (1Byte Integer, oder 1 Zeichen)
  - **bool** (boolean, True/False (wird auch in 4Byte gespeichert))
  - **float** (IEEE 754 floating point number mit 32bit)
  - **double** (double precision float, IEEE 754 64bit)
- *Qualifications*
  - **short**, **long**, **signed**, oder **unsigned**

Primitive Datentypen

- Beispiele:

```
int i = 100;
float f = 3.14159f;
char c = 'g';

unsigned int i = 42u;
signed char = -128;
short int = -32768; // 16 bit
long int = 2147483647; // 32 bit
long long int = 9223372036854775807; // 64 bit
```

- 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**. Auch Speicher dafür wird schon reserviert. Auf die einzelnen Elemente kann mit **a[0]** bis **a[9]** zugegriffen werden.

- Auf diese Art lassen sich nur Arrays definieren, deren Größe zur Compile-Zeit 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 (*Enumerations, Enums*)

- 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 }

Operatoren

- Folgende Operatoren sind auf den Standarddatentypen definiert:

```
// Assignments: =
int x = 5;
int z = x = y = 2;

// Arithmetic: +,-,*,/,%
x = y+z;
y = x%z;
…

// Compound Assignments: +=,-=,*=,/=,%=
y += z;
…

// Increment and Decrement: ++,--
y++;
z = ++x;
z = x++;
```

Operatoren

- Folgende Operatoren sind auf den Standarddatentypen definiert:

```
// Comparisons: ==,!=,>,<,>=,<=
bool b = x==y;

// Logical operators: !,&&,||
bool nb = !b;

// Bitwise operators: &,|,^,~,<<,>>
char c = 63;
char c1 = c & 1; // bitwise and
char c2 = c<<; // shift left,=>c*2=126
```

Controll Structures & Loops

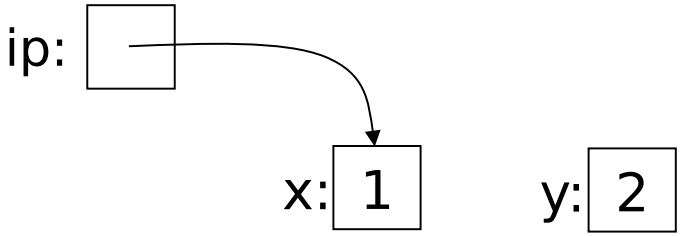
- Die Kontrollstrukturen und Schleifen in C++ sind (bis auf foreach) identisch mit denen von Java
- Kontrollstrukturen: if, switch
- Loops: for, while, do while

Zeiger (*Pointer*)

- Für einen beliebigen Datentyp **`type`** bezeichnet **`type*`** den Typ "Pointer 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:



- Ein Zeiger kann auch nicht belegt sein. Für diesen Fall gibt es den nullptr (Null-Pointer).

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

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

if ( ip )
   y = *ip;          // y = 1
```

```
int    *pi_p;     // Zeiger auf ein int, noch nicht init'ed!
double *pd_d;
int    *pi_q, i_x; // Achtung: i_x ist kein Zeiger

pi_p = nullptr;
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
```

- Folgende Anweisungen sind schwere Memory-Bugs:

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

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

- 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 ist nicht mehr
                // reserviert!! (wild/dangling pointer)

int *pi_p = new int;
pi_p = nullptr;// ändere keinen Zeiger, ohne den
                // Speicherplatz freizugeben, auf den er
                // vorher gezeigt hat (memory overflow)
```

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
  - In Java: jedes Vorkommen eines Variablennamens (dessen Typ nicht einer der built-in Skalartypen ist) ist eigtl. eine Referenz!
- Da eine Referenz immer an ein Objekt gebunden ist, muß man sie zwingend initialisieren
- Beispiel:

```
int    x = 17;
int & xr = x;  // now xr is a reference to x

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

- Anwendung: für *Call-by-Reference*

Funktionen

- Im Gegensatz zu Java können in C++ Funktionen außerhalb von Klassen existieren
- Funktionen erlauben es, Code auszulagern und in kleine, zusammengehörenden Teile zu zerschneiden
- Deklaration einer Funktion:

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

- Definition einer Funktion am Beispiel:

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

Funktionen

- Im allgemeinen werden globale Funktionen in C++ aber vermieden
- Eine Funktion hat trotzdem jedes C++-Programm:

```
// Definition einer Funktion
int main(int argc, char** argv)
{
      return 0;
}
```

- Die main(…)-Funktion ist der Startpunkt jedes C(++)-Programms
- Die Argumente sind Parameter aus dem Aufruf (Kommandozeilen-Parameter)

Parameterübergabe

- In Java erfolgt die Parameterübergabe durch Übergeben von Werten
- In C++ kann die Parameterübergabe erfolgen durch Übergabe von:
  - Einem Wert (*call-by-value*)
  - Einer Referenz auf einen Wert (*call-by-reference*)
    - Der Caller sieht dem Aufruf nicht an, dass der Wert sich ändern könnte!
  - Einer konstanten Referenz (*call-by-reference*)
    - Der Wert kann sich nicht ändern, aber man hat trotzdem die Vorteile von call-by-reference
  - Einem Pointer (auch *call-by-reference*)
    - Der Caller sieht dem Aufruf an, dass der Wert sich ändern könnte

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

- Erfolgt die Übergabe als Wert (*call by value*), 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);
   // jetzt gilt: 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

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!!!)

Parameterübergabe

- Parameter können bei der Übergabe als const deklariert werden

```
// Definition einer Funktion mit const
int* setup(const int param)
{
      param = 0; // Compiler-Fehler da const
      return new int[param]; // OK
}
```

- Bei Pointern als Parametern können Pointer auch als const deklariert werden
  - es gilt die Regel „const bezieht sich auf den Qualifikator/Typ auf seiner linken Seite“
  - falls links von const nichts steht bezieht es sich auf das rechte Element

Parameterübergabe

- Beispiel:

```
// Definition einer Funktion mit const
int get(int const * const param)
{
      param = new int[4]; // Nicht erlaubt, da Pointer const
      param[1] = 0; // Nicht erlaubt, da Typ const
      return param[0];
}
```

- 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 Parameter verändert werden soll, dann Pointer verwenden (Call-by-reference)!
    Z.B.:

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

    oder sogar:

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

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 dieser Anweisung einbinden:

```
#include "name"
```

- Zum Einbinden von System-Bibliotheken benutzt man die Variante

```
#include <name>
```

- Der *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)

Macros / Defines

- Bevor eine C++-Datei Kompiliert wird, wird sie vom sogenannten Pre-Processor durchlaufen
- Diesem Pre-Processor kann man rudimentäre Befehle geben
- Diese Befehle fangen im Code mit einer „#“ an
- Beispiel

```
#if OS = WIN32
#   include <windows.h>
#endif

#define PI 3.141592

float degree = 18.f;
float rad = degree * PI / 180.f;
```

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 (*conditional compilation*)

```
#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`)

Aufbau eines C++-Programms

```
#include <stdlib.h>

void print();          // forward declaration

int main()             // one "main" per program
{                      // = entry point
   print();
   return 0;
}

void print()         // this is the actual definition
{
   cout << "Hello world!" << endl;
}
```

C++

Structs

- Struct's in C++ :
  - Gruppieren von semantisch zusammenhängenden Daten
  - Ähnlich einer Klasse:

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

- Der Name eines Struct's definiert einen neuen Datentyp:

```
Student Student1, Student2;
```

- Zugriff auf Variablen eines Struct's:

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

- Verschachtelte Struct's:

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

Dynamic Data Structures

- Zugriff auf Variablen aus dynamischen Strukturen:

```
struct ListNode
{
   int m_data;
   ListNode *m_next;          // zeigt auf nächstes Element
                              // einer Liste
};
ListNode *head = nullptr;     // Zeiger auf Kopf der Liste;
                              // anfangs leer
int i_k;
while ( cin >> i_k )
{  // create new node storing value we've just read
   ListNode *new_node = new ListNode;
   new_node->m_data = i_k;
   // insert new node at head of list
   new_node->m_next = head;
   head = new_node;
}
```

C++ Klassen

- Klassen werden normalerweise in den *Header Files* deklariert
  - Klassenname.h
- Die Implementierung von Funktionen kommt in die *Source Files*
  - Klassenname.cpp
  - Manchmal sieht man auch: Klassenname.C, Klassenname.cc, etc.

Klassendefinition

- Allgemein:

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

   protected:
      // Geschützte Komponenten

   private:
      // Private Komponenten
};
```

Nicht vergessen!

Beispiel

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

      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 Source-Files verteilt werden

Beispiel

```
#include "Vector2D.h"

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

[...]

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

Klassen vs. Structs

- Eigentlich sind Klassen und Structs in C++ ein und das selbe.
- Auch structs können Konstruktoren/Destruktoren und Methoden haben
- Einziger Unterschied: Default Access
  - Methoden und Variablen in Klassen sind per default private
  - Methoden und Variablen in Structs sind per default public

  => Structs werden meistens für simple Speicherstrukturen benutzt, Klassen für „Black-Boxes“ mit Methoden

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.
- Konstruktoren können in anderen Konstruktoren aufgerufen werden (sogenannter delegating constructor)

Beispiel

- Definition:

```
class Vector2D {
public:
    Vector2D();
    Vector2D(float xy);
    Vector2D(float x, float y);
private:
    float m_x, m_y;
};
```

- Implementierung:

```
Vector2D::Vector2D()
    : Vector2D(0.f) // delegating Constructor
{}

Vector2D::Vector2D(float xy)
    : Vector2D(xy,xy)
{}

Vector2D::Vector2D(float x, float y)
    : m_x(x), m_y(y)
{}
```

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

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

  2. 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;
}
```

  3. 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
}
```

- Deklaration eines Copy-Konstruktors:

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

- Definition eines Copy-Konstruktors:

```
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];
   }
}
```

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!

Const ... again!

- In Klassen gibt es eine weitere Anwendung für const: Die konstante Methode
- In einer konstanten Methode können die Klassenvariablen nicht verändert werden
- Dies ist nützlich um dem Benutzer einer Klasse die Sicherheit zu geben, dass in dieser Methode nichts verändert wird
- Konstante Methoden werden durch ein nachstehendes const gekennzeichnet:

```
class Vector2D {
    float getX() const;
};
float Vector2D::getX() const { … }
```

Static

- Ein weiterer Qualifier für Klassen ist das static
- Wie in Java wird damit eine Methode oder Variable gekennzeichnet, die zur Klasse (oder Funktion) gehört (vs. zu einer Instanz gehört)
- Während auf Methoden/Variablen eines Objekts mit . oder -> zugegriffen wird, wird auf eine static Methode/Variable mit :: zugegriffen

```
class Widget {
    static int id;
    static int getNumberOfWidgets();
    …
};
int Widget::id = 0;
int Widget::getNumberOfWidgets() { … }
```

Überladen von Funktionen (*Overloading*)

- 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);   // calls float version of sqrt
double y = sqrt(d);   // calls double version
```

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;
}
```

Der Zuweisungsoperator

- **`operator =`**
  - In C++ kann damit ein Objekt einem anderem zugewiesen werden:

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

  - Ohne einen eigenen Zuweisungsoperator (**`operator =`**) müssen die Objekte Byte-weise kopiert werden (auch *flat copy* / *shallow copy* genannt)
    - Ist OK bei "flachen" Objekten / Datenstrukturen;
    - Geht schief (je nach Anwendung), falls eine Datenstruktur aus vielen verzeigerten Objekten besteht

- 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
}
```

- 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.

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);
```

Definieren neuer Datentypen

- Neue Datentypen können durch bereits existierende Datentypen definiert werden:

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

Vererbung (*Inheritance*)

- Eine Klasse kann Eigenschaften einer anderen Klasse durch die Ableitung/Vererbung 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 !!!

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

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

#include "Test.inl"
```

Test.h

```
// Implementation of template class Ctest

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

Test.inl

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
```

Standard Template Library

- 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);  // füge hinten an
```

  - 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:` sortierter Baum
    - `list:` doppelt-verkettete Liste
    - `stack:` Stack
    - `deque:` double-ended Queue
  - Algorithmen wie `find(), sort(), min(), max()`
  - Input-Output Streams:
    - `cin/cout:` console input und output
    - `ifstream, ofstream, fstream:` File input/output
  - Zeichenketten `string`
  - Zeitmessungen und -berechnungen `chrono`
  - Zufallszahlen `random`
  - Multithreading `thread, mutex, atomic`

- Nützliche Resource wenn man mit der STL arbeitet:

  cplusplus.com

- Ist eine Referenz über alle Container, Algorithmen, Streams etc. die die STL bereit stellt
- Hat auch (weiterführende) Tutorials für C++

C++11/14

C++11/14

- C++ ist ein ISO-Standard mit einer Arbeitsgruppe, die diesen auch erweitern kann
- In den letzten Jahren hat diese Arbeitsgruppe mehrere Updates veröffentlicht
- Gerade C++11 hat viele neue Konzepte eingeführt, z.B.
  - Smart-Ptr
  - Foreach-Schleifen
  - Lambda-Closures

Smart-Pointer

- Smart-Pointer kann man sich als Pointer mit automatischem Destruktor vorstellen
- intern macht ein Smart-Pointer ein Reference Counting
  - d.h. immer wenn der Smart-Pointer referenziert wird, also einer Variable zugewiesen wird, wird ein counter erhöht und beim Destruktor erniedrigt.
- Beispiel:

```
std::shared_ptr<int> foo;
{
    std::shared_ptr<int> bar(new int(100));
    foo = bar;
    *foo *= 2;
}
std::cout << *foo << std::endl;
```

Smart-Pointer

- Neben dem `shared_ptr` gibt es noch `unique_ptr` und `weak_ptr`
- Der `unique_ptr` besitzt seinen Pointer als „Alleinherrscher“ und kann ihn nicht teilen
- `weak_ptr` hingegen ist eine Version des `shared_ptr` der den Pointer nicht besitzt, d.h. den Use Count nicht erhöht

Auto und Type-deduction

- Mit dem Keyword `auto` kann eine Variable deklariert werden ohne ihren endgültigen Typ zu kennen
- Dies kann nützlich sein bei sehr vielen verschiedenen Anwendungen
- Z.B. for-Schleifen:

```
std::map<float, int> newmap;
…
for(auto it=newmap.begin(); it!=newmap.end(); ++it)
{
    it->second += 1;
}

auto size = newmap.size();

// davor:
for(std::map<float, int>::iterator it=newmap. …)
```

Auto und Type-Deduction

- Weiterer Einsatz: unintuitive Rückgabewerte, z.B. size()
- in der STL gibt size() die Größe von Containern zurück
- man würde dort ein `int` oder `unsigned int` erwarten
- stattdessen wird aber `size_t` (o.ä.) zurückgegeben, das (meistens) 64bit groß ist
- Einige C++-Experten empfehlen daher und anderer Probleme die durchgehende Verwendung von `auto`

- Der endgültige Typ einer Variable die mit `auto` deklariert wurde wird mit Type-Deduction ermittelt, sehr ähnlich der der Templates

Foreach

- Eine weitere Anwendung von `auto` findet man in den neu engeführten `foreach`-Schleifen (ja, die gibt es jetzt erst)
- Syntax: for(auto elem : container)
- Beispiel:

```
std::vector<Vector3D> normals;
…
for(auto n : normals)
{
    n.normalize();
}
```

Lambda Functions

- Die sogenannten Lambda Functions sind inline-Funktionen oder auch Anonymous Functions genannt
- Ihre Syntax startet mit einer Closure, `[]` die mit Variablen, die in der Lambda Funktion zur Verfügung stehen sollen gefüllt wird
- Beispiel:

```
std::vector<int> some_list{ 1, 2, 3, 4, 5 };
int total = 0;
std::for_each(begin(some_list), end(some_list),
       [&total](int x) {
          total += x;
       });
```

Lambda Functions

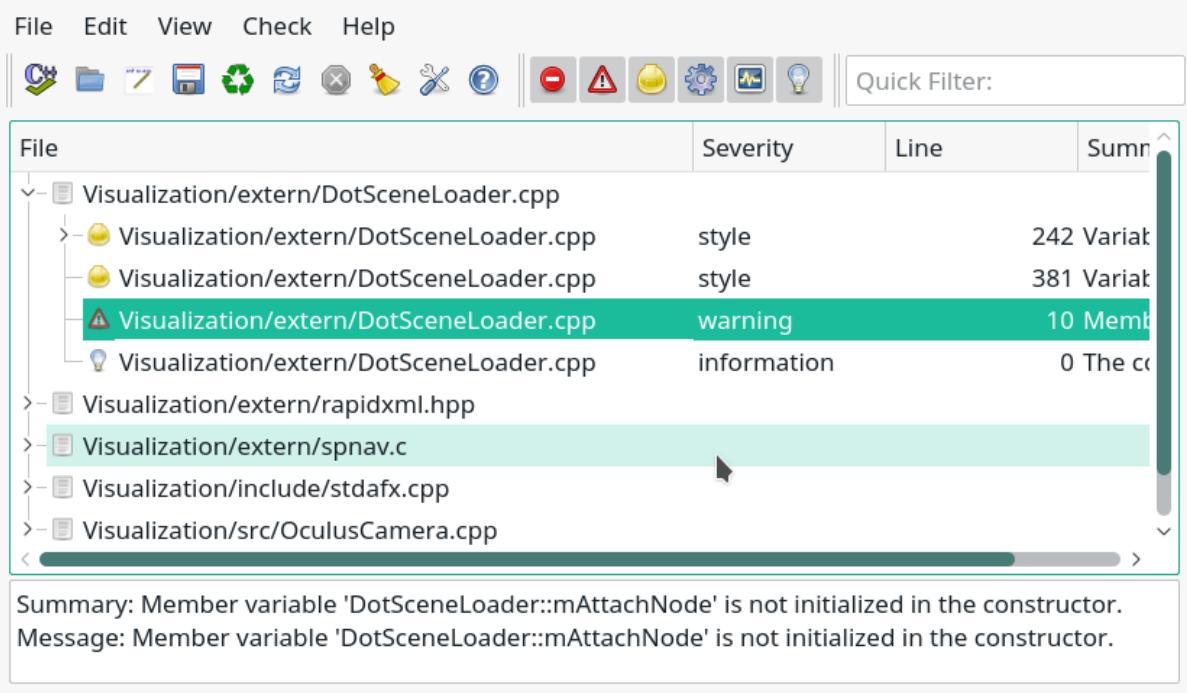
- Lambdas können auch einer Variable zugewiesen werden um sie später und mehrfach zu benutzen
- Beispiel:

```
#include <iostream>

using namespace std;

int main()
{
    auto func = [] () { cout << "Hello world"; };
    func(); // now call the function
}
```

Tools

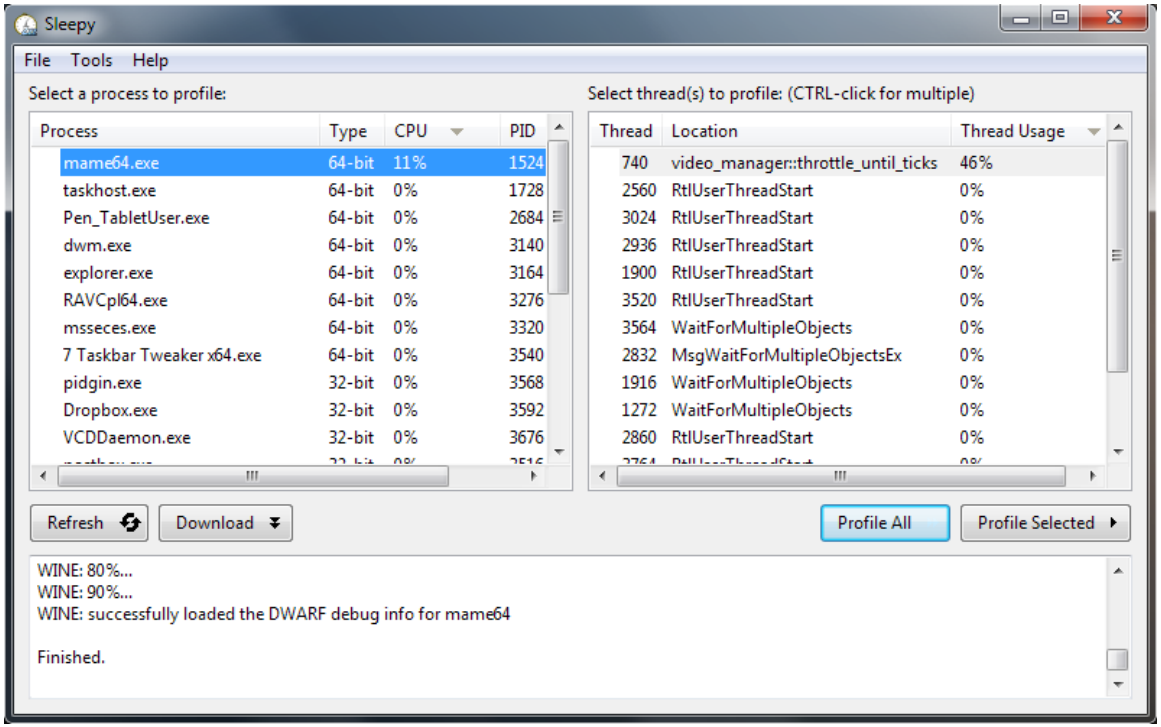
CPPCheck

- Das Programm cppcheck ist ein sogenannter Static Code Analyzer
- Er analysiert den Code ohne ihn auszuführen
- Gibt danach Hinweise auf Memory Leaks, eventuelle Fehler und schlechten Code



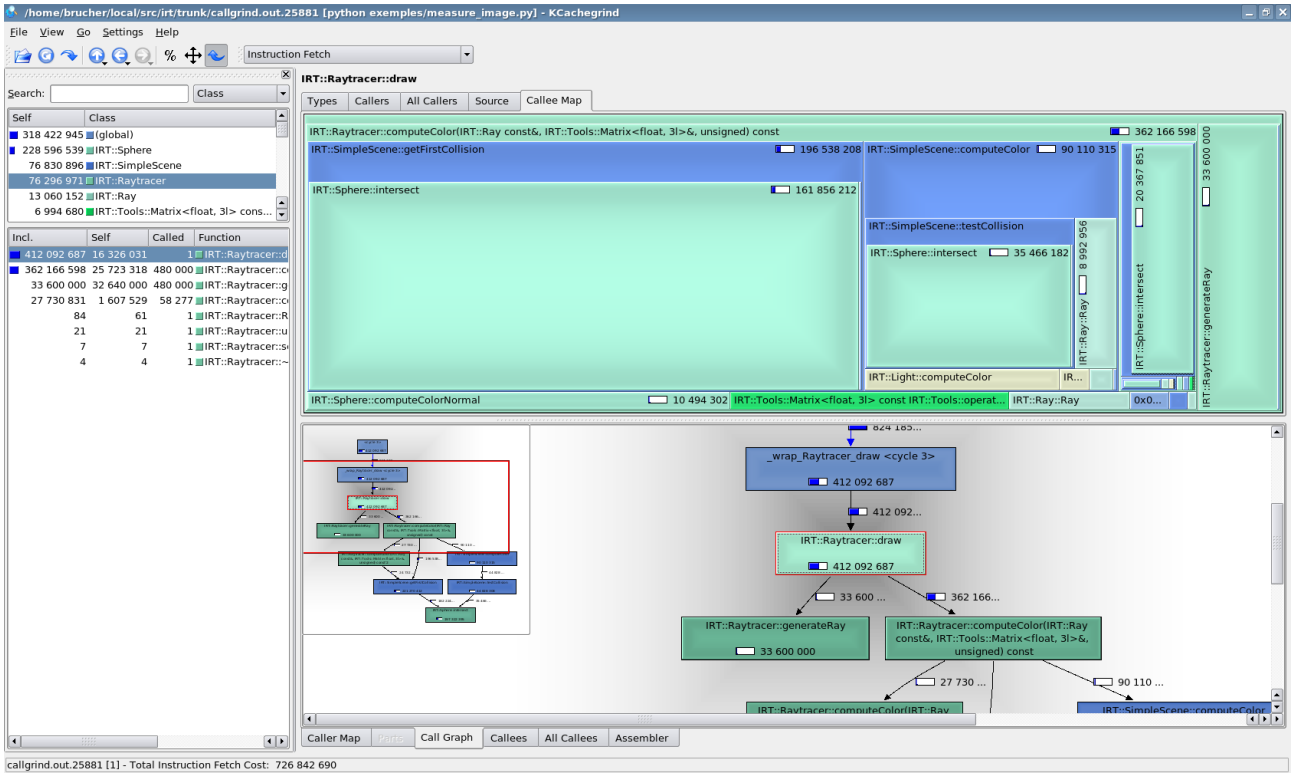
Very Sleepy

- Very Sleepy ist ein Programm für Windows, dass ein beliebiges Program *profilen* kann, d.h. Es zeichnet von diesem Programm die Laufzeiten jedes Methodenaufrufs auf
- Der Output kann sehr hilfreich sein zu langsame Methoden ausfindig zu machen und ein Programm zu optimieren



Valgrind

- Das Pendant zu Very Sleepy unter Linux ist Valgrind (--tool=callgrind)
- Valgrind hat noch mehr Funktionalität, aber keinen direkten Viewer, for that use KCachegrind



Test-Frameworks

- Vor allem für größere Projekte ist es wichtig einzelne Komponenten ständig auf ihre korrekte Funktionalität zu testen
- Hier kommen Test-Cases ins Spiel
- Mit Test-Frameworks wie Google Test (gtest) oder Qt Test können Testfälle geschrieben werden und unabhängig vom Programm ausgeführt werden
- Links:
  - Qt Test: http://doc.qt.io/qt-5/qttest-index.html
  - Google Test: https://github.com/google/googletest