Computergrafik Übung

Organisatorisches

- Übungsleiter
  - Emanuel Schrade (schrade@kit.edu)
  - Florian Simon (florian.simon@kit.edu)
  - Johannes Meng (meng@kit.edu)

- Übungsblätter alle zwei Wochen
- Übung wöchentlich
- Bonusübungsblatt am Ende des Semesters
- Scheinkriterium: 60% der Punkte
- Übungsmaterial in Ilias, Passwort: cg1617
- Abzugeben sind nur praktische Übungsaufgaben
- Programmierung in C++/OpenGL

Ilias / Forum

- Ankündigungen/Benachrichtigungen über Ilias.
- Forum zur Kommunikation (hauptsächlich) untereinander.
- Sie können Fragen stellen oder Probleme äußern.
- Keine Lösungen im Forum!

VM

- Wir verwenden eine VM zum Kompilieren der Abgaben
- Abgaben, die in der VM nicht kompilieren oder abstürzen -> 0 Punkte

- Kleines VM-Tutorial bei den Übungsunterlagen
- Bei Bedarf eigene Pakete mit Pacman nachinstallieren
  - https://wiki.archlinux.org/
  - https://wiki.archlinux.org/index.php/Pacman

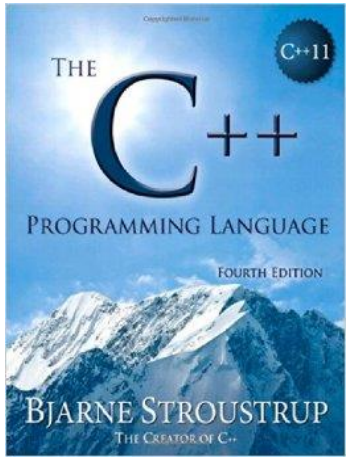
- Vorschlag: Auf dem Host-System coden, auf dem Gast-System kompilieren und ausführen.

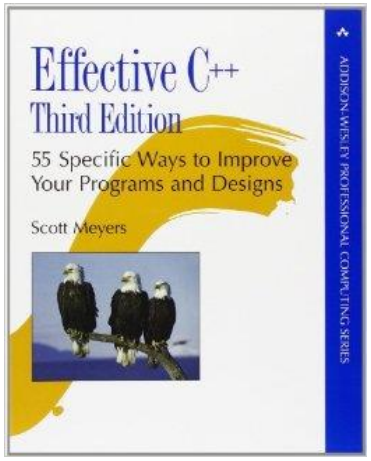
VM - Demo

C++

Mehr Info

- Wir gehen davon aus, dass Sie Java beherrschen.
- Bei grundlegenden Fragen sollten Sie ein entsprechendes Buch aus der Bibliothek ausleihen oder eines der zahlreichen Tutorials im Internet zu Rate ziehen.
- “In a Nutshell” auf den Folien





C++: Wichtige Konzepte

- Präprozessor
- auto
- Stack vs. Heap
- Zeiger und Referenzen
- Arrays: new/delete vs. new[] / delete[]
- Standard Template Library
  - std::vector, std::array, std::string, std::sort
- const, const und const
- class und struct: static, virtual, final, public, private, …
- Templates

Hallo Welt!

```
// Headerdatei für Standard-IO-Bibliothek einbinden
#include <iostream>

// Hauptprogramm
int main()
{
    // Textausgabe mit streams.
    std::cout << "Hallo Welt!" << std::endl;
    return 0;
}
```

- Präprozessor-Direktiven beginnen mit # (z. B. `#include`)
- Reine Textersetzung, die vor dem eigentlichen Kompilier-Vorgang durchgeführt wird
- std::cout ist ein C++ Stream, mit dem Text auf der Konsole ausgegeben werden kann

Kompilieren – Linux

```
~ckurs$ g++ -Wall hallo.cpp -o hallo
~ckurs$ ./hallo
Hallo Welt!
~ckurs$
```

- g++ ist der C++-Kommandozeilen-Compiler von GCC
  - in vielen Linux-Distributionen schon vorinstalliert
- Alternativ: clang++

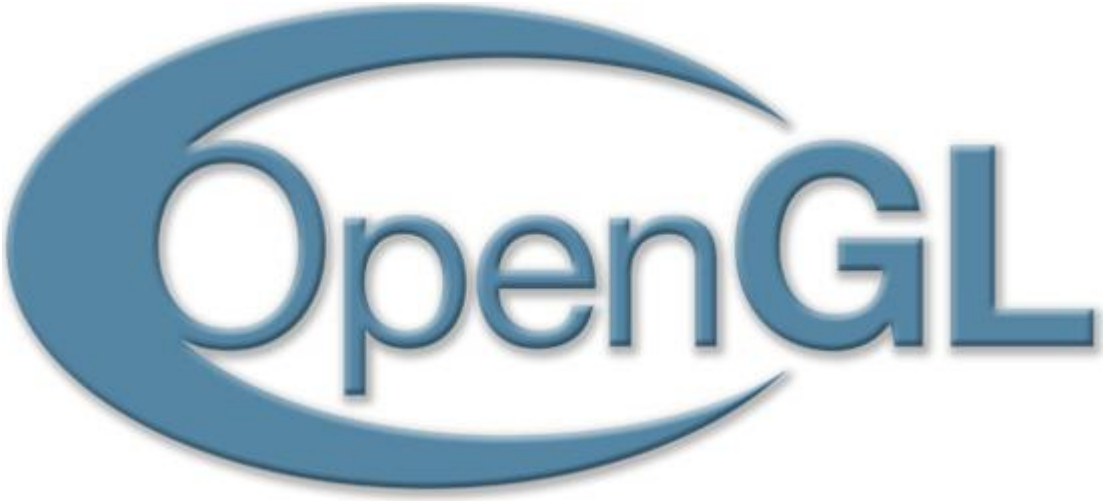
CMake

- Für Projekte mit mehreren Dateien nützlich
- Kann Makefiles und Projektdateien für Entwicklungsumgebungen generieren
- Integrierte Entwicklungsumgebungen
  - Vim + Make ;)
  - Visual Studio
  - Eclipse

CMake - Demo

Debugging

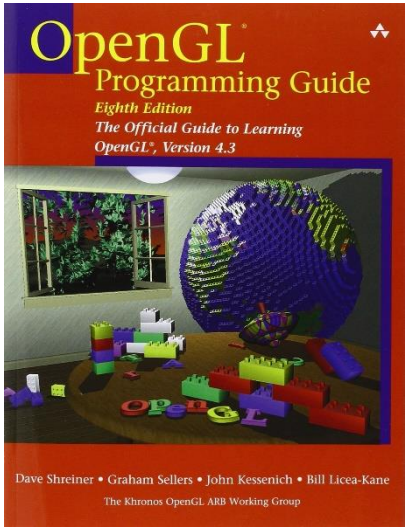
- GDB
  - Debugging auf der Konsole
  - Am besten ein bisschen mit kleinen Beispielen experimentieren und ein Tutorial durcharbeiten z.B.
    - das in Ilias
    - http://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf
- Valgrind
  - Memory Leaks, Profiling
- Beide Tools müssen mit Pacman auf der VM nachinstalliert werden!



Einführung in OpenGL

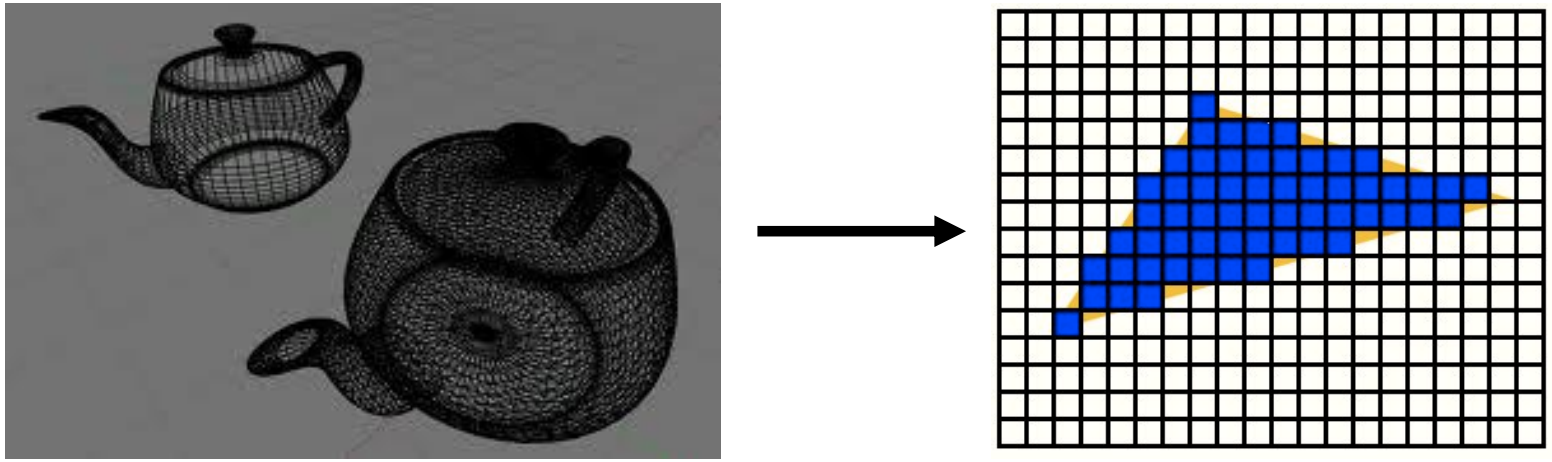
Was ist OpenGL?

- API zur Entwicklung von 2D und 3D Grafikanwendungen
  - Üblicherweise in Verbindung mit spezieller Grafikhardware
  - Platformunabhängig
    - Verschiedene Betriebssyteme
    - Verschiedene Hardware (Nvidia, AMD, Intel, …)
  - Offener Standard
  - Hersteller- und betriebssystemspezifische Implementierung
    - Darum die VM
    - Kein Fenstersystem
    - Wir benutzen glfw
  - Beinhaltet keine Mathe-Bibliothek
    - Wir benutzen glm (dazu später mehr)
- http://www.glprogramming.com/red/



Was ist OpenGL?

- CPU gibt Anweisungen an die Grafikhardware
- Diese transformiert 2D/3D Geometrie in ein 2D-Raster-Bild
- Als Zustandsautomat realisiert



Was ist OpenGL?

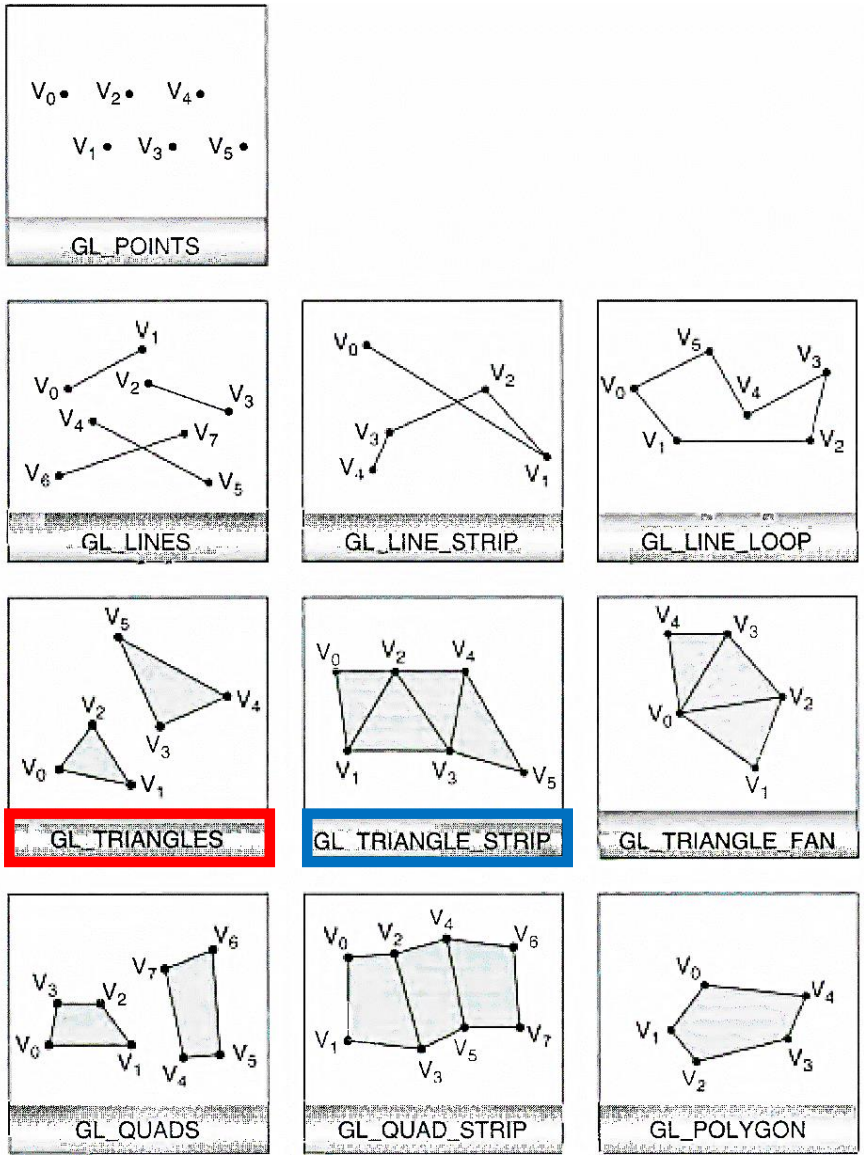
- Es gibt “altes” und “modernes” OpenGL
  - Modernes OpenGL behandeln wir nach Weihnachten
  - Wir machen jetzt altes OpenGL
    - da es immer noch weit verbreitet ist
    - weil man ohne großen Aufwand einfach Dinge zeichnen kann
    - damit wir schon im 1. ÜB etwas interessantes machen ;)

GLM

- Bibliothek für Vektor/Matrix Operationen
- Angelehnt an GLSL (später)
- http://glm.g-truc.net

```
#include <glm/glm.hpp>

glm::vec3 a(1.0, 2.0, 3.0);
glm::vec3 b(4.0, 5.0, 6.0);

glm::vec3 sum = a + b;   // = (5, 7, 9)
glm::vec3 mul = a * b;   // = (4, 10, 18)

float d = glm::dot(a, b);   // = 32

float x = a[0];     // = 1
float y = a.y;      // = 2
```

Primitive

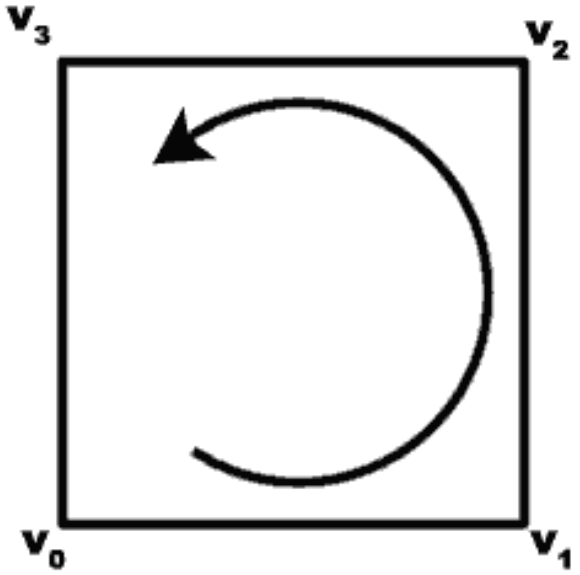



```
// Zeichne ein Quadrat mit 2
// Dreiecken
glBegin(GL_TRIANGLES);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(1.0, 0.0, 0.0);
    glVertex3f(0.0, 1.0, 0.0);

    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(1.0, 0.0, 0.0);
    glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

```
// Zeichne ein Quadrat als
// Triangle Strip
glBegin(GL_TRIANGLE_STRIP);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(1.0, 0.0, 0.0);
    glVertex3f(0.0, 1.0, 0.0);

    glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

Winding Order

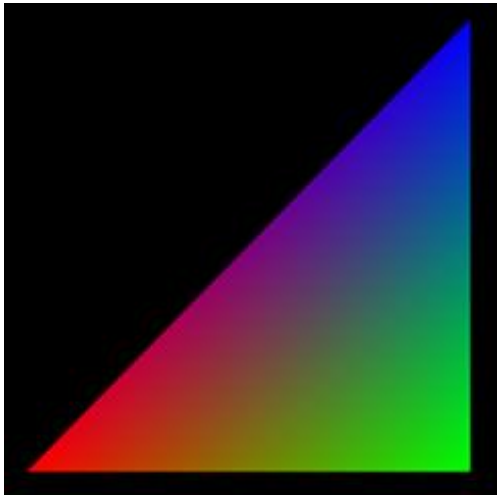
- Die Orientierung der Vertices spielt üblicherweise eine Rolle
- OpenGL sieht Dreiecke “von vorne” wenn die Vertices gegen den Uhrzeigersinn angegeben sind.



Farbe

- Pro Vertex kann eine Farbe als RGB-Tripel angegeben werden
- Die Farbe im Innern interpoliert OpenGL automatisch

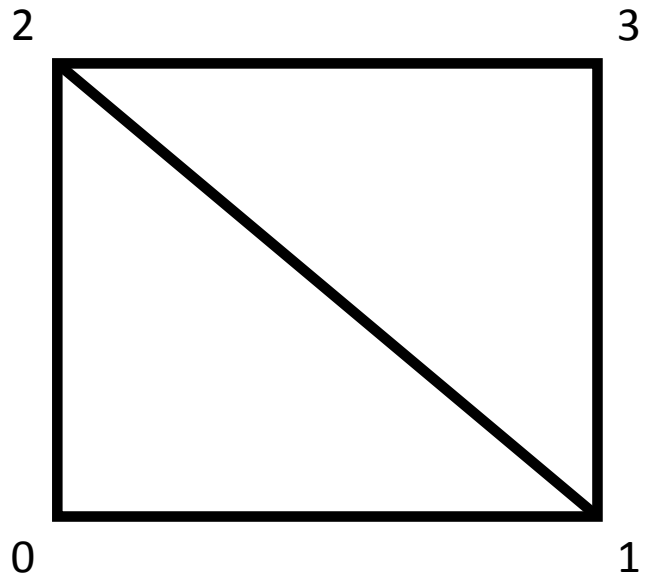
```
glm::vec3 blue(0.0, 0.0, 1.0);
glm::vec3 pos_blue(1.0, 1.0, 0.0);

glBegin(GL_TRIANGLES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(0.0, 0.0, 0.0);

    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(1.0, 0.0, 0.0);

    glColor3fv(&blue[0]);
    glVertex3fv(&pos_blue[0]);
glEnd();
```



Shared Vertex Darstellung

- Trennung von Geometrie und Topologie
  - Ein Array mit Vertex Positionen/Farbe/…
  - Ein Array mit Indizes, drei aufeinanderfolgende Indizes definieren ein Dreieck

```
std::vector<glm::vec3> positions {
   glm::vec3(0.0, 0.0, 0.0),
   glm::vec3(1.0, 0.0, 0.0),
   glm::vec3(0.0, 1.0, 0.0),
   glm::vec3(1.0, 1.0, 0.0)
};
std::vector<glm::vec3> colors {
   glm::vec3(1.0, 0.0, 0.0),
   glm::vec3(0.0, 1.0, 0.0),
   glm::vec3(0.0, 0.0, 1.0),
   glm::vec3(1.0, 0.0, 1.0)
};

// Dreiecke
std::vector<glm::uvec3> idx {
   glm::vec3(0, 1, 2),
   glm::vec3(2, 1, 3)
};
```



Shared Vertex Darstellung

- Trennung von Geometrie und Topologie
  - Ein Array mit Vertex Positionen/Farbe/…
  - Ein Array mit Indizes, drei aufeinanderfolgende Indizes definieren ein Dreieck

```
std::vector<glm::vec3> positions {
   glm::vec3(0.0, 0.0, 0.0),
   glm::vec3(1.0, 0.0, 0.0),
   glm::vec3(0.0, 1.0, 0.0),
   glm::vec3(1.0, 1.0, 0.0)
};
std::vector<glm::vec3> colors {
   glm::vec3(1.0, 0.0, 0.0),
   glm::vec3(0.0, 1.0, 0.0),
   glm::vec3(0.0, 0.0, 1.0),
   glm::vec3(1.0, 0.0, 1.0)
};

// Dreiecke
std::vector<glm::uvec3> idx {
   glm::vec3(0, 1, 2),
   glm::vec3(2, 1, 3)
};
```

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glVertexPointer(
     3, GL_FLOAT, 0,
     positions.data());
glColorPointer(
     3, GL_FLOAT, 0,
     colors.data());

glDrawElements(
    GL_TRIANGLES,
    idx.size() * 3,
    GL_UNSIGNED_INT,
    idx.data());

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```

Übungsblatt - Demo

C++ in a Nutshell

Einordnung von C++

- C++ wird kompiliert
  - BASIC, JavaScript, Python werden teilweise interpretiert
- C++-Compiler erzeugen nativen Maschinencode
  - Visual Basic, Java, C# erzeugen „Zwischencode“, der nicht direkt ausgeführt werden kann
- C++ ist portabel
  - Unterschiedliche Compiler können teilweise aus dem gleichen C++-Quellcode ausführbare Dateien für verschiedene Systeme generieren
- C++ ist hardwarenah
  - Größe von Datentypen hängt teilweise vom Zielsystem (32/64-Bit) ab
  - Direkte Speicherzugriffe möglich
- C++ erfordert kein Laufzeitsystem
  - Standardbibliothek kann in die ausführbare Datei integriert werden

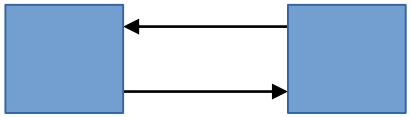
Abgrenzung von C

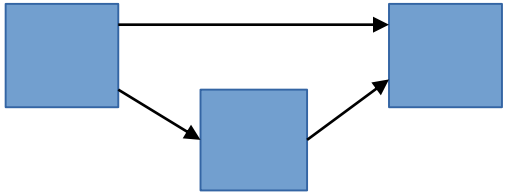
- C++ ist ursprünglich eine Erweiterung von C
- C++ erlaubt objektorientierte Programmierung
  - imperative Programmierung weiterhin möglich
- Großteils abwärtskompatibel
  - Viele C-Programm werden ohne oder mit nur geringen Änderungen von einem C++-Compiler akzeptiert
- C++ liefert neue Standardbibliotheken (STL)
  - C-Bibliotheken weiterhin nutzbar

Header

- Oft Aufteilung in Schnittstelle (.h) und Implementierung (.cpp/.cc)
- Einbinden mit der Präprozessoranweisung #include
  - `#include <iostream> // Oft: System-Header`
  - `#include "test.h"   // Oft: lokale Header`
- Mehrfaches Einbinden einer Datei sollte verhindert werden
  - Oder '#pragma once' am Anfang der Header-Datei
  - Oder Include-Guards

```
#ifndef SOME_UNIFORM_IDENTIFIER
#define SOME_UNIFORM_IDENTIFIER
// Eigentlicher Code
#endif
```





Grundlegende Datentypen

- `char, int, long int, float, double, bool`
- `unsigned char, unsigned int`
- `void`: 'nichts', wird verwendet als Rückgabetyp für Funktionen ohne Rückgabewert, oder für Zeiger, die auf alles zeigen können.
- auto (seit C++ 11): Typ wird automatisch abgeleitet, z.B.
  - `auto a = 0;     // a ist ein int`
  - `auto b = 0.0f; // b ist ein float`
  - `auto c = 0.0;  // c ist ein double`

Heap und Stack

```
{
    // a liegt auf dem Stack.
    int a = 0;
}
// Hier existiert a nicht mehr, der Speicher wurde freigegeben.

{
    int* b = new int;       // b ist ein Zeiger auf ein int auf dem Heap.
    *b = 0;                 // Schreibe 0 and die Adresse, auf die b zeigt.
}
// Hier existiert b nicht mehr, der allokierte Speicher wurde aber NICHT
// freigegeben. Das nennt man ein Speicherleck.

{
    int* c = new int;
    *c = 0;
    // Der Speicher muss explizit freigegeben werden.
    // Für jedes new braucht man ein delete!
    delete c;
}
```

Arrays

```
/*
 * Dieses Array liegt auf dem Stack und wird direkt initialisiert.
 */
int ar_stack[3] = {1, 2, 3};
ar_stack[0]  = 4;  // Zugriff über operator[]
ar_stack[-1] = 1;  // Legaler code, aber die Welt wird untergehen.

/*
 * Dieses Array liegt auf dem Heap.
 */
int* ar_heap = new int[2];
ar_heap[0] = 0; // Zugriff über operator[]
ar_heap[1] = 2;
delete[] ar_heap; // Achtung: delete[], nicht delete!
```

Heap und Stack, Teil 2

- Also, Speicher auf dem Heap
  - muss explizit freigegeben werden
  - hat Overhead beim Allokieren
- Darum soweit wie möglich den Stack benutzen!
- Als Alternative zu Arrays bietet sich std::vector an

```
#include <vector>

std::vector<int> vec1(4);         // Vektor mit 4 Elementen.
std::vector<int> vec2(4, 1);      // Elemente werden auf 1 initialisiert.
std::vector<int> vec3 {1, 2, 3}; // Explizite Initialisierung (C++ 11).

vec1[0] = vec2[1];        // Zugriff wie bei Arrays.
vec1.at(2) = vec2.at(3); // .at() prüft, ob der Index gültig ist.
                          // (langsamer aber sicher)
```

Zeiger und Referenzen

```
// Ein Zeiger ist eine Variable, die eine Adresse speichert.
int a = 4;
int* ptr = &a; // & ist der Adressoperator.

// Zugriff auf den referenzierten Speicher durch Dereferenzieren.
*ptr = 5; // Jetzt ist der Wert von a 5.

// Referenzen sind auch Zeiger, werden aber wie normale Variablen
// benutzt. Das ist syntaktisch einfacher:
int& ref = a;
ref = 6; // Jetzt ist der Wert von a 6.

// Zeiger können neu zugewiesen werden, Referenzen nicht:
int b = 3;
ptr = &b;
ref = b;   // Jetzt hat a den Wert 3. ref zeigt weiterhin auf a.

// Zeiger-Arithmetik:
int ar[4];
*(ar + 1) = 2; // Jetzt ist der Wert von ar[1] 2.

// Ungültige Zeiger (die auf nichts zeigen):
int* invalid1 = NULL;    // Früher.
int* invalid2 = nullptr; // C++ 11.
```

const

```
// const bedeutet in einer Deklaration, dass eine Variable
// nicht verändert werden kann.
int const a = 4; // Äquivalent: const int a = 4.
a = 5;           // Kompiliert nicht.

// Die Position des Wortes const ist wichtig:
int b = 0;
int c = 0;
int const* ptr1 = &b;
int * const ptr2 = &b;
*ptr1 = 4;  // Kompiliert nicht
ptr1  = &a; // Kompiliert, man kann den Zeiger 'umbiegen'.
*ptr2 = 4;  // Kompiliert.
ptr2  = &c; // Kompiliert nicht, ein int* const kann nicht umgebogen werden.
int const* const ptr3 = &a;

// Methoden, die const sind, können ihr Objekt nicht verändern
class Test
{
        void test() const
        {
                foo = 4; // Kompiliert nicht, test ist const.
        }
        int foo;
};
```

Strings

```
/*
 * Strings haben eigentlich den Typ char const*.
 */
char const* msg = "Nachricht.";

/*
 * Solche Strings sind aber gefährlich. Man sollte stattdessen
 * std::string verwenden.
 */
#include <string>
std::string msg("Nachricht.");
```

Textausgabe

```
/*
 * In c++ verwendet man streams für Operationen auf Dateien. Besonders sind
 * hier die standard input / output streams zu nennen.
 * Streams überladen üblicherweise operator<< oder operator>>.
 */
#include <iostream>
#include <string>

std::cout << "Hallo Welt!" << std::endl;
std::cerr << "Fehler!"     << std::endl;
std::string eingabe;
std::cin  >> eingabe;

/*
 * Nützlich sind auch stringstreams.
 */
#include <sstream>
#include <string>

std::ostringstream os;
int punkte = 4;
os << "Sie haben " << punkte << " Punkte." << std::endl;
std::string msg = os.str();
```

sizeof

```
/*
 * Die Größe eines Datentyps lässt sich mit sizeof() bestimmen.
 */
std::cout << "sizeof(int) = " << sizeof(int) << " bytes." << std::endl;

/*
 * Das funktioniert auch mit Variablen ...
 */
int a = 0;
std::cout << "sizeof(a) = " << sizeof(a) << " bytes." << std::endl;

/*
 * ... und mit Arrays.
 */
int a[] = {1, 2, 3};
std::cout << "Das Array hat "
          << sizeof(a) / sizeof(a[0])
          << " Elemente." << std::endl;
```

Die main-Funktion

```
/*
 * Optional können Kommandozeilenparameter übergeben werden
 */
#include <iostream>
int main(int argc, char** argv)
{
        for (int i = 0; i < argc; ++i)
        {
                std::cout << argv[i] << std::endl;
        }
        return 0;
}

$ ./program test1 test2
program
test1
test2
```

Klassen / Strukturen

```
// In C++ verhalten sich Klassen und Strukturen sehr ähnlich. Beide
// können Methoden haben, unterstützen Vererbung, usw.
// Einziger Unterschied ist die Standard-Sichtbarkeit.
struct S
{
        double x;      // public, jeder kann x benutzen.

        protected:
                double y; // Nur S und abgeleitete structs können y benutzen.

        private:
                double z; // Nur S kann y benutzen.
}; // Achtung, hier muss ein Semikolon stehen!

class C
{
        double x; // private!
};

// Zugriff auf member:
S s;
s.x = 5;
S* ptr = &s;
ptr->x = 4;  // Äquivalent zu (*ptr).x = 4.
```

Klassen / Strukturen

```
// Klassen / Strukturen können Konstruktoren und Destruktoren haben.
struct S
{
        S(int x) { m_x = new int(x); }
        ~S()     { delete m_x;      }

        private:
                int* m_x;
};
S s(4);

// Initialisierung im Konstruktor funktioniert auch in der
// Initialisierungsliste.
class Point
{
        public:
                Point(double x, double y) :
                        m_x(x), m_y(y)
                {}
        private:
                double m_x, m_y;
};
Point p(2.0, 3.0);
```

std::vector

```
std::vector<int> v;          // v = {}
int size0 = v.size();        // size0 = 0

v.push_back(1);              // v = {1}
int size1 = v.size();        // size1 = 1

v.resize(5);                 // v = {1, 0, 0, 0, 0}
int size2 = v.size();        // size2 = 5

v[4] = 2;                    // v = {1, 0, 0, 0, 2}
int size3 = v.size();        // size3 = 5

v.push_back(3);              // v = {1, 0, 0, 0, 2, 3}
int size4 = v.size();        // size4 = 6
```

Weitere Themen

- Standard Template Library (STL)
  - Die Standardbibliothek von C++
  - Datenstrukturen & Algorithmen
- Templates
  - Die C++-Variante von Generics
  - Turing-vollständig, leider auch ziemlich komplex
  - Kann zu hohen Kompilierzeiten und unleserlichen Fehlermeldungen führen
  - Aber sehr mächtig (siehe STL)
- Lambda-Ausdrücke
  - Erlauben funktionalen Programmierstil
- Multithreading
  - STL unterstützt dies seit C++11 (<thread>, <atomic>, <mutex>...)