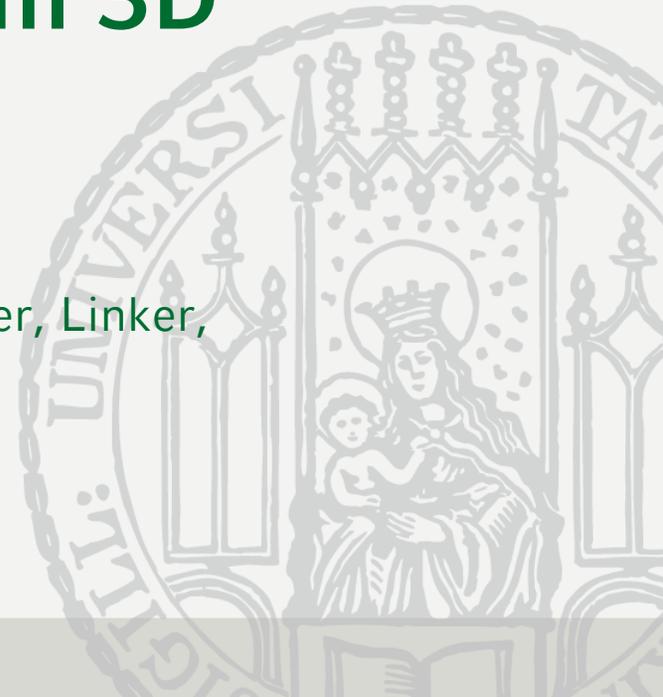


Dipl.Inf. Otmar Hilliges

Programmierpraktikum 3D Computer Grafik

Einführung in C++:

Header- u. Source files, Präprozessor, Compiler, Linker,
Standard I/O





Termin und Ort:

- Wöchentliche Besprechung: Mittwoch, 10:15 - 11:45
- Ort: Amalienstr. 17, Raum 107
- Pflichttermin – Bei Nichterscheinen vorher (begründet) entschuldigen.

Ablauf des Praktikums

- Es wird wöchentliche Übungsblätter geben (Phase I)
 - Dienstag 12:00 – Neues Aufgabenblatt.
 - Mittwoch 10:15-11:45 – Wöchentliches Treffen.
 - Dienstag – Montag – individuelle Bearbeitung der Aufgaben.
 - Montag 12:00 Abgabe der Aufgaben (per SVN commit).
- Zum Ende des Semesters wird es eine Projektarbeit geben die in Gruppen bearbeitet wird.



Diskussions-Liste:

- Anmeldung und Verwaltung:
<https://tools.rz.ifi.lmu.de/mailman/listinfo/3dpss07>
- Mails an: 3dpss07_at_lists.ifi.lmu.de
- Soll zum Austausch von Fragen, Problemen und auch Lösungen rund um das 3D Programmierpraktikum dienen.

SVN Repository:

- Abgaben erfolgen per SVN commit.
- Sie sollten username/pwd per e-mail erhalten haben.
- <svn://svn.medien.ifi.lmu.de/ss07/3d-prakt>
- Siehe auch http://www.medien.ifi.lmu.de/lehre/ss07/3dp/merkblatt_svn.pdf
- Tutorial: <http://svnbook.red-bean.com/>



- Erlernen von Grundlegenden Konzepten der 3D Programmierung.
- Erwerb von Kenntnissen in C++, OpenGL und GLUT.
- Verständnis der Mathematik für 3D-Grafik
- Fähigkeit zur selbständigen (Team-) Arbeit anhand eines komplexeren Systems.
 - Implementierung eines kleinen Spiels / Animation
 - Teams von 2-3 Personen
 - (Netzwerkkommunikation -> Teams können gegeneinander spielen)



- Bearbeitung aller Aufgaben
- Abgaben müssen
 - Pünktlich
 - Korrekt (kompilieren mit einem Klick)
 - Lauffähig
 - Und nicht abgeschrieben sein!
- Am Ende des Semesters muss das Team-Projekt präsentiert und demonstriert werden (aka real implementiert sein).
- Anwesenheit bei den Gruppentreffen ist zwingend.



- Zur Bearbeitung der Aufgaben stehen die Rechner des CIP-Pools in der Amalienstrasse zur Verfügung.
- C++, OpenGL und GLUT sind Plattform unabhängig -> Achten sie darauf, dass ihre Abgaben das ebenfalls sind.
- Zum entwickeln unter Linux empfehlen wir die GNU GCC Compiler-Suite
- Zum entwickeln unter Windows empfehlen wir MS Visual Studio
 - Alle Studenten des IFI können VS 2005 kostenlos über MSDNAA <http://www.rz.ifi.lmu.de/Dienste/MSDNAA> beziehen (ebenso MS Windows).
- Zur Abgabe ihrer Aufgaben sollen Sourcen und Projekt-Files eingereicht werden.
 - .cpp, .h, .sln, .vcproj, makefiles sollen eingereicht werden.
 - .exe, .o, .ncb, .suo, .obj etc. dürfen nicht eingereicht werden!

Aufbau eines C++ Programms



- Jedes (lauffähige) C++ Programm benötigt einen Einstiegspunkt.
- Das ist die reservierte (d.h. darf nur einmal existieren) Methode `int main()`
 - `int main()`
 - `int main(int argc, char** argv)`
- Die `Main` Methode ist statisch d.h. sie kommt pro Prozess nur *einmal* vor.
- Aufruf ohne Instanz einer Klasse möglich *aber* kein Zugriff auf Member von Instanzen.
- Die `Main` Methode wird als erstes ausgeführt, wenn ein Programm gestartet wird.
- Wenn die `Main` Methode beendet wird, wird auch das Programm beendet.

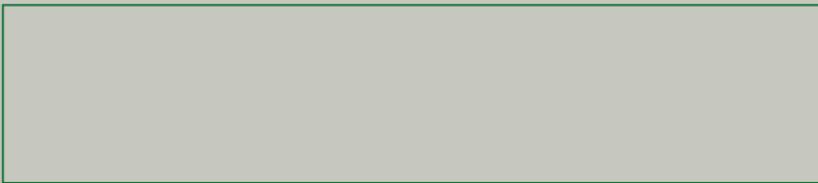


Ein einfaches Beispielprogramm:

```
#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "3D-PP" << std::endl;
    return 0;
}
```

- Kompilieren z.B. durch `gcc 3dp.cpp -o 3dp` oder `Strg+F5`
- Ausgabe bei Programmausführung: 3D-PP
- Rückgabewert 0 bedeutet, dass das Programm ohne Fehler beendet wurde
- Rückgabewerte ungleich 0 zeigen Fehler an
- Gilt auch für viele eingebaute C++ Funktionen.



Namespaces



- C++ Programme sind oft modular zusammen gesetzt.
- Klassen-/Funktionsnamen können also im Konflikt zu einander stehen
 - Code kommt von vielen Verschiedenen Quellen
- Um Konflikte zu vermeiden gibt es sogenannte `Namespaces`
- Im Beispielprogramm haben sie dieses Konstrukt kennen gelernt:

```
std::cout << "foo" << std::endl;
```
- `cout` ist eine Funktion des Namespace `std`.
- Normalerweise müssen Funktionen immer mit ihrem Namespace Präfix aufgerufen werden.
- Außer es wird eine `using` Direktive verwendet.

```
using namespace std;  
...  
cout << "Blah blah blah!" << endl;
```



- Um Konflikte mit Funktionen im globalen Namensraum (ohne Präfix) zu vermeiden sollten eigene Programme immer einen eigenen Namespace definieren.

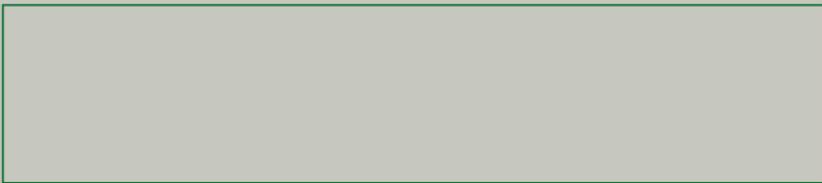
```
namespace A{  
    typedef ... String;           //definiert neuen Datentyp A::String  
    void foo (String);           //definiert A::foo(String)  
}
```

- Das Schlüsselwort `namespace` ordnet die darin befindlichen Symbole dem angegebenen Gültigkeitsbereich zu
- `A::String s1;` ist also unterschiedlich von `B::Strings s2;`



- C++ hat keine geschlossene API wie Java (es gibt aber einen ANSI Standard)
- Viele elementare Datentypen und Algorithmen sind in der STL implementiert (Vektoren, Listen, Iteratoren, Sortieralgorithmen,...)
- Ausführliche Beschreibungen im Web und in [N. Josuttis] oder [B.Strousstrup]

```
include <iostream>
include <string>
int main(){
    std::string s = "Was ist die Lösung";
    std::cout<< s << std::endl;
    std::cin>> t;
    if( t.compare("per aspera ad astra"==0) ){
        std::cout<< "Richtig!"<< std::endl;
    }
    return 0;
}
```



Kontrollstrukturen

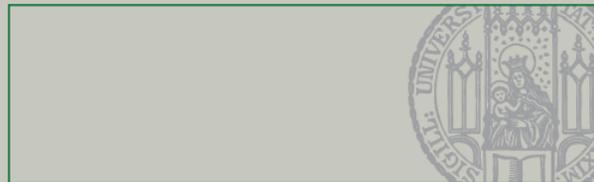


Drei Schleifen (analog zu Java):

- `for([start];[bedingung];[ende]){...}`
- `while([bedingung]){...}`
- `do{...}while([bedingung]);`

Beispiele:

- `for(int i=0; i<5; i++) {...}`
- `while(i < 5) {...}`
- `do {...} while(i < 5);`



Ein-/Ausgabe



Unformatierte Ausgabe:

- Ausgabestrom: `std::cout`
- Format muss nicht angegeben werden, da sich dies aus dem Datentyp ergibt (ähnlich der Variante `println()` in Java)
- Nicht auf Standardtypen beschränkt
- Alle Teil-Zeichenketten werden mit dem Operator `<<` getrennt

Beispiel:

```
int i = 7;  
std::cout << "#: " << i << std::endl;
```



Formatierte Ausgabe:

- Funktion `printf()`
- Argumente: Eine Zeichenkette plus ALLE vorkommenden Variablen

Beispiel:

```
printf("int: %d, float %f \n", 7, 4.2);
```

Weitere Erläuterungen:

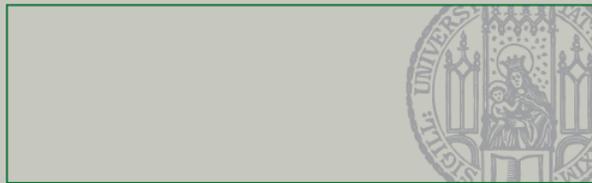
<http://www.cplusplus.com/ref/cstdio/printf.html>



- Eingabestrom: `cin`
- Kein `endl` erforderlich
- `fflush(stdin)` leert den Eingabestrom

Beispiel:

```
int i;  
cin >> i;    // Einlesen eines int  
fflush(stdin);
```



C++ vs. Java



Klassen und Objekte

- **Klasse** ist die Definition eines Objektes – also eine Prototypische Beschreibung.
- **Objekte** sind Instanzen von Klassen sie existieren nur zur Laufzeit eines Programms – verschiedene Instanzen einer Klasse können unterschiedliche Konfigurationen haben.
- **Objekte** kapseln Daten (Membervariablen) und bieten Funktionen (Memberfunktionen) zum ändern/abfragen dieser Daten und damit dem Status des Objektes an.



Dynamischer vs. Statischer Kontext

- Statische Variablen gibt es nur einmal pro Klasse und Prozess
 - Änderungen von statischen Variablen wirken sich auf alle Instanzen aus
 - Bsp. Zähler für aktuell erzeugte Instanzen.
- Dynamische Variablen sind Membervariablen und können sich von Instanz zu Instanz unterscheiden.
- Statische Funktionen können ohne Instanz (aus statischem Kontext) aufgerufen werden dürfen aber nicht auf Member von Instanzen zugreifen.
- Um auf Member zuzugreifen muss erst eine Instanz erzeugt werden.



C++ spezifische Konzepte

- Schlüsselwort `virtual` – nur virtuelle Funktionen können nach Vererbung überschrieben werden.
- Schlüsselwort `abstract` – nicht alle (virtuellen) Funktionen sind implementiert -> keine Instanzierung möglich.
- Mehrfachvererbung ist in C++ möglich in Java nicht.
- Definition/Deklaration und Implementierung von Variablen/Klassen + Methoden wird in C++ üblicherweise getrennt (muss aber nicht) -> Header und Source-Files



Die Header Datei

- Enthält Definitionen und Deklarationen -> eigentliche Implementierung erfolgt dann in der Source (.cpp) Datei.
- Außerdem enthält der Header Präprozessoranweisungen, z.B. `#include` um externe Klassen einzubinden.

1. Klassendeklaration

```
class MyClass {...};
```

2. Typdefinition

```
struct Position { int x, y };
```

3. Aufzählungen

```
enum Ampel { rot, gelb, grün };
```

4. Funktionsdeklaration

```
int rechtEckFlaeche { int width, int height};
```

5. Konstantendefinition

```
const float pi = 3.141593;
```

6. Datendeklaration

```
int a = null;
```

7. Präprozessoranweisungen

```
#include <iostream>  
#include „myheader.h“  
#define VERSION12  
#ifdef VERSION12
```



myClass.h

```
class MyClass : MyParentClass
{ // die Klasse MyClass erbt von MyParentClass
  public:
    MyClass(); //standard Konstruktor
    MyClass(std::stringtext); //zweiter Konstruktor
    virtual ~MyClass; //Destruktor

    virtual int func()=0; //eine rein virtuelle(=abstrakte) Funktion
    static double funct(); //eine statische Funktion
    static int m_someNumber; //eine statische Membervariable

  protected:
    virtual int fun(); //eine virtuelle Funktion

  private:
    void fu(); //eine Funktion
    std::string m_someString; //eine Membervariable
};
```



```
#include "myClass.h"
```

```
myClass.cpp
```

```
int MyClass::m_someNumber(5);
```

```
MyClass::MyClass() { //Standardkonstruktor  
    m_someString= "EineIntialisierungvomText";  
}
```

```
MyClass::MyClass(std::stringtext) //zweiter Konstruktor  
    m_someString= text;  
}
```

```
MyClass::~MyClass() {} //Destruktor
```

```
void MyClass::fu() {} //eine Funktion  
int MyClass::fun(){return 4;} //eine virtuelle Funktion  
double MyClass::funct(){return 2;} //eine statische Funktion
```



Zeiger (Pointer) und Referenzen

- Pointer sind Verweise auf Daten im Speicher -> Pointer zeigt auf die Speicherstelle.

```
int x = 42;    //ein Integer
int* p_x;     //(NULL)Pointer auf ein Integer
```

- Zur Verwaltung von Pointern dienen die Operatoren & und * .
 - & liefert die Adresse von einem Datum also den Pointer.
 - * dereferenziert einen Zeiger, d.h. die tatsächlichen Daten.

```
*p_x = &x;    //p_x zeigt auf x jetzt 7?
```



```
std::cout << *p_x << std::endl; //das, worauf p_x zeigt, ausgeben
```



C++ ist formatfrei, deswegen gibt es unterschiedliche Notationen für Pointer:

```
int *xp;  
int * xp;  
int*xp;  
int* xp;           //alle Notationen sind äquivalent
```

Vorsicht bei Verwendung der letzten Notation:

```
int* p1, p2;       //Erzeugt?  
int *p1, p2;
```



Arrays (Felder):

- Anlegen eines Arrays: `int werte[10];`
- Zugriff auf Feldelemente mit `[]` Operator

Beispiel:

```
// ein Byte Array (10MByte) um den Wert 5 erhöhen
char* array= newchar[10000000];
for (inti=0; i < 10000000; i++){
    array[i] += 5;
}
delete[] array;
```



Effizientes Programmieren mittels Zeigerarithmetik

```
char* array= new char[10000000];      //zeigt auf erstes Element
char* endOfArray= array+10000000;    //zeigt auf letztes +1 Element

for (char* i=array; i < endOfArray; i++){
    (*i) += 5;    //Wert des Elements, auf das Zeiger i zeigt
}
delete[] array;
```

Diese Implementierung ist deutlich schneller. Warum?



Auch die Geschwindigkeit von Methodenaufrufen kann durch den Einsatz von Pointern beeinträchtigt werden.

```
Eintrag suchFunktion(strings, Telefonbuch t){  
    //kopiert beim Aufruf das ganze Telefonbuch  
    ...  
    return e;  
}
```

```
Eintrag suchFunktion(string& s, Telefonbuch& t){ // kopiert nur 4 Byte  
    ...  
    return e;  
}
```

```
Eintrag& suchFunktion(string& s, Telefonbuch& t){  
    //wenn man den Eintrag ändert, ändert sich auch der im Telefonbuch!  
    ...  
    return e;  
}
```

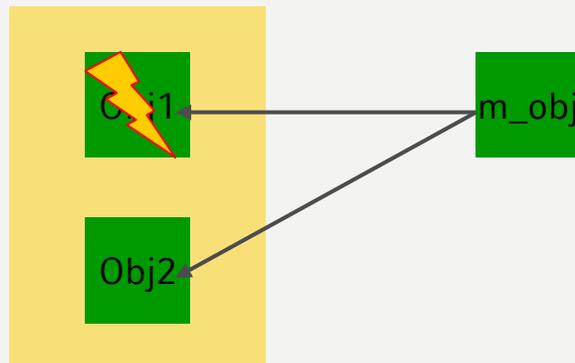


- Java verwaltet Speicher automatisch (Garbage Collection)
- In C++ muss man seinen Speicher (teilweise) selber Verwalten.
- Es gibt zwei Möglichkeiten Speicher anzufordern:

```
MyObject mo1; // Stack  
MyObject* mo2 = new MyObject(); // Heap
```

- **Stack:** Speicher wird freigegeben sobald die dazugehörige Variable ihre Gültigkeit verliert (z.B. lokale Variablen innerhalb einer Methode)
- **Heap:** Speicher wird erst wieder freigegeben, wenn man `delete` aufruft!

- `new/delete` ist die Fehlerquelle Nummer 1 für C++ Anfänger – warum?
 - Heap-Overflow weil sehr viele Objekte angelegt und nicht mehr gelöscht wurden.
 - Aufruf von `delete` auf Objekten die ihre Gültigkeit verloren haben.
- Zu jedem `new` gehört ein `delete`!
- Vorsicht mit Pointern
 - Pointer sollten immer initialisiert werden sonst ist `delete` ungültig.
 - Wenn für ein Objekt nur ein Pointer existiert darf dieser nicht umgebogen werden bevor `delete` aufgerufen wurde (Memory Leak)





Präprozessor, Compiler, Linker



Compiler Collection

- Eine Sammlung von Programmen, die man zur Programmerstellung benötigt
- Präprozessor – Textersetzung in Sourcefiles vor dem eigentlichen Kompilieren
- Übersetzer [Compiler] Übersetzt Programm(fragmente) von C++ in Binärcode (maschinenlesbar)
- Verknüpfer [Linker] Setzt Binärcodefragmente zu lauffähigem Programm zusammen



- Anweisungen für den Präprozessor fangen mit # an
- `#include "file.h"` fügt die Header-Datei `.h` ein
- `#define FOO 1` ersetzt überall im Programmtext `FOO` mit `1`

```
#ifdef BAR
    Code 1
#else
    Code 2
#endif
```

- Fügt im Programmtext Code 1 ein falls `BAR` definiert wurde, ansonsten Code 2



- Schutz vor mehrfach Einbindung von Headern

```
#ifndef _MYCLASS_H
#define _MYCLASS_H

#include ...

class MyClass{...};

#endif // _MYCLASS_H
```

- Ersetzungsworte immer in Großbuchstaben
- Präprozessor Einsatz ist unflexibel und sollte minimiert werden!



Bibliotheken (*.lib,*.so,*.dll) sind vorkompilierte Binärdateien ohne Einstiegspunkt

- Zeitvorteil, da nicht immer alles neu kompiliert werden muss
- Wiederverwendbarkeit -> kein mehrfach erfinden des Rades nötig
- Platz- und Geschwindigkeitsvorteil, da häufig benötigter Code nur einmal auf der Platte gespeichert werden muss (und teilweise auch nur einmal im Speicher)



- Das eigentliche Erstellen von Programmen wird vom Linker erledigt
- Linker ersetzt symbolische Funktionen in Binärcode mit Adressen im Speicher.
- Außerdem werden eigene Programmfragmente und verwendete Bibliotheken zusammen gesetzt.
- Zwei Arten des Verknüpfens möglich:
 - Statisches Verknüpfen
 - Dynamisches Verknüpfen



Statisches Verknüpfen:

- Einfachste Methode
- Erzeugt eine große Datei
- Bei Änderungen an der Bibliothek muss neu kompiliert werden

Dynamisches Verknüpfen:

- Komplizierter Mechanismus der auf unterschiedlichen BS verschieden funktioniert.
- Kleinere, modulare Dateien
- Bibliotheken können von mehreren Programmen gleichzeitig benutzt werden
- Bei Änderungen an der Bibliothek muss nicht neu kompiliert werden



Einbinden von fremden Bibliotheken

- Im eigenen Code den/die Header einbinden (für die Methoden Deklarationen)
- Den Linker-Suchpfad so anpassen, dass Bibliotheken gefunden werden können (lib path)
- Den Compiler-Suchpfad so anpassen, dass Headerdateien gefunden werden können (include path)

Tipps zur Fehlersuche:

- Meist wurde in den angegebenen Objektdateien ein Symbol (Funktion, Variable, Klasse, ...) nicht gefunden
 - Schreibfehler
 - Falscher oder fehlender Namensraum
 - Schlüsselwörter `virtual/static` vergessen
- Der Compiler prüft **nie** nach, ob eine deklarierte Funktion (im Header) implementiert wurde -> Laufzeitfehler
- Implementierte Methoden ohne Deklaration geben dagegen immer einen Fehler
- SegFaults sind das äquivalent zu Nullpointer Exceptions in Java



Weiterführende Literatur

- Nicolai Josuttis, “Objektorientiertes Programmieren in C++”, ISBN 3-8273-1771-1
- Bjarne Stroustrup, “The C++ Programming Language”, ISBN-13: 978-0201700732
- <http://www.cplusplus.com/reference/stl/>
- <http://www.cplusplus.com/ref/cstdio/>