C++ Kurs Teil 1

- "hello world"
- Vergleich von C++ und Java
  - Architektur, Anwendungsspektrum, primitive Datentypen, Funktionsaufrufe, Referenzen, Klassen
- C++ Spezialitäten
  - Schlüsselwort 'const', Copy Constructor, Destructor, Source Code Organisation, 'inline', Überladen von Operatoren
- Standard-Bibliothek: Streams
- Kodierungsrichtlinien
- Editor

"hello world!"

hello_world.cc:

```
#include <iostream>

int main( int argc, char** argv)
{
  std::cout << "hello world!\n";
  return 0;
}
```

iostream Bibliothek benutzen, (spitze Klammern heißt: Systembibliothek)

Jedes C / C++ Programm muß eine **main()** Funktion haben

std::cout ist der Standard Ausgabe Stream

Makefile:

```
all: hello_world
```

make soll die Datei **hello_world** auf den neuesten Stand bringen

Kommandos in der shell:

```
gutemine:~> make
g++  hello_world.cc  -o hello_world

gutemine:~> ./hello_world
hello world!
```

Da kein **hello_world**, aber ein **hello_world.cc** existiert, wird automatisch der C++ Compiler des Systems aufgerufen

Programm **hello_world** ausführen

Vergleich von C++ und Java

Architektur

C++

- **Compiler** (Quelltext wird einmal auf dem Rechner des Entwicklers in Maschinensprache übersetzt)
  - \+ optimale Geschwindigkeit
  - – Wenn das Programm auf mehreren Plattformen (z.B. Linux, Windows, Mac) laufen soll, muß der Entwickler entsprechend viele Versionen von dem ausführbaren Programm erzeugen.

Java

- **Interpreter** (Quelltext wird bei jedem Durchlauf auf dem Rechner des Anwenders neu interpretiert)
  - – langsam
  - \+ vom Entwickler muß nur eine Version des Programmes bereitgestellt werden

Vergleich von C++ und Java

Anwendungsspektrum

C++

- Low-Level Hardware-Treiber
- Betriebssysteme
- Bibliotheken
- Office-Pakete

Java

- Webseiten
- Sicherheitsrelevante, verteilte Anwendungen (Home-Banking)
- Lehre

Vergleich von C++ und Java

primitive Datentypen

Primitive Datentypen in C++
(garantierte **Mindest-**Genauigkeit)

- bool
- (unsigned) char
- (unsigned) wchar_t
- (unsigned) short
- (unsigned) int
- (unsigned) long
- float
- double
- void

Primitive Datentypen in Java
(garantierte Genauigkeit)

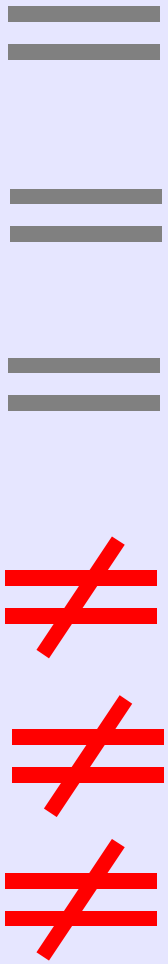
- boolean
- char
- int
- long
- float
- double
- void

Vergleich von C++ und Java

primitive Datentypen und Klassen

In **C++** verhalten sich Klassen genau so wie primitive Datentypen:

```
int a;              // Erzeugen eines
                    // integers
...
int b = a;          // b enthält eine
                    // Kopie von a

b = 42;             // nur b wird
                    // verändert

Body x;             // Erzeugen eines
                    // Body
...
Body y = x;         // y enthält eine
                    // Kopie von x

y.setVol(42);       // nur y wird
                    // verändert
```



In **Java** ist das Verhalten unterschiedlich:

```
int a;              // Erzeugen eines
                    // integers
...
int b = a;          // b enthält eine
                    // Kopie von a

b = 42;             // nur b wird
                    // verändert

Body x;             // es wird kein
                    // Body erzeugt
...
Body y = x;         // y enthält eine
                    // Referenz auf x

y.setVol(42);       // y und x
werden
                    // verändert
```

Vergleich von C++ und Java

Funktionsaufrufe

Funktions-Deklarationen in **C++**

```
void f1( int a)     // a ist eine
{                   // Kopie
  a = 42;
}

void f2( Body x)    // x ist eine
{                   // Kopie
  x.setVol( 42);
}

// Aufruf
f1(b);              // b wird nicht
                    // verändert

f2(y);              // y wird nicht
                    // verändert
```

Funktions-Deklarationen in **Java**

```
void f1( int a)     // a ist eine
{                   // Kopie
  a = 42;
}

void f2( Body x)    // x ist eine
{                   // Referenz
  x.setVol( 42);
}

// Aufruf
f1(b);              // b wird nicht
                    // verändert

f2(y);              // y wird
                    // verändert
```

Vergleich von C++ und Java

Referenzen in C++

```
int a;              // Erzeugen eines
                    // integers
...
int& b = a;         // b enthält eine
                    // Referenz auf a

b = 42;             // b und a werden
                    // verändert

Body x;             // Erzeugen eines
                    // Body
...
Body& y = x;        // y enthält eine
                    // Referenz auf x

y.setVol(42);       // y und x werden
                    // verändert
```

In C++ gibt es natürlich auch die Möglichkeit Referenzen statt Kopien zu erzeugen (sowohl für primitive Datentypen als auch für Klassen). Dafür wird ein '&' hinter den Datentyp geschrieben.

Vergleich von C++ und Java

Referenzen in C++

```
void f1( int& a)   // a ist eine
{                  // Referenz
  a = 42;
}

void f2( Body& x)    // x ist eine
{                    // Referenz
  x.setVol( 42);
}

...

// Aufruf
f1(b);             // b wird verändert

f2(y);             // y wird verändert
```

Dieselbe Syntax wird auch bei Funktions-Deklarationen angewandt

Vergleich von C++ und Java

Referenzen in C++

Achtung: In C++ kann eine Referenz nach der Erzeugung **nicht mehr** "verbogen" werden:

```
Body a;
Body b;
...

Body& c = a;      // c und a bezeich-
                  // nen jetzt
                  // dasselbe Objekt

c = b;            // b wird in c (und
                  // somit auch in a)
                  // kopiert!
```

```
Body a;
Body b;
...

Body c = a;       // c und a bezeich-
                  // nen jetzt
                  // dasselbe Objekt

c = b;            // c wird auf b
                  // umgebogen
```

Vergleich von C++ und Java

Syntax von Klassendefinitionen

```
class Zylinder : public Body
{
public:
   // Constructor
   Zylinder();
   // Copy Constructor
   Zylinder( const Zylinder& z);
   // Destructor
   ~Zylinder();
   float volume() const;
   void setHeight( float h);

protected:
   void updateVolumeCache();

private:
   float  _height;
   float  _radius;
};
```

```
class Zylinder extends Body
{

   // Constructor
   Zylinder() { ...}
   // Copy Constructor ...
   // ... so ähnlich wie clone()
   // Destructor
   public void finalize() { ...}
   public float volume() { ...}
   public void setHeight( float h) { ...}

   protected void updateVolumeCache()
   { ...}


   private float  _height;
   private float  _radius;
};
```

C++ Spezialitäten

Schlüsselwort 'const'

```
class Zylinder : public Body
{
public:
   // Constructor
   Zylinder();
   // Copy Constructor
   Zylinder( const Zylinder& z);
   // Destructor
   ~Zylinder();
   float volume() const;
   void setHeight( float h);
   void merge( const Zylinder& z);

protected:
   void updateVolumeCache();

private:
   float  _height;
   float  _radius;
};
```

Zu dem Schlüsselwort 'const' gibt es in Java kein Equivalent. Je nachdem, wo es auftaucht, hat es verschiedene Bedeutungen:

Die Funktion volume() darf die privaten Variablen der Klasse nicht verändern

Das übergebene Zylinder-Objekt darf nicht verändert werden

C++ Spezialitäten

Schlüsselwort 'const'

```
void doSomething( const Zylinder& z)
{
  std::cout << z.volume();    // OK
  z.setHeight( 123);          // Error!
}
```

Eine belibige Funktion, die ein 'const' – Objekt übergeben bekommt,...

... darf nur die 'const'-members dieses Objektes aufrufen

C++ Spezialitäten

Constructor, Copy Constructor, operator=, Destructor

```
void dummy()
{
  Body a;  // Default Constructor wird aufgerufen

  Body b = a; // Copy Constructor wird aufgerufen

  Body c;
  c = b;    // c.operator=(const Body& b) wird aufgerufen

}  // Destructor von a, b und c wird aufgerufen
```

Source Code Organisation

```
class Body
{
public:
   float volume() const;
...
};
```

Body.hh: Headerfile enthält nur die Deklaration der Body-Klasse

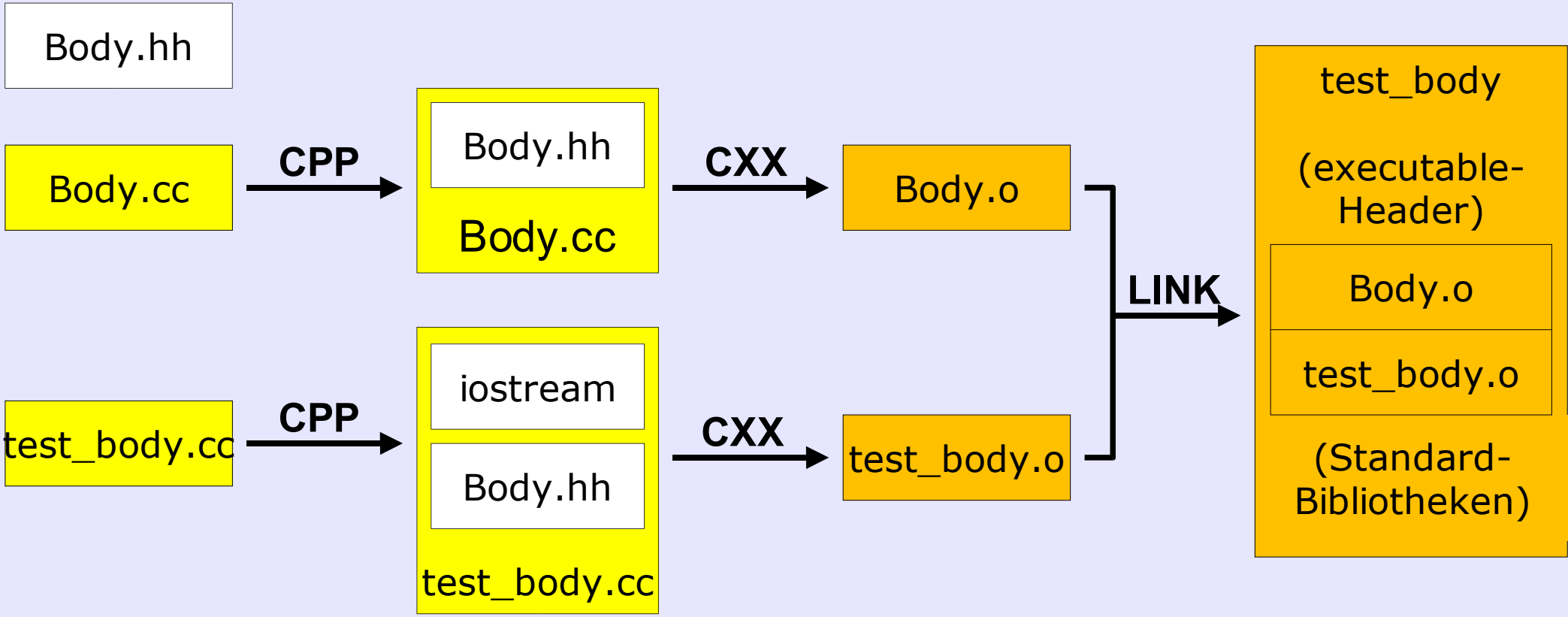
```
#include "Body.hh"

float Body::volume() const
{
   ...
}
...
```

Body.cc: enthält die Implementation der Body-Klasse

```
#include <iostream>
#include "Body.hh"

int main( int argc, char** argv)
{
   Body a;
   std::cout << a.volume() << std::endl;
   return 0;
}
```

test_body.cc: Hauptprogramm, das die Body-Klasse benutzt

Source Code Organisation



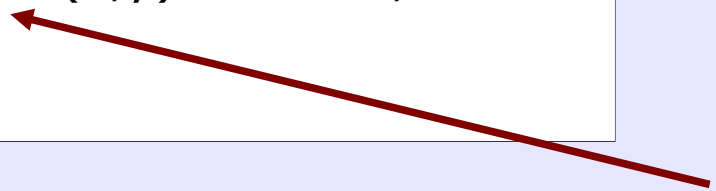
CPP: Präprozessor (wird normalerweise nicht explizit aufgerufen)
CXX: Compiler (z.B. '`g++ -c Body.cc -o Body.o`')
LINK: Linker (z.B. '`g++ Body.o test_body.o -o test_body`')

Mehrfaches Einbinden von Headerfiles verhindern

```
// Mehrfaches Einbinden des Headerfiles "GreyImage.hh" verhindern

#ifndef GREY_IMAGE_HH
#define GREY_IMAGE_HH

class ....


#endif
```

'inline' bei Funktionen

```
inline int min( int a, int b)
{
  if( a < b) return a;
  return b;
}

int main( int argc, char** argv)
{
  int x = 42;
  int y = 137;

  std::cout << min(x,y) << endl;
  return 0;
}
```



Das Schlüsselwort 'inline' vor der Funktion min() sorgt dafür, das hier der kein Funktionsaufruf erzeugt wird, sondern der Code aus min() direkt eingefügt wird.

'inline' bei Klassen

Methode 1: Direkte Implementation im Headerfile

MyComplex.hh:

```
class MyComplex
{
   ...
   float real() const
      {
         return  _real;
      }
   ...
};
```

Methode 2: Implementation in separater ".icc" – Datei

MyComplex.hh:

```
class MyComplex
{
   ...
   float real() const;
   ...
};

#include "MyComplex.icc"
```

MyComplex.icc:

```
inline float MyComplex::real() const
{
   return _real;
}
```

Überladen von Operatoren

- In C++ kann praktisch jeder Operator überladen werden (z.B. +, -, *, /, =, ==, etc...)
- Dies kann als member-Funktionen der jeweiligen Klasse formuliert werden (für binäre Operatoren kann man es alternativ als externe Funktion formulieren)

```
x = y;          // equivalent zu x.operator=( y);
i += 1;         // i.operator+=( 1);
a = b + c;      // a = b.operator+( c);   oder   a = operator+(b, c);
if( q == r )    // q.operator==( r)       oder   operator==( q, r)
cout << x;      // cout.operator<<(x);    oder   operator<<( cout, x);
```

Überladen von Operatoren

```
class MyComplex
{
public:
   MyComplex( float real, float imag);
   void operator+=( const MyComplex& v);
   MyComplex operator+( const MyComplex& v) const;

private:
   float  _real;
   float  _imag;
};
```

Überladen von Operatoren

```
void MyComplex::operator+=( const MyComplex& v)
{
   _real += v._real;
   _imag += v._imag;
}

MyComplex MyComplex::operator+( const MyComplex& v) const
{
   MyComplex retval( _real, _imag);
   retval += v;
   return retval;
}
```

Standard-Bibliothek: Streams

- Allgemeines Konzept für Ein-und Ausgabe (standard in/out, Dateien, serielle Schnittstelle etc.)
- Realisierung durch Überladen des '<<' und des '>>' operators ermöglicht mehrere Ausgaben in einem Befehl

```
int alter = 42;
std::cout << "Du bist " << alter << " Jahre alt\n";
```

- Einfache Definition der Ein-/Ausgabe für eigene Klassen

File-Streams

```
#include <iostream>
#include <fstream>
#include <string>

int main( int argc, char** argv)
{
  // Öffnen einer Datei zum Lesen
  std::ifstream textFile( "data.txt");

  // Einen String und einen Integer einlesen
  // (getrennt durch "white spaces")
  std::string word;
  int number;
  textFile >> word >> number;

  // und auf den Bildschirm schreiben
  std::cout << word << " " << number << std::endl;
  return 0;
}
```

Hier wird der Destructor von 'textFile' aufgerufen und die Datei dabei geschlossen

Kodierungsrichtlinien

- Die Programmiersprache soll (genauso wie die Deutsche Sprache) dazu genutzt werden, die Funktionalität knapp und klar auszudrücken
- Wenn Optimierungen nötig sind, die den Code schlecht lesbar machen, muß ein entsprechender Kommentar angebracht werden
- Kommentare sollen beschreiben, warum etwas gemacht wird. Nicht den C++ Code im Klartext wiederholen.
- Variablennamen und Kommentare immer Englisch

Literatur

- Bjarne Stroustrup: The C++ Programming Language
  - Die C++ Bibel
  - Als Tutorial geschrieben, leider kaum als Nachschlagwerk geeignet
- Bruce Eckel: Thinking in C++, 2nd edition, Volume 1 und Volume 2
  - viele ausführliche Beispiele und gut verständlich
  - online frei verfügbar
- Dinkum C++ Library Reference Manual.
  - Nachschlagwerk für die Standard Template Library
  - online verfügbar (ab und zu erscheint eine "Werbeseite")
- www.cppreference.com

Editoren: Vim vs.Emacs

- Syntax-Highlighting
- Einrückungen
- Abbreviations
- Dynamic Completion
- Compilieren
- TAGS

Übungsaufgabe
RGB-Pixel-Klasse

- Abgabe bis Montag Abend per Mail an
  - **Gruppe Sound-Memory:** fehr@informatik...
  - **Gruppe Computer Grafik:** sebastian.schulz@pluto.uni-freiburg.de
- http://lmb.informatik.uni-freiburg.de/lectures/praktika/SWPraktikum05SS/