

Besonderheiten von C++

Prof. Dr. Peter Jüttner

Überblick

- **C++ verfügt über spezifische Features zur Ein-/Ausgabe (wie die meisten Programmiersprachen)**
- **C++ hat Sprachelemente, die über den „normalen“ Umfang der Objektorientierten Programmierung hinausgehen bzw. diesen z.T. auch widersprechen**

Ein-/Ausgabe

- **in C werden Daten am Bildschirm und Dateien mittels Bibliotheksfunktionen (printf(...), scanf(...), getc(...), read(...), write(...)) eingelesen bzw. ausgegeben-**
- **Diese Bibliotheksfunktionen aus der Bibliothek stdio.h sind auch in der Regel auch in C++-Compilern verfügbar und können auch dort verwendet werden.**
- **In C++ wird Ein-/Ausgabe über sog. Streams durchgeführt**
- **Streams werden über die Iostream-Bibliothek (iostream) verwaltet**

Ein-/Ausgabe

- **Die lostream-Bibliothek besteht aus einer komplexen Klassenhierarchie mit Mehrfachvererbung und virtuellen Basisklassen.**
- **Vorteile von lostream gegenüber stdio:**
 - **Typprüfung bei der Ausgabe durch den Compiler**
 - **Eigene Ausgabefunktionen für eigene Datentypen oder Klassen können definiert werden.**

Ein-/Ausgabe

- **Standardausgabe**
 - **erfolgt über den vordefinierten cout-Stream mittels des <<-Operators (der <<-Operator hat hier nichts mit dem Links-Shift-Operator aus C zu tun, der <<-Operator ist hier überladen)**
 - **einfachste Form der Ausgabe:**

```
#include <iostream>
int main(void)
{ cout<<"Hello World\n";
  return(0);
}
```

Ein-/Ausgabe

- **Standardausgabe**
 - **Ausgabe von Zahlen, Buchstaben und Ausdrücken:
Das Programm**

```
int main(void)
{
    cout<<"Das Doppelte der Zahl ";
    cout<<5;
    cout<<" ist ";
    cout<<2*5;
    cout<<"\n";
    return(0);
}
```

erzeugt die Ausgabe: Das Doppelte der Zahl 5 ist 10

Ein-/Ausgabe

- **Standardausgabe**
 - **Die Ausgabe kann auch aneinander gehängt werden:**

```
int main(void)
{
    cout<<"Das Doppelte der Zahl " <<5 <<" ist " <<2*5 <<"\n";
    return(0);
}
```

Ein-/Ausgabe

- **Standardausgabe**
 - **Ähnlich wie in C kann die Ein-/Ausgabe formatiert erfolgen.**
 - **Die Formatierung erfolgt über so genannte Manipulatoren**
 - **Die Manipulatoren werden vor die auszugebende Information gesetzt, z.B. `cout << hex << 100` für die Ausgabe der Dezimalzahl 100 in hexadezimaler Form (Ausgabe 64)**

Ein-/Ausgabe

- **Standardausgabe mit Manipulatoren:**
 - **flush** - Ausgabedaten werden sofort ausgegeben, der Zwischenpuffer wird geleert
 - **endl** – Zeilenende ('\n') ausgeben mit anschließendem flush
 - **ends** – Stringende ('\0,') ausgeben mit anschließendem flush
 - **oct** – Integerzahlen oktale darstellen
 - **dec** – Integerzahlen dezimal darstellen
 - **hex** – Integerzahlen hexadezimal darstellen
 - **setbase(int b)** permanent Integerwert zur Basis b ausgeben. Gültige Wert für b sind: 8, 10 und 16

Ein-/Ausgabe

- **Standardausgabe mit Manipulatoren:**
 - **setw(int n)** legt die minimale Breite des Ausgabefeldes fest (nur für die nächste Ausgabe)
 - **setfill(int c)** spezifiziert das Füllzeichen zum Auffüllen des Ausgabefeldes (nur für die nächste Ausgabe)
 - **setprecision(int n)** definiert die Anzahl der gültigen Ziffern bei der Ausgabe von Gleitkommawerten
 - **die parametrisierten Manipulatoren erfordern die Einbindung des Headerfiles iomanip**

Ein-/Ausgabe

- **Standardausgabe mit Manipulatoren, Beispielprogramm:**

```
#include <iostream>
```

```
#include <iomanip>
```

```
using namespace std;
```

```
int main(void)
```

```
{
```

```
    cout << oct << 100 << endl << dec << 100 << endl << hex << 100 << endl;
```

```
    cout << setbase(8) << 100 << endl;
```

```
    cout << setw(20) << setfill('*') << "Hallo" << endl;
```

```
    double d = 12345.6789012345;
```

```
    cout << setprecision(5) << d << endl;
```

```
    cout << setprecision(10) << d << endl; // 10 Stellen Ausgabe
```

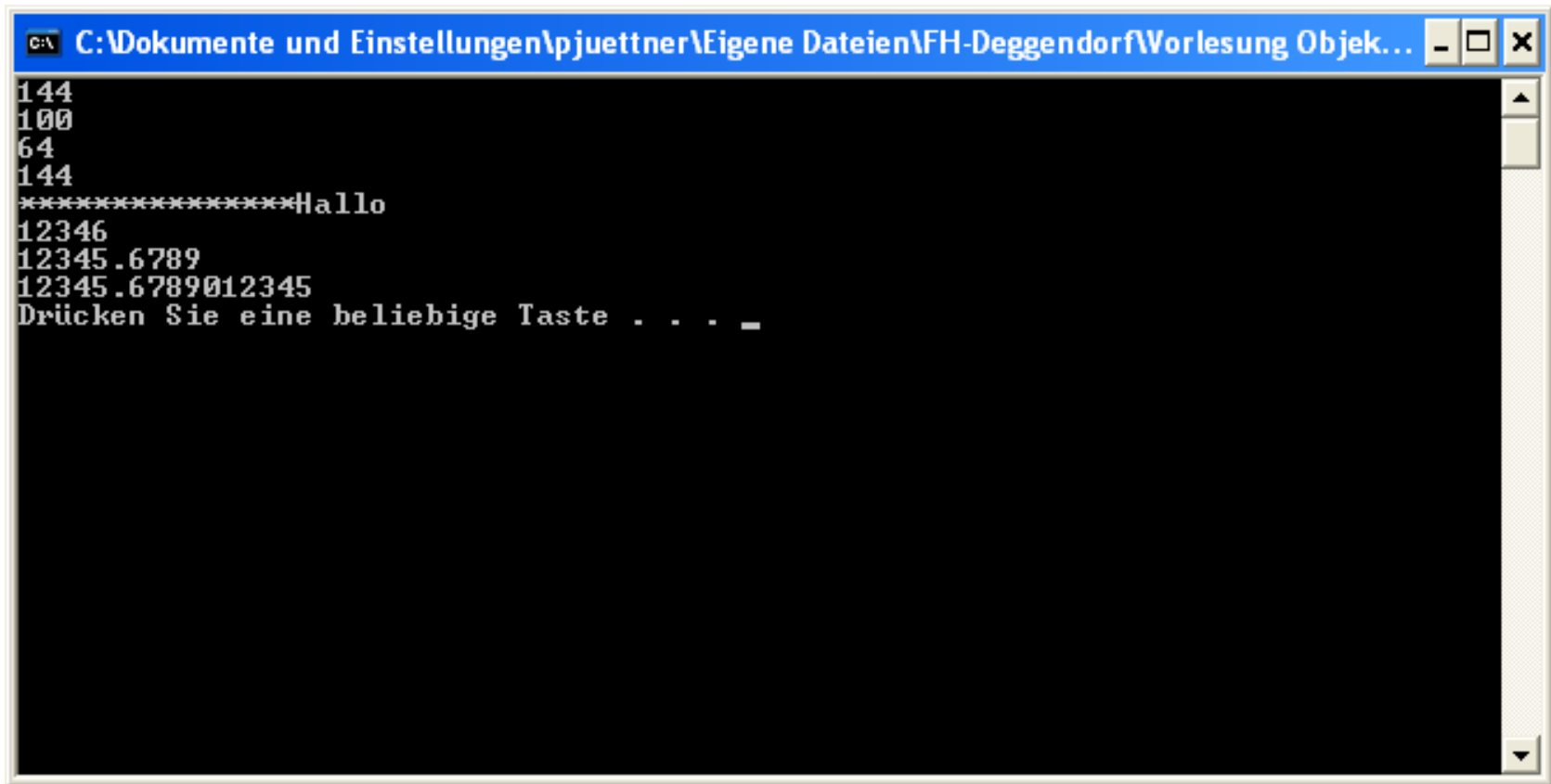
```
    cout << setprecision(10) << fixed << d << endl; // 10 Nachkommastellen Ausgabe
```

```
    return(0);
```

```
}
```

Ein-/Ausgabe

- Standardausgabe mit Manipulatoren, Ergebnis
Beispielprogramm:



```
C:\Dokumente und Einstellungen\pjüttner\Eigene Dateien\FH-Deggendorf\Vorlesung Objek...  
144  
100  
64  
144  
*****Hallo  
12346  
12345.6789  
12345.6789012345  
Drücken Sie eine beliebige Taste . . . _
```

Ein-/Ausgabe

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Ein-/Ausgabe

- **Standardeingabe**
 - Die Standardeingabe erfolgt über den Stream `cin`
 - Zur Standardeingabe ist der `>>`-Operator überladen
 - Die Eingabe erfolgt im Format `cin >> Variable` (z.B. `int i; cin >> i;`)
 - Eingaben können analog zur Ausgabe zusammengehängt werden im Format `cin >> v1 >> v2 >> v3`. Die Ausführung erfolgt von links nach rechts, die Variablen können unterschiedliche Typen haben.
 - Mittels der Manipulatoren `oct`, `dec`, `hex` können die Eingabeformate umgeschaltet werden.

Ein-/Ausgabe

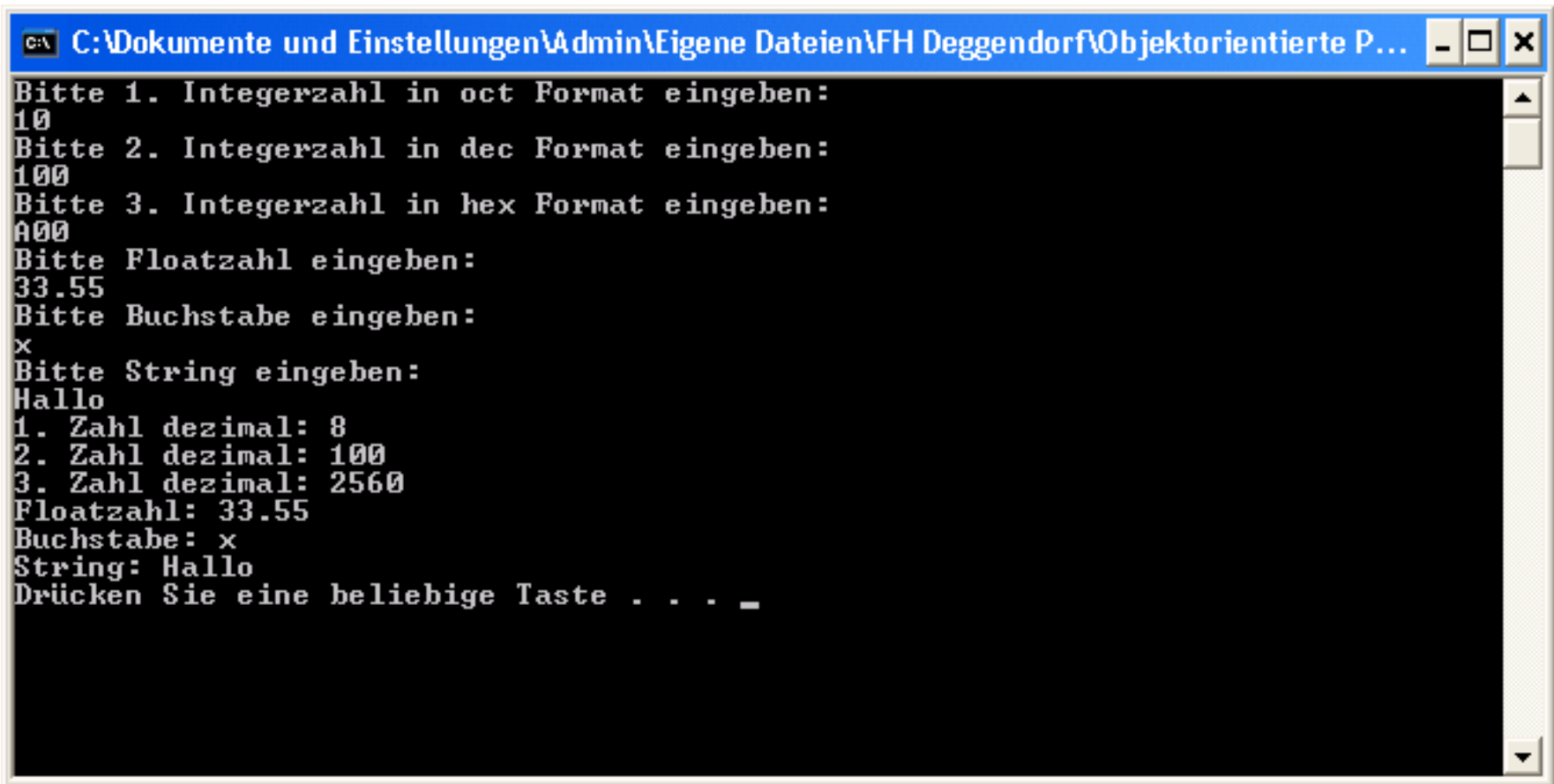
- **Standardeingabe mit Manipulatoren, Beispielprogramm:**

```
#include <iostream>
using namespace std;

int main(void)
{ int i1,i2,i3; float f;  char c;  char s[50];
  cout << "Bitte 1. Integerzahl in oct Format eingeben:" << endl; cin >> oct >> i1;
  cout << "Bitte 2. Integerzahl in dec Format eingeben:" << endl; cin >> dec >> i2;
  cout << "Bitte 3. Integerzahl in hex Format eingeben:" << endl; cin >> hex >> i3;
  cout << "Bitte Floatzahl eingeben:" << endl; cin >> f;
  cout << "Bitte Buchstabe eingeben:" << endl; cin >> c;
  cout << "Bitte String eingeben:" << endl; cin >> s;
  cout << "erste Zahl dezimal: " << dec << i1 << endl;
  cout << "zweite Zahl dezimal: " << dec << i2 << endl;
  cout << "dritte Zahl dezimal: " << dec << i3 << endl;
  cout << "Floatzahl: " << f << endl << "Buchstabe: " << c << endl;
  cout << "String: " << s << endl;
  return(0);
}
```

Ein-/Ausgabe

- **Standardeingabe mit Manipulatoren, Ergebnis**
Beispielprogramm:



```
C:\Dokumente und Einstellungen\Admin\Eigene Dateien\FH Deggendorf\Objektorientierte P...
Bitte 1. Integerzahl in oct Format eingeben:
10
Bitte 2. Integerzahl in dec Format eingeben:
100
Bitte 3. Integerzahl in hex Format eingeben:
A00
Bitte Floatzahl eingeben:
33.55
Bitte Buchstabe eingeben:
x
Bitte String eingeben:
Hallo
1. Zahl dezimal: 8
2. Zahl dezimal: 100
3. Zahl dezimal: 2560
Floatzahl: 33.55
Buchstabe: x
String: Hallo
Drücken Sie eine beliebige Taste . . . _
```


Motivation

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Ein-/Ausgabe

- **Ein-/Ausgabe in Dateien**
 - **Dateiverarbeitung (Ein-/Ausgabe) erfolgt durch Zuordnung einer physikalischen Datei zu einem Stream (analog zu fopen(...) in C).**
 - **Zuvor muss das Headerfile fstream inkludiert werden**
 - **Eingabedateien werden mit ifstream objekt ("Dateiname") im Programm definiert, Ausgabedateien mit ofstream objekt ("Dateiname").**
 - **Eingabedateien können mit der Funktion eof() auf das Dateiende abgefragt werden (Aufruf datei.eof())**

Ein-/Ausgabe

- **Einfache Dateiein-/ausgabe mit Manipulatoren, Beispielprogramm:**

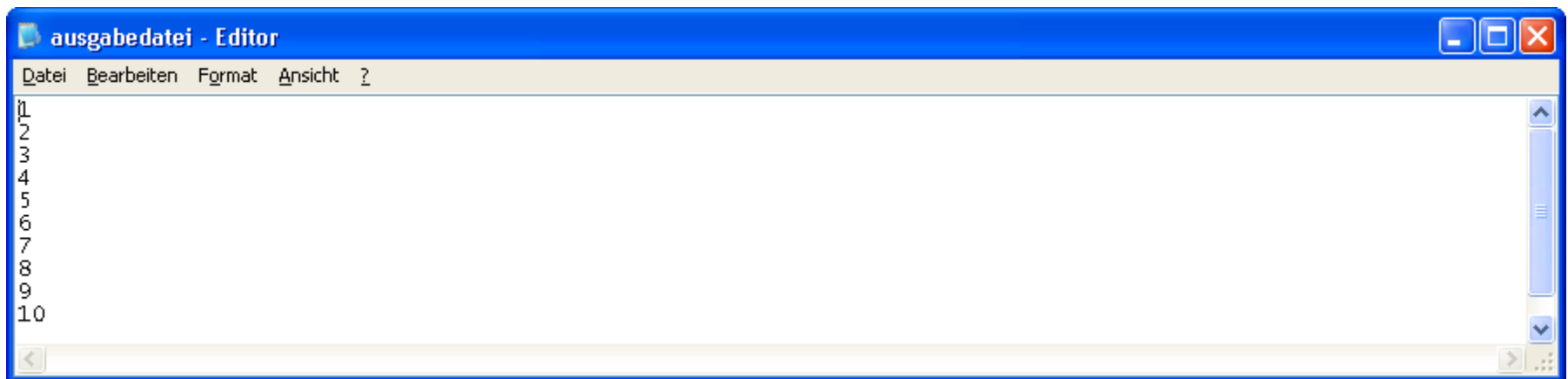
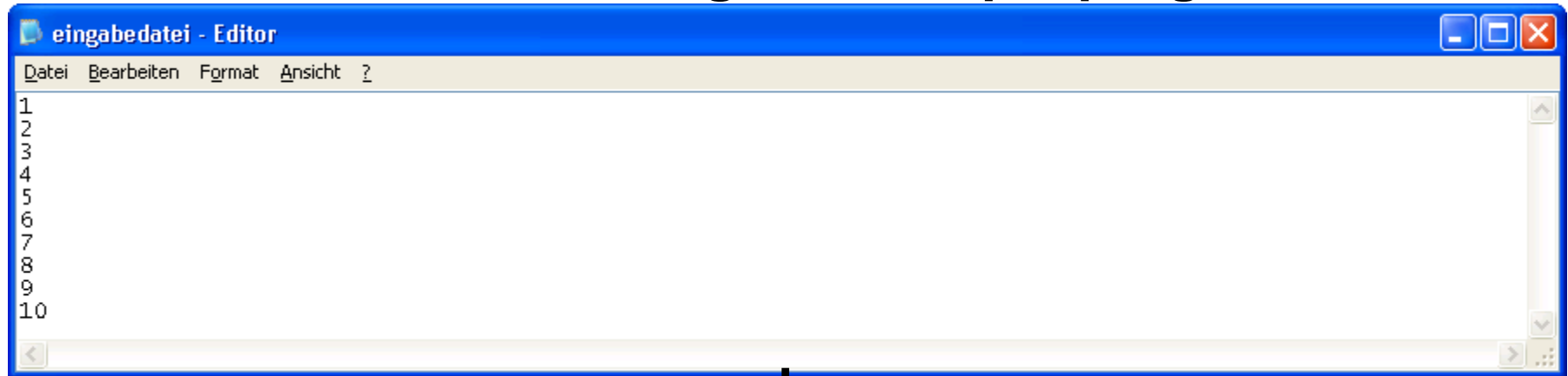
```
#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;

int main(void)
{ int i;
  ofstream ausgabe("ausgabedatei.txt"); // Datei an Stream zuordnen
  ifstream eingabe("eingabedatei.txt"); // Datei an Stream zuordnen
  if (!eingabe || !ausgabe) exit(1); // Programm beenden, falls Fehler beim Dateioffnen
  while (!eingabe.eof()) // So lange Eingabedatei nicht zu Ende
  { eingabe >> i; // Lese aus Eingabedatei
    ausgabe << i << endl; // Schreibe in Ausgabedatei
  };
  eingabe.close(); ausgabe.close(); // SchlieÙe Dateien
  return(0);
}
```

Ein-/Ausgabe

- **Einfache Dateiein-/ausgabe, Beispielprogramm:**



Ein-/Ausgabe

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Friend Funktionen und Friend Klassen

- **Vom Prinzip der Objektorientierung sind gekapselte Elemente nur der eigenen Klasse und Unterklassen zugänglich (in C++ weitere Einschränkung durch „private“)**
- **C++ durchbricht dieses Prinzip und ermöglicht anderen Funktionen (sog. Friend Funktionen“) und Klassen (sog. Friend Klassen“) den unbeschränkten Zugriff auf gekapselte Elemente**

Friend Funktionen und Friend Klassen

- **Das Prinzip der Datenkapselung bleibt insoweit gewahrt, dass eine Klasse selbst definiert, wer ihre „Freunde“ sind.**
- **Friend Funktionen und Friend Klassen müssen also innerhalb der Klasse, deren „Freunde“ sie sein sollen, definiert werden.**
- **Eine Friend Definition von „außen“ (d.h. eine Funktion erklärt sich zum „Freund“ einer Klasse) ist nicht möglich. Jede Klasse entscheidet selbst, ob sie Freunde haben will oder nicht.**

Friend Funktionen und Friend Klassen

- **Beispiel für eine Friend Funktionsdeklaration**

```
class stringklasse
```

```
{ char *p;
```

```
  unsigned short laenge;
```

```
  public:
```

```
    stringklasse (const char *s) // Konstruktor
```

```
    { ... };
```

```
    ~stringklasse() // Destruktor
```

```
    { delete p; };
```

- Deklaration einer Friend Funktion
- Zugriff auf gekapselte Daten durch Friend Funktion

```
friend void gebe_string_aus(const stringklasse str); // Friend Deklaration
```

```
void gebe_string_aus(const stringklasse str) // Friend Deklaration
```

```
{ cout << "String enthaelt folgenden Text: " <<str.p << endl;
```

```
  return;
```

```
};
```


Friend Funktionen und Friend Klassen

- Eine Friend Funktion ist eine normale C-Funktion, der ggf. ein Objekt per Parameter übergeben werden muss (im Gegensatz zu Methoden, die an einem Objekt aufgerufen werden).
- Eine Friend Funktion gehört nicht zu einer Klasse
- Friend Funktionen können dazu dienen, bestehende C-Programme um objektorientierte C++-Anteile zu erweitern.

Friend Funktionen und Friend Klassen

- **Bei Friend Klassen ist eine ganze Klasse Freund, d.h. alle Methoden der Friend Klasse haben Zugriff auf gekapselte Elemente.**
- **Eine Klasse K_1 kann eine andere Klasse K_2 als Friend deklarieren. In diesem Fall haben alle Methoden von K_2 Zugriff auf die gekapselten Elemente von K_1 .**

Friend Funktionen und Friend Klassen

- Beispiel:**

```
class stringklasse
{ ...
  public:
    stringklasse (const char *s)
    { ... };

    ~stringklasse()
    { ... };
    ...
    friend class test; // Friend Class Deklaration
};
```

Deklaration einer Friend Klasse

Zugriff auf gekapselte Daten in der Friend Klasse

```
class test
{ public:
  void ersetze_string(stringklasse *s, char* zeichenkette)
  { delete s->p;
    s->p = new char [strlen(zeichenkette)+1];
    strcpy(s->p, zeichenkette);
    s->laenge = strlen(zeichenkette);
  };
};
```

Friend Funktionen und Friend Klassen

- **Die Friend-Beziehung zwischen Klassen ist asymmetrisch und nicht transitiv.**
 - **Ist Klasse A Friend der Klasse B (A hat Zugriff auf gekapselte Daten von B), dann ist nicht automatisch B Friend von A.**
 - **Ist Klasse A Friend der Klasse B und B Friend der Klasse C, so ist nicht automatisch A Friend von C**
 - **Die Friend Eigenschaft wird vererbt, d.h. ist Klasse A Friend von Klasse B, so auch alle Unterklassen von A. A ist aber nicht Friend der Unterklassen von B.**

Friend Funktionen und Friend Klassen

- **Anwendung von Friend Klassen**
 - **Testzwecke: eine Testklasse ermöglicht das Testen und den Zugriff auf gekapselter Elemente einer Klasse.**
 - **Ein-/Ausgabeklassen, die Daten in gekapselte Elemente schreiben oder gekapselte Elemente auf ein bestimmtes Medium ausgeben.**

Friend Funktionen und Friend Klassen

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Überladene Operatoren

- **Durch die Definition einer Klasse wird ein neuer Datentyp definiert.**
- **Manchmal wäre es „wünschenswert“, existierende Standard-Operatoren auch auf neue Datentypen anzuwenden (anstatt Methoden für diesen Zweck zu definieren)**

Beispiel: neuer Zahlentyp, Anwendung von +, *, -, / auch auf den neuen Typ

Überladene Operatoren

- ➔ C++ bietet die Möglichkeit, existierende Operatoren für eine Klasse neu zu definieren.
- ➔ Überladen von Operatoren
- ➔ Definition der Semantik für existierende Operatoren angewandt auf neue Klassen

Überladene Operatoren

Beispiel: Bruchrechnen ohne Rundungsfehler

Motivation: Rechnen mit Gleitkommazahlen (double, float) führt schnell zu Rundungsfehlern:

```
main()
{
    double d = 0.0;
    for (d=0.0; d<10.0; d=d+0.1)
    { if (d == 1.0) break;
    };
    printf("Ergebnis: %.4f\n",d);

    system("PAUSE");
};
```

Überladene Operatoren

Definition einer Klasse Rationaler Zahlen

```
class Rational
{ long Zaehler; // enthält auch das Vorzeichen
  long Nenner; // immer > 0

public:
  void kuerzen()
  { long Zabs = Zaehler>0 ? Zaehler : - Zaehler;
    long Teiler = ggt(Zabs, Nenner);
    Zaehler = Zaehler / Teiler;
    Nenner = Nenner / Teiler;
  };
};
```

Überladene Operatoren

Definition einer Klasse Rationaler Zahlen

```
class Rational
{ ...
    Rational() { Zaehler = 0; Nenner = 1 };

    Rational(long z, long n = 1) // Konstruktor mit 1 oder 2 Parametern
    { if (n == 0) cout << "Fehler";
      else if (n < 0) { Zaehler = -z; Nenner = -n};
      else { Zaehler = z; Nenner = n);
      kuerzen();
    };
};
```

Überladene Operatoren

Definition einer Klasse Rationaler Zahlen

Ziel:

Folgendes Programmteil mit Operatoren soll funktionieren:

```
Rational r1(4,5), r2(-5,6), e;  
e = r1 + r2; // Addition zweiter rationaler Zahler durch + Operator  
cout << e; // Ausgabe einer rationalen Zahl durch << Operator  
c -= r1; // Subtrahieren einer rationalen Zahl durch -= Operator  
r2 = r1 + 5; // Addieren einer int Zahl (Typumwandlung) durch + Operator
```

Überladene Operatoren

Definition einer Klasse Rationaler Zahlen

Operatoren definieren:

- **als „normale“ Funktion (außerhalb einer Klasse)**
➔ als Friend Funktion in der Klasse angeben

```
class Rational
{ long Zaehler; // enthält auch das Vorzeichen
  long Nenner; // immer > 0
  ...
  friend Rational operator+ (Rational, Rational);
  friend Rational operator+ (Rational, long);
  friend ostream& operator<<(ostream&, Rational); // Ausgabe einer rat. Zahl
  ...
};
```

Überladene Operatoren

Definition einer Klasse Rationaler Zahlen

Operatoren definieren:

- als „normale“ Funktion (außerhalb einer Klasse)
→ als Friend Funktion in der Klasse angeben

```
ostream& operator<< (ostream& o, Rational r) // Ausgabe einer rat. zahl
{ o << r.Zaehler;
  if (r.Nenner != 1) o << "/" << r.Nenner;
  return o;
};
```

Überladene Operatoren

Definition einer Klasse Rationaler Zahlen

Operatoren definieren:

- **als Memberfunktion einer Klasse**
➔ **erster Operand ist immer das Aufrufobjekt (*this)**

```
class Rational
{ long Zaehler; // enthält auch das Vorzeichen
  long Nenner; // immer > 0
  ...
  Rational operator+ (Rational r) // a+b
  { Rational erg (Zaehler * r.Nenner + r.Zaehler*Nenner,
                  Nenner * r.Nenner);
    erg.kuerzen();
    return erg;
  };
```

Überladene Operatoren

Regeln für überladene Operatoren

- **Nur Standardoperatoren können überladen werden (neue Operatoren können nicht definiert werden)**
- **Nicht überladbar sind: ::, ?:, sizeof(), .***
- **Die Vorrangregeln bleiben erhalten („Punkt vor Strich“)**
- **Die Anzahl der Operanden bleibt erhalten**
- **Mindestens ein Operand muss ein selbst definierter Typ oder eine Klasse sein (für Standardtypen können Operatoren nicht überladen werden)**

Überladene Operatoren

Regeln für überladene Operatoren

- **`+=`, `-=`, `*=`, usw. müssen explizit überladen werden (das Überladen von `+`, `-`, `*`, usw. genügt nicht)**
- **Der Compiler erkennt keinen Zusammenhang zwischen „verwandten“ Operatoren, z.B. `+` und `+=` (die Verantwortung liegt beim Programmierer)**
- **Falls in Klassen kein überladener Zuweisungsoperator definiert wird, wird elementweise kopiert**

Überladene Operatoren

Regeln für überladene Operatoren

- die Operatoren `=` `()` `[]` und `->` sind nur als Memberfunktionen überladbar
- überladene Operatoren, die das Aufrufobjekt verändern, sollten als Memberfunktionen definiert werden (Programmierstil)
- Operatoren, bei denen der linke Operand einen anderen Typ als die eigene Klasse haben kann, können nicht als Memberfunktionen implementiert werden (z.B. Ausgabe mittels `<<`)

Überladene Operatoren

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Modularisierung

Programme

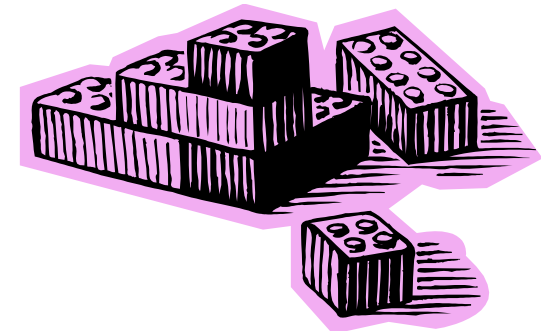
- sind meist (relativ) groß ($> 1000-100000000$ Zeilen)
- werden sehr oft in (örtlich und/oder organisatorisch) verteilten Entwicklungsteams erstellt
- nutzen sehr oft vorhandene Bibliotheken
- werden oft sequentiell oder parallel in verschiedenen Versionen entwickelt



Modularisierung

Folgerung

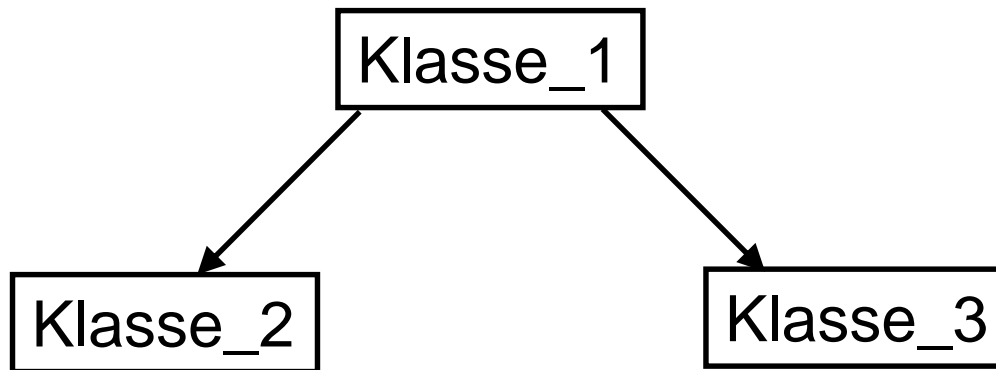
- Eine **Modularisierung** ist notwendig
- Zerlegung in unabhängig voneinander zu erstellende SW-Teile (**Klassen**)
- **Klassen** sind getrennt voneinander übersetzbar



Modularisierung

Folgerung

- Grundlage der Zerlegung ist eine **vor (!)** der Codierung definierte **SW Architektur**



- Die Gesamtfunktion des Programms wird durch das „Zusammenspiel“ der einzelnen Klassen realisiert

Modularisierung

Prinzipien

- Elemente einer Klasse
 - Attribute (meist gekapselt)
 - Methoden (meist öffentlich)
- öffentliche Elemente werden in anderen Klassen oder im Hauptprogramm verwendet
- öffentliche Elemente werden (wie in C) über ein Headerfile bekannt gemacht

Modularisierung

```
class class1
{
    int attribut;
public:
    int methode1();
    int methode2(int i);
    class1();
};

int class1::methode1()
{ return attribut * 2; };

int class1::methode2(int i)
{ return attribut + i; };

class1::class1()
{ attribut = 5; };
```

class1.cpp



„Extrahieren der
öffentlichen Teils in
eine eigene
Datei (Header)“

```
class class1
{
public:
    int methode1();
    int methode2(int i);
    class1();
};
```

class1.h

Modularisierung

```
class class1
{
    public:
        int methode1();
        int methode2(int i);
        class1();
};
```

class1.h



Klasse class1 wird durch inkludieren der Headerdatei im Hauptprogramm (main) bekannt gemacht. Danach kann die Klasse im Hauptprogramm verwendet werden

```
#include "class1.h"
using namespace std;

int main()
{
    class1 my_object;

    cout <<
        my_object.methode1() <<
        endl;

    cout <<
        my_object.methode2(7)
        << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

main.cpp

Modularisierung

Vorgehensweise

1. Für Klassen, die außerhalb der eigenen Datei verwendet werden sollen, wird ein Header-File erzeugt. Dieses enthält nur(!) die öffentlichen (public) Teile der Klasse, wobei Methoden nur(!) als Schnittstelle angegeben werden.
2. Dort, wo die Klasse verwendet werden soll, wird das Header-File inkludiert. Damit ist der öffentliche Teil der Klasse bekannt und kann verwendet werden.
3. Im .cpp-File der Klasse werden die Methoden implementiert

Modularisierung

Folgerungen

1. Die Kapselung der Attribute ist sichergestellt
2. Die Implementierung der Methoden bleibt verborgen
3. Änderungen an den Algorithmen der Methoden erfordern keine Änderung in anderen Dateien

Modularisierung

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Sonstiges

Klassenvariable (static Datenelemente)

- **Attribute, die nicht objektspezifisch, sondern klassenspezifisch sind**
 - ➔ **das Attribut existiert nur einmal im Programm (ähnlich einer globalen Variablen)**
 - ➔ **gehört zur Klasse**
 - ➔ **Sichtbarkeit wie bei „normalen Attributen“**
 - ➔ **alle Objekte der Klasse können auf das Attribut zugreifen**

Sonstiges

Klassenvariable (static Datenelemente)

- **Attribute, die nicht objektspezifisch, sondern klassenspezifisch sind**
 - ➔ **„Ersatz“ für globale Variable**
 - ➔ **existieren auch ohne, dass ein Objekt ihrer Klasse instanziiert wird**
 - ➔ **Anwendung, z.B.**
 - ➔ **Zählen der Objekte einer Klasse zur Laufzeit**
 - ➔ **Kommunikation zwischen Objekten einer Klasse**

Sonstiges

Klassenvariable (static Datenelemente)

- Deklaration durch Schlüsselwort *static*, zusätzlich durch globale Definition und Initialisierung

- **Beispiel:**

```
class Datum
{ short tag, monat, jahr;
  short wochentag;
  static char *namen_wochentage[];
  ...
};
```

```
char* Datum::namen_wochentage[] = { "Sonntag", "Montag", "Dienstag",
  "Mittwoch", "Donnerstag", "Freitag", "Samstag" };
```

Sonstiges

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Sonstiges

static Memberfunktionen

- **Klassenvariable existieren auch wenn keine Objekte ihrer Klasse instanziiert werden.**
 - **Eine Methode (Memberfunktion) einer Klasse, die nur mit Klassenvariablen arbeitet, benötigt ebenfalls kein Objekt.**
- ➔ Memberfunktionen, die nur auf Klassenvariable zugreifen, können ebenfalls per static deklariert werden.**

Sonstiges

static Memberfunktionen

- **Beispiel:**

```
class Datum
{ short tag, monat, jahr;
  short wochentag;
  static char *namen_wochentage[];
  ...
  static void Sprache(char *s)
  { // initialisiert wochentage in einer bestimmten Sprache s
    ...
  };
};
```

```
char* Datum::namen_wochentage[] = { "Sonntag", "Montag", "Dienstag",
  "Mittwoch", "Donnerstag", "Freitag", "Samstag" };
```

```
Datum::Sprache("Englisch"); // Aufruf einer static Memberfunktion
```

Sonstiges

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Sonstiges

Referenzparameter

- Rückgabe von mehr als einem Funktionsergebnis in C entweder über globale Variable oder Pointer
- In C++ gibt es die Möglichkeit Referenzparameter zu vereinbaren (durch Anfügen eines &-Zeichens an den Datentyp, z.B. `int& p`)
- Referenzparameter werden implizit als Adresse übergeben
- Formale Referenzparameter werden beim Aufruf durch Variable ersetzt, die Adressbildung und Dereferenzierung „übernimmt“ der Compiler, explizite Adressbildung / Dereferenzierung unnötig

Sonstiges

Referenzparameter

- **Beispiel:**

```
// in C
```

```
C_tausche (int *p1, int *p2)
```

```
{ int c;
```

```
  c = *p1;
```

```
  *p1 = *p2;
```

```
  *p2 = c;
```

```
};
```

```
int main()
```

```
{ int i1 = 5;
```

```
  int i2 = 10;
```

```
  C_tausche(&i1, &i2);
```

```
};
```

Sonstiges

Referenzparameter

- **Beispiel:**

```
// in C++
```

```
Cpp_tausche (int& i1, int& i2)
```

```
{ int c;
```

```
  c = i1;
```

```
  i1 = i2;
```

```
  i2 = c;
```

```
};
```

```
int main()
```

```
{ int i1 = 5;
```

```
  int i2 = 10;
```

```
  Cpp_tausche(i1, i2);
```

```
};
```

Sonstiges

Referenzparameter

- **Achtung:** Für Referenzparameter wird der Parametertyp streng(!) geprüft → keine Typkonvertierung und Fehlermeldung oder zumindest Warnung durch Compiler
- **Beispiel:**

```
short s1 = 5;  
short s2 = 10;  
Cpp_tausche(s1, s2);
```


→ Fehlermeldung beim Dev-C++-Compiler
- **Lösung ?**

Sonstiges

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Sonstiges

Qualifizierter Zugriff

- **Klassenname::Memberfunktion(...)**

oder

Klassenname::Klassenattribut

ermöglicht den qualifizierten Zugriff auf ein Element einer Klasse

Sonstiges

Qualifizierter Zugriff

- **Klassenname::Memberfunktion(...)**
 - **ermöglicht das Definieren einer Methode außerhalb des Klassenrumpfs (unter Angabe der Signatur)**

Beispiel:

```
class c
{ int m1 (int);
};

...
int c::m1 (int p)
{ return p+5;
};
```

Sonstiges

Qualifizierter Zugriff

- **Klassenname::Memberfunktion(...)**
 - **ermöglicht das Aufrufen einer bestimmten Methode innerhalb einer Klassenhierarchie, z.B. Verhindern der dynamischen Bindens**

Sonstiges

Qualifizierter Zugriff

- **Klassenname::Memberfunktion(...), Beispiel:**

```
class c
{ public: virtual int m1 (int); };
```

```
class uc : public c
{ public: virtual int m1 (int); };
```

```
int c::m1 (int p) { return p+5; };
```

```
int uc::m1 (int p) { return p+7; };
```

```
int main()
{uc* c_p;
  int j1 = c_p->c::m1(5); // Aufruf der Oberklassenmethode
  int j2 = c_p->uc::m1(5); // Aufruf der Unterklassenmethode};
```

Sonstiges

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Sonstiges

Inline (Member-)Funktionen

- **Motivation: C Makros, z.B.**

```
#define TAUSCH(X,Y) { (X)=(X)+(Y); (Y) = (X)-(Y); (X)=(X)-(Y); }
```

- **an der Aufrufstelle expandiert**
- **keinerlei Typüberprüfung der Parameter**
- **Seiteneffekte, z.B. Parameter a++**

Sonstiges

Inline (Member-)Funktionen

- **C++ erlaubt inline-Funktionen**
- **Schlüsselwort inline vor der Funktionsdefinition**
z.B.

```
inline void tausch(int& x, int& y)
{ int hv = x;
  x = y;
  y = hv;
};
```

- **Einsetzen des Funktionsrumpfs anstelle des Aufrufs
(keine Sprung, aber Parameterübergabe wie gewohnt →
keine unerwarteten Seiteneffekte)**

Sonstiges

Inline (Member-)Funktionen

- **Einschränkungen**
 - **keine Rekursion (☹)**
 - **nach einer Änderung müssen alle Module, die inline-Funktion verwenden neu übersetzt werden (bei nicht-inline genügt neues Linken)**
 - **Inline Funktionen haben keine Funktionsadresse, können daher nicht als Parameter übergeben werden.**

Sonstiges

Inline (Member-)Funktionen

- **Inline und Member-Funktionen**
 - **Member-Funktionen, die im Klassenrumpf definiert sind, werden implizit als Inline Member-Funktionen betrachtet**
 - **Virtuelle (d.h. dynamisch bindbare) Funktionen werden nicht als inline Funktionen compiliert (ggf. Compilerwarnung)**
 - **Definition einer Member-Funktion ohne inline und außerhalb des Klassenrumpfs führt zu normaler Funktion**
 - **inline Definition einer Member-Funktion außerhalb des Klassenrumpfs führt zu inline Funktion**

Sonstiges

Inline (Member-)Funktionen

- **Inline und Member-Funktionen**

```
class c
{ public:
  int m1 (int i) // implizit inline, da im Klassenrumpf definiert
  { return 2*i; };

  int m2 (int); // explizit nicht inline, da außerhalb des Klassenrumpfs definiert
                // ohne inline

  virtual int m3(int k)
  { return 4*k; } // explizit nicht inline da virtual

  int m4(int); // explizit inline, da außerhalb des Klassenrumpfs als inline definiert
};

int c::m2(int j) { return 3*j;};

inline int c::m4(int l) {return 5*l;};
```

Sonstiges

Zum Schluss dieses Abschnitts ...

Noch Fragen ??