Zusammenfassung C++
Streams, Präprozessoranweisungen, Zeiger

Streams

Stream ist abstrakter Datenfluss zwischen Programmen und E/A – Gerät

Aufgabe der SSIOL:
typische, effiziente und flexible Methoden für Umwandlung von ASCII-, Zeichenketten in vordefinierte oder benutzerdefinierte Datentypen vorzunehmen

SSIOL basiert auf Basisklassen ios und streambuf

ios und streambuf sind abstrakte Klassen – sie erzeugen keine Objekte

istream stellt Methoden zur Dateneingabe zur Verfügung (get, read, putback, seekg, tellg)
istream definiert auch >>
istream wird für cin verwendet

Eingabeströme müssen immer mit Pufferobjekt verbunden werden -> verwenden von ifstream und istrstream

ifstream definiert istream-Objekte (werden automatisch mit filbuf-Objekt verbunden)
filbuf-Objekt wird dynamisch angelegt (ist Benutzer verborgen = transparent)

iststream definiert istream-Objekte (sind mit streambuf-Objekt verbunden)
wird istrstream-Objekt erzeugt, muss Zeichenpuffer spezifiziert werden

ostream dient Datenausgabe (put, write, flush, seekp, tellp)
ostream definiert <<
cout (Standardausgabe für Daten) und cerr (Standardfehlerausgabe) werden von ostream abgeleitet

ofstream definiert ostream-Objekte (werden mit filbuf –Objekt verbunden)
filbuf-Objekte werden dynamsich angelegt und sind transparent.

oststream definiert ostream-Objekte (werden mit strstreambuf-Objekt verbunden)
wird oststream-Objekt erzeugt, muss Zeichenpuffer spezifiziert werden

iostream erbt Funktionalität von istream und ostream (für Datenein- und -ausgabe)

fstream (wird automatisch mit filbuf-Objekt verbunden) und strstream (benutzt Zwischenpuffer im Arbeitsspeicher) stehen für kontinuierliche Ein- und Ausgabeströme zur Verfügung.

vordefinierte Datentypen

Formatierte Eingabe:
>> wird überladen (heißt: "lies von")

```
istream & operator >>   (char *);
istream & operator >>   (char &);
          "             (short &);
          "             (unsigned short &);
          "             (int &);              ...
```

Aus Gründen der Widerspruchsfreiheit und Typsicherheit sind Methoden, die char akzeptieren, auch für unsigned char und signed char implementiert.

Operatoren greifen auf istream-Objekt zurück -> bilden von Ausdrücken mit mehreren Operatoren möglich (Abarbeitung von links nach rechts)

Unformatierte Eingabe

```
istream & get (char * ps, int n, char delim = '\n');
istream & get (char & c);
istream & getline (char * ps, int n, char delim = '\n');
istream & read (char * ps, int n);
istream & ignore (int n = 1, int delim = EOF);
int gcount ( ) const;
```

get (mit einem Parameter) liest ein Zeichen ein
get (mit drei Parametern) bzw. getline liest eine Textzeile ein
Textzeile wird durch Whitespace, New Line, Space, Tab, FF begrenzt

read liest bestimmte Zeilenzahl ein.
ignore für Überlesen einer bestimmten Zeichenzahl (bei Erreichen eines besonderen Zeichens wird Überlesen gestoppt)

gcount gibt Anzahl Bytes zurück, die mit letztem Methodenaufruf eingelesen worden sind

Steuerfunktionen

```
istream & putback (char c);
istream & seekg (streampos pos);
streampos tellg ( );
int sync
```

putpack stellt zuletzt gelesenes Zeichen in Eingangspuffer zurück; Operation undefiniert, wenn c nicht letztes gelesenes Zeichen entspricht

tellg ( ) gibt Eingabezeiger in Bytes zurück

Bei Plattendateien wird Byteposition vom Dateianfang gezählt, beim Arbeitsspeicher vom Pufferanfang

seekg ( ) setzt Byteposition pos als neue absolute Position des Eingabezeigers in Datei oder Arbeitsspeicherpuffer

sync ( ) synchronisiert nächste Eingabe mit physischer Eingabe (nicht ausgewerteter Pufferinhalt geht verloren)

ws entfernt führende Whitespacezeichen
Bsp.:
cin >> ws >> c;

Headerdateien

istream.h -> Eingabe cin
iostream.h -> Ausgabe cout, cerr, cin

Formatierte Ausgabe
Operator << wird für folgende Datentypen überladen
....
ostream & operator << (const char * pos);
" << (char c);
" << (short s);
" << (unsigned short us);
" << (int i);
" << (unsigned int ui); .....

wegen Typsicherheit muss unsigned und signed char unterschieden werden

Operatoren geben Referenz auf ostream-Objekt zurück -> bilden von Ausdrücken mit mehreren Operatoren
Bsp:
cout << "a= " << a;

Interpretation:
cout.operator<<("a= ").operator<<(a);

Unformatierte Ausgabe

ostream & put (char c);
ostream & write (const char * ps, int n);

Byte c oder definierte Byteanzahl von einem Puffer ps wird ausgegeben

Steuerfunktionen:
ostream & flush ( );
streampos & tellp ( );
ostream & seekp (streampos pos);

flush leert Ausgangspuffer
tellp gibt Ausgabezeiger in Bytes zurück
seekp setzt Byteposition pos als neue absolute Position des Ausgabezeigers in der Datei oder im Arbeitsspeicherpuffer

Verwendung der Operatoren für formatierte Ein- und Ausgabe (Beispielblatt)

Lösung hat Einschränkungen:
1. es werden nur Teilzeichenketten bis zum Whitespace eingelesen
2. leere Zeichenkette eingeben unmöglich (-> Schleifenabbruch mit Eingabe #)
3. Länge der Zeichenkette wird nicht kontrolliert (wenn Zeichenketten größer als Zielbereich sind, kommt es zu Speicherverletzugen)

Verwendung der unformatierten Ein- und Ausgabe mit Löschen des Eingabe-Puffers (Beispielblatt)

get-Methode liest max. 9 Zeichen bis zum NL ein.
NL und restliche Zeichen bleiben im E-Puffer (-> beim nächsten get wird keine weitere Eingabe angefordert)
sync löscht E-Puffer -> neue Eingabe erforderlich

Benutzung der unformatierten E/A mit Ignorieren des NL im Puffer

NL im Puffer wird mit ignore aus Puffer gelöscht -> mehr Zeichen als in der Arraydeklaration vereinbarte Zeichen können als Restzeichen verarbeitet werden (es fehlt aber immer das letzte Zeichen)

Benutzung der unformatierten E/A mit korrekter Verarbeitung der Restzeichenkette

mit read wird nächstes Zeichen eingelesen (nach Eingabe der Zeichenkette)
kein NL -> es wird in den Puffer gestellt -> korrekte Verarbeitung

Fileverarbeitung

Eingabefiles
Methoden des ifstream
ifstream ( );
ifstream (const, char * name, int mode = ios::in , int prot);
void open (const char * name, int mode = ios::in, int prot);
void close ( );
void attach (int fd);
int filedesc ( ) const;

Konstruktur ifstream ( ) definiert Objekt -> damit kann File geöffnet werden
ifstream (.....) definiert Objekt und öffnet File, mit name für bestimmten Modus und bestimmten Schutzcode

open (.....) öffnet File.
close ( ) schließt File. Files werden am Laufzeitende des Progamms automatisch geschlossen

attach (.....) bindet stream an geöffnetes File (Filedeskriptor des Files muss bekannt sein)
filedesc ( ) gibt Filedeskriptor zurück

Headerfiles

Für Einlesen erforderlich:
ifstream.h
oder
fstream.h

Ausgabefiles:
ofstream stellt folgende Methoden zur Vefügung
ofstream ( );
ofstream (const char * name, int mode = ios::out, int prot);
void open (const char * name, int mode = ios::out, int prot);
void close ( );
void attach (int fd);
int filedesc ( ) const;

ofstream ( ) und ofstream (.....) definieren Objekt und (in Abhängigkeit von Parameterliste) ein Ausgabefile

Klassen ifstream, ofstream und fstream stehen in fstream.h zur Verfügung.
fstream.h enthält auch iostream.h

Zeichenketten in File schreiben

- Objekt mit Namen outfile als Ojekt der Klasse ofstream bilden
- Initialisierung dieses Objekts als Datei mit Namen test.txt
- << in der Klasse ofstream überladen (für unkomplizierte Ausgabe)
- am Ende wird Gültigkeitsbereich des outfile-Objekts verlassen
- Destruktor wird aufgerufen -> schließt File und eliminiert Objekt

Zeichenketten aus File lesen
Objekt der Klasse ifstream definieren
getline ( ) liest aus dem File infile Text Zeile für Zeile in Pufferbereich ein
getline liest bis zum NL
Nach jeder Eingabezeile erfolgt Ausgabe aus Puffer

Dateiendeerkennung
Objekte haben einen Wert (auch infile) -> kann man nach Fehlerbedingungen abtesten
zurückgeben von 1, solange Ende der Datei nicht erreicht ist (while-Bedingung erfüllt)
while-Bedingung ist 0 wenn EOF erreicht ist

Zeichenein- und -ausgabe in eine Datei
get (Klasse istream) und put (Klasse ostream) für Einlesen und Ausgeben

Objektein- und –ausgabe
Objekte einer Klasse werden in Datei gespeichert und ausgelesen
Methode write ( ) gehört zu ofstream.

Unterschied zwischen Text- und Binärdateien
write ( ) arbeitet nicht mit Zeichen (so wie put) sondern mit deren Binärwerten
Daten des Objekts werden als Folgen von Binärwerten (stream) ausgegeben -> keine Angabe über Weiterverarbeitung einzelner Binärwerte

Zeichenorientierte Methoden arbeiten anders:
\n wird auf zwei Bytes erweitert, bevor es in Datei geschrieben wird
Vorteil: jeder Editor kann Datei lesen!
Modi für Dateieröffnung:

ios::app	wenn File schon existiert, wird auf Fileende positioniert -> File wird fortgeschrieben
ios::ate	wenn File schon existiert, wird auf Fileende positioniert -> File wird fortgeschrieben verschiedene Positionierungen sind erlaubt -> auf Position des aktuellen Ausgabezeigers wird geschrieben
ios::in	File wird für Eingabe eröffnet. mit ios::out ist kombinierte Ein- und Ausgabe möglich
ios::out	File wird neu angelegt. Existierendes File wird auf 0 Byte gekürzt
ios::trunc	Existierendes File wird auf 0 Bytes gekürzt
ios::nocreate	File wird nur eröffnet, wenn es bereits existiert -> ansonsten Fehler!
ios::noreplace	File wird nur eröffnet, wenn es noch nicht existiert

Wahlfreier (direkter) Zugriff
zu jeder Datei gehören Lese- und Schreibzeiger (= Bytenummern innerhalb der Datei, wo nächstes Lesen und Schreiben stattfinden soll)
Bei Bearbeitung von sequentiellen Dateien wird jeweilige Dateizeiger automatisch gehandelt.
Bei wahlfreiem Zugriff auf beliebige Stelle der Datei muss Dateizeiger kontrolliert werden.
seekg ( ) und tellg ( ) erlauben Setzen und Positionieren von Lesezeiger
Für Schreibzeiger: seekp ( ) und tellp ( )

Bsp.:
infile.seekg ( );
Setzen auf Dateianfang
Datei beginnt mit Byte 0!
-> benötigt ein Argument (Argument beschreibt absolute Position)

Angabe eines Abstands:
seekg ( ) kann auch mit zwei Argumenten aufgerufen werden.

1. Argument → Abstand zu einer bestimmten Position innerhalb der Datei
2. Argument → Position, von der aus der Abstand gemessen wird
   3 Möglichkeiten:
   beg -> Anfang der Datei
   cur -> aktuelle Zeigerposition
   end -> Ende der Datei

Bsp.:
infile.seekg (-10; ios::end) -----> 10 Byte vor dem Ende

Formatieren der Ausgabe

Weite und Füllzeichen:

width ( ) bestimmt Weite der unmittelbar folgenden Zahlen- oder Stringausgabe
Leerstellen werden mit Leerzeichen aufgefüllt
fill ( ) ersetzt Leerzeichen durch anderes Füllzeichen
Bsp.:

```
cout.width (7);
cout.fill ('0');
cout << '(' << 13 << ')';
cout << '(' << 35 << ')';
```

Ausgabe:
(0000013) (35)

Steuerung der Ausgabe über Flags
Flag ist Zeichen für Merkmal.
Merkmal vorhanden = Flag gesetzt
Merkmal nicht vorhanden = Flag nicht gesetzt

folgende Flags für Formatsteuerung:

Flag	Hex-Wert	Bedeutung
skipws	01	Whitespace ignorieren
left	02	linksbündige Ausgabe
right	04	rechtsbündige Ausgabe
internal	010	zwischen Vorzeichen & Wert auffüllen
dec	020	dezimale Ausgabe
oct	040	oktale Ausgabe
hex	0100	hexadezimale Ausgabe
showbase	0200	Basis anzeigen
showpoint	0400	nachfolgende Nullen auffüllen
uppercase	01000	E, X statt e, x
showpos	02000	+ bei pos. Zahlen anzeigen
scientific	04000	Exponentialformat
fixed	010000	Gleitpunktformat
unitbuf	020000	nach jeder A-operation Puffer leeren
stdio	040000	nach jedem Textzeichen Puffer leeren

setf ( ) setzt Flags für Ausgabeobjekt
Bsp.:
cout.setf (ios::fixed);
cout.setf (ios::fixed | ios::showpos);

Weite von Gleitpunktzahlen

precision ( ) steuert Weite von GPZ (Flags fixed oder scientific nicht setzen!)
precision ( ) legt Anzahl Nachpunktstellen fest

Bis zum nächsten precision ( ) ist eingestellte Stellenanzahl gültig
Bsp.:
.....
f = 10.123456789;
cout << f << endl;

Ausgabe Standard-Precision = 5:
10.123

cout.precision (5);
cout << f << endl;
Ausgabe:
10.123

cout.setf (ios::fixed);
cout << f << endl;
Ausgabe:
10.12346
Begründung:
precision steht weiterhin mit 5, wegen fixed sind es 5 Nachpunktstellen

Präprozessoranweisungen (-direktiven)

include, # define, # undef, # if, # else, # ifdef, # ifndef, # elseif, # endif, # pragma

#include:
Form:
include "dateiname" oder
include <dateiname>

Mit Anweisung wird Inhalt der angegebenen Datei als Quelltext in geschriebenen Quelltext eingefügt.
Quelltexte können sein: Funktionsdefinitionen, Makros, Definitionen, Anweisungsfolgen
Bei # include <dateiname>: Suche in Standardverzeichnissen (von IDE gesetzt)
Bei # include "dateiname": Suche im aktuellen VZ danach in Standardverzeichnissen

Präprozessoranweisungen können an beliebiger Stelle im Quelltext stehen

define
Möglichkeit, Konstanten mit Namen zu versehen
Bsp.:

```
# include <iostream.h>
# include <math.h>
# define U_GRENZE 0
# define O_GRENZE 2*M_PI
# define SCHRITT (M_PI / 8)
int main ( )
{double x;
  cout << 'x' << '\t' << "sin(x)";
  for (x=U_GRENZE; x<O_GRENZE + SCHRITT / 2; x+=SCHRITT)
  cout << 'x' << '\t' << "sin(x)";
```

Definition und Aufruf von Makros

Bei # define wird Ersatz aus Programmanweisungen gebildet
Bsp.:
Makrodefinition

```
# define QUADRAT (a)                    ((a)*a))
```

Aufruf des Makros:

```
....
z=QUADRAT (x+1);
```

Präprozessor ersetzt:

```
z = ((x+1) * (x+1))
```

Achtung!
Sorgfältige Klammerung!

Konstanten in Headerdateien schreiben, wenn sie in verschieden Quelltexten verwendet werden sollen -> includieren von jeder Quelldatei möglich
Jeder Programmierer verwendet gleichen Satz an Definitionen und Vereinbarungen
Makros mit Parametern können wie Funktionen aufgerufen werden
wichtige Unterschiede:

- Makros:
  Definition über Präprozessor mit # define
  entsprechende Erweiterung wird eingefügt -> ausführbare Datei wird länger
  kürzere Programmlaufzeit, weil Funktionsoverhead entfällt
  keine Typprüfung

- Funktion
  werden eigenständig übersetzt
  Linker verbindet sie in ausführbare Datei
  Funktion wird verzweigt -> Laufzeit wird verlängert
  kürzere ausführbare Datei

undef
undef hebt # define – Anweisung auf (ab da, wo # undef steht)
Bsp.:

```
# define NACHRICHT "\n Fehler-Nr.: "
......
unsigned int i;
.....
cout << NACHRICHT << nummer;
......
# undef NACHRICHT
........
# define NACHRICHT "\n E/A – Fehler"
```

#if # else #elif # endif
Übersetzung von bestimmten Programmteilen anzuschließen ist möglich
einfachste Form:

```
#if     konst_Ausdruck
        Quellprogrammteil
# endif
```

Unmittelbar folgender Programmteil wird ausgeführt, wenn konst_Ausdruck true ist, sonst nicht

Erweiterte Formen:

```
# if    konst_Ausdruck
        Programmteil a

# else  konst_Ausdruck
        Programmteil b

# endif
```

oder

```
# if      konst_Ausdruck
          Programmteil 1

# elif    konst_Ausdruck
          Programmteil 2

# elif    konst_Ausdruck
          Programmteil 3

.........

# endif
```

Bsp.:

```
# include <iostream>
# define A1
# define B0
# define C (A) = = (B)

int main ( )
{ # if C
  cout << "Programmteil A";
   # else
  cout << "Programmteil B";
```

Bedingte Compilierung

```
# define DEMO 1
....
# if DEMO
   float vektor [20];
   sleep (60);
# else
   float vektor [2000];
# endif
```

ifdef # ifndef
ifdef und # ifndef ersetzen # if- Direktive

ifdef name
bzw.
ifndef name

name steht für Name, der schon in # define-Direktiven verwendet wurde (oder nicht)

Äquivalente Schreibweise:

# ifdef	äquivalent	# if defined
# ifndef	äquivalent	# if !defined

error
Übersetzung des Quelltextes wird abgebrochen, wo # error steht. Kommentar kann eingefügt werden (wird mit ausgegeben)

```
# error Nachricht
```

IDE – Meldung mit der Nachricht erscheint!

line
Zeilennummerierung (beginnend mit 1), ab da, wo # line steht

```
# line Zeilennummer "dateiname"
```

Zur Fehlernachricht gehörende Dateiname kann neu festgelegt werden

pragma
Compiler kann Direktiven übergeben

```
# pragma Compilerparameterwerte
```

Unterschiedliche # pragma – Anweisungen (von Compiler abhängig)

Zeiger

wozu?
- Zugriff auf Elemente eines Vektors
- Übergabe von Argumenten an Funktion, wenn Funktion Argumente verändern soll
- Speicherplatzanforderung an System, falls sie dynamisch erfolgt
- Verarbeitung von Datenstrukturen (verkettete Listen und Bäume)

Zeiger und Adressen
Konzept:
jedes Byte im Speicher ist unter einer Adresse gespeichert
Adressen sind natürliche Zahlen (0 bis n)
jede Variable beginnt bei einer Adresse
Bestimmung der Adresse einer Variablen mit &-Operator

Variablen werden auf Stack geschrieben
Stackadressen werden absteigend genutzt

Zeigervariablen
Vorteile im Zusammenhang mit Zeiger – dient der Speicherung von Adressen
Typ von Zeigervariablen?
ist angepasst an Datentyp für den sie verwendet werden (alle vordefinierten Datentypen, wie float, double, int....)
sowie benutzerdefinierten Datentypen (Strukturtypen und Klassen)

Bsp.:
```
int var1;
float var2;
int *pint1, *pint2;
float *pfl;

pint1 = &var1;
pfl = & var2;
pint2 = & var1;
```

Zeigervariablen müssen Wert zugewiesen bekommen (eine Adresse), bevor man mit ihnen arbeitet

Inhaltsoperator

Um auf Inhalt der Adresse zuzugreifen, muss Inhaltsoperator * verwendet werden.
Bsp.:
Verwendung Inhaltsoperator

```
* pint1 = 13;          // var1 = 13
* pfl = 14.5;          //var2 = 14.5
var3 = * pfl;         // var3 = 14.5
```

Zeiger auf void
ist typloser Zeiger.
Enthält Adressen von Variablen beliebiger Datentypen
Bsp.:

```
void * ptr;
```

Bsp.:
Verwendung

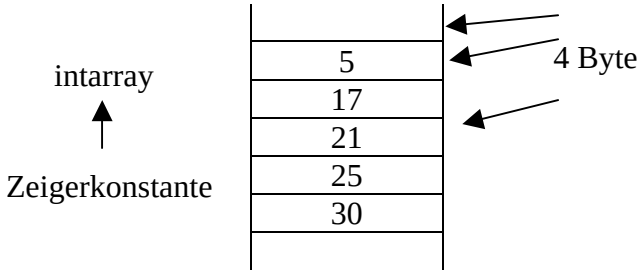
```
int var1;
float var2;
int * pint;
float * pfl;
void * ptr;

pint = & var1;          // OK!
pfl = & var2;           // OK!
pint = & var2;          //Fehler!
ptr = & var2;           // OK!
ptr = & var1;           // OK!
```

Anwendung:
Übergabe von Argumenten an Funktion, wenn Argumente von unterschiedlichen Typen sein können

Zeiger und Vektoren
Enge Beziehung zwischen Zeigern und Vektoren
Zugriffsmöglichkeiten auf Vektor:

Bsp.:
mit Vektorschreibweise
int intarray [5] = {5, 17, 21, 25, 30}
for (int i=0; i<5; i++)
cout << intarray [i] << endl;



Bsp.:
mit Zeigerschreibweise
int intarray [ ] = {5, 17, 21, 25, 30};
.....
for (int i =0; i<5, i++)
cout << *(intarray + i) << endl;

Bsp.:
Zugriff auf Vektor mit Zeigern (Zeigervariable)

int intarray [ ] = {5, 17, 21, 25, 30};
int *pint;
pint = intarray; //da Zeigerkonstante kein &-1
for (int i=0; i<5; i++)
cout << *(pint ++) << endl;