Praktikum

zu der Veranstaltung

Programmierung mit C++ 2

Bachelor-Studiengang Informatik

Prof. Dr.-Ing. Fritz Mehner

Fachhochschule Südwestfalen

Fachbereich Informatik und Naturwissenschaften

Email: mehner.fritz@fh-swf.de

Version 1.8

Stand 21. März 2014

Inhaltsverzeichnis

Einführung v

1. Bildverarbeitung 3 1
1.1. Verwendung von Strukturen . . . 1

2. The Game of Life 3
2.1. Implementierung . . . 4
2.2. Erweiterungen . . . 6

3. Verkettete Listen 7
3.1. Einfach verkettete Liste aus statischen Elementen . . . 7
3.2. Rohdaten aus einer Datei einlesen . . . 8
3.3. Personenbeschreibung als Struktur . . . 9
3.4. Personenbeschreibungen in einer Liste . . . 9
3.5. Zerlegung in Funktionen . . . 10

4. Stapel als Klasse 11
4.1. Basisimplementierung eines Stapels als *C++*-Klasse . . . 11
4.2. Konstruktor und Destruktor für Stapel beliebiger Größe . . . 12
4.3. Stapel-Objekte kopieren . . . 12

5. Reihenfolge und Wirkung von Methodenaufrufen 15
5.1. Implementierung einer Testklasse . . . 15
5.2. Analyse der Programmausgaben . . . 17

6. Überladen von Operatoren 19
6.1. Implementierung der Klasse `Vek3` . . . 19
6.2. Implementierung der Operatorüberladung . . . 21
6.3. Mittelpunkt eines Kreises in beliebiger Lage . . . 21
6.4. Berechnung der Bogenlänge . . . 24
6.5. Implementierung und Test . . . 24

7. Ausnahmebehandlung 27
7.1. Implementierung der Klasse `Datum` . . . 27
7.2. Verwendung von Fehlerklassen . . . 28

8. Verwendung von Templates 29
8.1. Implementierung einer Ringliste . . . 29
8.2. Generische Ringliste . . . 30

A. Programmdokumentation 31
A.1. Dateibeschreibung . . . 31
A.2. Abschnittskommentare und Zeilenendkommentare . . . 32
A.3. Einrückung und Schachtelung . . . 33

B. Fehlersuche **35**
B.1. Die häufigsten Anfängerfehler 35
B.2. Der Debugger `DDD` 36

C. Build Tools **39**
C.1. Die Verwendung von `make` 39
C.2. Die Verwendung von `qmake` 41

Stichwortverzeichnis **43**

Einführung

Lehre bildet Geister,
doch Übung macht den Meister.
Deutsches Sprichwort

Das vorliegende Dokument enthält die Praktikumsaufgaben zu der Veranstaltung ***Programmierung mit C++ 2*** im Studiengang *Informatik* des Fachbereiches *Informatik und Naturwissenschaften.*

Die Durchführung der Praktikumsaufgaben gibt Gelegenheit, das in der Vorlesung Gehörte anzuwenden und zu vertiefen. Programmieren kann man nur lernen indem man programmiert. Wenn keine Vorkenntnisse vorhanden sind, können die Anfänge durchaus mühevoll sein. Die alleinige Teilnahme am Praktikum, das heißt Programmieren am Rechner im Umfang von 2 Semesterwochenstunden, reicht *keinesfalls* zum Erwerb gefestigter Grundkenntnisse aus.

> Es ist zwingend erforderlich, auch außerhalb der Lehrveranstaltungen das in der Vorlesung erworbene Wissen praktisch nachzuvollziehen, anzuwenden und zu vertiefen.

Dazu sollte auf dem eigenen Rechner eine entsprechende Entwicklungsumgebung zur Verfügung stehen. Grundsätzlich ist jeder ***C++***-Compiler oder jede Entwicklungsumgebung geeignet. Aus Kostengründen bietet sich freie Software an (***Linux*** oder ***FreeBSD***). Bei allen ***Unix***-Betriebssystemen gehören ***C***-/***C++***-Compiler zur Grundausstattung und werden durch weitere leistungsfähige Entwicklungswerkzeuge ergänzt. Selbstverständlich können stets auch die Einrichtungen der Fachhochschule benutzt werden.
Für Windows-Benutzer steht der freie GNU-***C***/***C++***-Compiler zusammen mit verschiedenen integrierten Entwicklungsumgebung ebenfalls zur Verfügung.

Der zur Lösung der jeweiligen Aufgaben benötigte Stoff entspricht natürlich dem Stand, der in der Vorlesung bis zur Bearbeitungszeit erreicht wurde. Das heißt jedoch nicht, daß es für einzelne Aufgabenteile nicht elegantere Lösungen gibt, die aber Kenntnisse erfordern, die im Augenblick noch nicht vermittelt wurden.

Durchführung des Praktikums

Teilnahme und Testat Die selbständige Bearbeitung der Praktikumsaufgaben ist Pflicht. Die *Bearbeitungspflicht* gilt als erfüllt, wenn für mindestens 80 Prozent der Praktikumsaufgaben die selbständige und erfolgreiche Bearbeitung durch den Betreuer bescheinigt wurde. Daraufhin wird der Erwerb der Vorleistung bescheinigt. Nur diese Vorleistung berechtigt zur Teilnahme an der Klausur. Die Abgabetermine regelt der Terninplan (siehe unten).

Eine Pflicht zur regelmäßigen Anwesenheit im Praktikum gibt es nicht. Die Praktikumstermine dienen dem Testieren der Abgaben und sollen zur persönlichen Beratung genutzt werden.

Terminplan Für die Testierung der Aufgaben gibt es einen Terminplan, der auf der Internetseite zu der jeweiligen Veranstaltung eingesehen werden kann. Dieser Terminplan ist *verbindlich.* Eine Nachfrist für eine Aufgabe wird nur dann gewährt, wenn zum festgesetzten Termin die Lösung im wesentlichen vorlag und deshalb nur Schwächen oder Fehler zu beheben sind.
Der Praktikumsbetreuer kann für einen vorher festgelegten Teil einer Praktikumsgruppe einen

um eine Woche späteren Termin festlegen, um die Abgaben zu entzerren und damit mehr Zeit für die Beratung aufwenden zu können.
Die zu testierenden Aufgaben müssen im wesentlichen zu *Beginn dieses Praktikumstermins* vorliegen, um den Betreuern die Durchsicht der Aufgaben aller Teilnehmer zu ermöglichen.

Vorbereitung und Durchführung Die erfolgreiche Durchführung des Praktikums setzt voraus, daß der zugrundeliegende *Stoff im wesentlichen bekannt* ist und daß die Aufgabenstellung *vollständig gelesen und verstanden* wurde.
Die Zeitdauer von 2 Wochenstunden, die formal für das Praktikum angesetzt ist, wird in der Regel nicht zur vollständigen und richtigen Bearbeitung der geforderten Aufgaben ausreichen, so daß wesentliche Teile der Lösung außerhalb des Praktikums erarbeitet werden müssen.
Die Praktikumstermine dienen unter anderem zur Klärung der Aufgabenstellung und bieten Gelegenheit, Einzelfragen zum Stoff der Vorlesung mit dem Betreuer zu besprechen. Weiterhin ist die Diskussion der gewählten Lösungswege und die Festlegung von Verbesserungen und Berichtigungen ein wesentlicher Zweck der Veranstaltung.

Programmdokumentation Für die zu erstellenden Programme gilt ein *Mindeststandard* für die Programmdokumentation der in Anhang A erläutert ist. Nicht oder mangelhaft dokumentierte Programme werden nicht anerkannt.

Programmtests Grundsätzlich ist *jedes vorzulegende Programm* vorher vom Autor zu testen. Das geschieht in der Regel durch die Implementierung der jeweils geforderten Tests. Diese sind je nach Aufgabenstellung durch Nachrechnen oder durch stichprobenartige Überprüfung zu kontrollieren.
Wenn Testfälle vorgegeben sind (zum Beispiel Zahlenwerte für Gleichungskoeffizienten und ähnliches), dann müssen diese zur Vereinfachung der Kontrolle natürlich in der angegebenen Form verwendet werden.
Tests sind selbstverständlich auch dann durchzuführen, wenn Sie nicht ausdrücklich in der Aufgabenstellung gefordert werden.
Im Abschnitt B.1 sind einige typische Anfängerfehler aufgelistet. Sie sollten unbedingt lernen, diese Fehler zu vermeiden! Überprüfen Sie Ihre Lösungen vor der Abgabe auch mit Hilfe dieser Liste.

Rechnerplattform Die vorzulegenden Programme müssen auf den im Praktikum zur Verfügung stehenden ***Linux***-Installationen ohne Fehler und Warnungen übersetzbar, bindbar und ausführbar sein.

Freiwillige Zusatzaufgaben Einzelne Übungen enthalten freiwillige Bestandteile die in der jeweiligen Überschrift mit der Kennzeichnung *freiwillig* ausgewiesen sind.
Die Bearbeitung dieser Teile wird empfohlen, ist aber nicht zur Erlangung der Vorleistung erforderlich. Die erarbeiteten Lösungen können natürlich trotzdem den Praktikumsbetreuern zur Beurteilung vorgelegt werden.

Bewertung der Aufgaben

Ziel der Bearbeitung einer Aufgabe ist natürlich die vollständige Bearbeitung und Lösung der Aufgabenstellung. Bei der Testierung können jedoch Teilaufgaben anerkannt werden, sodaß ein mangelhafter Anteil nicht das gesamte Testat in Frage stellt. Tabelle 1 listet die Punkte auf, die bei den einzelnen Teilaufgaben erzielt werden können. Die Numerierung der Teilaufgaben stimmt mit den Abschnittsnummern überein (Aufgabe 3, Teil 2 findet sich in Abschnitt 3.2). Einige wenige Abschnitte sind nicht aufgeführt, weil diese Erläuterungen zu den davorstehenden Aufgabenteilen enthalten und somit keine eigene Wertung besitzen.

Programmierung mit C++ 2

Aufgabe	Teil 1	Teil 2	Teil 3	Teil 4	Teil 5	Summe
1	10			-	-	10
2	10	-	-	-	-	10
3	2,5	2,5	2,5	2,5	-	10
4	4	3	3	-	-	10
5	3	7	-	-	-	10
6	2	2	-	-	6	10
7	5	5	-	-	-	10
8	4	6	-	-	-	10
						$\sum 80$

Tabelle 1.: Aufteilung der Teilpunkte

Gesamtpunktzahl Insgesamt können 80 Punkte erreicht werden. Mit **64 Punkten** (80 Prozent) ist die Bearbeitungspflicht erfüllt.

Praxisbezug der Praktikumsaufgaben

Die Praktikumsaufgaben dienen dazu, den in der Vorlesung behandelten Stoff anzuwenden und zu vertiefen. Die Veranstaltungen *Programmierung mit* ***C++ 1*** und *Programmierung mit* ***C++ 2*** sind als Erstausbildung in einer höheren Programmiersprache angelegt. Bei der Auswahl der Beispiele muß deshalb entweder auf geeigneten Stoff aus gleichzeitig laufenden Veranstaltungen (zum Beispiel *Grundlagen der Informatik*) oder auf Beispiele aus der Mathematik zurückgegriffen werden, da hier mindestens der Kenntnisstand der Fachhochschulreife erwartet werden kann.
Bei Anwendungen, wie etwa einer Gerätesteuerung, einer Bildauswerte-Software, einem Compiler oder einem Zeichenprogramm ist der Praxisbezug ganz offensichtlich gegeben. Die Lehrerfahrung zeigt jedoch, daß die Behandlung solcher Beispiele, oder auch nur größerer Teile daraus, bereits erhebliche Kenntnisse der jeweiligen Anwendungsgebiete und fortgeschrittene Programmierkenntnisse voraussetzen. Der Anfänger müßte sich mit zwei Problemfeldern gleichzeitig auseinandersetzen.
Das bedeutet jedoch keineswegs, daß die gewählten Praktikumsaufgaben nicht praxisbezogen sind – ganz im Gegenteil! Das Erlernen einer Programmiersprache beginnt mit dem Kennenlernen der Sprachbestandteile und deren Bedeutung. In weiteren Schritten müssen unter anderem einfache Vorgehensweisen und Trivialalgorithmen für immer wiederkehrende Anwendungssituationen erlernt werden. Hierzu zählen zum Beispiel wechselnde Vorzeichen in Zahlen- und Aufruffolgen, Wertetausch bei Variablen, die Bestimmung von Summen, Minimal- und Maximalwerten von Zahlenmengen und ähnliches. Der Umgang mit Zahlen, die im Gegensatz zur Mathematik endlich und nicht beliebig genau sind, muß ebenso geübt werden, wie etwa die Vermeidung häufig anzutreffender Fehler (siehe auch Abschnitt B.1) oder die Erzielung von logisch vollständigen Lösungen. Die saubere Formatierung und Kommentierung von Programmen, sowie die Durchführung von systematischen Tests und die Fehlersuche sind als nächste Schritte zu nennen. Diese und weitere Kenntnisse und Fähigkeiten sind unabdingbar für jeden, der später im Beruf verantwortlich Programme entwickeln möchte.
Aus den genannten Gründen sind die gerade genannten erworbenen Fähigkeiten praktisch in höchstem Maße wichtig, da man ohne sie nicht davon sprechen kann, eine Programmiersprache zu beherrschen.

Zur Darstellung

Programmcode, Programmausgaben, Programm- und Dateinamen, Schlüsselwörter von Programmiersprachen und Menüeinträge erscheinen in `Schreibmaschinenschrift mit fester Zeichenbreite`. In Codebeispielen und Listen werden für Code und Kommentar zur Verbesserung der Lesbarkeit unterschiedliche Schriftstile der Schriftart `luximono` verwendet: Schlüsselwörter sind halbfett gesetzt (zum Beispiel **`double`**), Kommentare sind kursiv gesetzt (zum Beispiel *`// Datei einlesen`*), alle anderen Programmbestandteile sind im Normalschnitt gesetzt (zum Beispiel `erg = fkt3( y );`).

Dieses PDF-Dokument enthält aktive Verweise zur Erleichterung der Navigation: Seitenzahlen im Inhalts- und Stichwortverzeichnis, Email- und Web-Adressen, sowie Verweise auf Seiten, Listen, Abbildungen, Formeln und Tabellen im laufenden Text. Auf die Verwendung farbiger Verweise wurde verzichtet, da die schwächere Einfärbung bei der gedruckten Fassung die Lesbarkeit mindert.

Dieses Dokument wurde in LaTeX 2_ε unter ***Linux*** erstellt.

1. Bildverarbeitung 3

1.1. Verwendung von Strukturen **10 Punkte**

Erstellen Sie eine neue Version der Bildverarbeitungsprogramme aus den Praktikumsaufgabe 7 und 8 des Moduls *Programmierung mit C++ 1*. Die Grauwertbilder sollen hier in einer Struktur `pgm_bild` (Liste 1.1) abgelegt werden. Außerdem sollen alle Bildverarbeitungsfunktionen in jeweils einer eigenen Funktion dargestellt werden. Die Bilder werden mittels Zeiger an die Funktionen übergeben werden.

Liste 1.1: Strukturdefinition für ein pgm-Bild

```
1  // #####   TYPE DEFINITIONS   ##########################################################
2  typedef unsigned char   Pixel;                  // Datentyp Pixel
3
4  // #####   DATA TYPES  -  LOCAL TO THIS SOURCE FILE   ###############################
5  struct pgm_bild
6  {
7    char    magic[2];                              // Bildtyp
8    int     nx;                                    // Zeilenzahl
9    int     ny;                                    // Spaltenzahl
10   int     graumax;                               // max. Grauwert
11   Pixel **bild;                                  // Bildmatrix
12 };        // ----------  end of struct pgm_bild  ----------
13
14 // #####   LOCAL PROTOTYPES   ##########################################################
15 void    bild_lesen          ( pgm_bild *bild, string ifs_file_name );
16 void    bild_schreiben      ( pgm_bild *bild, string ofs_file_name );
17 void    delete_pixel_matrix ( Pixel **m );
18 Pixel** new_pixel_matrix    ( size_t rows, size_t columns );
19 void    glaetten            ( pgm_bild *bild1, pgm_bild *bild2 );
20 void    invertieren         ( pgm_bild *bild1, pgm_bild *bild2 );
21 void    kantenbildung       ( pgm_bild *bild1, pgm_bild *bild2 );
22 void    kopiere_bildkopf    ( pgm_bild *bild1, pgm_bild *bild2 );
23 void    schwellwert         ( pgm_bild *bild1, pgm_bild *bild2, Pixel schwellwert );
```

Die Funktionen `bild_lesen` und `bild_schreiben` brechen das Programm mit einer Fehlermeldung ab, wenn die Eingabedatei beziehungsweise die Ausgabedatei nicht geöffnet werden kann. Beim Schreiben der Ausgabedatei eine Typkonvertierung verwendet werden, um die Grauwerte der Bildpunkte als Zahlenwert zu erhalten. Hier ein Beispiel:

```
ofs << setw(4) << (int)bild->bild[i][j];
```

Die Bildpunkte werden in einer Bildmatrix abgelegt, die in der Einlesefunktion `bild_lesen` in der passenden Größe beschafft wird. Die Bildmatrizen werden am Ende des Hauptprogrammes freigeben. Zum Anlegen und Löschen der Bildmatrizen dienen die beiden Funktionen `new_pixel_matrix` und `delete_pixel_matrix` (Liste 1.2).

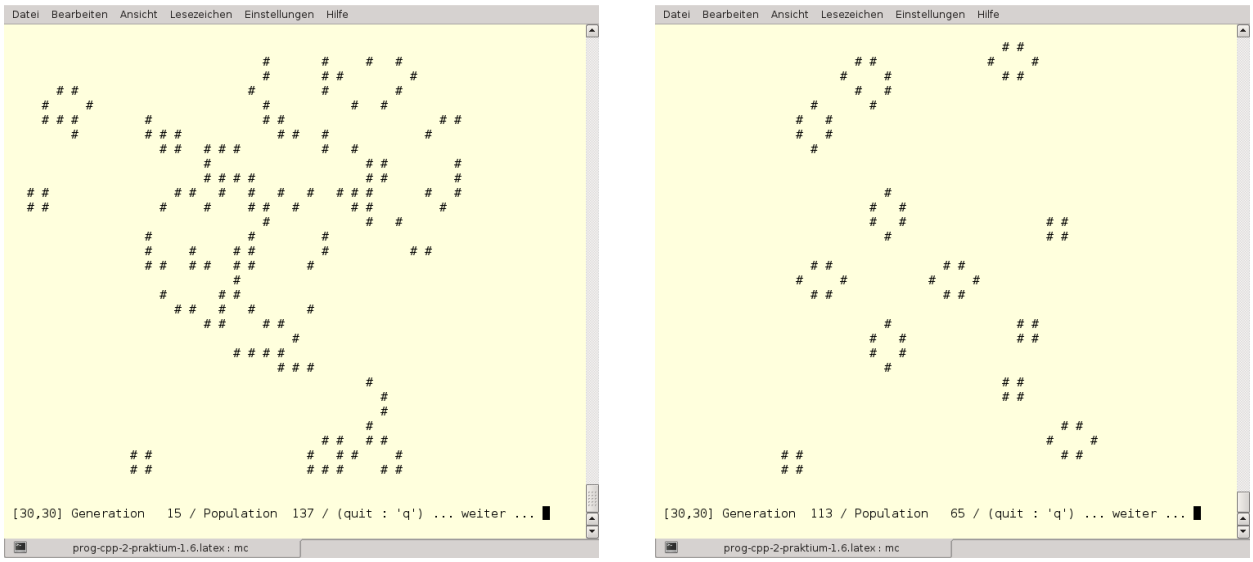
Liste 1.2: Dynamische Bildmatrix beschaffen und freigeben

```
// ===  FUNCTION  ======================================================================
//         Name:  new_pixel_matrix
//  Description:  Allocate a dynamic Pixel-matrix of size rows*columns; return a pointer.
// =====================================================================================
Pixel** new_pixel_matrix ( size_t rows, size_t columns )
{
  size_t  i;
  Pixel   **m;
  m   = new Pixel* [rows];                      // allocate pointer array
  *m  = new Pixel  [rows*columns];              // allocate data array
  for ( i=1; i<rows; i+=1 )                     // set pointers
    m[i]  = m[i-1] + columns;
  return m;
}  // ----------  end of function new_int_matrix  ----------

// ===  FUNCTION  ======================================================================
//         Name:  delete_pixel_matrix
//  Description:  Free a dynamic Pixel-matrix.
// =====================================================================================
void delete_pixel_matrix ( Pixel **m )
{
  delete[] *m;                                  // delete data array
  delete[]  m;                                  // delete pointer array
}  // ----------  end of function delete_int_matrix  ----------
```

Versuchen Sie, die beiden Funktionen in Liste 1.2 zu verstehen und ihre Arbeitsweise zu erklären.

2. The Game of Life



(a) 15. Generation (b) stabiler Endzustand

Abbildung 2.1.: Ein Spiel auf einem 30×30–Feld

Das Spiel des Lebens (engl. Game of Life) ist ein vom Mathematiker John Horton Conway 1970 entworfenes Spiel, das eine sehr einfache Art von Evolution simuliert. In einem zweidimensionalen Feld gibt es lebende und tote Zellen (Abbildung 2.1). Nach vier einfachen Regeln (siehe unten) wird aus der gerade vorhandenen Generation eine neue Generation berechnet.

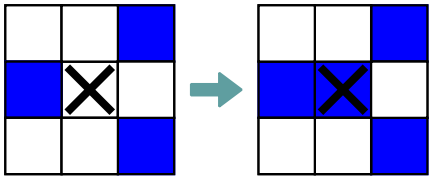
Die Regeln

Jede innere Zelle hat 8 Nachbarzellen, eine innere Randzelle hat 5 Nachbarzellen, eine Eckzelle hat 3 Nachbarzellen. Die nächste Generation wird durch Anwendung der folgenden Regeln aus der vorherigen bestimmt (siehe auch Abbildung 2.2).

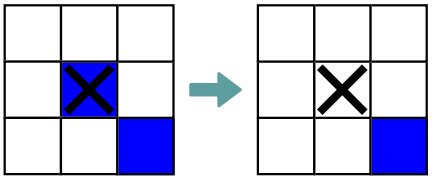
1. Eine tote Zelle mit genau drei Nachbarn wird in der nächsten Generation neu geboren.
2. Eine lebende Zelle mit weniger als zwei Nachbarn stirb in der folgenden Generation an Vereinsamung.
3. Eine lebende Zelle mit zwei oder drei Nachbarn bleibt in der folgenden Generation am Leben.
4. Eine lebende Zelle mit mehr als drei Nachbarn stirbt in der folgenden Generation an Überbevölkerung.

Alle Zellen, auf die keine der Regeln anzuwenden ist, werden unverändert in die nächste Generation übernommen.

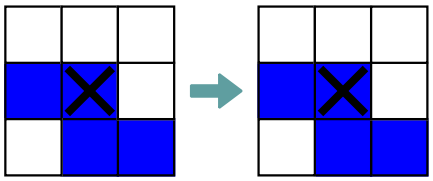
Das Spiel führt zu verblüffenden Beobachtungen. Nach einigen Generationen entstehen meist Zellgruppen mit besonderen Eigenschaften. Manche Zellanordnungen bilden statische Muster (zum Beispiel die Quadrate und die Sechsecke in Abbildung 2.1b). Weiterhin gibt es oszillierende Anordnungen mit unterschiedlichen Zykluslängen und sich bewegende Gruppen, die über das Spielfeld wandern.



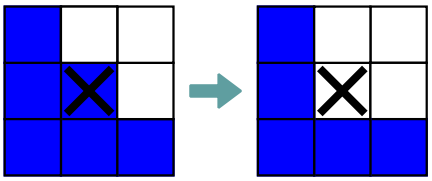
(a) Eine tote Zelle mit genau drei Nachbarn wird in der nächsten Generation neu geboren



(b) Eine lebende Zelle mit weniger als zwei Nachbarn stirb in der folgenden Generation an Vereinsamung



(c) Eine lebende Zelle mit zwei oder drei Nachbarn bleibt in der folgenden Generation am Leben



(d) Eine lebende Zelle mit mehr als drei Nachbarn stirbt in der folgenden Generation an Überbevölkerung

Abbildung 2.2.: Die vier Regeln des Game of Life

2.1. Implementierung

10 Punkte

Liste 2.1: Prototypen

```
1 typedef unsigned int  uint;
2
3 void    zufallsbelegung   ( int **feld, uint hoehe, uint breite );
4 uint    next_generation   ( int **feld1, int **feld2, uint breite, uint hoehe );
5 void    print_feld        ( int **feld, uint hoehe, uint breite );
6 int**   new_int_matrix    ( int rows, int columns );
7 void    delete_int_matrix ( int **m );
```

Das Spiel soll als Konsolanwendung implementiert werden. Die Anwendung ist in mehrere Funktionen aufzuteilen (Prototypen in Liste 2.1). Das Spielfeld wird als `int`-Matrix dargestellt, deren Größe am Programmanfang eingelesen wird (zum Beispiel 30 Zeilen, 40 Spalten). Die intern verwendete Matrix wird mit 2 zusätzlichen Zeilen und Spalten angelegt (im Beispiel dann 32 Zeilen und 42 Spalten). Die Randzellen werden zwar mit Null initialisiert, aber bei der Berechnung der nächsten Generation nicht besetzt. Das vereinfacht die Berechnung der vorhandenen Nachbarn, weil die 30×40 inneren Zellen immer 8 Nachbarn besitzen.

Hauptprogrammes Spielfeldgröße einlesen und zwei Spielfelder anlegen (`int`-Felder). Die inneren Zellen des ersten Feldes werden zufällig mit 0 und 1 initialisiert (0 entspricht tot, 1 entspricht lebendig). Alle Randzellen werden mit 0 initialisiert. In einer Schleife wir solange die nächste Generation berechnet und ausgegeben, bis der Benutzer das Programm mit der Eingabe `q` (quit)

abbricht. Wie in Abbildung 2.1 zu sehen ist, wird jeweils die Größe des (inneren) Spielfeldes, die Nummer der Generation und die Anzahl der lebenden Zellen ausgegeben.

zufallsbelegung Zufällige Belegung der inneren Felder der Spielfeldmatrix mit 0 und 1.

next_generation Berechnung der nächsten Generation. Die nächste Generation muß zunächst in einem zweiten Feld aufgebaut werden (warum ?). Am Ende wird das erste Feld (alte Generation) mit der neuen Generation überschrieben oder die Zeiger auf die Felder getauscht. Das Hauptprogramm kann dann immer das 1. Feld ausgeben. Die Funktion gibt die Anzahl der lebenden Zellen (Populationsgröße) zurück.

print_feld Ausgabe der jeweils aktuellen Generation. Die Nullen werden durch ein Leerzeichen dargestellt, die Einsen durch das Nummernzeichen #.

new_int_matrix Beschaffung eines zweidimensionalen Feldes zur Laufzeit des Programmes (Liste 2.2). Felder werden wie folgt vereinbart und nach der Beschaffung wie statische zweidimensionale Felder verwendet (Zeilen-, Spaltenindex).

```
int **feld1;
  ...
feld1 = new_int_matrix( hoehe, breite );
  ...
feld1[i][j] = ...
```

delete_int_matrix Freigabe eines zweidimensionalen Feldes zur Laufzeit des Programmes (Liste 2.2).

Liste 2.2: Die Funktionen `new_int_matrix` und `delete_int_matrix`

```
1  // ===  FUNCTION  ======================================================================
2  //         Name:  new_int_matrix
3  //  Description:  Allocate a dynamic int-matrix of size rows*columns; return a pointer.
4  // =====================================================================================
5  int**
6  new_int_matrix ( int rows, int columns )
7  {
8    int   i;
9    int **m;
10   m     = new int* [rows];                    // allocate pointer array
11   *m    = new int  [rows*columns]();          // allocate data array; initialize!
12   for ( i=1; i<rows; i+=1 )                   // set pointers
13     m[i]  = m[i-1] + columns;
14   return m;
15 }  // ----------  end of function new_int_matrix  ----------
16
17 // ===  FUNCTION  ======================================================================
18 //         Name:  delete_int_matrix
19 //  Description:  Free a dynamic int-matrix.
20 // =====================================================================================
21 void
22 delete_int_matrix ( int **m )
23 {
24   delete[] *m;                                // delete data array
25   delete[]  m;                                // delete pointer array
26 }  // ----------  end of function delete_int_matrix  ----------
```

2.2. Erweiterungen *freiwillig*

Weitere Einzelheiten zum Spiel und Varianten der Regeln findet man auf einigen Web-Seiten, unter anderem bei *Wikipedia*. Weiterhin findet man unterschiedliche Implementierung zum Herunterladen. Folgende Erweiterungen liegen nahe:

- Speichern eines Spielstandes.
- Laden einer Anfangsbelegung aus einer Datei.
- Automatischer Ablauf: die Hauptschleife berechnet nach einer Verzögerung (zum Beispiel $0,5$ Sekunde) die nächste Generation und gibt diese aus.
- Textorientierte graphische Oberfläche (zum Beispiel mit der Bibliothek `ncurses`).
- Graphische Oberfläche.

3. Verkettete Listen

3.1. Einfach verkettete Liste aus statischen Elementen **2,5 Punkte**

Liste 3.1: Struktur der Listenelemente

```
1 struct Element {
2   long             key;          // Schluessel des Listenelements
3   long             info;         // die im Listenelement zu verwaltende Information
4   struct Element  *next;         // Zeiger auf das naechste Element
5 };        // ----------  end of struct element  ----------
```

In Abbildung 3.1 ist ein Liste mit vier Elementen gezeigt. Die Listenelemente werden als Struktur angelegt und besitzen den Datentypnamen `Element`. Der Listenanker ist dementsprechend ein Zeiger auf ein Element. Die Strukturvereinbarung ist in Liste 3.1 wiedergegeben.
Bei einer leere Liste hat der Zeiger `L` den Wert `NULL`, bei einer nichtleeren Liste zeigt er auf das erste Listenelement, das heißt er enthält dessen Adresse. Die Listenelement werden untereinander verkettet: ein Listenelement zeigt auf den Nachfolger, falls ein solcher vorhanden ist.

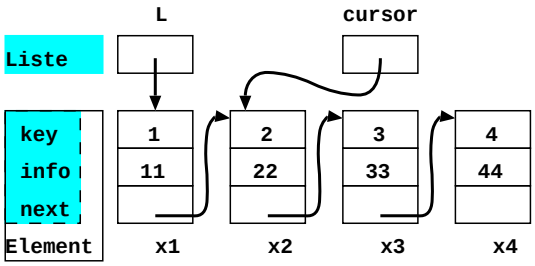


Abbildung 3.1.: Liste mit 4 Elementen

Schreiben Sie nun ein *C++*-Hauptprogramm in welchem folgende Schritte durchgeführt werden:

- Zwei Listenzeiger (Datentyp `Element*`) anlegen: `L` zeigt als Anker auf das 1. Listenelement (falls vorhanden), `cursor` wird als Zeiger beim Durchlaufen der Liste verwendet.
- Vier Listenelemente `x1`, `x2`, `x3`, `x4` anlegen. Als Schlüsselwerte (`key`) werden fortlaufend die Werte 1 bis 4 zugewiesen. Als Informationen (`info`) werden fortlaufend die Werte 11 bis 44 zugewiesen.
- Elemente verketten: durch entsprechende Adreßzuweisungen an die Strukturelemente `next` soll die Verkettung in Abbildung 3.1 hergestellt werden. Der `next`-Zeiger des letzten Element erhält den Wert `NULL`.
- Das Element mit dem Schlüsselwert 3 suchen: der Zeiger `cursor` wird auf den Listenanfang gesetzt und die Liste solange in einer `while`-Schleife durchlaufen bis entweder ein Listenelement mit dem Schlüsselwert 3 gefunden wurde oder das Listenende erreicht wurde (Wert von `next` ist `NULL`)
- Wenn das Element gefunden wurde soll die Elementadresse, der Schlüssel- und Informationswert sowie die Adresse des nachfolgenden Elementes ausgeben werden. Die Programmausgabe sollte wie folgt aussehen:

```
Listenelement gefunden : 0xbfa456d8

 key     =            3
 info    =           33
 next    =   0xbfa456cc
```

Die Zahlen- und Adreßwerte können mit dem Ausgabemanipulator `setw(12)` mit einer Breite von 12 Zeichen ausgegeben werden. Die Speicheradressen sind natürlich von der augenblicklichen Speicherbelegung abhängig und könnten allenfalls zufällig mit den hier gezeigten Beispielwerten übereinstimmen.

- Das Programm muß auch dann einwandfrei arbeiten, wenn die Liste leer ist und wenn der gesuchte Schlüsselwert nicht in der Liste vorhanden ist.

3.2. Rohdaten aus einer Datei einlesen **2,5 Punkte**

Schreiben Sie als Ausgangspunkt für die weiteren Teilaufgaben ein *C++*-Programm, in das gemäß Skriptum, Teil 2, Abschnitt 1.7, Datensätze eingelesen, zerlegt und zur Kontrolle wieder ausgegeben werden. Die Daten sind in der Datei *personen.dat* enthalten (Liste 3.2).

Liste 3.2: Personenbeschreibungen in der Datei `personen.dat`

```
1  10001222 DOLBEL              DAVID          QC18 4700
2  10002220 JAYALATH            THILANAKA      QC18 4701
3  10002222 GRANT               MARK           QC18 4714
4  10002222 ILLICH              ANDREW         QC18 4726
5  10002223 ATTWELL             SIMON          QC18 4702
6  10003222 GAMMON              JAMIE          QC15 4718
7  10022049 BAILEY              JASON          QC18 4703
8  10022194 ALDOUS              MATTHEW        QC15 4719
9  10022252 CHATRANGSAN         WERACHAT       QC15 4222
10 10022266 FU                  NING           QC15 4224
11 10022284 GILLETT             NATHAN         QC05 4731
12 10022865 JACKSON             SUZANNE        QC18 4752
```

Die Datensätze sind fest formatiert. Die Bestandteile haben folgende Bedeutungen:

Nr.	Bestandteil
1	Idenditätsnummer, eindeutig
2	Nachname
3	Vorname
4	Abteilungsbezeichnung
5	Telefondurchwahl

Die Datensätze werden zeilenweise in Variablen eingelesen (siehe unten). Diese werden dann in dieser Teilaufgabe ohne weitere Verwendung in einer eigenen Schleife zur Kontrolle ausgegeben. Die Ausgabe soll so formatiert sein, daß sie dem Bild der Eingabedatei in Liste 3.2 entspricht.

```
  string       identnummer;                    // Idenditätsnummer
  string       nachname;                       // Nachname
  string       vorname;                        // Vorname
  string       abteilung;                      // Abteilungsbezeichnung
  unsigned int durchwahl;                      // Telefondurchwahl

  // ... weiter unten :

  while ( ifs >> identnummer  >> nachname >> vorname >> abteilung >> durchwahl ) {
    satz++;

    // ... Variablenwerte formatiert ausgeben

  }
```

3.3. Personenbeschreibung als Struktur **2,5 Punkte**

Erstellen Sie auf der Grundlage des Programmes aus Abschnitt 3.2 eine neue Fassung, in der die Personenbeschreibungen in der Reihenfolge des Einlesens in einem globalen Feld von Strukturen (`struct mitarbeiter`) abgelegt werden. Die Strukturinhalte werden zunächst zur Kontrolle einfach wieder ausgegeben.

```
// #####   TYPE DEFINITIONS  -  LOCAL TO THIS SOURCE FILE   ##############

struct mitarbeiter {
  string           identnummer;
  string           nachname;
  string           vorname;
  string           abteilung;
  unsigned int     durchwahl;
};        // ----------  end of struct mitarbeiter  ----------

// #####   VARIABLES  -  LOCAL TO THIS SOURCE FILE   ####################

const unsigned int MitarbeiterMax  = 1000; // max. Anzahl Mitarbeiter
mitarbeiter ma[MitarbeiterMax];            // Feld mit Mitarbeiter-Beschr.
```

3.4. Personenbeschreibungen in einer Liste **2,5 Punkte**

Erweitern Sie in einer weiteren Programmfassung die Struktur aus dem letzten Abschnitt um einen Verkettungszeiger `next`, mit dessen Hilfe Strukturen zu einer Liste verkettet werden können:

```
struct mitarbeiter {
  string           identnummer;
  string           nachname;
  string           vorname;
  string           abteilung;
  unsigned int     durchwahl;
  mitarbeiter     *next;
};        // ----------  end of struct mitarbeiter  ----------
```

Richten Sie nun, ähnlich wie in Abschnitt 3.1, zwei Zeiger ein. Der Zeiger `ma` dient dabei als Listenanker, der Zeiger `cursor` dient als Durchlaufzeiger:

```
mitarbeiter *ma     = 0;                    // Listenanker
mitarbeiter *cursor = 0;                    // Durchlaufzeiger
```

In der Einleseschleife wird vor der Zerlegung des gerade gelesenen Datensatzes dynamisch eine neue Struktur angelegt und dann gefüllt:

```
mitarbeiter *maNeu  = new mitarbeiter;      // neue Struktur
```

Anschließend wird die neue Struktur am Ende der Liste angehängt. Der Zeiger `cursor` kann hierbei als Zeiger auf das Listenende (das jeweils letzte Element) verwendet werden.

Wenn die ganze Datei eingelesen und die Liste aufgebaut ist, dann soll ein bestimmter Datensatz gesucht werden. Setzen Sie dazu den Durchlaufzeiger wieder auf den Listenanfang und durchlaufen Sie die Liste so lange, bis die Personenbeschreibung mit der Telefondurchwahl 4731 gefunden ist. Der Vor- und Nachname ist in diesem Falle auszugeben. Überprüfen Sie die richtige Arbeitsweise ihrer Lösung auch für die folgenden Fälle:

- Die Liste ist leer.
- Die Liste enthält genau ein Element.
- Das gesuchte Element ist das erste Element der Liste.
- Das gesuchte Element ist das letzte Element der Liste.
- Das gesuchte Element steht weder am Anfang noch am Ende der Liste.
- Die gesuchte Telefondurchwahl ist in der nichtleeren Liste nicht vorhanden.

`Liste abbauen.` Am Programmende muß die Liste wieder abgebaut werden, da der Speicherplatz für die Elemente dynamisch beschafft wurde. Lassen Sie zu diesem Zwecke eine Schleife über die Liste laufen, die solange das erste Element entfernt, bis die Liste leer ist.

3.5. Zerlegung in Funktionen *freiwillig*

Zerlegen Sie die Lösung aus Abschnitt 3.4 in Funktionen mit den folgenden Prototypen:

```
int          datenbank_lesen     ( mitarbeiter **ma, string filename );
void         liste_loeschen      ( mitarbeiter **ma  );
mitarbeiter* suche_durchwahl     ( mitarbeiter **ma, unsigned int durchwahl );
void         mitarbeiter_aus     ( mitarbeiter  *ma  );
void         alle_mitarbeiter_aus( mitarbeiter **ma  );
```

`datenbank_lesen` Die Funktion erhält einen Zeiger auf den Listenanker (Zeiger auf einen Zeiger) und den Namen der Rohdatendatei. Die gesamte Eingabe und der Aufbau der Liste werden in dieser Funktion abgewickelt.

`liste_loeschen` Diese Funktion löscht die gesamte Liste.

`suche_durchwahl` Diese Funktion sucht nach der übergebenen Durchwahl. Im Erfolgsfall wird ein Zeiger auf das gefundene Element zurückgegeben, sonst ein Nullzeiger.

`mitarbeiter_aus` Diese Funktion übernimmt einen Zeiger auf ein Listenelement und gibt den Inhalt formatiert aus.

`alle_mitarbeiter_aus` Diese Funktion übernimmt einen Zeiger auf einen Listenanker und gibt die gesamte Liste aus. Für die einzelnen Elemente wird die Funktion `alle_mitarbeiter_aus` benutzt.

4. Implementierung eines Stapels als Klasse

4.1. Basisimplementierung eines Stapels als *C++*-Klasse **4 Punkte**

Abbildung 4.1 zeigt die Skizze eines Stapels mit 100 Datenelementen. Der Stapelzeiger `next` zeigt auf den nächste freien Platz.

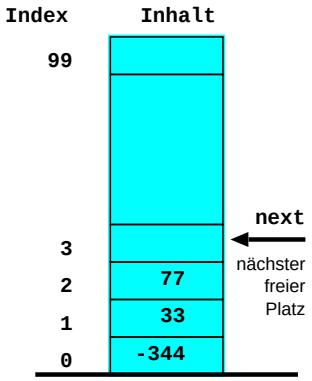


Abbildung 4.1.: Datenstruktur Stapel

Datenelemente des Stapels:

- Feld, 100 Elemente, zur Aufnahme der Daten
- Stapelzeiger (`next`)
- Feldumfang (zur Überwachung; globale `const`-Größe)

Operationen auf dem Stapel:

Initialisieren Zeiger `next` auf den Index 0 stellen.

`push` Element auflegen.

`pop` Oberstes Element vom Stapel nehmen und zurückgeben (falls vorhanden).

`isempty` Feststellen, ob der Stapel leer ist.

`top` Wert des obersten Elementes zurückgeben (falls vorhanden).

`hight` Anzahl der Elemente auf dem Stapel zurückgeben.

`dup` Oberstes Element, falls vorhanden, nochmals auf den Stapel legen (duplizieren).

`exch` Die beiden obersten Elemente vertauschen (falls vorhanden).

Schreiben Sie ein *C++*-Programm (Dateiendung `.cc`), in welchem ein Stapel mit 100 Elementen als Klasse `stack` für die Aufnahme von `int`-Elementen implementiert ist. Die Datenelemente sind als `private`-Elemente, die Operationen (Methoden, Komponentenfunktionen) als `public`-Elemente einzurichten.

Die Implementierung der Komponentenfunktionen soll außerhalb der Klassendefinition stehen und muß deshalb den scope-resolution-Operator (`::`) verwenden:

```
void stack :: push ( int datum ) { . . . }
```

Die Einrichtung eines Stapels geschieht zum Beispiel durch

```
stack stapel1;
```

Der Aufruf einer Komponentenfunktion geschieht zum Beispiel durch

```
stapel1.push( 37 );
```

- Schreiben Sie ein Hauptprogramm, in welchem der Stapel in einer Schleife mit einigen Elementen gefüllt wird.
- Geben Sie die Belegung des Stapels aus und bauen Sie den Stapel wieder ab (mit Ausgabe) um das richtige Funktionieren zu testen.
- Verhindern Sie Unterlauf und Überlauf des Stapels durch entsprechende Vorkehrungen in den Komponentenfunktionen.
- Ersetzen Sie die Initialisierungsfunktion durch einen Konstruktor.
- Verwenden Sie zur Eingabe und Ausgabe `cin` und `cout` (`#include <iostream>`).

4.2. Konstruktor und Destruktor für Stapel beliebiger Größe **3 Punkte**

Ändern Sie die Klassendefinition so ab, daß mit Hilfe eines ersten Konstruktors ein Stapel beliebiger Länge zur Laufzeit angelegt werden kann (Operator `new`). Ein zweiter Konstruktor, ohne Parameter, soll Stapel der Länge 1024 anlegen. Ein Destruktor soll den dynamische belegten Platz eines Stapels freigeben (Operator `delete`). Überprüfen Sie die Wirkung durch folgendes Experiment: Schreiben Sie eine rekursive Funktion

```
void belege (int n)
```

die einen Stapel der Größe 1024 als lokale Datenstruktur besitzt. Die Funktion `belege` ist dabei *keine Methode* der Klasse `stack`. Im Innern wird `n` um 1 vermindert (solange es ungleich Null ist). Anschließend ruft sich die Funktion selbst auf. Wenn n gleich Null ist, wird mit dem Systemaufruf

```
system("free");
```

die augenblickliche Belegung des Hauptspeichers ausgegeben und die Funktion mit `return` beendet. Im Hauptprogramm wird vor und nach dem Aufruf von `belege(1024)` ebenfalls `system(free")` aufgerufen (Prototyp in `cstdlib` . Der Vergleich der Speicherbelegungen sollte etwa 4 MByte ergeben (warum?).

4.3. Stapel-Objekte kopieren **3 Punkte**

Schreiben Sie eine Komponentenfunktion `copy`,

```
void copy ( const Stack &s );
```

die einen Stapel `s` (Aufrufargument) auf die Originalstruktur kopiert. Der zu kopierende Stapel wird als Referenz übergeben. Folgende Maßnahmen sind erforderlich (siehe auch das Skript):

- Zeiger auf das dynamisch belegte Feld des Zielobjektes zwischenspeichern,
- Feld gemäß Längenangabe von `s` im Zielobjekt neu belegen (`new`),
- Feldinhalt kopieren,
- die Datenelemente übernehmen,
- das Originalfeld mittels zwischengespeichertem Zeiger freigeben (`delete`).

Schreiben Sie weiterhin eine Komponentenfunktion `print`,

```
void  print  ( string msg );
```

die einen Stapel etwa in der folgenden Form ausgibt:

```
Stapel stack1

 0 :      1     2     3     4     5     6     7     8     9    10
10 :     11    12    13    14    15    16    17    18    19    20

Stack-Länge : 40            Belegung : 20
```

Die Funktion übernimmt eine beliebige Zeichenkette (Parameter `msg`) und gibt diese als Kopfzeile aus (hier: `Stapel stack1`). Es folgt der Stapelinhalt in Zeilen zu maximal 10 Elementen. Abschließend werden die Feldgröße und die Anzahl der tatsächlich gespeicherten Daten ausgegeben.

Zur Kontrolle wird ein Stapel der Länge 30 und ein Stapel der Länge 20 angelegt und beide bis zur jeweiligen Hälfte mit Testdaten gefüllt. Danach werden die beiden Stapel mit Hilfe von `print` ausgegeben.
Nun wird mit Hilfe von `copy` der längere Stapel auf den kürzeren kopiert und beide Stapel nochmals ausgegeben

Anmerkung: Die Funktion `copy` kann durch die Überladung des Zuweisungsoperators `=` ersetzt werden.

5. Reihenfolge und Wirkung von Methodenaufrufen

Der Zweck dieser Aufgabe ist die Vertiefung des Verständnisses

- von Objekten, Zeigern auf Objekte und Referenzen auf Objekte,
- der Abläufe beim Erzeugen und Entfernen von Objekten, sowie die dazu erforderlichen Methodenaufrufe,
- der Abläufe bei Parameterübergaben (Adreß- und Wertübergabe), sowie
- bei der Rückgabe von Objekten, Zeigern auf Objekten und Referenzen auf Objekte.

Die Aufgabe besteht deshalb im wesentlichen NICHT in der Implementierung des geringen Umfangs der Klasse `Class` und des Testprogramms (deshalb nur 3 Punkte), sondern in Analyse der Ausgaben, die die Anweisungen in der `main`-Funktion erzeugen (7 Punkte). Wenn Sie sich nicht sicher sind, arbeiten Sie deshalb zunächst die Abschnitte aus dem Skript und Ihrer Vorlesungsmitschriften durch, die die Themen Objekte, Referenzen und Zeiger behandeln.

5.1. Implementierung einer Testklasse **3 Punkte**

Implementieren Sie zunächst die Klasse `Class` gemäß Klassendefinition in Liste 5.1 .

Liste 5.1: Klassendefinition

```
1  class Class {
2    public:
3      // ====================  LIFECYCLE     ======================================
4      Class ( int _value=0, string _name="" );    // constructor
5      Class ( const Class &other );               // copy constructor
6      ~Class ();                                  // destructor
7      // ====================  ACCESSORS     ======================================
8      void    print ( );
9      // ====================  MUTATORS      ======================================
10     Class& set_name   ( string _name );
11     Class& set_value  ( int    _value );
12     // ====================  OPERATORS     ======================================
13     Class& operator = ( const Class &other );
14     Class  operator + ( const Class &other );
15
16   private:
17     // ====================  DATA MEMBERS  ======================================
18     int     value;
19     string  name;
20 }; // -----  end of class Class  -----
21
22 Class f ( Class arg );              // prototyp function f
23 Class g ( Class *arg );             // prototyp function g
```

Die Klasse `Class` besitzt zwei Datenelemente, die **`int`**-Größe `value` und den Namen `name`. Folgendes ist zu beachten:

Konstruktor Der Konstruktor übernimmt einen ganzzahligen Wert und weist diesen an `value` zu und gegebenenfalls den Namen des Objektes, der dann an `name` zugewiesen wird.

Kopierkonstruktor Der Kopierkonstruktor weist den `value`-Wert eines anderen Objektes an das eigene Objekt zu. Der Name des Originals wird nicht übernommen.

Destruktor Der Destruktor gibt nur eine Meldung aus (siehe unten).

`operator =` Weist den `value`-Wert eines anderen Objektes an das eigene Objekt zu.

`operator +` Addition zweier `value`-Werte.

`print` Ausgabe des `value`-Wertes und des Objektnamen.

`set_name` Diese Methode übernimmt eine Zeichenkette und weist diese an `name` zu. Zurückgegeben wird eine Referenz auf das eigene Objekt.

`set_value` Diese Methode übernimmt einen ganzzahligen Wert und weist diesen an `value` zu. Zurückgegeben wird eine Referenz auf das eigene Objekt.

Außerhalb der Klasse `Class` soll eine Funktion `f` implementiert werden, die nichts anderes macht, als einen Parameter vom Typ `Class` zu übernehmen (Wertübergabe) und diesen mittels **`return`** wieder zurückzugeben. Weiterhin soll außerhalb der Klasse `Class` eine Funktion `g` implementiert werden, die nichts anderes macht, als einen Parameter vom Typ `Class*` zu übernehmen (Adreßübergabe) und dessen Wert mittels **`return`** wieder zurückzugeben. Alle Methoden enthalten eine Ausgabe der folgenden Art, um die Objekte besser verfolgen zu können:

```
cout  << " ====  xxxxxxx ( this = " << this << " )\n";
cout  << " ====  xxxxxxx ( this = " << this << ", other = " << &other <<  " )\n";
```

Die Zeichenkette `xxxxxxx` wird dabei durch den Methodennamen (`constructor`, `destructor`, `print`, ...) ersetzt.

Schreiben Sie nun ein Hauptprogramm, in dem die folgenden Anweisungen in der hier aufgeführten Reihenfolge stehen.

- **Vereinbarungen.** Vereinbarung von drei Objekten durch unterschiedliche Konstruktoraufrufe:

  ```
  Class x1(11, "x1");
  Class x2 = x1;
  Class x3(33, "x3");
  x2.set_name("x2");
  ```

- **Einfache Zuweisung.** Einfache Zuweisung mit Hilfe des Zuweisungsoperators `=`.
- **1. Addition.** Addition eines Objektes mit einer **`int`**-Konstanten und Zuweisung des Ergebnisses an ein anderes Objekt.
- **2. Addition.** Addition zweier Objekte und Zuweisung des Ergebnisses an ein anderes Objekt.
- **Einfacher Funktionsaufruf.** Funktionsaufruf der Form `x3 = f( x2 )`.
- **Funktionsaufruf in einer Addition.** Funktionsaufruf der Form `x3 = x1 + f( x2 )`.
- **Funktionsaufruf in einer Addition.** Funktionsaufruf der Form `x3 = x1 + g( &x2 )`.
- **Zuweisung einer `int`-Konstanten.** Zuweisung einer Konstanten in der Form `x3 = 234`.
- **Verwenden der Methode `set`.** Zuweisung einer Konstanten in der Form `x3.set(333)`.

Alle Ergebnisse sind durch Ausgaben mit Hilfe der Methode `print` zu überprüfen. Zur besseren Gliederung der Ausgabe ist zwischen diesen Anweisungen jeweils eine Zwischenüberschrift auszugeben, zum Beispiel in der Form

```
cout  << "\n ##########  x3 = x1 + 4                          ########## \n\n";
```

Liste 5.2 zeigt einen Teil der möglichen Ausgaben des Programmes.

Liste 5.2: Mögliche Ausgaben bei Verwendung der Klasse `Class`

```
  ...

##########  x3  = x1 + 4                      ##########

====       constructor ( this = 0x7fff4030c030 )
====        operator + ( this = 0x7fff4030bfe0, other = 0x7fff4030c030 )
====       constructor ( this = 0x7fff4030c020 )
====        operator = ( this = 0x7fff4030bfc0, other = 0x7fff4030c020 )
====        destructor ( this = 0x7fff4030c020 )
====        destructor ( this = 0x7fff4030c030 )
Class-Object x3        ( this = 0x7fff4030bfc0 )  :   value =     15, name =     x3

  ...
```

Aufgrund der ausgegebenen Objektadressen (Wert des jeweiligen **`this`**-Zeigers) können die beteiligten Objekte unterschieden werden.

5.2. Analyse der Programmausgaben **7 Punkte**

Analysieren Sie die Ausgaben. Voraussetzung für die Abnahme ist, dass Sie für jeden Test in der Lage sein, die **Notwendigkeit**, die **Reihenfolge** und die **Wirkung** der einzelnen Methodenaufrufe zu erläutern und den Weg der Objekte nachzuverfolgen.

Beispiel. Ein Beispiel für eine Erklärung, die bei der Abgabe erwartet wird:

```
x3 = x1 + 4
```

Zunächst wird zu Beginn der Konstruktor

```
Class ( int _value=0, string _name="" );
```

aufgerufen, weil die Methode `operator +` eine Referenz auf ein Objekt vom Typ `Class` als Parameter erwartet (rechter Operand der Addition) und deshalb aus dem übergebenen `int` Wert 4 zunächst ein anonymes Objekt vom Typ `Class` erzeugt werden muß (automatische Typumwandlung). Dessen Referenz wird danach als Parameter an die Methode `operator +` übergeben.
Nun erfolgt der Aufruf von `operator +`. Diese Methode enthält einen Konstruktoraufruf in der `return`-Anweisung. Das hier entstandene Objekt wird per Zuweisungsoperator zugewiesen. Danach werden die beiden Hilfsobjekte beseitigt (zwei Destruktoraufrufe).

6. Überladen von Operatoren

6.1. Implementierung der Klasse `Vek3` **2 Punkte**

Die Implementierung der Klasse `Vek3` soll die wichtigsten Grundoperationen der Vektorarithmetik für Vektoren der Länge 3 bereitstellen. Verwenden Sie die Beispielklasse aus der Vorlesung als Grundlage.

Datenelemente: `double`-Variablen `x, y, z`

Methoden:

Methode	Beschreibung
`Vek3`	0 - 3 Komponente werden gesetzt (Ersatzwerte jeweils 0.0)
`null`	alle Komponenten zu 0 setzen
`e_x`	Einheitsvektor in x-Richtung erzeugen
`e_y`	Einheitsvektor in y-Richtung erzeugen
`e_z`	Einheitsvektor in z-Richtung erzeugen
`set_x`	x-Komponente zuweisen
`set_y`	y-Komponente zuweisen
`set_z`	z-Komponente zuweisen
`get_x`	x-Komponente zurückgeben
`get_y`	y-Komponente zurückgeben
`get_z`	z-Komponente zurückgeben
`norm`	Vektor auf die Länge 1.0 normieren
`laenge`	Rückgabe der Vektorlänge
	Die folgenden 3 Operationen verändern den eigenen Vektor und geben eine Referenz auf sich selbst zurück.
`plusgleich`	Vektoraddition
`minusgleich`	Vektorsubtraktion
`mulgleich`	Multiplikation mit einem Skalar
	Die folgenden 3 Operationen erzeugen jeweils einen neuen Vektor und geben diesen zurück; der eigene Vektor bleibt unverändert.
`add`	Vektoraddition
`sub`	Vektorsubtraktion
`mul`	Multiplikation mit einem Skalar
`skalarprodukt`	Skalarprodukt zweier Vektoren
`kreuzprodukt`	Kreuzprodukt zweier Vektoren (siehe unten)
`out`	einfache Ausgabefunktion

- Alle Methoden sind in einem Hauptprogramm geeignet zu testen.
- Verwenden Sie die nachstehende Klassendefinition (Liste 6.1).

Liste 6.1: Definition der Klasse `Vek3`

```
1  // ============================================================================
2  //        Class:  Vek3
3  //  Description:  Vektorarithmetik; die Vektoren haben die feste Länge 3
4  // ============================================================================
5  class Vek3
6  {
7    public:
8
9      // ====================  LIFECYCLE     ======================================
10     Vek3 ( double xx=0.0, double yy=0.0, double zz=0.0 ); // constructor
11
12     // ====================  ACCESSORS     ======================================
13     double  laenge  (  );                 // Vektorlänge zurückgeben
14     double  get_x() { return x; }         // x-Wert zurückgeben
15     double  get_y() { return y; }         // y-Wert zurückgeben
16     double  get_z() { return z; }         // z-Wert zurückgeben
17
18     // die folgenden Operationen erzeugen jeweils einen Ergebnisvektor;
19     // der eigene Vektor bleibt unverändert
20     Vek3  add ( Vek3 &v2 );               // Vektoraddition
21     Vek3  sub ( Vek3 &v2 );               // Vektorsubtraktion
22     Vek3  mul ( double s );               // Vektor x Skalar
23
24     Vek3    kreuzprodukt  ( Vek3 &v2 );
25     double  skalarprodukt ( Vek3 &v2 );   // Skalarprodukt
26     void    out  ( char *name );          // einfache Ausgabefunktion
27
28     // ====================  MUTATORS      ======================================
29     Vek3& null  (  );                     // Komponenten zu Null setzen
30     Vek3& e_x   (  );                     // Einheitsvektor in x-Richtung
31     Vek3& e_y   (  );                     // Einheitsvektor in y-Richtung
32     Vek3& e_z   (  );                     // Einheitsvektor in z-Richtung
33     Vek3& set   ( double xx=0.0,          // alle Komponenten zuweisen
34                   double yy=0.0,
35                   double zz=0.0 );
36     void  set_x ( double xx ) { x = xx; } // x-Wert zuweisen
37     void  set_y ( double yy ) { y = yy; } // y-Wert zuweisen
38     void  set_z ( double zz ) { z = zz; } // z-Wert zuweisen
39     Vek3& norm  (  );                     // Vektor normieren
40
41     // die folgenden Operationen verändern den eigenen Vektor und
42     // geben eine Referenz auf sich selbst zurück
43     Vek3& plusgleich  ( Vek3 &v2 );       // Vektoraddition
44     Vek3& minusgleich ( Vek3 &v2 );       // Vektorsubtraktion
45     Vek3& mulgleich   ( double s );       // Vektor x Skalar
46
47     // ====================  DATA MEMBERS  ======================================
48   private:
49     double  x;
50     double  y;
51     double  z;
52
53 }; // -----  end of class Vek3  -----
```

Berechnung des Kreuzproduktes

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \times \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$$

6.2. Implementierung der Operatorüberladung **2 Punkte**

Die Klasse `Vek3` aus dem vorhergehenden Abschnitt soll nun mit Hilfe der Operatorüberladung so eingerichtet werden, daß die übliche mathematische Schreibweise für elementare Vektoroperationen möglich wird.

Erzeugen Sie eine neue Version der Klasse `Vek3`, in der die Operatoren in den Tabellen 6.1 und 6.2 implementiert sind und testen Sie alle Operatoren.

Operator	1. Operand	Bemerkung
`+`		positives Vorzeichen
`-`		negatives Vorzeichen
`+=`	`const Vek3 &`	Vektoraddition mit Zuweisung
`-=`	`const Vek3 &`	Vektorsubtraktion mit Zuweisung
`*=`	`double`	Streckung mit Zuweisung
`/=`	`double`	Streckung mit Zuweisung
`+`	`const Vek3 &`	Vektoraddition
`-`	`const Vek3 &`	Vektorsubtraktion
`/`	`double`	Streckung

Tabelle 6.1.: Zuweisungsoperatoren der Klasse `Vek3`, Implementierung als Methode

Operator	1. Operator	2. Operand	Bemerkung
`*`	`const Vek3 &`	`double`	Vektor*Skalar
`*`	`double`	`const Vek3 &`	Skalar*Vektor
`*`	`const Vek3 &`	`const Vek3 &`	Skalarprodukt
`%`	`const Vek3 &`	`const Vek3 &`	Kreuzprodukt
`<<`	`ostream &`	`const Vek3 &`	Ausgabe

Tabelle 6.2.: `friend`-Funktionen der Klasse `Vek3`

6.3. Mittelpunkt eines Kreises in beliebiger Lage

In Maschinensteuerungen (zum Beispiel in Robotersteuerungen) werden Bewegungsbahnen in der Regel durch Stützpunktfolgen dargestellt. Ein Stützpunkt kann als Ortsvektor im dreidimensionalen

Raum aufgefaßt werden. Aufeinanderfolgenden Stützpunkten wird eine Bahnform zugeordnet (Teilbahn, Bahnabschnitt). Zwei Stützpunkte sind hinreichend für die Darstellung einer Geraden, drei Stützpunkte für die Darstellung eines Kreisbogens und so weiter. Zur Berechnung aller zur Bewegungsführung auf einer Teilbahn erforderlichen Informationen, wie zum Beispiel der Bahnlänge, muß aus den Stützpunkten zunächst eine mathematische (zum Beispiel eine vektorielle) Beschreibung der Teilbahn gewonnen werden.

Ein Kreis in beliebiger Lage (im dreidimensionalen Raum) ist durch drei nicht kollineare Kreispunkte P_1, P_2, P_3 vollständig bestimmt. Diese Kreispunkte können durch ihre Ortsvektoren $\vec{p}_1$, $\vec{p}_2$ und $\vec{p}_3$ angegeben werden. Die Kreispunkte legen gleichzeitig die Ebene fest in der der Kreis liegt.

Zur Bestimmung des Kreismittelpunktes M werden die Mittelsenkrechten der Strecken $\overline{P_1P_2}$ und $\overline{P_2P_3}$ in der Kreisebene geschnitten (Abbildung 6.1).

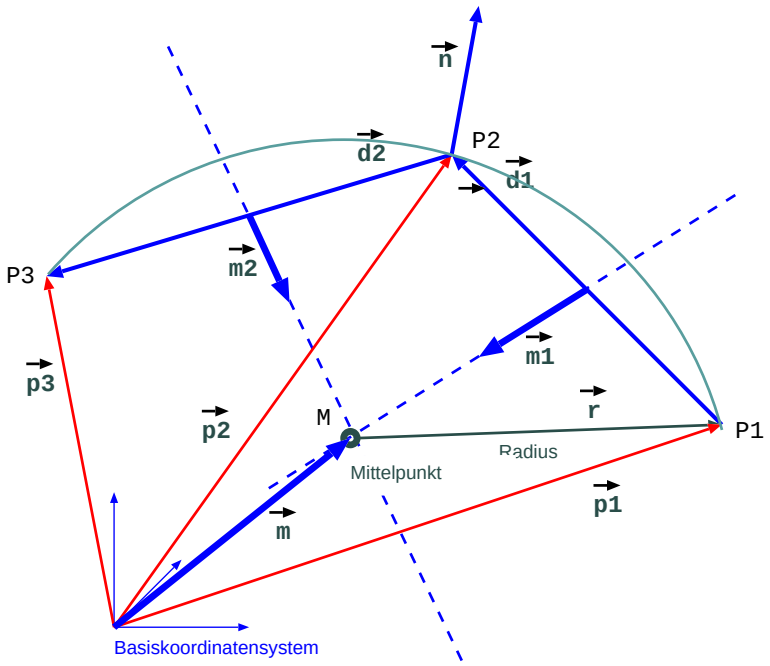


Abbildung 6.1.: Ermittlung des Mittelpunktes eines Kreises in beliebiger Lage

Diese Aufgabe kann vektoriell gelöst werden. Der Ortsvektor $\vec{m}$ kann durch die Auflösung der folgenden Vektorgleichung bestimmt werden.

$$\vec{d_1} = \vec{p_2} - \vec{p_1} \tag{6.1}$$
$$\vec{d_2} = \vec{p_3} - \vec{p_2} \tag{6.2}$$
$$\vec{n} = \vec{d_1} \times \vec{d_2} \tag{6.3}$$
$$\vec{m}_1 = \vec{d_1} \times \vec{n} \tag{6.4}$$
$$\vec{m}_2 = \vec{d_2} \times \vec{n} \tag{6.5}$$
$$\vec{p_1} + \frac{1}{2}\vec{d_1} + \lambda \cdot \vec{m}_1 = \vec{p_2} + \frac{1}{2}\vec{d_2} + \mu \cdot \vec{m}_2 \tag{6.6}$$

Durch Umstellung von Gleichung 6.6 ergibt sich die Gleichung

$$\lambda \cdot \vec{m}_1 - \mu \cdot \vec{m}_2 = \vec{p}_2 - \vec{p}_1 + \frac{1}{2}\vec{d}_2 - \frac{1}{2}\vec{d}_1$$

und daraus durch Vereinfachung der rechten Seite

$$\lambda \cdot \vec{m}_1 - \mu \cdot \vec{m}_2 = \frac{1}{2}(\vec{p}_3 - \vec{p}_1) \tag{6.7}$$

Durch Hinzufügen der (eigentlich nicht notwendigen) Komponente ν in Normalenrichtung der Kreisebene ergibt sich eine Vektorgleichung mit den drei Unbekannten λ, μ und ν :

$$\lambda \cdot \vec{m}_1 \quad - \quad \mu \cdot \vec{m}_2 \quad + \quad \nu \cdot \vec{n} \quad = \quad \frac{1}{2}(\vec{p}_3 - \vec{p}_1) \tag{6.8}$$

$$[\ \vec{m}_1\ ,\ -\vec{m}_2\ ,\ \vec{n}\ ] \cdot \begin{pmatrix} \lambda \\ \mu \\ \nu \end{pmatrix} \quad = \quad \frac{1}{2}(\vec{p}_3 - \vec{p}_1) \tag{6.9}$$

Gleichung 6.8 stellt durch diesen Kunstgriff ein lineares Gleichungssystem mit drei Unbekannten dar und kann somit in der Matrix-Vektor-Schreibweise 6.9 dargestellt werden. Die Koeffizientenmatrix $[\vec{m}_1, -\vec{m}_2, \vec{n}]$ ist aus Spaltenvektoren zusammengesetzt. Der Wert der Komponenten ν des Lösungsvektors wird sich zwangsläufig zu Null ergeben.
Die Bestimmung der Lösung (Cramersche Regel) einer derartigen Gleichung in Matrix-Vektor-Darstellung mit drei Unbekannten ist durch die Bestimmungsgleichungen 6.10 gegeben. Die Größen D_0, D_1, D_2 und D_3 sind Determinanten, deren Wert nach der Sarrusschen Regel (hier durch Vektorprodukte ausgedrückt) berechnet wird. Dabei stellt $\vec{rs}$ den Vektor der rechten Seiten dar und $\vec{x}$ den Vektor der gesuchten Lösungen:

$$\begin{aligned} [\ \vec{s}_1\ ,\ \vec{s}_2\ ,\ \vec{s}_3\ ] \cdot \vec{x} &= \vec{rs} \\ D_0 &= (\vec{s}_1 \times \vec{s}_2) \cdot \vec{s}_3 \\ D_1 &= (\vec{rs} \times \vec{s}_2) \cdot \vec{s}_3 \\ D_2 &= (\vec{s}_1 \times \vec{rs}) \cdot \vec{s}_3 \\ D_3 &= (\vec{s}_1 \times \vec{s}_2) \cdot \vec{rs} \end{aligned}$$

$$x_1 = \frac{D_1}{D_0} \quad , \quad x_2 = \frac{D_2}{D_0} \quad , \quad x_3 = \frac{D_3}{D_0} \tag{6.10}$$

Wenn das Element λ des Lösungsvektors in Gleichung 6.9 bestimmt ist, dann kann mit Hilfe der linken Seite von Gleichung 6.6 der Ortsvektor $\vec{m}$ des Mittelpunktes bestimmt werden:

$$\vec{m} = \vec{p}_1 + \frac{1}{2}\vec{d}_1 + \lambda \cdot \vec{m}_1 \tag{6.11}$$

Berechnen Sie mit Hilfe der Klasse `Vek3` nach dem angegebenen Verfahren den Mittelpunkt des Kreises (Ortsvektor $\vec{m}$), der durch drei Raumpunkte gegeben ist (siehe auch Abschnitt 6.5) :

Berechnen Sie anschließend zur Kontrolle die Abstände vom ermittelten Mittelpunkt zu den drei Stützpunkten (Radius):

$$\begin{aligned} l_1 &= |\vec{p}_1 - \vec{m}| \\ l_2 &= |\vec{p}_2 - \vec{m}| \\ l_3 &= |\vec{p}_3 - \vec{m}| \end{aligned}$$

6.4. Berechnung der Bogenlänge

Berechnen Sie abschließend die Länge des Kreisbogens b zwischen den Stützpunkten nach folgender Vorschrift:

$$\begin{aligned} \vec{v}_1 &= \| \vec{p}_1 - \vec{p}_2 \| \\ \vec{v}_2 &= \| \vec{p}_3 - \vec{p}_2 \| \\ \frac{\alpha}{2} &= \pi - \arccos(\vec{v}_1 \cdot \vec{v}_2) \\ r &= \frac{|\vec{p}_3 - \vec{p}_1|}{2 \sin \frac{\alpha}{2}} \end{aligned}$$

$$b = \alpha \cdot r \tag{6.12}$$

$\| \cdot \|$ stellt die Normierung eines Vektors dar. Der Vektor hat nach der Normierung die Länge 1. Der Winkel α ist im Bogenmaß zu verwenden. Für die trigonometrische Funktion *arccos* wird die Bibliotheksfunktion `acos` verwendet (`cmath`).

6.5. Implementierung und Test **6 Punkte**

Die Lösung einer Gleichung nach (6.10), die Mittelpunktberechnung nach (6.11) und die Berechnung der Bogenlänge nach (6.12) sollen in Funktionen mit folgenden Prototypen implementiert werden:

```
Vek3    lingl3           ( Vek3 s1, Vek3 s2, Vek3 s3, Vek3 rs );
Vek3    kreismittelpunkt ( Vek3 p1, Vek3 p2, Vek3 p3 );
double  kreisbogenlaenge ( Vek3 p1, Vek3 p2, Vek3 p3 );
```

Die Funktion `lingl3` berechnet die Lösung eines vektoriell dargestellten linearen Gleichungssystems nach der Sarrusschen Regel (Gleichung 6.10). Wenn der Betrag der Größe D_0 sehr klein oder gleich Null ist, dann sind die Stützpunkte kollinear und die Gleichung ist nicht lösbar. Das Programm soll in diesem Fall mit Hilfe der Funktion `assert` abgebrochen werden.
Die Konstante π kann bei Verwendung von GNU-C++ im Programmtext durch das Makro `M_PI` dargestellt werden (`include`-Datei `cmath`). Eine vollständig portable Möglichkeit stellt die Verwendung einer globalen Variablen dar:

```
static const double Pi = 4.0*atan(1.0);
```

Die drei Funktionen sind völlig eigenständig, also weder Methoden noch **friend**-Funktionen der Klasse Vek3.

Testfälle

Bestimmen Sie Mittelpunkt m_1, Radius r_1 und Bogenlänge b_1 für den Kreis mit folgenden Stützpunkten:

$$\vec{p}_1 = \begin{pmatrix} +1 \\ +1 \\ 0 \end{pmatrix}, \quad \vec{p}_2 = \begin{pmatrix} 0 \\ +2 \\ 0 \end{pmatrix}, \quad \vec{p}_3 = \begin{pmatrix} -1 \\ +1 \\ 0 \end{pmatrix}$$

$$\vec{m}_1 = \begin{pmatrix} \_\_\_\_ \\ \_\_\_\_ \\ \_\_\_\_ \end{pmatrix}, \quad r_1 = \_\_\_\_, \quad b_1 = \_\_\_\_$$

Bestimmen Sie für diesen ersten Testfall den Mittelpunkt, den Radius und die Bogenlänge des Bogens auf Papier und legen Sie diese Ergebnisse mit dem Programm vor.

Bestimmen Sie Mittelpunkt m_i, Radius r_i und Bogenlänge b_i für die Kreise mit folgenden Stützpunkten:

$$\vec{p}_4 = \begin{pmatrix} +2 \\ -1 \\ -1 \end{pmatrix}, \quad \vec{p}_5 = \begin{pmatrix} -2 \\ -2 \\ -1 \end{pmatrix}, \quad \vec{p}_6 = \begin{pmatrix} 0 \\ +1 \\ +4 \end{pmatrix}$$

$$\vec{m}_2 = \begin{pmatrix} \_\_\_\_ \\ \_\_\_\_ \\ \_\_\_\_ \end{pmatrix}, \quad r_2 = \_\_\_\_, \quad b_2 = \_\_\_\_$$

$$\vec{p}_7 = \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix}, \quad \vec{p}_8 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad \vec{p}_9 = \begin{pmatrix} -1 \\ -1 \\ +1 \end{pmatrix}$$

$$\vec{m}_3 = \begin{pmatrix} \_\_\_\_ \\ \_\_\_\_ \\ \_\_\_\_ \end{pmatrix}, \quad r_3 = \_\_\_\_, \quad b_3 = \_\_\_\_$$

Weiterhin sind zur Kontrolle für jeden Kreis die drei Abstände vom Mittelpunkt zu den jeweiligen Stützpunkten (=Radius) auszugeben.

Führen Sie die Berechnung für die Testfälle durch und geben Sie die Ergebnisse aus.

7. Ausnahmebehandlung

Liste 7.1: Die Klasse `Datum`

```
1  // #####   TYPE DEFINITIONS  -  LOCAL TO THIS SOURCE FILE   ######################
2
3  typedef unsigned int  uint;
4
5  // ============================================================================
6  //        Class:  Datum
7  //  Description:  Handhabung von Kalenderdaten (Minimalversion)
8  // ============================================================================
9  class Datum
10 {
11   public:
12
13     // ===================  LIFECYCLE   ========================================
14     Datum ()  {};
15     Datum ( uint tg, uint mnt, uint jhr );
16
17     // ====================  ACCESSORS     ======================================
18     uint  getTag    () const  { return tag;   }
19     uint  getMonat  () const  { return monat; }
20     uint  getJahr   () const  { return jahr;  }
21
22   private:
23     bool  monatOk   ();
24     bool  tagOk     ();
25     bool  schaltjahr();
26
27     // ====================  DATA MEMBERS  ======================================
28   private:
29     uint  tag;
30     uint  monat;
31     uint  jahr;
32
33 }; // -----  end of class  Datum  -----
```

7.1. Implementierung der Klasse `Datum` **5 Punkte**

Implementieren Sie eine Klasse `Datum` gemäß Schnittstellenbeschreibungen in Liste 7.1. Die Klasse besitzt für interne Prüfungen drei **`private`**-Methoden:

`monatOk` Diese Methode gibt **`true`** zurück, wenn die Monatsnummer zwischen 1 und 12 liegt.

`tagOk` Diese Methode gibt **`true`** zurück, wenn die Tagesangabe gültig ist. Hierzu wird geprüft, ob die Tagesnummer zwischen 1 und dem jeweils Letzten des Monats liegt. Schaltjahre sind beim Februar natürlich zu berücksichtigen. Dazu wird die Methode `schaltjahr` verwendet.

`schaltjahr` Prüft, ob ein Schaltjahr vorliegt. Schaltjahre sind alle Jahre, die ohne Rest durch 4 teilbar sind. Alle Jahreszahlen, die ohne Rest durch 100 teilbar sind (zum Beispiel 1800, 1900), sind keine Schaltjahre. Alle Jahreszahlen, die ohne Rest durch 400 teilbar sind (zum Beispiel 1600, 2000), sind jedoch wieder Schaltjahre.

`Datum ( uint tg, uint mnt, uint jhr )` Der zweite Konstruktor übernimmt drei Datumsanteile und prüft die Gültigkeit der Tages- und Monatsangabe. Im Fehlerfall soll zunächst eine Textausgabe erfolgen.

Erstellen Sie weiterhin eine Funktion (keine Methode)

```
ostream & operator << ( ostream & os, const Datum & obj );
```

die mit Hilfe der get-Methoden eine einfache Datumsausgabe zu Kontrollzwecken ermöglicht.

In einem Hauptprogramm wird ein Feld von Objekten des Typs `Datum` mit 10^6 Elementen angelegt. Aus der Datei `datumsliste.dat` (Archiv auf der Modul-Seite) werden alle Elemente ausgelesen und der Reihe nach in dem Feld abgelegt. Die Datei enthält einige fehlerhaften Angaben, die zu Fehlermeldungen des Konstruktors führen. Am Programmende wird die Anzahl der eingelesenen Datensätze zur Kontrolle ausgegeben.

7.2. Verwendung von Fehlerklassen **5 Punkte**

Erstellen Sie gemäß Vorlesungsskript (Abschnitt 5.3.1) eine Fehlerklasse `FehlerDatum` als Basisklasse für weitere Fehlerklassen.
Leiten Sie davon zwei weitere Fehlerklassen, `FehlerTag` und `FehlerMonat`, ab. Diese beiden Klassen besitzen nur einen Konstruktor, der jeweils den Basisklassenkonstruktor zur Übernahme eines Meldungstextes aufruft. Beide Klassen dienen nur dazu, eine genauere Unterscheidung von Fehlerursachen zu ermöglichen.

Entfernen Sie nun im zweiten Konstruktor der Klasse `Datum` die Textausgabe und werfen Sie stattdessen im Fehlerfall (Tages- oder Monatsfehler) ein Fehlerobjekt mit einem kurzen Text aus (zum Beispiel `"Falsche Tagessangabe"` und `"Falsche Monatsangabe"`).

In der Methode `Datum::tagOk` muß der Monat ebenfalls überprüft werden, da er zur Feststellung des Monatsletzten verwendet werden muß. Sollte der Monat hier falsch sein, wird ebenfalls ein Fehlerobjekt ausgeworfen.

Im Hauptprogramm wird nun der Konstruktoraufruf (Erzeugung der Feldelemente in der Einleseschleife) durch eine **`try-catch`**-Anweisung überwacht. Getrennt aufgefangen werden die Fehlerobjekte vom Typ `FehlerTag` und `FehlerMonat`. Als Fehlerbehandlung werden die Nummer des fehlerhaften Datensatzes, die in den Fehlerobjekten enthaltene Meldung und die fehlerhafte Angabe ausgegeben:

```
Satz Nr.    15 : Falsche Tagessangabe : 32
Satz Nr.    19 : Falsche Monatsangabe :  0
Satz Nr.  1995 : Falsche Tagessangabe : 34
Satz Nr.  1996 : Falsche Monatsangabe : 13

2000 von 2004 Datumsangaben übernommen
```

8. Verwendung von Templates

Ringlisten werden verwendet, um die letzten n Werte in einer Reihe sich ändernder oder sich vermehrender Größen zu verwalten. Beispiele sind die Ermöglichung des Zugriffes auf die letzten n Meßwerte einer fortlaufenden Messung oder auf die letzten n Meldungen, Eingaben und Ausgaben, die in einer Terminal-Sitzung auftreten (die sogenannte history).
Diese Praktikumsaufgabe besteht aus zwei Teilen. Im ersten Teil soll zunächst eine Ringliste für **`double`**-Größen geschrieben und getestet werden. Im zweiten Teil soll auf dieser Grundlage eine generische Ringlisten-Klasse entwickelt werden, die für beliebige Objekte geeignet ist.

8.1. Implementierung einer Ringliste **5 Punkte**

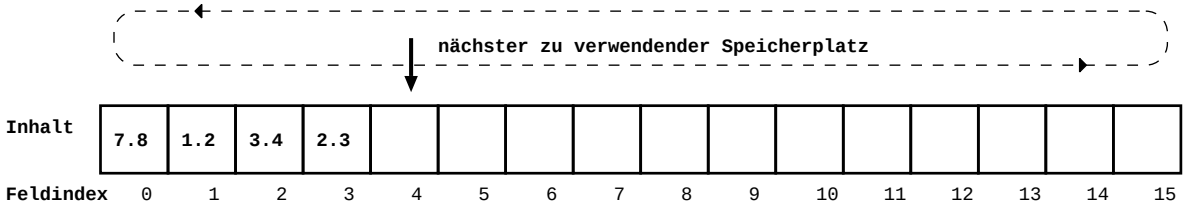


Abbildung 8.1.: Ringliste der Länge 16

Abbildung 8.1 zeigt die Arbeitsweise einer Ringliste der Länge 16. Die Listenverwaltung hält einen Index vor, der die Indexposition des nächsten zu verwendenden Platzes enthält (anfänglich Null). Kommt ein neuer Wert hinzu, wird er unter diesem Index gespeichert und der Index anschließend um 1 erhöht. Wenn das Listenende erreicht ist, wird der Index auf 0 zurückgesetzt. Der nächste Wert der gespeichert wird, überschreibt dann bereits den Wert, der augenblicklich auf dieser Position steht. Auf diese Weise stehen immer die 16 letzten Werte zur Verfügung.

Liste 8.1 zeigt die Definition einer Klasse `Ringliste`, die ein Minimum an Methoden enthält. Die Klasse ist für Listen beliebiger Länge ausgelegt. Der Platz für die eigentliche Liste, in dieser Teilaufgabe ein **`double`**-Feld, wird im Konstruktor dynamisch beschafft. Die Methode `add()` speichert einen neuen **`double`**-Wert in der Liste, die Methode `last()` gibt den zuletzt gespeicherten Wert zurück. Die Methode `dump()` dient nur zu Testzwecken. Sie gibt die gesamte Liste in einer einfachen Tabellenform aus (zum Beispiel in Zeilen zu jeweils 8 Werten).

Implementierung Sie diese Klasse und testen Sie alle Methoden in einem entsprechenden Hauptprogramm. In der Methoden `add()` kann der Zugriff auf eine leere Liste mit `assert()` überwacht werden.

Liste 8.1: Die Klasse `Ringliste`

```
1  typedef unsigned int  uint;
2
3  static const  uint  RinglisteErsatzLaenge = 20; // Ersatzwert für die Listenlänge
4
5  class Ringliste
6  {
7    public:
8
9      // ====================  LIFECYCLE     ======================================
10     Ringliste ( uint lng = RinglisteErsatzLaenge ); // constructor
11     Ringliste ( const Ringliste &other );           // copy constructor
12     ~Ringliste ();                                  // destructor
13
14     // ====================  ACCESSORS     ======================================
15     double  last ( );                          // letzten Wert zurückgeben
16     void    dump ( );                          // gesamte Ringliste ausgeben (Test)
17
18     // ====================  MUTATORS      ======================================
19     Ringliste& add ( double wert );            // neuen Wert aufnehmen
20
21     // ====================  OPERATORS     ======================================
22     const Ringliste& operator = ( const Ringliste &other );  // assignment operator
23
24     // ====================  DATA MEMBERS  ======================================
25   private:
26     uint     laenge;                           // Listenlänge
27     int      index;                            // nächste Position
28     double  *element;                          // Zeiger auf das Feld der Listenelem.
29
30 }; // -----  end of class Ringliste  -----
```

8.2. Generische Ringliste **5 Punkte**

Verwenden Sie in dieser Teilaufgabe die Lösung aus Abschnitt 8.1 als Ausgangspunkt für die Erstellung einer generischen Klasse

```
template < class T > class Ringliste
{
  // ...
}
```

Alle **double**-Größen aus der Lösung aus Abschnitt 8.1 müssen durch die allgemeine Typbezeichnung `T` ersetzt werden. Für die Klasse und die Methoden sind jetzt natürlich die entsprechenden **template**-Schreibweisen zu verwenden.

Prüfen Sie in einem ersten Schritt, ob das entsprechend übernommene und angepaßte Testprogramm aus Abschnitt 8.1 bei der Verwendung des Datentyps **double** die selben Ergebnisse liefert.

In einem zweiten Test soll der Datentyp `string` verwendet werden. Prüfen Sie das einwandfreie Verhalten Ihrer Lösung mit kurzen Zeichenketten und geben Sie die Listen zur Überprüfung jeweils mit `dump()` aus.

Die Anzahl der gespeicherten Werte muß größer als die Listenlänge sein, um die richtige Arbeitsweise der Indexfortschaltung erkennen zu können.

A. Programmdokumentation

Programme sollen grundsätzlich richtig, zweckmäßig, vollständig, verständlich und leicht wartbar sein. Diese Ziele werden unter anderem durch eine Reihe von Merkmalen und Eigenschaften sichergestellt, die Bestandteile des Programmierstils sind. Dazu gehören:

- die Form der Kommentierung
- die Art der Formatierung
- die Art der Benennung von Bezeichnern
- der Aufbau eines Moduls

Die praktische Umsetzung kann, besonders bei größeren Projekten, nicht allein dem persönlichen Geschmack des einzelnen Programmierers überlassen werden. Es ist üblich, die notwendigen Vorgaben in firmen- oder projektbezogenen Programmierrichtlinien niederzulegen und alle Beteiligten auf die Einhaltung dieser Richtlinien zu verpflichten.
Die durchgängige Einhaltung eines festgelegten Programmierstils erlaubt dann das automatische Herausziehen von Kommentaren zur Erzeugung von Projektdokumentationen, sowie das automatische Durchsuchen großer Quellcode-Bäumen. Dazu sind eine Reihe gebräuchlicher Werkzeuge vorhanden.
Auch im Rahmen dieses Praktikums muß für jedes Programm ein Mindestumfang an Programmdokumentation vorhanden sein. Dieser wird in den nachfolgenden Abschnitten beschrieben. Grundsätzlich gilt folgendes:

- Kommentierungssprache ist in der Regel **Deutsch**. Die **Rechtschreibung** ist zu beachten!
- **Personenangaben** sind vollständig aufzuführen (Vor- *und* Nachname, gegebenenfalls Matrikelnummer). Die Verwendung von Spitznamen, Phantasiebezeichnungen, Schimpfwörtern und ähnliches ist zu unterlassen.
- In *C++*-Dateien werden *C++*-Kommentare verwendet (`//...`).

A.1. Dateibeschreibung

Jede Datei muß an ihrem Anfang eine Dateibeschreibung in Form eines Kommentares enthalten. Wenn die Datei in einem Editor geöffnet wird ist dieser Kommentar sofort sichtbar. Ein Kommentar, wie in Liste A.1 dargestellt, sollte als Schablone in einer Datei vorliegen, die bei Bedarf einkopiert wird. Damit wird sichergestellt, daß die Dateibeschreibungen stets die gleiche Form besitzen.

Manche Editoren und integrierte Entwicklungsumgebungen (sogenannte IDEs) können Dateibeschreibungen erzeugen oder eine vorgegebene Schablone einsetzen.

Liste A.1: Kopfkommentar als Dateibeschreibung

```
1  // ============================================================================
2  //
3  //       Filename:  p-1-1.cc
4  //
5  //    Description:  Praktikum Programmierung mit C++ 2
6  //                  Blatt 1 / Aufgabe 1.1
7  //                  Zweidimensionale Felder, Bilddatei lesen und speichern
8  //
9  //        Version:  1.0
10 //        Created:  23.05.2007 13:35:58 CEST
11 //       Revision:  none
12 //       Compiler:  g++
13 //
14 //         Author:  Dr.-Ing. Fritz Mehner (Mn), mehner@fh-swf.de
15 //        Company:  Fachhochschule Südwestfalen, Iserlohn
16 //
17 // ============================================================================
```

A.2. Abschnittskommentare und Zeilenendkommentare

Kurze Funktionen und kurze, logisch zusammenhängende Programmabschnitte werden zur optischen Gliederung und zur Erläuterung des Inhaltes mit Abschnittskommentaren versehen (Liste A.2, Zeilen 1-3, 10-12, 21-23, 32-34). Bei Bedarf kann der eigentliche Kommentar natürlich mehrzeilig sein.

Liste A.2: Abschnittskommentare

```
1  //---------------------------------------------------------------------
2  //  Hauptprogramm
3  //---------------------------------------------------------------------
4    int
5  main ( int argc, char *argv[] )
6  {
7    int wert;                                    // jeweils eingelesener Wert
8    int summe = 0;                               // Summe der positiven Werte
9
10   //-------------------------------------------------------------------
11   //  Eingabedatei öffnen
12   //-------------------------------------------------------------------
13   char *ifs_file_name = "ea-2-in.dat";         // Name der Eingabedatei
14   ifstream  ifs;                               // ifstream-Objekt erzeugen
15   ifs.open(ifs_file_name);                     // Eingabedatei öffnen
16   if (!ifs) {
17     cerr << "\nERROR : failed to open input file " << ifs_file_name << endl;
18     exit (1);
19   }
20
21   //-------------------------------------------------------------------
22   //  Ausgabedatei öffnen
23   //-------------------------------------------------------------------
24   char *ofs_file_name = "ea-2-out.dat";        // Name der Ausgabedatei
25   ofstream  ofs;                               // ofstream-Objekt erzeugen
26   ofs.open(ofs_file_name);                     // Ausgabedatei öffnen
27   if (!ofs) {
```

```
28      cerr << "\nERROR : failed to open output file " << ofs_file_name << endl;
29      exit (2);
30    }
31
32    //--------------------------------------------------------------------
33    //  Eingabedatei lesen, Ausgabedatei schreiben
34    //--------------------------------------------------------------------
35    while( ifs >> wert ) {                          // unbekannte Dateilänge
36      if ( wert > 0.0 ) {
37        summe += wert;
38        ofs << summe << endl;                       // Wert in die Ausgabedatei
39      }
40    }
41
42    ifs.close();                                    // Eingabedatei schließen
43    ofs.close();                                    // Ausgabedatei schließen
44
45    return 0;
46  }       // ----------  end of function main  ----------
```

Die Abschnittskommentare (Blockkommentare) richten sich in ihrer Einrücktiefe stets nach der Schachtelungstiefe des zu kommentierenden Codes. Der Kopf des Hauptprogrammes in Liste A.2 hat die Schachtelungstiefe 0, der Kopfkommentar (Zeilen 1-3) ist dementsprechend nicht eingerückt. Der Abschnittskommentar über der Öffnung der Eingabedatei (Zeilen 10-12) hat die Schachtelungstiefe 1 und ist dementsprechend eine Tabulatorweite eingerückt (hier 2 Zeichenpositionen).
Kurze Erläuterungen werden als Zeilenendkommentare an die Codezeile angehängt (Liste A.2, Zeile 7, 8, 13-15, und so weiter). Wenn möglich, sollten Zeilenendkommentare (abschnittsweise) jeweils in der selben Spalte beginnen. Zu kommentieren sind

- **Variablen** (Bedeutung und Verwendung; der Kommentar steht bei der Vereinbarung).
- **Funktionen** (mindestens: Zweck, Parameter und Rückgabewert; bei sehr kurzen oder einfachen Funktionen reicht ein Kopfkommentar).
- **Klassen** (mindestens: Zweck der Klasse, kurze Beschreibung der Datenkomponenten und Methoden; die Implementierungen der Methoden sind wie Funktionen zu behandeln).

Kommentare sollen kurz, treffend und aussagekräftig sein. Der Kommentartext `Schleifenzähler` stellt eine sinnvolle Aussage dar:

```
int i, j;                       // Schleifenzähler
```

Der Kommentartext `Hilfsvariable` ist überflüssig. Jede derart vereinbarte Variable ist eine Hilfsvariable:

```
int i, j;                       // Hilfsvariable
```

Die Gesamtlänge einer Zeile (mit oder ohne Kommentar) sollte 90 Zeichen nicht überschreiten. Damit kann das Suchen in Querrichtung vermieden werden und die Datei bleibt bei üblichen Papierbreiten ohne abgeschnittene Zeilenenden abdruckbar.

A.3. Einrückung und Schachtelung

Die Einrückung von Programmkonstrukten muß in der Regel mit der Schachtelungstiefe dieser Konstrukte übereinstimmen.

Liste A.3: Abschnittskommentare (Ausschnitt aus Liste A.2)

```
32  //---------------------------------------------------------------------
33  //  Eingabedatei lesen, Ausgabedatei schreiben
34  //---------------------------------------------------------------------
35  while( ifs >> wert ) {                          // unbekannte Dateilänge
36    if ( wert > 0.0 ) {
37      summe += wert;
38      ofs << summe << endl;                       // Wert in die Ausgabedatei
39    }
40  }
```

- Die `while`-Anweisung (Liste A.3, Zeilen 35-40) besitzt die Schachtelungstiefe 1.
- Die `if`-Anweisung in den Zeilen 36 bis 39 gehört zum Rumpf der umgebenden `while`-Schleife und besitzen deshalb die Schachtelungstiefe 2.
- Die Addition in Zeile 37 und die Ausgabe in Zeile 38 gehören zum Rumpf der `if`-Anweisung und besitzen deshalb die Schachtelungstiefe 3.

Einrückung ist die wichtigste Maßnahme zur Erzeugung lesbarer Programme und ist daher *zwingend* anzuwenden! Unter anderem werden Strukturfehler die sich in der Schachtelung widerspiegeln sofort erkennbar.

Gute Programmiereditoren unterstützen die Einrückung beim Einfügen. Zur Formatierung vorhandener Quellen gibt eine ganze Reihe von Formatierungsprogrammen (engl. *beautifier*), die eine Vielzahl von Darstellungsoptionen unterstützen. Ein bekanntes Beispiel ist das Programm `indent` [1].

[1] `http://www.gnu.org/software/indent/`

B. Fehlersuche

B.1. Die häufigsten Anfängerfehler

Die folgende Liste enthält Programmierfehler, die erfahrungsgemäß häufig von Programmieranfängern gemacht werden. Es wird dringend empfohlen, jedes Programm auf diese Fehler hin durchzusehen. Das sollte auch dann geschehen, wenn das Programm anscheinend die richtigen Ergebnisse liefert.

Nr.	Fehler	Beschreibung
1	**Variable besitzt keinen Anfangswert**	Variablen vom Speichertyp `auto` werden **nicht** mit Null initialisiert und müssen immer einen Anfangswert erhalten, wenn ihr erster Gebrauch auf der rechten Seite einer Zuweisung oder in einer Parameterliste stattfindet.
2	**Ganzzahlige Division**	Verwendung der ganzzahliger Division anstatt der reellen Division, also zum Beispiel `laenge *= 1/2;` anstatt `laenge *= 0.5;` . Der Faktor `1/2` im ersten Beispiel wird auf Grund der Regeln für die ganzzahlige Division zu Null berechnet !
3	**Schleife läuft zu kurz oder zu weit**	Eine Schleife hat einen Durchlauf zuviel oder zu wenig, weil die Abbruchbedingung nicht richtig ist (zum Beispiel `i<=n` anstatt `i<n`) oder weil die Initialisierung der Steuervariablen falsch ist (zum Beispiel `i=1` anstatt `i=0`).
4	**Zugriff über die Feldgrenzen hinaus**	Zugriff auf einen Feldindex, der nicht vorhanden ist.
5	**Feld zu klein**	Die Größenangabe eines Feldes in einer Vereinbarung bezeichnet die Elementanzahl, **nicht** den letzten Index ! Bei Zeichenpuffern (`char`-Felder) muß meist ein zusätzliches Element für das Abschlußzeichen (die binäre Null) vorgesehen werden.
6	**Zuweisung statt Vergleich**	Der Operator zur Feststellung der Gleichheit zweier Werte besteht aus zwei Gleichheitszeichen (`==`). Die Bedingung in `if ( n = n_alt )` stellt eine Zuweisung dar: der Wert der Variablen `n` wird überschrieben und anschließend als Wahrheitswert verwendet !

7	**Strichpunkt hinter dem Kopf einer Schleife oder Bedingung**	Ein Strichpunkt hinter einem Schleifenkopf oder hinter dem Kopf einer Verzweigung (`if`, `switch`) trennt den nachfolgenden Rumpf ab; der Kopf wird dadurch zu einer selbständigen, abgeschlossenen Anweisung. Wenn die Bedingung wahr ist wird die Schleife dadurch zur Endlosschleife:
		`while ( n <= n_max );`
		`{ ... }`

B.2. Der Debugger DDD

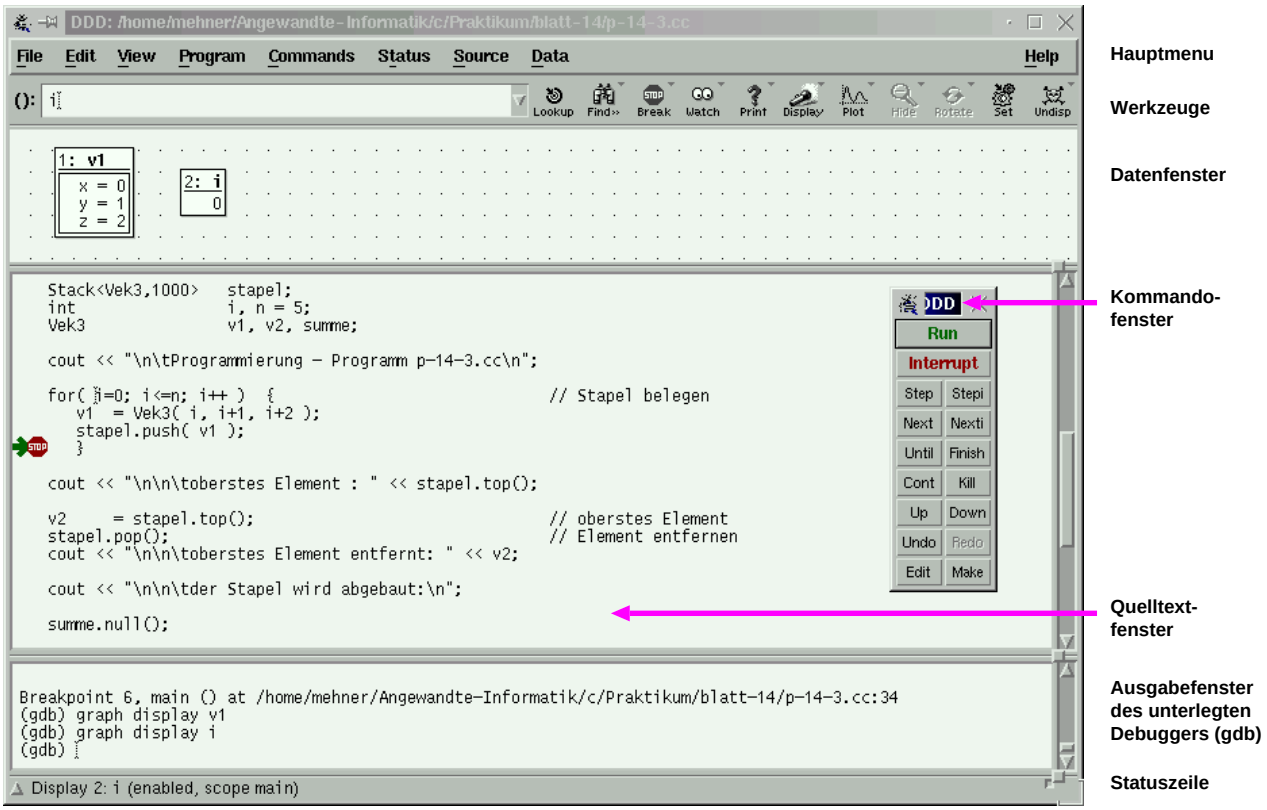
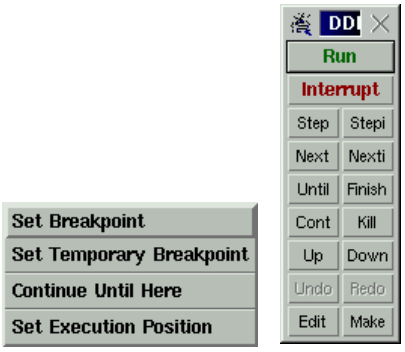


Abbildung B.1.: Der Debugger DDD

Zum Testen von Programmen kann der Debugger DDD[1] verwendet werden. Das zu testende Programm muß dazu als ausführbare Datei vorliegen, das heißt es muß übersetzt und gebunden sein und enthält damit keine syntaktischen Fehler mehr.
Der DDD kann über das entsprechende Symbol auf der Arbeitsfläche gestartet werden. Im Dialog wird das zu testende und ausführbare Programm geöffnet. Die zugehörige Quelldatei erscheint nun im Quelltextfenster (Abbildung B.1). Außerdem erscheint im Quelltextfenster das frei verschiebbare Kommandofenster.



(a) Haltepunktmenü (b) Kommandomenü

Abbildung B.2.: Haltepunkt- und Kommandomenü

Haltepunkte (breakpoints). Setzt man die Schreibmarke an den Anfang einer ausführbaren Quellcodezeile und betätigen die rechte Maustaste, dann erscheint das Haltepunktmenü (Abbildung B.2a). Die erste Auswahl (`Set Breakpoint`) setzt einen festen Haltepunkt. Der zweite Eintrag (`Set Temporary Breakpoint`) setzt einen Haltepunkt, der nach dem Erreichen wieder gelöscht wird. Der dritte Eintrag (`Continue Until Here`) ermöglicht die Programmausführung oder -weiterführung bis zu der Zeile vor der Schreibmarke.

[1] `http://www.gnu.org/software/ddd/`

Programm schrittweise testen. Durch die Anwahl des Eintrages `Run` im Kommandomenü (Abbildung B.2b) wird das Programm gestartet und bis zur letzten ausführbaren Programmanweisung *vor* dem Haltepunkt ausgeführt. Die erreichte Stelle wird durch einen roten Pfeil gekennzeichnet (Abbildung B.1). Mit Hilfe der anderen Möglichkeiten des Kommandomenüs kann das Programm Zeile für Zeile ausgeführt werden (Tabelle B.2). Nach jedem Schritt können dann zum Beispiel Variablenwerte angesehen oder verändert werden.

Der Debugger `DDD` ist äußerst leistungsfähig und steht für mehrere Rechnerplattformen zur Verfügung. Er ist nicht auf *C/C++* beschränkt. Eine umfassendere Darstellung würde den Rahmen an dieser Stelle sprengen. Alles weitere muß der Originaldokumentation entnommen werden.

Haltepunkt setzen	- Cursor am Anfang der gewünschten Zeile positionieren - rechte Maustaste niederhalten (Haltepunktmenü) - `Set Breakpoint` oder `Delete Breakpoint` wählen - am Zeilenanfang erscheint ein Stop-Schild
Haltepunkt löschen	- Cursor über dem Stop-Schild positionieren - rechte Maustaste niederhalten (Haltepunktmenü) - `Delete Breakpoint` wählen
Programm bis zum Haltepunkt ausführen	- Menüeintrag `Run` wählen
Wert einer Variable ansehen	- Mauszeiger im Quelltextfenster über dem Variablennamen positionieren
Datenelement im Datenfenster anzeigen	- Mauszeiger im Quelltextfenster über dem Namen positionieren - rechte Maustaste niederhalten (Display-Menü erscheint) - `Display` wählen - im Datenfenster erscheint die entsprechende Anzeige
Datenelement im Datenfenster erweitern	Bei Strukturen, Feldern, komplexen Objekten und so weiter kann die Darstellung erweitert oder zusammengefaßt werden: - Mauszeiger im Datenfenster über dem Datenelement positionieren - rechte Maustaste niederhalten (Menü) - `Show All` oder `Hide All` wählen

Tabelle B.1.: `DDD` - grundlegende Bedienung

Run	Starte das zu testende Programm
Interrupt	Unterbreche das zu testende Programm
Step	eine Zeile weiter (bei Unterprogammaufrufen wird in das Unterprogramm gesprungen)
Next	eine Zeile weiter (bei Unterprogammaufrufen wird der Aufruf übersprungen)
Until	Programm bis zur nächsten ausführbaren Zeile fortsetzen (zeilenweise Ausführung)
Cont	Programm nach einem Haltepunkt fortsetzen
Kill	Programm abbrechen

Tabelle B.2.: `DDD` - die Befehle des Kommandomenüs

C. Build Tools

C.1. Die Verwendung von `make`

Programmierprojekte werden ab einer bestimmten Größe üblicherweise in mehrere Dateien zerlegt, die jeweils getrennt compiliert werden können. Im Falle von *C*- und *C++*-Dateien werden die Schnittstellenbeschreibungen der so entstandenen Module in header-Dateien (Dateinamenendung `.h`) zusammengefaßt. Diese h-Dateien enthalten üblicherweise die Funktionsprototypen, `define`-Makros, globale Konstanten und Klassendefinitionen. Die Implementierungen der Funktionen und Klassen stehen in den eigentlichen Quelldateien (Dateinamenendung `.c`, `.cc` und so weiter). Diese Quelldateien enthalten dann `include`-Anweisungen für ihre eigenen header-Dateien sowie für weitere header des Projektes und von Bibliotheken.

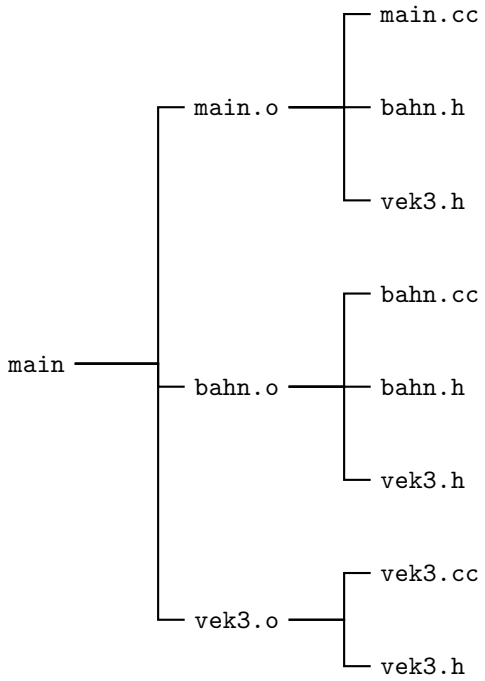


Abbildung C.1.: Abhängigkeiten der Projektbestandteile aus Aufgabe 6

Eine ausführbare Datei ist damit von den Objektdateien abhängig, aus denen sie zusammengebunden wird. Jede Objektdatei ist von ihrer Quelldatei und von den header-Dateien abhängig, die zu ihrer Übersetzung benötigt werden. Dadurch entsteht zumindest bei mittleren und größeren Projekten ein Geflecht von schwer überschaubaren Abhängigkeiten. Insbesondere erfordert die Änderung einer Datei (`h`- oder Objekt-Datei) die Neuübersetzung aller abhängigen Dateien und eine Neuerstellung der ausführbaren Zieldatei.

Abbildung C.1 zeigt die Abhängigkeiten der Projektbestandteile eines kleinen Projektes. Ein Änderung in der Datei `vek3.h` wirkt sich offenbar auf alle Quelldateien aus und erfordert eine vollständige Neuübersetzung mit Binden. Ein Änderung in der Datei `bahn.cc` erfordert lediglich die Übersetzung dieser Datei mit anschließendem Binden.

Das Programm `make` stellt automatisch fest, welche Teile eines Projektes neu compiliert werden müssen und erzeugt die Befehle für diese Compilierungen. Dazu wird eine Skriptdatei verwendet, die üblicherweise `Makefile` oder `makefile` genannt wird. Hierin sind die statischen Abhängigkeiten beschrieben, wie sie zum Beispiel im Abbildung C.1 dargestellt sind. Hinzu kommen Regeln, die festlegen, wie die Compilierungen durchzuführen sind (Compiler, Komandozeilenschalter und so weiter).

Die Datei `Makefile` besteht im einfachsten Fall aus Regeln der folgenden Form:

Ziel : ***Voraussetzungen***
↦ ***Befehl***
↦ ...

Ein ***Ziel*** ist meistens eine zu erzeugende Datei (ausführbare Datei oder Objekt).

Eine ***Voraussetzung*** ist eine Datei, die zur Erzeugung des Zieles erforderlich ist (Quelle oder Objekt). Zu den Voraussetzungen können header-Dateien hinzukommen.

Ein ***Befehl*** ist eine Anweisung, die zur Erzeugung des Zieles ausgeführt werden muß. Meist sind das Compiler- oder Binderaufrufe. *Achtung: Eine Befehlszeile muß mit mindestens einem Tabulatorzeichen* (↦) *beginnen* ! In der Befehlszeile können jedoch auch andere Kommandozeilenbefehle stehen. Mehrere Befehle sind ebenfalls möglich. Diese stehen dann in weiteren Zeilen unter dem Ziel (ohne eingeschobene Leerzeilen).

Liste C.1: `Makefile` für das Projekt „Bahnlängenberechnung“

```
1  #
2  #  Makefile : Projekt Bahnlaengenberechnung
3  #
4  main:     main.o vek3.o bahn.o
5            g++ -lm -o main vek3.o bahn.o main.o
6
7  main.o:   main.cc bahn.hh vek3.hh
8            g++ -Wall -c main.cc
9
10 bahn.o:   bahn.cc bahn.hh vek3.hh
11           g++ -Wall -c bahn.cc
12
13 vek3.o:   vek3.cc vek3.hh
14           g++ -Wall -c vek3.cc
```

Für das Projekt „Bahnlängenberechnung“ ist die Datei `Makefile` in Liste C.1 wiedergegeben. Die Ziele und die Voraussetzungen beschreiben die Abhängigkeiten aus Abbildung C.1 in Textform.

Zum vollständigen Verständnis dieses Beispiels ist die Kenntnis der Kommandozeilenoptionen des Compilers erforderlich. In den Zeilen 5, 8, 11 und 14 wird der ***C++***-Compiler `g++` aufgerufen. Zeile 5 enthält den Befehl zum Binden des Programmes `main` aus den einzelnen Objektdateien.

`g++`	GNU ***C++***-Compiler; wird hier als Binder aufgerufen
`-o main`	legt den Namen der ausführbaren Datei fest
`vek3.o bahn.o main.o`	Liste der einzubindenden Objektdateien
`-lm`	Einbinden der Standardbibliothek `libm.a` (Mathematik)
`-Wall`	beim Compilieren alle auftretenden Warnungen ausgeben

Der Schalter `-c` in den Zeilen 8, 11 und 14 sorgt dafür, daß die jeweilige Quelldatei übersetzt wird. Das Binden macht für diese Teilziele keinen Sinn und unterbleibt deshalb.

Wenn das Programm `make` ohne weitere Zusätze von der Kommandozeile oder aus dem Editor heraus aufgerufen wird, dann wird die Datei `Makefile` oder `makefile` im augenblicklichen Arbeitsverzeichnis gesucht und ausgeführt. Dabei wird versucht, das erste dort angegebene Ziel zu erzeugen (hier `main`). Wenn die Voraussetzungen dieses Zieles selbst Ziele enthält die vorher erzeugt werden müssen (zum Beispiel `bahn.o`), dann sucht `make` Regeln für diese Teilziele und führt diese zuerst aus.

Die Erzeugung eines Zieles ist dabei nicht nur durch die statischen Abhängigkeiten in den Voraussetzungen bestimmt. Wenn alle Dateien die in den Voraussetzungen aufgeführt sind älter sind als das Ziel, dann ist das Ziel aktuell und muß nicht neu erzeugt werden. Ist mindestens eine Datei neuer als das Ziel, dann wird das Ziel neu erzeugt. Damit werden nur diejenigen Dateien neu übersetzt, die sich tatsächlich geändert haben. Von allen anderen werden die bereits vorhandenen Objektdateien verwendet.

`make` ist der „Klassiker“ unter den sogenannten build tools. Es gehört bei den Betriebssystemen der ***Unix***-Familie zur Grundausstattung, ist aber auf fast allen anderen Plattformen verfügbar. In der Regel ist `make` auch Bestandteil der Kommandozeilenwerkzeuge von Entwicklungsumgebungen. Bei guten graphischen Entwicklungsumgebungen (IDEs) besteht meist die Möglichkeit, die Projektstruktur im Dialog festzulegen und die Abhängigkeiten automatisch ermitteln zu lassen. Damit kann die Erstellung einer eigenen `Makefile`-Datei umgangen werden. Es gibt neben `make` eine ganze Reihe ähnlicher Werkzeuge.

Das angegebene Beispiel für ein `make`-Skript ist sehr einfach. `make` bietet umfangreiche Steuerungsmöglichkeiten mit deren Hilfe die Erzeugung großer Projekte (zum Beispiel Betriebssystemkern, graphische Oberflächen und ähnliches) möglich ist. Die Befehlsdarstellung wird jedoch meist als schwer lesbar empfunden. Weitere Informationen enthalten zum Beispiel folgende Quellen:

Mecklenburg, Robert Managing Projects with make
O'Reilly, 2004, ISBN 978-0-596-00610-5, 3. Auflage

Stallman, Richard / McGrath, Roland GNU Make
Free Software Foundation, 2010, ISBN 1-882114-83-3
Als PDF-Datei unter `http://www.gnu.org/software/make/` erhältlich.

C.2. Die Verwendung von `qmake`

Eine Alternative zum klassischen `make` aus Anhang C.1 ist der Makefile-Generator `qmake`. Er ist Bestandteil der Qt-Bibliothek, die zur Programmierung grafischer Benutzeroberflächen eingesetzt wird.

Wenn in einem Verzeichnis die fünf Dateien

```
bahn.cc
bahn.hh
main.cc
vek3.cc
vek3.hh
```

aus dem Beispiel des letzten Abschnitts vorhanden sind, dann erzeugt der Aufruf

```
qmake -project
```

die Projektdatei `bahn.pro` (Liste C.2). Die Benennung erfolgt nach der Datei `bahn.cc`, da hier bei der Analyse das Hauptprogramm gefunden wurde. Ein anschließendes

```
qmake
```

erzeugt aus dieser Projektdatei die Datei `Makefile` als Steuerdatei für das Standard-`make`. Der Aufruf von `make`

```
make
```

baut daraus die ausführbare Datei `bahn`. Die ersten beiden Schritte müssen nur wiederholt werden, wenn sich Abhängigkeiten geändert haben.

Liste C.2: Die automatisch erzeugte Projektdatei `bahn.pro`

```
#############################################################################
# Automatically generated by qmake (2.01a) Fri Mar 1 16:15:45 2013
#############################################################################

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += bahn.hh vek3.hh
SOURCES += bahn.cc main.cc vek3.cc
```

Die Datei `Makefile` enthält stets einige Standardziele:

clean Beseitigt alle Objektdateien.

distclean Beseitigt alle Objektdateien, sowie dem Makefile und die ausführbare Datei.

dist Erstellt ein gepacktes `tar`-Archiv (einen sogenannten tarball).

Weitere Möglichkeiten zur Verwendung von `qmake` sind in der zugehörigen Dokumentation beschrieben.

Index

Abschnittskommentar, 32
Ausnahme, 27

beautifier, 34
Bildverarbeitung, 1
Bogenlänge, 24
breakpoint, 36
build tools, 39
    `make`, 39
    `qmake`, 41

Cramersche Regel, 23

Dateibeschreibung, 31
`DDD`, 36
Debugging, 35
Destruktor, 12, 15

Einheitsvektor, 19
Einrückung, 33
exception, 27

Fehlerklasse, 28
Fehlerobjekt, 28
`friend`-Funktionen, 21

Game of Life, 3

Haltepunkt, 36
header-Dateien, 39

`indent`, 34

Klasse, 11
Konstruktor, 12, 15

Liste
    einfach verkettet, 7

`make`, 39
`Makefile`, 39

Operator, 19
Operatorüberladung, 21

Pi
    Programmdarstellung, 24
Programmdokumentation, 31
Programmierrichtlinien, 31
Programmierstil, 31

`qmake`, 41

Ringliste, 29

Sarrusschen Regel, 23
Schachtelung, 33
Schachtelungstiefe, 33
Stapel, 11
Struktur, 1

Testen, 35
`try-catch`-Anweisung, 28

UNIX
    `free`, 12
    `indent`, 34
    `make`, 39
    `qmake`, 41

Vektor, 19
    Addition, 19
    Kreuzprodukt, 19, 21
    Länge, 19
    Multiplikation mit einem Skalar, 19
    normieren, 19
    Skalarprodukt, 19
    Subtraktion, 19
Vektorarithmetik, 19

Zeilenendkommentar, 32
Zuweisungsoperator, 13, 15