Kapitel 3
Nützliche Werkzeuge

3.1 Namensbereiche und hilfreiche Templates

Für wiederkehrende Aufgaben werden meist Bibliotheksfunktionen eingesetzt. Neben der STL werden Sie im Laufe der Zeit einige andere Bibliotheken laden und auch die eine oder andere Methode selbst programmieren und irgendwann feststellen, dass eine Lösung in mehreren Bibliotheken existiert und möglicherweise sogar den gleichen Namen verwendet. Möglicherweise passiert es Ihnen sogar in eigenen Projekten, dass Sie Namen aus früheren Projekten wiederverwenden. Die erste Empfehlung lautet daher, Namensbereiche zu verwenden:

```
namespace MyName{
   namespace Helpers {
   ...
   }//end namespace
}//end namespace
...
using namespace STD;
using namespace MyName::Helpers
```

Tauchen unter solche Umständen den Compiler verwirrende Mehrfachbezeichnungen auf, können Sie die von Ihnen gewünschte problemlos durch Verwendung des Scope-Operators spezifizieren:

```
var1 = MyName::Helpers::myFunction(..);
```

In C werden viele einfacher Hilfsfunktionen durch Macros definiert. Die Möglichkeit besteht in C++ natürlich auch, und wir werden sie auch nutzen, aber in den meisten Fällen empfiehlt es sich, die Makros durch `template`–Methoden zu ersetzen, da die Kontrollen durch den Compiler besser sind, ohne dass es zu einer Leistungsverminderung bei Optimierung kommt, und eben auch mit dem `namespace`-Konzept gearbeitet werden kann. Einige häufig benötigte Methoden stellen wir hier vor.[1]

---

[1] Die STL stellt bereits die meisten zur Verfügung.

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_3, © Springer-Verlag Berlin Heidelberg 2010

Häufig gilt es das größere (oder kleinere) zweier Elemente bestimmen. Hier kommen folgende `template`–Funktionen zum Einsatz, die wir gleich um eine Austausch- und eine Vergleichsmethode ergänzen:

```
template <class T> inline
T const& min(const T& a, const T&b){
    return (a<b?a:b);
}//end function

// in gleicher Weise die Methode max

template<class T> inline void swap(T& a, T& b){
T temp(a);
    a=b;b=temp;
}//end function

template <class T>
inline int compare(const T& t1, const T& t2){
    if(t1==t2)      return 0;
    else if(t2<t1) return 1;
    else           return -1;
}//end function
```

Beachten Sie in den Funktionen `min` und `max` die Rückgabe als Referenz. Die Auswahl des jeweils kleineren oder größeren Elements beschränkt sich undabhängig von der Komplexität des Vergleichs auf ein Verschieben einer Adresse in das Arbeitsregister der folgenden Operation.

Die Methoden besitzen nur einen Typparameter. Sollen Variablen unterschiedlichen Typs verarbeitet werden, so bestehen folgende Möglichkeiten:

- Einer der Typen besitzt einen Konstruktor, der ein Objekt des anderen Typs als Parameter besitzt. In den meisten Fällen wird dieser vom Compiler implizit verwendet und man braucht nichts weiter zu tun, ggf. kann man den Konstruktor aber auch von Hand angeben.
- Einer der Typen wird durch explizites Typecasting auf den anderen abgebildet.
- Man schreibt Methoden mit mehr als einem Template-Parameter. In der Regel ist hiervon jedoch abzuraten, da sich das Problem dann nur auf den unteren Ebenen wie Vergleichsoperatoren fortsetzt und wenig gewonnen ist.

Operatoren werden sehr häufig in C++ überschrieben. Es ist jedoch nicht notwendig, ob der großen Anzahl in Panik zu verfallen, da auch hier die Template-Technik einiges an Arbeit beseitigt. Die Definition von Vergleichsoperatoren kann auf `operator==(..)` und `operator<(..)` beschränkt werden, fast arithmetische Operatoren auf die Versionen `operator+=(..)`, `operator*=(..)`, usw. Die weiteren werden durch die Templates

```
template <class T>
inline bool operator >=(const T& a, const T& b){
    return !(a<b);
```

```
}//end function
template <class T>
inline bool operator !=(const T& a, const T& b){
    return !(a==b);
}//end function

...

template <class T>
inline T operator +(const T& a, const T& b){
    T temp(a);
    return temp+=b;
}//end function

...
```

Als unäre Methoden definieren wir den Absolutwert sowie die Signum–Funktion:

```
template <class T> inline
T abs(const T& a){
    return (a>=Constant<T>::null() ? A : -a );
}//end function

template <class T> inline int sign(const T& a){
    if(a<Constant<T>::null())
        return -1;
    else if(a==Constant<T>::null())
        return 0;
    else
        return 1;
}//end function
```

Anstelle des Vergleichs `a==0` haben wir hier den Methodenaufruf `Constant<T=::null()` verwendet, der durch die Templateklasse

```
template <typename T> struct Constant {
    static inline T const& null(){
        static T Null(0);
        return Null;
    }//end constant
    ...
```

definiert wird und eine Referenz auf eine statische Variable mit dem Wert Null zurückgibt. Das mag nun besonders in Verbindung mit den C-Standardtypen aufwändig erscheinen (*immerhin muss ja zunächst die Adresse geladen werden, bevor der eigentliche Wert in die CPU transferiert wird*), hat aber Vorteile bei komplizierteren Datentypen. Dies müssen nämlich bei dem Vergleich `a==0` entweder einen Konstruktor mit int-Parameter oder eine Vergleichsmethode mit int-Parameter besitzen, was mehr Aufwand bei der Programmierung sowie bei der Vergleichsimplementation durch den Compiler bewirkt. Innerhalb der statischen Methode

der Klasse Constant lässt sich die Initialisierung durch Spezialisierung auf einen Datentyp aber beliebig gestalten und der Konstruktoraufruf erfolgt nur einmalig.

In der Klasse `Constant` können auf die gleiche Art weitere häufig benötigte Konstanten wie `eins()` oder `zwei()` definiert werden. Das Problem des größeren Aufwandes mit Standardtypen beseitige man durch Spezialisierung der Funktionen, die die `Constant`-Klasse nutzen, mit einfacheren Vergleichskonstrukten.

3.2 Umwandeln in Strings

Um Daten mit dem Anwender oder auch systemunabhängig zwischen Maschinen austauschen oder auch längerfristig auf der Festplatte speichern zu können, ist eine Darstellung in lesbarer Form notwendig. Sinnvollerweise geschieht dies in Form von ASCII-Strings. Wir schaffen dazu weitere Standardfunktionen:

```
template <typename T> inline
string to_string(T const& t);

template <typename T> inline
bool from_string(T& t, string s);
```

Die Methoden besitzen keine Implementation, d.h. wir müssen sie für jeden Datentyp spezialisieren. Das macht Anfangs etwas Aufwand, aber wie wir später sehen werden, kann man rekursiv immer wieder auf bereits implementierte Teile zurückgreifen, so dass sich der Aufwand insgesamt in Grenzen hält.

Bevor wir mit den ersten Implementationen beginnen, empfiehlt sich eine Festlegung von Regeln.

(a) Jede Umwandlung muss die binäre Genauigkeit wiedergeben, d.h. durch aufeinander folgende String- und Objektumwandlung ändert sich am ursprünglichen Inhalt nichts.

(b) Ein String muss sich eindeutig einem Datentyp zuordnen lassen.

Um (b) zu realisieren, machen wir eine Anleihe bei XML.[2] Die Kodierung einer `int`-Größe sieht dann folgendermaßen aus:

---

[2] Wir befinden uns da auf sicherem Boden, denn diese Kodierungsmethode wird in sehr vielen Bereichen genutzt, so dass zu einer Kompatibilität unserer Anstrengungen mit denen anderer Programmierer oft nur ein kurzer Schritt ist. Die zu markierende Größe wird hierbei mit einer öffnenden und einer schließenden Marke (englisch tag) eingerahmt. Die öffnende Marke unterscheidet sich von der schließenden lediglich durch das Zeichen / an Position zwei. Die Zeichen < und > sind ebenfalls vorgeschrieben, der Name der Marke darf beliebig gewählt werden. Marken dürfen geschachtelt oder hintereinander angeordnet sein, dürfen einander aber nicht überlappen. Im Weiteren werden wir statt `Marke` den gebräuchlicheren Ausdruck `Tag` verwenden, auch wenn dieses „Denglisch“ mir persönlich wenig liegt. Wer weitere Informationen benötigt, sei auf das Internet verwiesen.

```
<int>12345</int>
```

```
template <> string to_string(int const& i){
    char buf[10];
    sprintf(buf,"%d",i);
    return string("<int>")+buf+"</int>";
}//end function
```

```
template <> bool from_string(int& t, string s){
    string is;
    if(!extract_string(s,is,"<int>"))
        return false;
    if(is.find_first_not_of("+-0123456789")
              !=string::npos) return false;
    t=atoi(is.c_str());
    return true;}//end function
```

Diese Implementation erlaubt die Kodierung von vorzeichenbehafteten ganzen Zahlen als Dezimalzahlen. Sind andere Kodierungen wie beispielsweise Hexadezimalzahlen gewünscht, ist der Code entsprechend anzupassen.

Während die Kodierung sehr einfach zu gestalten ist, sind in der Dekodierung Prüfungen eingebaut. Die Methode `extract_string` erledigt zunächst das Ausschneiden eines Tags aus einem String. Das ist komplizierter, als man vielleicht im ersten Augenblick meint. Im String

```
vorspann <tag> tagbereich</tag> nachspann
```

muss man nur die Positionen der beiden Tagstrings mit Hilfe der Stringmethode `find()` bestimmten und dann Quelle und Ziel geeignet zuschneiden, aber was ist mit rekursiven Tags der Art

```
vor <tag> htag <tag> ntag</tag> stag</tag> nach
```

Um hier richtig zuschneiden zu können, muss überprüft werden, wie viele öffnende Tags ohne schließendes Tag hintereinander stehen, um dann auch eine entsprechende Anzahl von schließenden Tags überspringen zu können. Sehen wir uns die folgende Methode an:

```
bool extract_string(string& source, string& dest,
                    string const& tag){
    string etag; int pos1, pos2,tpos;
```

Zunächst unterstützen wir den Nutzer, indem wir fehlenden Tagbegrenzer ergänzen. Anschließend werden die Positionen von öffnendem und schließendem Tag ermittelt und abgebrochen, wenn bereits dies keinen Erfolg zeigt:

```
if(tag.find('<')==string::npos){
    tag=string("<")+tag+">";
}//endif
```

```
etag=tag;
etag.insert(1,"/");
tpos=pos1=source.find(tag);
pos2=source.find(etag);
if(pos1>pos2) return false;
```

Nun folgt die rekursive Suche, wobei wir abbrechen, sofern die Suche nach dem nächsten Tagpaar erfolglos ist (was bereits bei der ersten Suche, die wir durchgeführt haben, der Fall gewesen sein kann)

```
while(tpos!=string::npos && pos2!=string::npos){
    tpos=source.find(tag,tpos+1);
    if(tpos!=string::npos && tpos<pos2)
        pos2=source.find(etag,pos2+1);
}
```

Der Rest ist nun trivial: ist kein Fehler aufgetreten, werden die Strings entsprechend zurecht geschnitten.

```
    if(pos1==string::npos || pos2== string::npos)
        return false;
    dest=source;
    source.erase(pos1,pos2+etag.length()-pos1);
    dest.erase(pos2,dest.length()-pos2);
    dest.erase(0,pos1+tag.length());
    return true;
}//end function
```

Die Tag-Bezeichnung muss der XML-Konvention entsprechen, wird aber nicht weiter geprüft. Der komplette erste Tag-String wird aus dem String `source` ausgeschnitten und (ohne Tags) in `dest` abgespeichert. Treten Fehler auf, gibt die Methode `false` zurück.

Lässt sich der String sauber zerlegen, prüft `find_first_not_of` (siehe STL), ob der String ausschließlich erlaubte Zeichen enthält. Ist die Dekodierung fehlerfrei gelungen, gibt die Methode `true` als Rückgabewert zurück.

In diesem Stil können nun alle weiteren Standardtypen durch Spezialisierungen implementiert werden. Treffen wir nun auf zusammengesetzte Typen, können die ersten Ergebnisse verwendet werden. Beispielsweise lässt sich der Typ `complex`, eine Template-Klasse der STL, problemlos rekursiv implementieren:

```
template <typename T>
string to_string(complex<T> const& t){
    return string("<complex>")+to_string(t.real())+
           to_string(t.imag())+"</complex>";
}//end function

template <typename T>
bool from_string(complex<T>& t, string& s){
    string ss;
```

```
    if(!extract_string(s,ss,"<complex>"))
       return false;
    return from_string(t.real(),ss) &&
           from_string(t.imag(),ss);
}//end function

Kodierungsbeispiel:
------------------
complex<int> a(2,3);
<complex><int>2</int><int>3</int></complex>
```

Das ist bei länger werdenden String zugegebenermaßen nicht ganz einfach zu lesen, aber es hindert Sie niemand daran, zwischen den Tags Formatierungen und weitere Kennungen unterzubringen, die für bessere Lesbarkeit oder weitere Prüfmöglichkeiten sorgen. Nach dieser Strategie können Sie nun beliebige Klassen kodieren, indem Sie eine eindeutige Tagmarke für die Klasse als Rahmen um die Einzelkodierungen aller Attribute setzen.
Auch sehr einfach wird die Kodierung kompletter Conatiner unter Nutzung des Iteratorkonzeptes. Nur mit Iteratoren operieren die folgenden Methoden:

```
template <typename iter>
string to_string(iter const& beg, iter const& end){
    string s="<container>";
    s+=to_string(distance(beg,end));
    s.replace(s.find("int"),3,"size");
    s.replace(s.find("int"),3,"size");
    for(iter it=beg;it!=end;++it){
        s=s+to_string(*it);
    }//endfor
    return s+"</container>";
}//end function

template <typename iter>
bool from_string(iter const& beg, iter const& end,
                 string& t){
    string s; iter it;
    if(!extract_string(t,s,"<container>"))
        return false;
    for(it=beg;it!=end && s.length()>0;++it)
        if(!from_string(*it,s)) return false;
    return (it==end)&&(s.length()==0);
}//end function
```

Zumindest beim Rücklesen muss der aufnehmende Container auf die Anzahl der kodierten Elemente, die im Tag `<size>` angegeben ist, eingestellt sein. Wir können aber auch einen Container direkt in der Template-Funktion angeben:

```
template <class T,
          template <class> class container>
string to_string(container<T> const& c){
    return to_string(c.begin(),c.end());
}//end function

template <class T,
          template <class> class container>
bool from_string(container<T>& c, string& s){
    string t,tt; int l;
    c.clear();
    if(!extract_string(s,t,"<container>"))
        return false;
    if(!extract_string(t,tt,"<size>"))
        return false;
    if(!from_string(l,string("<int>")+tt+"</int>"))
        return false;
    c.resize(l);
    return from_string(c.begin(),c.end());
}//end function
```

Während die Kodierungsmethode einfach die vorhergehende Iteratorversion aufruft, wird der Tag `<size>` nun in der dekodierenden Methode benutzt, um die Containergröße einzustellen. Auch diese Methoden können Sie natürlich weiter spezialisieren, wenn Sie bestimmte Container in der Kodierung kenntlich machen wollen.

Aufgabe. Implementieren sie Spezialisierungen für elementare Typen sowie die Eingabe in ein Feld über eine ASCII-Liste. Erstellen Sie dazu ein Projekt „Testkonversion", in dem Sie im Laufe der Zeit alle Kodierungen sammeln und leicht kontrollieren können. Zur Vereinfachung implementieren Sie eine Testmethode

```
template <class T>
bool test_conversion(T const& obj, string ms="")
{...}
```

die Kodierungen und Dekodierungen durchführt und im Fehlerfall eine Nachricht ausgibt.

Anmerkung. Die Kodierungsmethoden führen nun nicht gerade zu besonders gut lesbaren Daten, was insbesondere bei der Verwendung in Programmtests auffällt. Eine Kodierung der Form

```
<complex><int>2</int><int>3</int></complex>...
```

sähe so

```
<complex>
     <int>  2  </int>
     <int>  3  </int>
</complex>
```

für das Auge sicher angenehmer aus, besonders wenn noch mehrere Daten folgen. Eine solche „Verschönerung“ der Daten besteht aus dem Einfügen von Leerzeichen und Zeilenvorschüben, was bei der Ausgabe kein Problem darstellt, das Rücklesen aber komplizierter macht, da Textdateien in der Regel zeilenweise gelesen werden, Zeilen aber nicht interpretiert werden dürfen, da sie noch unvollständig sind.

Aufgabe. Führen Sie zunächst einen Softwareschalter ein, der zwischen komprimierter und verschönerter Ausgabe umschaltet. Zeilenvorschübe innnerhalb von Strings können durch `\n` kodiert werden. Der gesamte fertige Text wird durch die Tags

```
<beautifier> ...</beautifier>
```

eingerahmt.

Entwerfen Sie eine Klasse, die beliebig viele Strings entgegen nimmt und in einen Ausgabenstring verdichtet. Ab dem Tag `<beautifier>` werden sämtliche Leerzeichen und Zeilenvorschübe bis zum Auftreten des Tags `</beautifier>` entfernt, außer bei Tags des Typs `<string>.. </string>`.

3.3 Parameterstrings

3.3.1 Grundgerüst

Bei der Kodierung von Objekten komplexer Klassen entstehen mit den vorstehend beschriebenen Methoden Strings mit komplex geschachtelten HTML-Tags. Die Technik eignet sich auch für andere Darstellungen komplexer Abhängigkeiten, die aber auch unabhängig von bestimmten Klassen bearbeitbar sein sollten. Wir schaffen uns dazu das Werkzeug „Parameterstring“, das eine solche Kodierung dynamisch in seine Teile zerlegt.[3]
Spezifizieren wir zunächst alle Regeln, die für die Kodierung gelten sollen:

(a) Zu einem Objekt gehörende Daten werden durch ein öffnendes Tag und ein schließenden Tag eingerahmt.

```
<tag>...</tag>
```

[3] In der ersten Auflage dieses Buches wurden an dieser Stelle noch mit anderen als XML-Mechanismen kodierte Parameterstrings beschrieben. Die XML-Technik führt zwar zu etwas umfangreicheren Strings, ist aber im Gegenzug einfacher zu interpretieren und außerdem die allgemein verwendete Technik. Eigene Entwürfe sind zwar vielleicht intellektuell interessant, aber nicht besonders universell einsetzbar, weshalb ich hier darauf verzichtet habe.

(b) Tags dürfen geschachtelt werden

```
<tag1>..<tag2>...</tag2>..</tag1>
```

(c) Tags dürfen hintereinander auftreten.
(d) Tags dürfen **nicht** überlappen.
(e) Um für die Tagkennzeichnung verwendeten öffnenden und schließenden Klammern < und > von einer Verwendung in normalem Text abzuheben, werden diese im Text durch vorangestellte inversen Schrägstrich gekennzeichnet

```
\< , \> , \\
```

Tauchen diese Zeichenkombinationen im Text auf, sind sie keine Tagsteuerzeichen. In reinen Textdarstellungen sind die \ -Zeichen zu Löschen, in Kodierungen hinzuzufügen.

XML enthält noch weitere Vereinbarungen für die Darstellung bestimmter Zeichen. Wer möchte, kann die Liste der Vereinbarungen entsprechend erweitern. Auch ist sinnvoll, die Kodierung von Strings im vorhergehenden Kapitel auf die Einhaltung dieser Regeln zu überarbeiten.

Durch die XML-Kodierung entstehen verzeichnisartige Strukturen. Auf einer Ebene können mehrere „Verzeichnisse“ angeordnet sein, die jeweils wieder mehrere „Unterverzeichnisse“ besitzen können. Da bei unserer Datenkodierung bereits der Fall aufgetreten ist, dass mehrere Tags mit gleicher Tagbezeichnung hintereinander auftreten können, muss unser Werkzeug die Reihenfolge von Einträgen ebenfalls beibehalten.

Unser Modell enthält auch den Fall, dass auf einer Ebene mit Tags weiterer, nicht durch Tags geklammerter Text zu finden ist

```
<t1>text1<t2>..</t2><t3>..</t3>Text2</t1>
```

Diese Textbestandteile interpretieren wir als tagfreie Daten. Sie werden ebenfalls in der auftretenden Reihenfolge im Arbeitsobjekt abgelegt.

Für den Zugriff auf Daten und Tags werden zwei verschiedene Methoden vorgesehen, die mit einem Schlüsselsystem der Form

```
data("0.1.0")
tag("0.1.0")
```

die zugehörenden Daten ausliefern. Die Schlüssel bestehen aus einer Indexnummer des Objektes in jeder Ebene.[4] Um taggebundene Daten von tagfreien Daten zu unterscheiden, müssen beide Einträge ausgelesen werden. Damit haben wir nun alle Vereinbarungen zusammen, um eine Klasse zu definieren:

---

[4] Dies ist die anwenderorientierte Schnittstelle für den Zugriff. Für programminterne Zugriffe sind zahlenorientierte Schnittstellen sinnvoller, die wir weiter unten berücksichtigen werden.

```
class XMLString {
...
    struct entry {
        string tag;
        string data;
        vector<entry> sub;
    };//end struct

    vector<entry> ent;
};
```

Die zu einem Tag gehörenden Daten werden auf dem String `data` gespeichert,[5] die Tagbezeichnung auf `tag`. Enthaltene Subtags sind auf dem enthaltenen Vektor kodiert.

3.3.2 Das Zerlegen und Rekonstruieren eines Strings

Mit Hilfe von rekursiven Methoden lässt sich ein XML-String sehr leicht aufarbeiten. Grundlage ist die Methode `extract_string`, die wir zunächst in einer arbeitstechnisch angenehmeren und auch allgemeiner nutzbaren Variante implementieren:

```
vector<string> explode(string source,
                       string delim,
                       int cnt=INT_MAX){
    vector<string> v; int pos;
    while((pos=source.find(delim))!=string::npos
          && count-- >0){
        v.push_back(source.substr(0,pos));
        source.erase(0,pos+delim.length());
    }//end
    if(source.length()>0) v.push_back(source);
    return v;
}
```

Die Methode spaltet von einem String bei jedem Auftreten des Trenners einen Teilstring ab und speichert diesen in einem Vektor. Über den optionalen Zähler kann die Anzahl der Aufspaltungen kontrolliert werden.

Zunächst entfernen wir die \< - und \> - Einträge aus dem zu zerlegenden String. Da die Strings vereinbarungsgemäß nur lesbare Zeichen enthalten sollen, können wir dazu auf nicht-lesbare zurückgreifen. Das Maskieren erledigt die Methode

[5] Die zum Tag gehörenden Daten sind ggf. nach dem oben angegebenen Schema mit \ abgelegt, um sie bei Rekonstruktion des Strings sicher wieder an die korrekte Position schreiben zu können.

```
string mask_slashes(string s){
    int pos;
    while((pos=s.find("\\<"))!=string::npos)
        s[pos+1]=0xff;
    while((pos=s.find("\\>"))!=string::npos)
        s[pos+1]=0xfe;
    return s;
}//end function
```

Aufgabe. Implementieren Sie dazu passend die unmask_slashes-Methode.

Das rekursive Zerlegen der Strings lässt sich nun folgendermaßen durchführen:

- Spalte an der Position < auf. Der erste Text ist, falls nicht leer, ein Kommentar.
- Spalte an der Position > auf. Der erste Text ist die Tagbezeichnung tag.
- Spalte an der Position </ + tag + > auf. Der erste Text ist der Taginhalt, der zweite Text enthält möglicherweise weitere Tags.
- Arbeite den Taginhalt rekursiv auf, falls er weitere Tags enthält, oder speichere ihn als Taginformation.
- Arbeite den zweiten Text, sofern nicht leer, in der gleichen Weise auf.

Mit ein wenig Fehleranalyse, was in einem String alles daneben gehen kann, führt diese Strategie auf folgenden Code

```
bool XMLString::parse(string s){
    ent.clear();
    return do_parsing(ent,s);
}//end function
```

Die Rekursion operiert auf dem Datentyp vector<entry>, der ja im Knotentyp entry wiederum auftaucht. Der Vektor ist daher Übergabeparameter der Rekursionsmethode. Diese entfernt zunächst die \< - Sequenzen und spaltet dann am ersten < - Zeichen auf.

```
bool XMLString::do_parsing(vector<entry>& ent,
                           string s){
    int pos; bool cnt=false;
    vector<string> v1,v2,v3;
    entry e;
    s=mask_slashes(s);
    v1=explode(s,'<',1);
```

Sofern der erste Teilstring das Zeichen > enthält, handelt es sich nicht um einen gültigen XML-String und wir brechen ab. Falls nur eine Antwort geliefert worden ist, haben wir die Taginformation identifiziert und brechen ab. Ansonsten fahren wir mit der Identifizierung des Tags fort.

```
if(v1.front().find('>')!=string::npos)
    return false;
e.data=unmask_slashes(v1.front());
ent.push_back(e);
if(v1.size()==1) return true;
v2=explode(v1.back(),'>',1);
if(v2.size()==1 ||
    v1.front().find('<')!=string::npos)
    return false;
```

Falls nur ein Ergebnisstring bei der Aufspaltung entsteht oder der String ein < - Zeichen enthält, ist wiederum etwas faul und wir brechen wiederum ab. Andernfalls kümmern wir uns um die Identifikation des Endtags

```
e.tag=unmask_slashes(v2.front());
s=string("</")+v2.front()+">";
if(v2.back().find(s)==string::npos)
    return false;
v3=explode(v2.back(),s,1);
e.data=unmask_slashes(v3.front());
```

Falls die Taginformation noch < - Zeichen enthält, sollten weitere Tags enthalten sein, und wir können die Rekursion auf der nächsten Ebene fortsetzen.

```
if(v3.front().find('<')!=string::npos &&
    !do_parsing(e.sub,v3.front())) return false;
ent.push_back(e);
```

Besitzt der String weitere Einträge, setzen wir die Auswertung durch einen rekursiven Aufruf der Methode fort, verzweigen jedoch nicht in die nächste Ebene, sondern bleiben auf derselben Ebene.

```
if(v3.size()>1 && v3.back().length()>0)
        return do_parsing(ent,v3.back());
    return true;
}//end function
```

Damit haben wir den XML-String komplett zerlegt und in unserer rekursiven Datenstruktur abgelegt bzw. die Operation mit einer Fehlermeldung unterbrochen, falls der String gegen die Regeln verstößt.

Aufgabe. In ähnlich rekursiver Weise lässt sich aus dem XMLString-Objekt wieder ein XML-String erzeugen. Implementieren Sie die Methode `c_str()`, die dies bewerkstelligt.

3.3.3 *Arbeiten mit dem XMLString*

Für den Zugriff auf die Daten eines Parameterstrings sehen wir zwei Methoden vor:

```
string data(string) const;
string data(int size,int) const;
```

In der ersten Methode erfolgt der Zugriff durch einen String, der die Indizes der einzelnen Ebenen enthält:

```
s=obj.data("0.1.0.3");
```

Dieser Schlüssel liefert, sofern vorhanden, die Daten des Eintrags mit dem Index Null auf der ersten Ebene, dem Index 1 auf der zweiten usw. Mit der zweiten Methode lautet der entsprechende Parametersatz der Methode

```
s=obj.data(4,0,1,0,3);
```

Den Schlüsselstring können wir mit Hilfe bereits implementierter Methoden in ein Feld von ganzen Zahlen überführen, dass wir der eigentlichen Zugriffsmethode übergeben:

```
string XMLString::data(string s) const{
    vector<string> v; vector<string>::iterator it;
    vector<int> vi;   int i;
    v=explode(s,'.');
    for(it=v.begin();it!=v.end();it++){
        *it="<int>"+*it+"</int>";
        if(!to_string(i,*it) return "";
        vi.push_back(i);
    }
    return get_data(ent,0,vi.size(),&vi[0]);
}
```

Diese arbeitet sich wieder rekursiv durch die Baumstruktur:

```
string XMLString::get_data(
       vector<entry> const& c,
       int i, int size, int* index) const {
    if(c.size()<=index[i]) return "";
    if(i==size-1) return c[index[i]].data;
    if(c[index[i]].sub.empty()) return "";
    return get_data(c[index[i]].sub,
                    ++i,size,index);
}
```

Bei Wechsel in die nächste Ebene wird der Ebenenzäher `i` inkrementiert sowie eine Referenz auf die aktuelle Unterliste übergeben. Der gesuchte Dateneintrag ist gefunden, wenn der Ebenenzähler die ebenfalls übermittelter Schlüsseltiefe erreicht hat.

Die zweite Zugriffsmethode greift ebenfalls auf diese Methode zurück:

```
string XMLString::data(int size, int index) const{
    return get_data(ent,0,size,&index);
}
```

Die Methode arbeitet mit einer variablen Anzahl von Parametern im Funktionskopf, was hier allerdings nicht explizit angegeben ist. Alle Parameter im Übergabeaufruf werden vom Compiler nacheinander auf dem Stack abgelegt, so dass für den Zugriff die Adresse des ersten Parameters sowie der Typ der abgelegten Werte genügt. Hier ist beispielsweise der `int`-Typ `index` im Funktionskopf angegeben, und `&index` gibt die Startadresse des `int`-Feldes an, das sämtliche Parameter enthält. Da das Laufzeitsystem natürlich nicht in der Lage ist, zwischen einem `int`-Wert und irgendeinem anderen Bitmuster auf dem Stack zu unterscheiden, muss zusätzlich im ersten Parameter die Anzahl der übergebenen Werte angegeben werden.

Unter weiterer Verwendung dieser Zugriffsmechanismen kann die Klasse komplettiert werden, was dem Leser überlassen sei.

Aufgabe. Implementieren Sie weitere Methoden zum Ändern der Daten, zum Ausgeben und Ändern der Tag-Bezeichnungen, zum Einfügen weiterer Tags mit Daten und zum Löschen von Tags.

3.4 Ablaufverfolgung (TRACE)

3.4.1 Debugger oder Tracer?

Bei der Suche nach den Gründen, warum eine Anwendung nicht das macht, was man von ihr erwartet, benötigt man häufig Informationen über innere Zustände, das heißt welche Werte nehmen welche Variable an usw. Eine Möglichkeit, genau zu verfolgen, was ein Programm alles so macht, ist die Nutzung eines Debuggers. Moderne Debugger vermögen Programme an einer bestimmten Stelle unter genau definierten Bedingungen anzuhalten und den Inhalt aller existierenden Variablen, der Prozessorinhalte und des Funktionsstacks anzuzeigen.[6] Daten können manipuliert und das weitere Verhalten des Programms im Einzelschrittverfahren oder in größeren Abständen beobachtet werden. Wie Debugger effektiv eingesetzt werden, hängt in hohem Maße vom Verständnis des Entwicklers für den Prozess ab, den er beobachten will, und ich bin ziemlich sicher, dass sich dicke Bücher über das Debuggen mit einer Unmenge an Beispielen schreiben lassen, die unbrauchbar sind, weil ein Entwickler entweder seinen Prozess nicht versteht und daher den Buchinhalt nicht anwenden kann oder seinen Prozess sehr gut versteht und dann ein Buch nicht benötigt.

---

[6] Allerdings geht dabei häufig die Performanz in die Knie. Ergebnisse, die sonst unmittelbar nach dem Programmstart bereits vorliegen, lassen dann schon einmal einige Minuten auf sich warten.

Debugger sind jedoch nicht unter allen Bedingungen geeignete Werkzeuge. Zunächst halten sie das Programm an, was für ereignisorientiert Anwendungen tödlich ist. Sie geben auch immer nur ein augenblickliches Bild eines Prozesses, nie aber eine Übersicht über den Gesamtverlauf. Die Verfolgung längerer Abläufe kann wegen des häufigen Anhaltens äußerst mühsam werden. Neben Debuggern bilden daher Tracer, also Ablaufverfolger, ein zweites Werkzeug bei der Programmuntersuchung. Tracer erzeugen mehr oder weniger große Datenmengen bestimmter vorher festgelegter Prozesszustände, die im Anschluss an den Programmablauf analysiert werden können. Im Grunde handelt es sich um Ausgabebefehle für Daten in Dateien, die zusätzlich zum normalen Ablauf im Programm untergebracht werden.
Allerdings gibt es dabei zwei neue Probleme:

- Zwangsweise entsteht beim Ausdruck größerer Datenmengen auch sehr viel Müll, aus dem die interessierenden Informationen herausgesucht werden müssen. Wie in der modernen Gesellschaft muss Müllsortierung und Müllvermeidung betrieben werden, um die Daten überhaupt noch in sinnvoller Zeit überschauen zu können.
- Wenn die Probleme beseitigt sind, sind die Trace–Befehle wieder zu entfernen. Das ist zunächst ein sehr mühsames Geschäft, wenn sehr viele Kontrollen eingebaut wurden, und mit ziemlicher Garantie stellt sich das nächste Problem ein, wenn endlich alle Trace–Befehle entfernt sind, und man fängt von vorne mit dem Einbau an (*die Produktion neuer Flüche ist zu solchen Zeitpunkten oft höher als die Produktion neuen Kodes*).

In der Praxis neigen die einzelnen Programmierer häufig der weitgehenden Verwendung eines Werkzeuges zu. Ich beispielsweise bevorzuge (nach eine längeren Phase der Debuggernutzung) das Trace-Werkzeug, während ein Freund, der mit mir am gleichen Projekt arbeitet, auf den Debugger schwört. Vermutlich wird es Ihnen ähnlich gehen, jedoch sollten Sie besonders Anfangs beide Möglichkeiten ausprobieren. Das bietet sich meist dann an, wenn man bei der Lösung eines Problems nicht so richtig weiterkommt. Wie im Kapitel über Arbeitstechniken beschrieben, sollte man sich hier ein Zeitlimit setzen und nach Ablauf die andere Technik ausprobieren.

3.4.2 Eine einfache Trace-Klasse

Wir schaffen uns nun Werkzeuge, um das Tracen von Programmen effektiver gestalten zu können. Problem Zwei ist dabei sehr einfach zu erledigen. Implementieren wir dazu eine spezielle Funktion für den Kontrollvorgang:

```
#define TRACE_ON

template <class T>
inline void TRACE(T const& t){
    #ifdef TRACE_ON
        trace_stream << to_string(t) << endl;
```

```
    #endif
};//end function

void main(){
    ...
    TRACE("Das sollte mal ausgedruckt werden");
    ...
```

Der Ablauf ist schnell analysiert: Ist `TRACE_ON` gesetzt, wird die Kontrollinformation ausgedruckt. Kommentieren wir `TRACE_ON` aus, verschwindet der Implementationskode der Methode und der Kontrollausdruck unterbleibt, ohne dass wir die `TRACE`–Anweisung im Programm zu entfernen brauchen. Allerdings ist das noch nicht alles: aktivieren wir nun noch den Optimierer, so sollte der erkennen, dass hinter `TRACE(..)` ein leerer Methodenkörper steckt und die Zeile komplett weg optimiert werden kann. Damit sind dann auch mögliche Effizienzprobleme beseitigt.

Wie Sie bemerkt haben werden, nutzen wir für die Ausgabe der Informationen unsere Stringkodierungsmethoden. Das ist zwar einerseits mit recht komplexen Ausgabestrings verbunden, andererseits sind diese aber sehr aussagekräftig und sie entheben uns der Notwendigkeit, für jeden Datentyp den `operator<<` aus der Streamklasse zu implementieren, d.h. diese Tracemethode funktioniert immer, sobald wir die Konversionsmethoden implementiert haben.

3.4.3 Konditionelle Trace-Klassen

Diese einfache Trace–Technik lässt sich leicht an den Bedarf anpassen. Soll beispielsweise nur unter bestimmten Bedingungen der Ausdruck erfolgen und sind mehr Parameter auszugeben, so kann das Trace–Modul entsprechend erweitert werden:

```
template <class T1, class T2>
inline void TRACE(bool c, T1 const& t1,
                          T2 const& t2){
    #ifdef TRACE_ON
        if(c)
            trace_stream << to_string(t1) <<
                         << to_string(t2) << endl;
    #endif
};//end function

...
long i; double d1; string s;
...
TRACE(i>5,s,d1);
```

Was den Detailreichtum und die Struktur der ausgegebenen Information angeht, so überlasse ich dies Ihrem Bedarf und Ihrer Fantasie. Handelt es sich beispielsweise um Langläufer, in denen aber relativ wenig passiert, oder um Prozesse mit mehreren parallelen Teilprozessen (*threads*), so kann eine Ergänzung durch Zeitmarken recht sinnvoll sein.

3.4.4 Trace-Gruppen

Die Trace–Anweisungen lassen sich zwar nun nach Bedarf ein- oder ausschalten, aber eine Programmverfolgung liefert unter Umständen immer noch zu viel Datenmüll. Die bedingte Ausgabe ist da auch nur eine Teilhilfe, da die Kontrollen immer nur lokal eingebaut werden können und gegebenenfalls bei unterschiedlichen Fragestellungen in weiten Programmteilen ausgetauscht werden müssen. Als Modellerweiterung sehen wir zusätzlich folgende Steuerkriterien vor (die im Übrigen auch bei Debuggern zu finden sind):

(a) Die Informationserfassung soll erst nach einer bestimmten Anzahl von Trace–Ereignissen erfolgen und danach auch nur für eine begrenzte Zahl von Ereignissen aufgezeichnet werden. Genauer: Das Programm soll zunächst n-mal die Trace–Anweisung x in der Methode y passieren, bevor die Aufzeichnung aktiviert wird, nach weiteren m Durchläufen ist die Aufzeichnung wieder zu deaktivieren.
(b) Die Informationserfassung soll für Gruppen von Trace–Anweisungen ein- und ausgeschaltet werden können. Die bedingte Aktivierung/Desaktivierung nach (a) und sinnvollerweise auch eine konditionelle Aktivierung, wie wir sie bereits oben beschrieben haben, schaltet also nicht eine Trace–Anweisung, sondern direkt beispielsweise alle Anweisungen im betreffenden Programmbereich.

Damit kommen wir schon ein erhebliches Stück weiter. Zentrale Schalter lassen sich in den meisten Fällen gut Positionieren und Bedienen, und um die vielen abhängigen Detailschalten muss man sich dann meist nicht mehr kümmern.

Zur Realisierung dieser Anforderungen müssen wir nun ein echtes Trace–Modul implementieren (*wir habe zwar oben schon von einem Modul gesprochen, aber bisher handelt es sich nur um eine Header–Datei mit Template–Funktionen*). Damit der Optimierer weiterhin in der Lage ist, ungenutzten Code bei abgeschaltetem Tracing zu entfernen, dürfen wir an dem oben eingeführten Programmschema nichts ändern (*Trace–Objekte einer speziellen Trace–Klasse anstelle der Trace–Methoden kommen also nicht in Frage*). Da aber einiges an Kontrollinformationen gespeichert werden muss, benötigen wir eine Implementationsdatei mit statischen Variablen, die natürlich auch nur bei aktiviertem Tracing aktiv werden.

```
static ofstream tr_out("TRACE.txt");

static struct TraceOptions{
    bitset<MAX_TRACE_GROUPS> activ_groups;
    vector<int> t_start(0,MAX_TRACE_GROUPS),
```

```
                  t_end(numeric_limits<int>::max(),
                                   MAX_TRACE_GROUPS),7


         t_count(0,MAX_TRACE_GROUPS);
} tr_opt;//end struct
```

Die statischen Variablen öffnen beziehungsweise überschreiben eine Ausgabedatei und stellen Merker für die Aktivitätskennung und Aufrufzählung von Gruppen bereit. Hierzu bietet sich natürlich der Einsatz der im letzten Kapitel vorgestellten Container für logische und ganzzahlige Größen an. Das Aktivieren und Deaktivieren von Gruppen wird mit Hilfe von Funktionen mit offenen Parameterlisten durchgeführt. Der erste Parameter gibt dabei die Anzahl der folgenden an, die von der Funktion sonst nicht festgestellt werden können

```
void TRACE_Activate(int anz, int gr,...){
#ifdef TRACE_ON
    int i, *a;
    a=&gr;
    for(i=0;i<anz;++i){
        if(a[i]<MAX_TRACE_GROUPS)
            tr_opt.activ_groups.set(a[i]);
    }//endif
#endif
}//end function
```

Durch Einfügen eines zusätzlichen logischen Parameters können die angegebenen Gruppen auch konditionell aktiviert oder deaktiviert werden. Auf ähnliche Art werden die Zähler bedient:

```
void TRACE_CountProperties(int anz, int parms,...){
#ifdef TRACE_ON
    int i, *a;
    a=&parms;
    for(i=0;i<anz;i+=3){
        if(a[i]<MAX_TRACE_GROUPS){
            tr_opt.t_start[a[i]]=a[i+1];
            if(a[i+2]==-1)
                tr_opt.t_end[a[i]]=
                        numeric_limits<int>::max();
            else
                tr_opt.t_end[a[i]]=a[i+2];
```

[7] Die Klasse numeric_limits<..= wird in einem späteren Kapitel erläutert. Hier soll sie unabhängig vom implementierten Typ den größten positiven Wert für eine int-Variable ausgeben.

```
            tr_opt.t_count[a[i]]=0;
        }//endif
    }//endfor
#endif
}//end function
```

Die eigentlichen Informationsausgaben müssen nun nur noch durch Gruppennummern ergänzt werden. Für die konditionelle Ausgabe einer Information für eine bestimmte Gruppe erhalten wir beispielsweise

```
template <class T1>
inline void TRACEC(int group, bool c, T1 const& t){
#ifdef TRACE_ON
    if(c && TRACE_Active(group,true))
        TRACE_Stream() << "TRACE(Group(" << group
                       << "),"
                       <<  to_string(t) << ")"
                       << endl;
#endif
};//end function
```

Eine Referenz auf die Ausgabedatei sowie die Abfrage, ob die Gruppe aktiviert ist, werden durch Hilfsfunktionen übermittelt. Die Aktivitätsabfrage führt gleichzeitig die Zählung durch. Hierzu verwenden wir das Vorzeichen des Übergabeparameters. Bei negativem Parameter wird die Zählung durchgeführt, bei positivem nicht, so dass wir durch das Vorzeichen die Zählpositionen auswählen können. Die Trace–Information nur dann ausgegeben, wenn die lokale logische Variable den Wert `true` aufweist, die Gruppe aktiviert ist und der Aufrufzähler für die Gruppe zwischen dem Start- und dem Endwert liegt.

```
bool TRACE_Active(int group){
#ifdef TRACE_ON
    if(Abs(group)<MAX_TRACE_GROUPS){
        if(group<0)
            ++tr_opt.t_count[group];
        group=Abs(group);
        return (tr_opt.activ_groups[group] &&
                (tr_opt.t_count[group]>=
                 tr_opt.t_start[group]) &&
                (tr_opt.t_count[group]<
                 tr_opt.t_end[group]));
    }else
#endif
    return false;
}//end function
```

Einer Ablaufverfolgung mit Steuerungsmöglichkeiten der Ausgabe, wie sie ein Debugger bietet, steht nun nichts mehr im Wege. Für den persönlichen Bedarf werden Sie im Laufe der Zeit vermutlich noch die eine oder andere Funktion ergänzen, aber dieses „Schönen“ sei den jeweiligen Aufgabestellungen überlassen.

3.5 Objektstatistiken

Bei komplexeren Objekten, insbesondere solchen, die über `new` und `delete` erzeugt und vernichtet werden, kann die Gesamtausführungszeit eines Programms zu einem nicht unerheblichen Teil von der Objektverwaltung verursacht werden. Abhilfemaßnahmen bestehen meist aus der Optimierung der Speicherstrategien; die vom System angebotenen Strategien sind so angelegt, dass sie immer funktionieren müssen und im Mittel ein einigermaßen zufriedenstellendes Zeitverhalten aufweisen, jedoch muss die für einen speziellen Anwendungsfall keineswegs optimal sein.

Bevor wir uns in einem späteren Kapitel mit Optimierungsstrategien beschäftigen, ist es sinnvoll, erst einmal ein Werkzeug zu konstruieren, dass uns darüber Auskunft gibt, ob eine Optimierungsarbeit überhaupt sinnvoll ist. Die folgende Klasse gibt uns Auskunft,

- wie viele Objekte seit dem Programmstart überhaupt erzeugt worden sind,
- wie viele Objekte zum Zeitpunkt der Anfrage noch existieren und
- wie viele Objekte maximal gleichzeitig nebeneinander vorgelegen haben.

```
template <class T> class Statistik {
private:
    static int& total(){
        static int tot(0);return tot;}
    static int& living(){
        static int liv(0);return liv;}
    static int& maxliv(){
        static int mliv(0);return mliv;}
public:
    Statistik(){
        Statistik::total()++;
        Statistik::living()++;
        Statistik::maxliv() =
     max(Statistik::maxliv(),Statistik::living());
    }

    virtual ~Statistik(){
        Statistik::living()--;
    }

    static int total_objects(){
         return Statistik::total();}
```

```
    static int living_objects() {
         return Statistik::living(); }
    static int max_living_objects() {
         return Statistik::maxliv(); }

};//end class
```

Die Klasse wird nun für die Durchführung einer Statistik als Basisklasse einer Vererbungshierarchie eingesetzt,

```
class MyClass: public Statistik<MyClass> { ..
...
cout << Statistik<A>::total_objects() ;
```

und zwar, indem der Klassenname der erbenden Klasse direkt als Templateparameter eingesetzt wird. Da jeder in ein Template eingesetzte Datentyp zu einem individuellen Templatetyp wird, ermöglicht uns die Template-Definition der Statistikklasse, Objekte unterschiedlicher Klassen getrennt zu verfolgen.

Im `private`-Teil werden so genannte Singleton -Größen erzeugt. Singeltons sind Größen, die im gesamten Programm nur einmalig vorhanden sind und nicht dupliziert werden können. In C++ lassen sich Singletons auf sehr elegante Art in der angegeben Weise erzeugen: eine statische Methode benötigt kein Objekt, um aufgerufen zu werden, und wird programmweit nur einmal implementiert, und eine statische Variable innerhalb der statischen Methode behält ihren Inhalt zwischen verschiedenen Aufrufen bei. Zusätzlich sorgt der Compiler dafür, dass ein statisches Objekt korrekt initialisiert wird, bevor man es benutzt.

Man kann nun nicht nur jederzeit abfragen, wie die statistischen Werte einer Klasse aussehen, man kann außerdem auch prüfen, ob die Lebenszyklen der Objekte eingehalten werden oder ob irgendwo ein `delete` fehlt. Nimmt nämlich die Anzahl der lebenden Objekte ständig zu, obwohl dies aus theoretischen Gründen nicht der Fall sein dürfte, so stimmt etwas nicht.[8]

3.6 Laufzeitmessungen

Viele Entwicklungssysteme stellen zwar auch Werkzeuge zur Laufzeitkontrolle zur Verfügung, jedoch sind eigene Werkzeuge, die auch im Programm zum Beispiel für Nutzungsabrechnungen abgefragt werden können, an vielen Stellen ebenfalls recht hilfreich. Die C–Standardbibliothek stellt für die Laufzeitmessung die Funktion `clock()` zur Verfügung, die die vom Prozess verbrauchte Zeit in der Einheit `[CLOCKS_PER_SEC]` misst.[9] Um diese Zeitmessung herum muss

---

[8] Man kann natürlich auch an den Stellen die Anzahl prüfen, an denen man sie genau zu kennen glaubt.

[9] Damit ist nicht die Zeit seit Programmstart, sondern die tatsächliche Laufzeit, also Zeit seit Start abzüglich der Zeiten, in denen das Betriebssystem anderen Anwendungen Rechenzeit zugebilligt

implementieren wir die Klasse `Zeitmessung`, von der eine beliebige Anzahl von Objekten erzeugt werden kann, so dass auch Detailmessungen möglich sind. Einer Randbedingung müssen Sie sich allerdings von Anfang an bewusst sein: Die Zeitticks liegen bestenfalls in der Größenordnung von Millisekunden, was auf der Zeitskala der Prozessoren zwar noch nicht ganz an die Verhältnisse *Mensch–Kontinentalverschiebung* heranreicht, aber doch recht groß ist. Anwendungen, in denen dieses Werkzeug zum Einsatz kommt, sollten doch schon Laufzeiten im Sekundenbereich oder größer aufweisen. Das sind bei Entwicklungsarbeiten relativ wenige, und für einen Effizienztest von Algorithmen wird man dann schon mal auf

```
for(i=0;i<1000000;++i)...
```

zurückgreifen müssen, um die gewünschten Ergebnisse zu erhalten. Die Klasse erhält folgende Schnittstelle:

```
class Zeitmessung {
public:
    Zeitmessung(string bezeichnung);
    ~Zeitmessung();
    bool Zeitmarke(string kennung);
    bool Suspend(string kennung);
    bool Resume(string kennung);
    bool Stop();

    double TotalTime()const;
    friend ostream& operator<<(ostream& os,
                const Zeitmessung& z);
private:
    clock_t cl;
    double elapsed;
    Parameter p;
    enum {run,suspend,stop} status;
};//end class
```

Ein Objekt der Klasse beginnt mit seiner Deklaration mit der Messung. Die Objekte sind für zwei Anwendungszwecke ausgelegt:

(a) Messung von Ausführungszeiten für die Erstellung von Kontenstatistiken.

- ♦ `Suspend(..)` unterbricht die laufende Messung,
- ♦ `Resume(..)` setzt die Messung fort.

---

hat, gemeint. Angegeben wird die vom Prozess in Anspruch genommene Zeit, eine Unterteilung auf Thread-Ebene wird nicht durchgeführt. Ob das gemäß Handbuch so festgelegte Verhalten tatsächlich eingehalten wird, müssen Sie gegebenenfalls überprüfen (*vermutlich wird es häufig doch die Zeit seit Programmstart sein*). Wir gehen hier erst mal davon aus, dass das so ist.

Mit der Funktion `TotalTime()` kann die in Anspruch genommene Prozessorzeit in der Einheit [Sekunde] abgefragt werden.

(b) Protokollierung von Abläufen für eine Programmstatistik.

- `Zeitmarke(..)` generiert Zwischenmeldungen ohne Unterbrechung der Messung. Dabei kann es sich um Informationen handeln, an welcher Stelle sich das Programm gerade befindet, aber beispielsweise auch um Zwischenergebnisse aus der Rechnung usw. Sofern es sich um Informationen handelt, die aus mehreren Teilinformationen bestehen, sind diese als Parameterstring zu organisieren.
- `operator<<(..)` ermöglicht das Schreiben der Zeitprotokolls auf eine Ausgabeeinheit.

Alle Meldungen können durch einen Text genauer gekennzeichnet werden. Die `Stop()`–Funktion hält die Messung entgültig an. Die Protokolldaten werden in einen Parameterstring geschrieben. Abgesehen davon, dass die im letzten Kapitel definierte Klasse schnell zum Einsatz kommt,[10] eröffnen wir dadurch auch die Möglichkeit, die Zeitprotokolle automatisch auswerten zu können. Der Inhalt des Parameterstrings ist selbst erklärend; wir verteilen ihn hier zur besseren Übersichtlichkeit auf mehrere Zeilen:

```
Zeitmessung(
    Job(Start),
    Datum(Mon,Jun,03,14:48:46,2002),
    RuntimeInfo(
        Marke(Marke,t_ink(2.140),t_ges(2.140)),
        Suspend(Stop,t_ink(1.590),t_ges(3.730)),
        Resume(Start,t_pause(1.540)),
        Stop(,t_ink(1.480),t_ges(5.210))
    )
)
```

Die Zeitangaben `t_ink` und `t_pause` geben die Intervalle seit dem letzten Zugriff auf das Zeitobjekt an, `t_ges` gibt die gesamte seit der Objekterzeugung (*abzüglich der Pausen*) abgelaufenen Zeit an. Das Innenleben der Methoden ist recht einfach. Aufgrund verschiedener `const`–Vereinbarungen muss die Positionierung im Parameterstring in den Aufzeichnungsmethoden erfolgen:

```
p.Mark(0);
p.Mark(2);
p.Insert(p.Anzahl(),kennung);
p.Laynull();
```

Aufgabe. Implementieren Sie die Klasse.

[10]Wenn man sich den String genauer anschaut, wird man feststellen, dass die Verwendung eines Parameterstrings in der Klasse Zeitmessung nicht unbedingt notwendig gewesen wäre, jedoch lässt sich so die Verwendung der Klasse ganz gut üben.

Anmerkung. Da die Rechner immer schneller werden und Arbeiten je nach Programmdesign (und zukünftig möglicherweise auch unabhängig davon) auf mehreren Prozessoren erledigen, ist es bei der Prüfung von Algorithmen oft nicht ganz einfach, auf dieser einfachen Basis vernünftige Ausführungszeiten zu erhalten. Zu berücksichtigende Gesichtspunkte sind:

- System. Betriebssystem, Systemumgebung und Hardware können zu unterschiedlichen Aussagen führen. Beispielsweise kann auf einem anderen System der Hauptspeicher unzureichend sein, was zu Plattenauslagerungen und damit zu einem anderen Zeitverhalten führt.
- Compiler. Über normalen und optimierten Code haben wir bereits hinreichend oft gesprochen, so dass klar sein sollte, dass man sich hierum zu kümmern hat.
- Anwendungsbezug. Steckt hinter einem zu testenden Programmteil eine konkrete Anwendung, so sollten die Datenmengen auch unter diesem Gesichtspunkt konfektioniert werden.
- Wiederholungen. Wenn die Erhöhung der Datenmenge ausscheidet, die erreichbaren Zeiten aber immer noch zu klein sind, kann man den Algorithmus auch mehrfach hintereinander ausführen. Achten Sie aber dabei darauf, dass das Verhalten des Programms in jedem Durchlauf auch das Gleiche ist ! Bubblesort ändert beispielsweise seine Laufzeit, wenn man ihn mehrfach auf das gleiche Feld loslässt.
  Wenn Sie zwischen den einzelnen Läufen des zu testenden Programmteils weiteren Code unterbringen (beispielsweise eine Reinitialisierung des Feldes für Bubblesort), ist die Laufzeit mit und ohne den zu testenden Programmteil zu ermitteln, um dessen tatsächliche Laufzeit zu erhalten. Denken Sie daran: der zu testenden Programmteil soll eine hinreichende Laufzeit verursachen, nicht das ganze Programm !

3.7 Datenkompression

3.7.1 Ein wenig Theorie ...

Speicherplatz im Hauptspeicher eines Rechners ist angesichts der heutigen Technik zwar kaum noch ein Engpass, aber da mit den Möglichkeiten auch häufig die Anforderungen steigen, wollen wir uns trotzdem diesem Thema widmen. Dass es eng wird, merkt man meist an intensiver Plattenarbeit und langen Laufzeiten. Eine Möglichkeit zur besseren Resourcennutzung bei solchen Problemen ist die Kompression vorübergehend nicht benötigter Daten.[11] Voraussetzungen dafür:

[11] Außer Platzgewinn kann auch Zeitgewinn dabei herausspringen: Ein leider nicht mehr aktuelles Betriebssystem pflegte Programmkode in verschlüsselter Form auf der Platte abzulegen, da das Lesen weniger Daten von der Platte und deren Dekompression etwa 20% schneller zu erledigen war als das reine Lesen der unkomprimierten Daten. Heute ist der Trend anders herum: Selbst

- In der Anwendung sind hinreichend große Datenblöcke vorhanden, die zeitweise nicht in der Rechnung benötigt werden und daher komprimiert werden können.
- Bei Reaktivierung eines Teil der komprimierten Daten kann ein adäquater Anteil bisher benötigter Daten komprimiert werden, so dass sich die Anforderungen an die Menge des Speichers insgesamt nicht ändern.
- Die Daten sind für eine Kompression geeignet, das heißt durch eine Kompression wird ein merklicher Anteil des Speicherplatzes frei.

Sind die Daten nicht für eine Kompression geeignet, so müssen andere Maßnahmen wie beispielsweise eine Auslagerung in eine Datei ergriffen werden (*siehe nächstes Teilkapitel*). Eine Datenkompression beruht im wesentlichen auf der geschickten Zusammenfassung gleicher Datenmuster. Ob zu bearbeitende Daten diese Eigenschaften besitzen, lässt sich aber im Einzelfall oft schlecht abschätzen.

Als Grundlage für die Datenkompression sollte einer der frei im Internet verfügbaren Quellkodes für Kompressionssoftware verwendet werden. Wir werden hier nur kurz auf Kompressionsalgorithmen eingehen. Die Grundprinzipien sind zwar leicht zu verstehen, und erste Implementationsversuche werden Sie vermutlich auch schnell hinter sich bringen. Allerdings sind diese meist recht frustrierend (*ich schließe mich da selbst mit ein*), da der Rechner Minuten dafür benötigt, womit die Bibliotheksfunktion in weniger als einer Sekunde fertig ist. Die Optimierung ist hier ein recht hartes und spezielles Geschäft, und wenn man nicht speziell darauf aus ist, in diesem Bereich zu entwickeln, ist es besser, nach Kenntnisnahme der Theorie auf Vorhandenes zurück zu greifen.

Bei Kompressionsverfahren müssen wir strikt zwischen informationserhaltenden (*zum Beispiel lzw*, *zip*) und informationsverändernden (*zum Beispiel jpeg*) Algorithmen unterscheiden. Die zweite Gruppe ist nur für die Informationsversorgung des Menschen sinnvoll: Wenn von einer Nachricht bei einer bestimmten Darstellungsart nur jedes zweite Bit wahrgenommen werden kann und auf andere Darstellungsweisen, bei der auch die restlichen Informationen erkannt werden könnten, verzichtet wird, können diese Informationen fortgelassen werden – mit der Folge erstaunlicher Kompressionsraten. Als Beispiel denken Sie an Bilder, in denen das Auge zwar Strukturen von 0,1 mm auflösen kann, jedoch nur, wenn der Kontrast einen bestimmten Grenzwert überschreitet. Darunter muss die Gebietsgröße erheblich größer sein, um dem Auge eine Trennung zu ermöglichen. Verzichten wir auf eine Vergrößerung, können die Unterschiede im Originalbild natürlich entfallen.[12]

Dateien, die praktisch leer sind, werden in „voller Länge" auf der Platte abgelegt. Die immer weiter steigende „Rechen- und Speicherpower" verführt offenbar viele Entwickler dazu, die eigene Arbeit auch nur noch „mit der groben Kelle" zu erledigen.

[12] Die Unterschiede in Kodierungs- und Dekodierungsrichtung sind oft erheblich. Dauert die Überlegung, was in einem digitalen Video an Information alles fortfallen kann, beispielsweise eine Stunde, so ist das Ergebnis dem Betrachter meist in wenigen Sekunden präsentiert, wobei der Rechner auch dabei die meiste Zeit noch philosophische Betrachtungen darüber anstellen könnte, welches Vergnügen der Mensch wohl dabei empfindet, sich einzelne Bilder mit geringfügigen Veränderungen über längere Zeiträume hin anzusehen.

Wir können hier natürlich nur informationserhaltende Algorithmen ausnutzen. Hier werden folgende Mechanismen zur Datenverkürzung genutzt:

(a) **Wiederholungszählung**. betrachten wir die Nachricht

```
Blah blah blah blah blah blah
```

Die Zeichenfolge „`lah b`“ tritt wiederholt auf. Wir geben sie nur einmal wieder zuzüglich der Länge und der Anzahl der Wiederholungen:

```
[1,1] B [5,5]lah b [3,1] lah
```

Insbesondere Bilder weisen viele sich wiederholende Teile auf, so dass größere Einsparungen möglich sind.

Kompression und Dekompression haben ein unterschiedliches Zeitverhalten. Eine komprimierte Sequenz wiederherzustellen ist sehr effizient durchführbar (*Aufgabe 3.7-1*), ein Algorithmus, der Wiederholungen erkennt, ist schon schwieriger effizient zu gestalten.

(b) **Häufigkeitsalphabet**. bei der Zählung der verschiedenen Buchstaben eines Textes stellt man schnell fest, dass beispielsweise der Buchstabe „e“ sehr häufig auftritt, der Buchstabe „y“ aber fast nie. Beide belegen im normalen Alphabet den gleichen Speicherplatz. Wir können jedoch ein neues Alphabet konstruieren, bei dem einige Zeichen eine kürzere, andere eine längere Kodierung aufweisen. Die häufigsten Zeichen sollen natürlich kurze Kodierungen besitzen:

- ♦ Der Text wird analysiert und die Buchstaben der Häufigkeit nach geordnet notiert.
- ♦ Jedes Zeichen des neuen Alphabets beginnt mit einem 0–Bit als Startzeichen.
- ♦ An das Startbit werden so viele 1–Bits angehängt, wie der Position des neu zu kodierenden Buchstabens in der Häufigkeitstabelle entspricht.

Das Wort „ein“ weist mehr oder weniger die häufigsten Buchstaben in einem Text auf und hätte beispielsweise die Kodierung `01 011 0111`. Wer mitgezählt hat, wird festgestellt haben, dass für die Kodierung des Wortes 11 Bit ausreichen gegenüber 24 im Normalalphabet, was nur 46% des ursprünglichen Speicherplatzes entspricht.

Kodierungen dieser Art sind insbesondere für Texte interessant, da einige Zeichen meist gar nicht auftreten, andere dafür um so häufiger. Mit mehr Aufwand lässt sich das Verfahren erweitern, in dem komplette Silben dem Alphabet als weitere Zeichen zugeordnet werden. Neben dem Text muss in dem komprimierten Daten natürlich auch eine Tabelle der Zeichenreihenfolge bereitgestellt werden, was den Gesamterfolg wieder etwas schmälert..

(c) **Zeichenkettenhäufigkeit**. Diese Kodierungsform ist die komplexeste, insbesondere was die Konstruktion schneller Algorithmen angeht. Zur Initialisierung

wird eine Tabelle mit den Buchstaben des Alphabets und ein leerer Zeichenkettenspeicher P angelegt. Anschließend läuft folgender Algorithmus ab (*formale Sprachdarstellung, nicht C !*):

```
while(C){                    // C: Nächstes Zeichen
   P′ ← P + C
   if(!tab.find(P′)){
      OutBuffer C tab.pos(P)
      tab.insert(P′)
      P ← C                  //**//
   }else{
     P ← P′
   }//endif
}//endwhile
```

Im Klartext: an den Zeichenkettenspeicher werden so lange weitere Zeichen angefügt, wie die entstehende Zeichenkette noch in der Tabelle zu finden ist. Ist eine noch nicht notierte Zeichenkette entstanden, wird die letzte Position im Ausgabepuffer notiert und die neue Kette angefügt.

Soll beispielsweise die Zeichenkette „111111111" gespeichert werden, so liegt zunächst nur die Zeichenkette „1" in der Tabelle vor, die im „leeren" Zustand beispielsweise 25 Einträge aufweist. Die zugehörende Indexnummer 1 wird im Ausgabepuffer gespeichert, gleichzeitig die Kette „11" an Position 26 angefügt. Nach Lesen der nächsten beiden Zeichen steht die Folge 1 26 im Ausgabepuffer, „111" als neue Kette 27 in der Tabelle. Ist der String abgearbeitet, steht die Folge 1 26 27 27 im Ausgabepuffer, „1111" als letzte (*nicht im Ausgabepuffer verwendete*) Kette in der Tabelle. Aus neun Zeichen sind nun vier geworden. Je nach Wiederholungsfrequenz können bei diesem Verfahren sehr lange Zeichenketten entstehen, die nur durch eine Ziffer kodiert werden.

Das Suchen nach einer Zeichenfolge im Speicher ist natürlich eine recht delikate Angelegenheit. Um zu lange Suchvorgänge zu vermeiden, werden die Tabellen nach Erreichen einer bestimmten Größe regeneriert. Da sie außerdem den Kompressionseffekt großenteils zunichte machen würden, werden sie auch nicht mit den komprimierten Daten gespeichert, sondern beim Dekodiervorgang rekonstruiert. Um diesen Vorgang erkennen zu können, betrachte man //**// im Kodierungsalgorithmus. Das letzte Zeichen eines neuen Tabelleneintrags in der Kodierungstabelle bildet das erste Zeichen einer neuen Kodierungskette. Das erste kodierte Zeichen ist immer einer der Einzelbuchstaben der Initialisierungstabelle, so dass sich der Algorithmus eindeutig starten lässt. Ist cp die Kodenummer des letzten dekodierten Strings, so lautet der Forsetzungsalgorithmus

```
while(cw){                    // nächster Code der Eingabe
   if(tab.find(cw)){
     OutPut C tab.string(cw)
     P C tab.string(cp)
     tab.insert( tab.string(cp) + tab.string(cw)[1] )
```

```
   }else{
     OutPut C tab.string(cp) + tab.string(cp)[1]
     tab.insert( tab.string(cp) + tab.string(cw)[1] )
   }//endif
   cp C cw
}//endwhile
```

In unserem obigen Beispiel lesen wir zunächst die erste Eins ein und schreiben sie auf den Ausgabepuffer und den Zwischenspeicher. Wird nun die Kodenummer „`26`" gelesen, so wird „`11`" auf den Ausgabepuffer, in die Tabelle und in den Zwischenspeicher geschrieben. Bei der Kodenummer „`27`" steht nun überall einheitlich „`111`". Beim zweiten Auftreten der „`27`" wird wiederum „`111`" auf die Ausgabeeinheit geschrieben, nun aber „`1111`" mit der Kodenummer „`28`" in die Tabelle. Vollziehen Sie an weiteren Beispielen oder abstrakt noch einmal nach, dass auf diese Wiese die vollständige Tabelle wiederhergestellt wird.

> **Aufgabe.** Falls Sie zum besseren Verständnis es doch einmal mit einer Implementation versuchen wollen, können Sie mit einem `map`–Container für die Listen und der Pufferklasse aus einem späteren Kapitel für die Zeichenketten beginnen. Schrittweise können Zeichenketten mit zwei bis drei Zeichen durch Indexcontainer ersetzt werden, je nach Quelle (*Text, ASCII, Binärdaten*). Da bei der Kodierung noch einige weitere Strukturierungen anfallen, können Sie aber nur die Laufzeiten Ihres Kodes mit dem der Bibliothek vergleichen.

3.7.2 .. und eine Kompressionsklasse für die Praxis

Wir bedienen uns einer der freien Bibliotheken. Die dort gefundenen Quellen werden zwar meist recht umfangreich sein, benötigt werden aber im allgemeinen nur zwei Funktionen:[13]

```
compress(UC* buffer,long* compLen,const UC* buf,
          long inLen, long level);
uncompress(UC* buf,long* orgLen,const UC* buffer,
           long compLen);
```

Die Datenpuffer sind vom Typ (`typedef UC unsigned char`), und bei Beachtung unserer Regeln aus Kapitel drei dürfte die Bedeutung der Schnittstelle

[13] Hier zum Beispiel für eine „zlib"- Implementation nach RFC 1931. Aus der Masse der Funktionen schnell die benötigten herauszufinden, ist ebenfalls eine der Standardübungen für angehende Programmierer. Es gilt den Blick dafür zu schärfen, was man will und welche Funktionsschnittstelle dem eigenen Wunsch entspricht. Zusammen mit dem Funktionsnamen und der Dokumentation sollten die Funktionen dann relativ schnell identifiziert sein. Anfänger stehen allerdings oft relativ ratlos vor der großen Menge an Bibliotheksfunktionen. Auch hier hilft nur Üben: Versuchen Sie an der Datenkompression zu erkennen, worum es geht, und wiederholen Sie das Ganze anschließend mit dem Einlesen von JPEG- oder TIFF-Bildern durch Untersuchung der hierfür frei erhältlichen Bibliotheken.

(*fast*) klar sein. `inLen` und `compLen` sowie `compLen` und `orgLen` beschreiben die Längen von Eingabe- und Ausgabepuffern, wobei die gültigen Längen der Ausgabepuffer von den Funktionen berechnet werden. Die Ausgabepuffer sind allerdings keine Rückgabeparameter, das heißt sie müssen vom rufenden Programm in hinreichender Größe bereitgestellt werden.

Was heißt nun „hinreichende Größe"? Wie die Header–Dateien der Kompressionsmodule den Nutzer informieren, ist das Ergebnis eines Kompressionsversuches nicht vorhersagbar. Es kann sogar passieren, dass der Speicherbedarf nach der Kompression größer ist als vorher. Das ist dann der Fall, wenn überhaupt keine Muster in den Originaldaten zu erkennen sind. Im Grunde sucht ja jeder Kompressionsalgorithmus nach wiederkehrenden Sequenzen, die dann durch eine kürzere Sequenz ersetzt werden. Wiederholt sich jedoch kaum etwas, so werden immer mehr Testsequenzen erzeugt und gespeichert. Die den Testsequenzen entsprechenden „Alphabetzeichen", die anstelle der Originalzeichen in den Datenstrom eingefügt werden, werden länger als die Originale selbst, was zu der Zunahme des Speicherbedarfs führt.[14]

Das wird teilweise sogar in Prüfverfahren ausgenutzt: Datenverschlüsselungen lassen um so weniger Rückschlüsse auf das Original zu, je mehr das Ergebnis einer Zufallgröße ähnelt. Ein erfolgloser Kompressionsversuch mit einem Chiffrat zeigt zumindest an, dass der Verschlüsselungsalgorithmus diese Anforderung erfüllt (*was aber in der Regel noch nicht allzu viel heißt, wenn man Profis zu Gegnern hat*)

Zurück zu unserem Kompressionsalgorithmus : Es macht unter dem Gesichtspunkt der Speicherplatzeinsparung wenig Sinn, für die Kompression riesige Speichermengen zu belegen, um allen Eventualitäten begegnen zu können. Statt dessen empfiehlt es sich, die Daten in kleinere Blöcke zu zerlegen (*beispielsweise 64 kByte*) und diese nacheinander zu verarbeiten. Die komprimierten Daten lassen sich problemlos auf einem linearen Puffer ablegen:

Len_1	Daten_1	Len_2	Daten_2	0

```
        -- Len_1 Bytes -->      ---Len_2 Bytes  ---->
```

Der Speicherbereich wird zunächst nach einem „Erfahrungswert" eingerichtet. Geht man im Mittel von einem Kompressionserfolg von 45% aus, so ist *Originallänge*0,55* ein passender Startwert. Ist dieser Wert zu groß, so hat man Speicherplatz verschenkt und wird gegebenenfalls den Speicher verkleinern, ist er für die Aufnahme der komprimierten Daten zu klein, muss muss er vergrößert werden, wobei die Anzahl der Vergrößerungsschritte klein bleiben muss und gleichzeitig nicht ins Gegenteil, das heißt eine Überreservierung von Speicherplatz,

[14] Am leichtesten ist dies beim Häufigkeitsalphabet (*Hufman.Kodierung*) einzusehen. Jedes ASCII-Zeichen belegt ein Byte, die acht Bit eines Bytes sind jedoch nach sieben ASCII-Zeichen bereits „verbraucht" und die weiteren ASCII-Zeichen belegen mehr Platz als die Originalzeichen. Sind alle Zeichen ungefähr gleich häufig, so wird der Vorteil bei sieben Zeichen schnell vom Nachteil bei den restlichen des Alphabets aufgefressen.

umschlagen darf. In beiden Fällen sind aufwendige Kopieraktionen notwendig. Bei mehreren Datenkompressionen kann man das Kompressionsverhältnis und die nachträgliche Speicherplatzvergrößerung als gleitenden Mittelwert nachführen und sich so einem guten Arbeitswert nähern (*gleichbleibende Datenstrukturen vorausgesetzt*). Wir setzen diese Strategie im weiteren voraus, ohne noch weiter darauf einzugehen.

Aufgabe. suchen Sie im Internet nach Quellkode unter den Stichworten „zip" oder „zlib". Sie könen natürlich auch irgendein anderes Kompressionsverfahren einsetzen (LZW, LHA, ...), wenn Sie an geeignete Quellen kommen.

Wir definieren nun eine Speicherklasse, die entgegengenommene Daten komprimiert ablegt und erst bei der Abfrage wieder dekomprimiert. Nach den Eingangsbemerkungen ist es sicher sinnvoll, die Speicherung mehrerer Datenelemente zu ermöglichen. Ein Speicherobjekt liefert deshalb einen Handle zurück, mit dem auf die Daten wieder zugegriffen wird. Beim Auslesen der Daten wird zwischen Festhalten der komprimierten Daten und Löschen sowie zwischen der Erzeugung eines neuen Puffers für die unkomprimierten Daten und der Kopie in einen vom Anwender bereitgestellten Pufferbereich unterschieden:

```
class StoreCompress{
public:
    StoreCompress();
    ~StoreCompress();
    int   Push(const char* buf, int len);
    bool  Exchange(int key,const char* buf,
                   int len);
    int   Get(int key, char * buf, int len);
    char* Get(int key, int& len);
    int   Pop(int key, char * buf, int len);
    char* Pop(int key, int& len);
    bool  Erase(int key);
    long  Length(int key);
    int   CompressedLength(int key);
    bool  PutToStream(ostream& os, int key);
    int   GetFromStream(istream& is);
private:
    struct ce {
        ce(){..};
        ~ce(){..};
        inline void resize(){..};
        int key;
        int orgLen;
        int compLen;
        int res_len;
```

```
        char* buf;
    };
    std::vector <ce> data;
};//end class
```

Die Funktionsweisen der Methoden erklären sich weitgehend selbst: Wir unterscheiden zwischen Methoden, die die komprimierten Daten weiter gesichert lassen (`get`) und solchen, die mit dem Rücklesen die komprimierten Daten löschen (`pop`). Jede Funktion ist in einer Version für die Rückgabe auf vorhandenen Speicher und auf neu zu beschaffenden Speicher vorhanden. Bei der Kompression von Daten bleiben die Eingabedaten unverändert erhalten. Die Speicherung der komprimierten Daten eines Objektes erfolgt in einer Variablen der Struktur `ce`, die eine fortlaufende Zahl als Handle auf die Daten (*Schlüssel*) sowie die aktuellen Längen enthält. Für die Speicherung mehrerer komprimierter Datenobjekte deklarieren wir eine Variable des Typs `vector<ce>` . Die Verarbeitung erfolgt wie oben beschrieben blockweise. Die Speicherung eines Datenblocks erfolgt durch den Algorithmus:

```
ce * e;
data.push_back(ce());
e=&data.at(data.size()-1);
e->key=rand();
e->orgLen=len;
e->compLen=0;
e->buf.resize((ulong)(len*ratio));

for(i=0,j=0;i<len;i+=blen,j+=(dlen+sizeof(ulong))){
    compress(bb,&dlen,&buf[i],_min(BLKLEN,len-i));
    e->compLen+=dlen;
    if(e->buf.size()<(j+2*sizeof(ulong)+dlen))
        e->buf.resize(e->buf.size()+expand*blen);
    memmove(&(e->buf[j]),&dlen,sizeof(ulong));
    memmove(&(e->buf[j+sizeof(ulong)]),bb,dlen);
}//endfor
dlen=0;
memmove(&(e->buf[j]),&dlen,sizeof(ulong));
```

Entsprechend ist das Rücklesen der Daten zu organisieren. Zwei Methoden erlaube das Sichern der komprimierten Daten in eine Datei sowie das Rücklesen. Wir machen hierbei wieder von bereits implementierten Werkzeugen Gebrauch und geben dem Dateieintrag folgende Form:

```
<storecompress>
    <OrgLen> <int>...</int></OrgLen>
    <compLen><int>...</int></CompLen>
    <BufLen> <int>...</int></BufLen>
    <octet>...</octet>
</storecompress>
```

Wie aus dem Code für das Sichern der Daten abzulesen ist, enthält das Attribut `compLen` die Länge der komprimierten Daten ohne Steuerzeichen auf dem Puffer. Da der Zusammenhang zwischen der Länge der Daten und der Anzahl der Steuerzeichen bei einer Dateispeicherung nicht mehr zwingend gegeben sein muss (*die Blockgröße kann sich seit der Sicherung verändert haben*), speichern wir die Blockgröße zusätzlich ab.

Aufgabe: Führen Sie die komplette Implementation durch.

Wie Sie schnell feststellen werden, sind für den Datenteil `<octet>` spezielle Maßnahmen notwendig. Eine Möglichkeit, die Binärdaten problemlos in den String zu packen, wäre eine Umwandlung in Hexadezimalkode. Da sich hierbei die Datenmenge allerdings verdoppelt, muss die Datenkompression deutlich höher als 50% liegen, um noch irgendeinen Nutzeffekt zu haben (schließlich kostet das Alles auch Laufzeit).

Mit der ebenfalls problemlosen base64-Kodierung, die an anderer Stelle diskutiert wird, werden aus drei Binärzeichen „nur noch" vier kodierte Zeichen. Größere Texte lassen sich in dieser Weise schon effizient stauchen, und auch bei reinen Binärdaten, die ja ebenfalls in `<octet>` - Kodierung gespeichert werden, macht sich die Kompression meist positiv bemerkbar.

Als dritte und datentechnisch effektivste Option können die Binärdaten „as is" abgelegt werden. Auch wenn das in diesem Zusammenhang wohl nicht eintritt, müssen Sie allerdings Vorsorge treffen, dass ein zufällig im Oktetstring auftretender Tag `</octet>` nicht als Ende interpretiert wird (die Kodierungsmethode soll ja universell und nicht speziell für diesen Fall sein). Beispielsweise können alle Schrägstriche innerhalb des Binärdatenteils gedoppelt und beim Rücklesen jeweils einer wieder entfernt werden.

Ein weiteres Problem entsteht allerdings bei der Speicherung der Daten. Während die ersten beiden Methoden keinerlei Probleme bereiten, MUSS die Datei im dritten Fall als Binärdatei deklariert werden. Unterbleibt dies, so interpretiert das System die Daten als Textdaten und fügt zusätzliche Steuerzeichen ein, die zusammen mit dem etwas anders gearteten Rücklesen von Textdaten zu nicht mehr interpretierbarem Müll führen.

3.8 Temporäre Dateien

Alternativ zu einer Datenkompression bietet sich die Auslagerung von Daten in Dateien an. Das ist im Grunde trivial, wenn auch im allgemeinen bei der Strukturierung von Dateien etwas Sorgfalt sinnvoll ist, da sie vielfach eine deutlich längere Lebenserwartung haben als eine laufende Anwendung und auch zu einem späteren Zeitpunkt noch zweifelsfrei bearbeitet werden sollen. Ein Beispiel haben wir im letzten Abschnitt mit der Speicherung komprimierter Daten bereits vorgestellt und wollen an dieser Stelle nicht weiter darauf eingehen, sondern uns auf die Anlage und Verwaltung temporärer Dateien beschränken, also solcher Dateien, die (*spätestens*)

nach Ablauf der Anwendung gelöscht werden können. Spezifizieren wir zunächst die Anforderungen an eine Verwaltung:

- Auf Anforderung sind Namen für temporäre Dateien zu generieren.
- Die Sicherung in temporären Dateien kann mehrstufig erfolgen, das heißt die aktuellen neuen Daten eines bereits gesicherten Objektes können ebenfalls gesichert werden, ohne dass die alten Daten dabei gelöscht werden dürfen (*Kellerliste oder Stack*).
- Temporäre Dateien können in permanente Dateien umgewandelt werden.
- Bei Ende der Anwendung sind alle noch vorhandenen temporären Dateien zu löschen.

Zur Durchführung dieser Aufgaben ist eine Buchführung über die erzeugten und vergebenen Dateinamen notwendig. Wir realisieren diese mit Hilfe der Parameterstringklasse. Mit `insert` oder `erase` können einzelne Dateinamen hinzugefügt oder entfernt werden und durch die Unterebenenstruktur ist auch die Verwaltung der internen Kellerlisten realisierbar. Wir legen dazu fest: müssen auch neuere Versionen einer bereits gesicherten Datenstruktur erneut gesichert werden, so führt dies zu folgendem geschachtelten Parameterstring:[15]

```
...(datei_1) ...             -->
...(datei_2(datei_1)) ...    -->
...(datei_3(datei_2(datei_1))) ...
```

Für die Verwaltung der Namen der temporären Dateien deklarieren wir in einem Verwaltungsmodul eine statische Variable, deren Klasse von `Parameter` erbt und als einzige Erweiterung das rekursive Löschen aller Dateinamen am Ende der Bearbeitung vorsieht:

```
static class TFNames: public Parameter{
public:
    TFNames(){};
    ~TFNames()  {erasefile(parsed);};
    void erasefile(vector<entry>& ve);
} tfnames;

void TFNames::erasefile(vector<entry>& ve){
    int i;
    for(i=0;i<ve.size();i++){
        remove(ve[i].value.c_str());
        erasefile(ve[i].sub);
    }//endfor
}//end function
```

[15] Zu besseren Lesbarkeit verwende ich an dieser Stelle nicht die oben eingeführte Tagschreibweise, sondern runde Klammern, die hier die Tags ersetzen. In der Implementation bleibt es natürlich bei den Tagvereinbarungen

Die Kommunikation mit dem Verwaltungsmodul erfolgt mittels der Funktionen:

```
string  TFileName();
bool    TInsert(string name);
string  AllNames();
int     TNameErase(string fname);
int     TFileErase(string fname);
int     TFileEraseAll();
string  TPush(string fname);
string  TPop(string fname);
string  TSwap(string fname,int depth);
```

Die Funktionsnamen sagen vermutlich schon genug über die Funktion aus, und auch die Implementationen der einzelnen Funktionen sind relativ einfach beziehungsweise nochmals recht brauchbare Übungen für die Verwendung des Parameterstring–Konzepts. Wir können die Ausführungen daher recht kurz halten.

Zur Anforderung eines neuen Dateinamens dient die Funktion `TFileName`. Sie erzeugt mit Hilfe eines Zufallzahlengenerators einen Dateinamen: Die vom Generator erzeugten Zufallzahlen werden mit einem Hexdazimal–Konverter in einen für Dateinamen gültigen String umgewandelt, bis eine vorgegebene Namenslänge erreicht ist. Ein hinzugefügter Suffix *.tmp* kennzeichnet den Namen als temporären Dateinamen, falls eine manuelle Bearbeitung eines Verzeichnisses notwendig wird. Der erzeugte Dateiname darf natürlich noch nicht in der Namenssammlung vorhanden sein. Halbwegs sinnvolle Generatoren und Namenslängen vorausgesetzt, muss das nicht extra überprüft werden; der etwas paranoide Entwickler, der auch dies sicherheitshalber prüfen möchte, denke allerdings daran, dass aufgrund der Schachtelung rekursiv zu prüfen ist! Der neue Dateiname wird an den anfordernden Programmteil ausgeliefert und gleichzeitig an die Namenslisten in der ersten Ebene angehangen.

`TInsert` erlaubt das Einfügen weiterer Dateinamen, die vom Anwender selbst erzeugt wurden oder aus anderen Quellen stammen, in die Liste, um das automatische Löschen zu ermöglichen. In der Routine wird nicht auf Doppeleinträge geprüft.[16] Der übergebene String wird in der ersten Ebene eingefügt. Da hierzu das Einfügeschema der Klasse `Parameter` verwendet wird, können mehrere Dateinamen gleichzeitig sowie auch Schachtelungen eingefügt werden. Wegen des einschränkten Zugriffs der Funktionen dieses Moduls auf die Namensverwaltung, die wir im weiteren genauer beschreiben, sollte man sich aber schon recht gut überlegen, was hier Sinn macht.

`TNameErase` und `TFileErase` erlauben das Löschen von Namen aus der Verwaltungsliste, wobei zwischen Löschen der Namen alleine und dem gleichzeitigen Löschen der Dateien unterschieden wird. Das selektive Löschen des Dateinamens aus der Verwaltungsliste erlaubt es dem Anwender, ursprünglich temporäre Daten auch über das Ende des laufenden Programms hinaus zu sichern. Der

[16] Möglicherweise liegt es sogar in der Absicht des Anwenders, Mehrfacheinträge zu erzeugen.

Löschbefehl bezieht sich immer auf die erste Namensebene, das heißt Namen in tieferen Ebenen werden nicht gefunden. Ein Löschen in der Hauptebene führt allerdings auch zum Löschen aller gegebenenfalls vorhandenen Unterebenen, das heißt je nach Vorgehensweise verschwindet ein ganzer Schwarm von Dateien von der Platte oder bleibt nach Beenden des Programms ungelöscht zurück..

Für Schachtelungen führen wir die Funktionen `Tpush()`, `Tpop()` und `Tswap()` ein: sollen gespeicherte Daten weiter in den Hintergrund geschoben werden, weil in der Zwischenzeit davon abgeleitete Daten ebenfalls gesichert werden müssen, so erzeugt die Funktion `Tpush()` nach Auffinden des übergebenen Namens in der Hauptebene einen neuen Dateinamen und schiebt den alten Namen in die Unterebene (*siehe oben*). Der neue Dateiname wird als Rückgabewert an das rufende Programm geliefert. Das Verfahren ist rekursiv, das heißt bereits vorhandene Unterebenen werden weiter in den Hintergrund geschoben. Ist der übergeben Name nicht vorhanden, so hat die Funktion die gleiche Wirkung wie `TFileName()`. Der umgekehrte Vorgang wird durch die Funktion `TPop` ausgeführt:

```
...(Datei_3(Datei_2(Datei_1)))...        ====>
...(Datei_2(Datei_1)) (Datei_3) ...
TPop("Datei_3")   -->   "Datei_2"
```

Im Beispiel liefert der Aufruf mit dem Dateinamen „*Datei_3*“ den Rückgabewert „*Datei_2*“. Der erste Name sowie die dazugehörende Datei werden gelöscht. Liegt keine Schachtelung vor, so hat der Funktionsaufruf die gleiche Wirkung wie `TfileErase()` und der Rückgabewert ist ein leerer String. Obwohl im eigentlichen Schema nicht vorgesehen, müssen wir an dieser Stelle aber einen etwas größeren Aufwand treiben, da mit der Funktion `TInsert()` auch kompliziertere Schachtelungen eingefügt werden können. Wir vereinbaren folgenden Arbeitsablauf:[17]

```
..., D_1(D_2(D_3),D_4(D_5)), ...     ====>
..., D_2(D_3,D_4(D_5)), ...          ====>
..., D_3(D_4(D_5)), ...              ====> ...
```

Aufgabe. Setzen Sie dieses Arbeitsschema mit Hilfe der für die Klasse `Parameter` definierten Methoden um.

Als letzte Funktion sehen wir einen Austausch von Dateinamen zwischen den Ebenen vor:

```
string TSwap(string oldName, int depth){...}
// Wirkung von swapTFName("Datei_3",1):
..., Datei_3(Datei_2(Datei_1)),...        ====>
..., Datei_1(Datei_2(Datei_3)),...
```

[17] Zur einfacheren Lesbarkeit habe ich auch hier nochmals einige Klammerpaare ausgelassen und Kommata eingeführt, um die einzelnen Inklusionsrelationen besser herauszuarbeiten.

Die Methode wirkt immer nur auf auf den ersten Eintrag in den Unterebenen, das heißt auf die in `Tpop()` berücksichtigten komplexeren Schachtelungen besteht kein Zugriff. Die Funktionsweise ist mit den zu Verfügung stehenden Funktionen etwas umständlich, und wir beschreiben hier auch nur den Fall der Standardnutzung und lassen auch Fehlerprüfungen aus:

- Nach Identifizierung des übergebenen Dateinamens wird die Unterliste des nach oben zu holenden Namens (*in der Variablen „t“*) als Parameterstring ausgelesen (*se1*)

```
pos=tfnames.Find(s);
tfnames.Mark(pos);
for(i=0;i<depth;i++)
    tfnames.Mark(0);
t=tfnames.Wert(0);
tfnames.Mark(0);
se1=tfnames.Sequence();
```

- Der komplette Eintrag wird gelöscht und anschließend die verbleibende Sequenz des nach unten zu verschiebenden Namens ausgelesen (*se2*). Anschließend wird auch diese Sequenz komplett gelöscht.

```
tfnames.Layback();
tfnames.Erase(0);
tfnames.Laynull();
tfnames.Mark(pos);
se2=tfnames.Sequence();
tfnames.Layback();
tfnames.Erase(pos);
```

- Die Dateinamen und die Sequenzen werden nun wieder in den Parameterbaum eingefügt:

```
tfnames.Insert(0,t);
tfnames.Mark(0);
tfnames.Insert(0,se2);
do{
    tfnames.Mark(0);
} while(tfnames.Anzahl()>0);
tfnames.Insert(0,s);
tfnames.Mark(0);
tfnames.Insert(0,se1);
tfnames.Laynull();
```

Der Code kommt in dieser Form noch nicht mit komplexeren Schachtelungen klar, wie sie bei Import entstehen können. Falls Sie vorhaben, irgendwelche exotische Anwendungen zu konstruieren, die so etwas benötigen, sollten Sie hier tätig werden.

Es ist wohl unnötig zu erwähnen, dass auch der Nutzer der Bibliotheksklasse einige Sorgfalt walten lassen muss. Die Bibliothek verwaltet lediglich die Namen temporärer Dateien; für die Erzeugung und Nutzung ist der Anwender selbst verantwortlich. So kann es bei nicht sauber abgestimmten Aktionen durchaus dazu kommen, dass einzelne Dateien noch in der Verwaltungsliste stehen, aber gar nicht mehr existieren, oder bei Programmende eine Reihe von Dateien nicht mehr gelöscht werden, da die Einträge im Verwaltungssystem vorher aufgelöst wurden.

Aufgabe. Die temporären Dateien sind im aktiven Verzeichnis der Anwendung oder in einem fest installierten Verzeichnis unter dem vom Verwaltungssystem gelieferten Namen anzulegen. Wenn andere Pfade verwendet werden sollen, muss die Anwendung dies dem Verwaltungssystem durch einen Einfügebefehl bekannt machen. Erweitern Sie die Schnittstellen so, dass Verzeichnisauswahlen möglich sind.

Aufgabe. Implementieren Sie die Methode `FILE * ftmpopen()`, die einen Handle auf eine temporäre Datei zurückgibt.

3.9 Verschlüsselte Dateien

3.9.1 Die Aufgabenstellung

Noch in der Startphase über ein so komplexes Thema wie Verschlüsselung zu reden, mag vielleicht etwas verwegen erscheinen, aber um es vorweg zu nehmen, werden wir auch hier wieder fertige Bibliotheken nutzen und uns auf die Dateiarbeit beschränken. Formulieren wir zunächst die Anforderungen:

(a) Abgesehen von der Übergabe eines Schlüssels beim Öffnen einer Datei sollen die gleichen Funktionen wie bei unverschlüsselten Dateien nutzbar sein. Die Datei soll mit wahlfreiem Zugriff les- und schreibbar sein, sich also zum Anwender hin nicht anders verhalten als eine gewöhnliche Text- oder Binärdatei.
Wir werden uns zunächst auf die für Binärdaten notwendigen Methoden beschränken. Weitere Methoden können Sie nach Bedarf implementieren. Zur Beachtung: Diese Forderung impliziert, dass die Datei nicht einfach komplett ver- oder entschlüsselt wird, da dies bei großen Dateien zu zeitaufwendig und außerdem unsicher ist (*der Dateiinhalt liegt zumindest zeitweise unverschlüsselt vor*).

(b) Die Software soll über Kontrollfunktionen verfügen, die die Integrität der Datei sicherstellen.[18] Bereits beim Lesen der Daten soll somit sichergestellt werden, dass die Datei sich noch immer in dem Zustand befindet, in dem sie bei der

[18] Wir verzichten aber auf Reparaturfunktionen, die der Leser ebenfalls nach Bedarf ergänzen kann.

letzten Bedienung verlassen wurde. Ist der Dateiinhalt aus irgendeinem Grunde verfälscht, so sind natürlich auch die entschlüsselten Daten ziemlicher Müll. Dies kann aber bei Binärdaten relativ schlecht feststellbar sein, weshalb eine unabhängige Prüfung sinnvoll ist.

(c) Die Verschlüsselung soll auch bei gleichem Dateninhalt keine Wiederholungen aufweisen, um Plausibilitätsangriffe zu verhindern. Das gilt sowohl für verschiedene Dateien gleichen Inhalts, Blöcke in einer Datei mit gleichem Inhalt oder das erneute Schreiben eines Blocks mit unverändertem Inhalt auf die Platte.

Beispielsweise könnte ein Angreifer aus der Häufigkeit bestimmter Konstruktionen im Klartext Rückschlüsse auf den Inhalt einer Datei ziehen, in der Blöcke mit vergleichbarer Häufigkeit auftreten.

Für die Verschlüsselung steht eine Vielzahl unterschiedlicher Verfahren zur Auswahl, aus denen wir anhand unserer Randbedingungen ein geeignetes auswählen müssen. Eine Analyse ergibt:

- Symmetrische Verfahren verschlüsseln und entschlüsseln Daten mit dem gleichen Schlüssel (*und meist mit dem gleichen Algorithmus*), asymmetrische Verfahren verwenden für Ver- und Entschlüsselung verschiedene Schlüsselsätze.

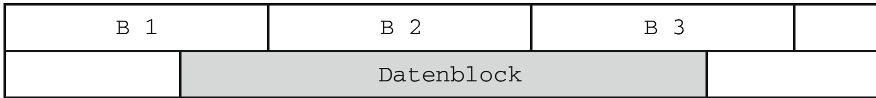
  Da asymmetrische Verfahren im Vergleich zu symmetrischen extrem langsam sind und eine Asymmetrie nicht zu unserem Aufgabenkatalog gehört, verwenden wir ein symmetrisches Verfahren.
- Man unterscheidet zwischen Block- und Stromchiffren . Stromchiffren verschlüsseln lange Datenblöcke am Stück, wobei gleiche Datensequenzen nicht zum gleichen Chiffrat führen. Fehlererkennung und Behebung ist meist vorhanden, jedoch ist immer der gesamte Datenstrom zu verarbeiten. Blockchiffren verschlüsseln nur kurze Datenblöcke jeweils gleicher Länge. Gleiche Daten ergeben stets das gleiche Chiffrat.

Keine der beiden Methoden erfüllt unsere Randbedingungen. Aufgrund des wahlfreien Zugriffs auf die Datei ist ein Zugriff auf relativ kurze Blöcke notwendig, um effizient arbeiten zu können. Die Individualisierung eines Blocks müssen wir selbst vornehmen. Wir verwenden deshalb ein Blockchiffrierverfahren, das wir um die weiteren Eigenschaften –Individualisierung und Stromchiffrierung über den Kurzblock– ergänzen.

3.9.2 Der Algorithmus

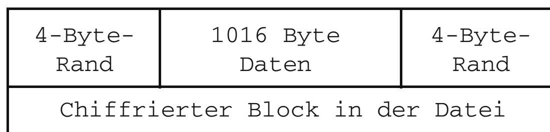
Als Basisverfahren können Sie ein beliebiges Blockchiffrierverfahren auswählen, für das Sie Quellen finden, beispielsweise AES mit einer Blocklänge und einer Schlüssellänge von 16 Byte, DES3 usw. Die Schlüssellänge sollte mindestens

16 Byte = 128 Bit betragen. Die Datei verschlüsseln wir in Einheiten mittlerer Größe, zum Beispiel 1 kByte = 1.024 Byte.



Soll beispielsweise der Bereich „Datenblock“ gelesen werden, so sind die Dateiblöcke „B1, B2, B3“ einzulesen und die Randblöcke B1 und B3 jeweils nur teilweise auszuwerten. Diese Arbeitsweise führt zu folgenden weiteren Eigenschaften unseres Verschlüsselungssystems:

- Die physikalischen Dateizeiger stimmen nicht mit den logischen Dateizeigern überein, da sie jeweils auf die Blockgrenzen in der Datei zeigen, während die logischen Zeiger auf jede beliebige Position verweisen können. Wir können daher vom Dateisystem bereit gestellte Dateizeiger nicht verwenden, sondern müssen eigene definieren.
- Jeder verschlüsselte Dateiblock muss individualisiert werden, um Wiederholungen bei gleichem Dateninhalt zu vermeiden. Dazu werden wir jedem Datenblock in der Datei vor der Verschlüsselung eine Zufallzahl voranstellen und sie am Ende wiederholen.



Das System kann bei der Entschlüsselung die Zahl leicht entfernen. Da bei jedem Schreibvorgang eine neue Zufallzahl vergeben wird, hat ein Datenblock nach einem erneuten Schreiben in die Datei einen anderen Inhalt, selbst wenn sich der unverschlüsselte Inhalt nicht geändert hat.

Als Konsequenz wird die Beziehung zwischen logischem und physikalischem Zeiger komplexer. Bezeichnet `lpos` die logische Position, so befindet sich diese im Block

$$B(lpos) = [lpos/1.016]^{19}$$

und dieser hat in der Datei die Position

$$FILEPOS = B(lpos) * 1.024$$

[19] [.] ist die so genannte Gaussklammer und bezeichnet den ganzzahligen Anteil der Division

Die der logischen Position entsprechende physikalische Position des Datenbytes ist

$$ppos = [lpos/1.016] * 1.024 + (lpos/1.016 - [lpos/1.016]) + 8^{20}$$

Sie werden sich sicher schon gewundert haben, weshalb die Zufallzahl am Ende des Datenblocks ein weiteres Mal auftaucht. Bei der Arbeit mit verschlüsselten Daten ist sinnvollerweise auch sicherzustellen, dass die Daten zwischen zwei Arbeitsvorgängen nicht verändert wurden. Wenn auch derjenige, vor dem der Dateiinhalt verborgen werden soll, mit den verschlüsselten Daten nicht viel anfangen kann, so könnte er uns dennoch ärgern, indem er Teil der Datei gegen beliebige andere Daten austauscht. Nach der Entschlüsselung steht dann nur noch Unfug in den entsprechenden Blöcken, und das sollte möglichst bereits durch das Leseprogramm erkannt werden und nicht erst durch eine Plausibilitätskontrolle der Daten.

Zur Integritätssicherung, wie man diese Prüfung nennt, könnte eine Signatur über die gesamte (*verschlüsselte*) Datei in Form eines Hashwertes berechnet werden. Eine Signatur lässt sich mit einem Fingerabdruck vergleichen: Wie ein Mensch anhand eines Fingerabdrucks sehr sicher erkannt werden kann, ohne dass man alle möglichen anderen Körpereigenschaften prüfen müsste, ist eine Signatur ein kurzer Datenblock, der einem unter Umständen sehr langen Datenblock mit hoher Sicherheit eindeutig zuzuordnen ist. Mit dem Begriff „Hashfunktion“ haben wir uns bei der Untersuchung der STL bereits beschäftigt und vertiefen dies hier noch einmal weiter unten. Ein solcher Hashwert wird beim Schließen der Datei berechnet und beim Öffnen der Datei geprüft.

Diese Standardvorgehensweise besitzt jedoch Nachteile. Zunächst ist beim Öffnen und beim Schließen der Datei ein Arbeitsaufwand zu leisten, der proportional zur Dateigröße ist und bei sehr großen Dateien durch ein Stocken der Anwendung auffallen kann. Außerdem ergeben sich daraus keine Hinweise, welche Bereiche der Datei noch brauchbar sind. Wir umgehen dies durch die Wiederholung der Zufallzahl. Bei Verfälschungen eines Datensatzes tritt am Ende des Blockes die Zufallzahl nicht mehr auf. Fehler werden so eindeutig erkannt, und im weiteren können unbeschädigte Datenblöcke sicher zurückgewonnen werden.

3.9.3 Der Einsatz des Algorithmus

Bevor wir zur Implementation kommen können, sind noch zwei Ergänzungen notwendig. Zum einen ist da das Problem des Dateiendes. Wenn wir einen Verschlüsselungsalgorithmus verwenden, der exakt so viele Bytes liefert wie der

[20]Die Zahlen sind bei Verwendung anderer Blockgrößen und Zufallzahlbreiten entsprechend auszutauschen.

unverschlüsselte Datensatz besitzt, ist nichts mehr zu tun. Vom letzten Datenblock ziehen wir die acht Byte für die Zufallzahlen ab und haben damit den kompletten Datenbestand. Das macht jedoch nicht jeder Algorithmus; die meisten verschlüsseln die Daten in Blöcken bestimmter Größe. Bei Verwendung eines solchen Algorithmus besitzt auch der letzte Block typischerweise 1.024 Byte (*womit auch die echte Dateigröße wirksam verschleiert wird*), und wir müssen irgendwo notieren, wie viele echte Datenbyte in diesem letzten Satz tatsächlich vorhanden sind.

Zum anderen werden je nach Vorliebe unterschiedliche Verschlüsselungsalgorithmen angewandt, und um später eine Datei wieder entschlüsseln zu können, muss auch diese Information irgendwo notiert werden. Die Datei erhält für beide Informationseinheiten einen „Vorspann" von einem Datenblock Länge. Im ersten Teil sind Klartextinformationen untergebracht, die in einem zweiten verschlüsselten Bereich des Vorspanns wiederholt werden, um Täuschungen zu entdecken. Zu Beginn der Datei legen wir einen Parameterstring ab (*hier der Übersichtlichkeit halber in mehrere Zeilen aufgeteilt*):

```
// Klartextbereich, Pos. 0-159:
SecureFile(
      Hash( SHA(20),172),
      Cipher(AES(16,16)),
      Pre(160,96),
      Data(4,1016),
      Rand(4,123456,164),
      Eof(4,168))
```

Dies erlaubt folgende Prüfungen:

- Der Datenstring `SecureFile` stellt sicher, dass es sich um den richtigen Dateityp handelt.
- Die Einträge `Hash` und `Cipher` erlauben die Auswahl der richtigen Algorithmen für die Entschlüsselung.[21]
- Die Zahlenparameter zum Bezeichner `Pre` sind die Länge des unverschlüsselten und des verschlüsselten Vorspanns.
- `Data` bezeichnet die Länge der Zufallzahl sowie die Länge der Nutzdaten pro Dateiblock.
- Die Zufallzahl `Rand` mit der Datenbreite vier besitzt im Beispiel den Wert $r = 123456$ und wird als Binärzahl an der Position 164 wiederholt. Die Position liegt im verschlüsselten Teil des Vorspanns und ist erst nach Entschlüsselung für einen Vergleich zugänglich.
- Stimmen die Werte nicht überein, so ist der Schlüssel nicht korrekt (*oder die Datei wurde verfälscht*).

[21] Wir verwenden hier nur jeweils einen bestimmten Algorithmus, eine Erweiterung auf andere Algorithmen ist damit aber jederzeit möglich.

- `Eof()` gibt die Position des Zeigers auf das logische Dateiende an (*ebenfalls im verschlüsselten Bereich*).
- Mit `Hash` wird die Signatur über den unverschlüsselten Vorspann und sämtliche verschlüsselten Daten berechnet und anschließend im verschlüsselten Bereich des Vorspanns gesichert.

  Ist der Hash–Wert nicht korrekt, so ist der Vorspann verändert worden (*bei korrekt lesbarer Zufallzahl*) oder der Schlüssel stimmt nicht (*dann ist auch die Zufallzahl nicht lesbar*).

3.9.4 Die Implementation

So viel zur Theorie. Für die Umsetzung besorgen Sie sich zunächst Quellkode für einen Hash–Algorithmus, beispielsweise für „SHA–1 Secure Hash Algorithm " oder einen anderen. Die Quellen sind frei im Internet verfügbar. Frischen wir noch einmal die Kenntnisse auf: Ein Hash–Algorithmus ist eine Einwegverschlüsselungsfunktion, die Informationen beliebiger Länge auf eine Information fester Länge, zum Beispiel 128, 160, 192 oder 256 Bit verdichtet (*das ist eine andere Größenordnung als in der STL, bei der Längen zwische 10 und 20 Bit interessant sind; die Aufgaben sind allerdings auch völlig anders geartet*). Die Algorithmen arbeiten nicht informationserhaltend, das heißt mehrere Eingangsinformationen können zum gleichen Funktionswert führen. Sie sind jedoch so konstruiert, dass nicht nur weitgehende Kollisionsfreiheit herrscht, also verschiedene Informationen auch unterschiedliche Hash–Wert ergeben, sondern dass auch aus einem Hash–Wert nicht auf Original zurück geschlossen oder durch gezielte Veränderung einiger Informationsbits der Hash–Wert nicht noch einmal erzeugt werden kann. Benötigt werden meist zwei Funktionen:

```
void processblock(const uchar* data, uint len)
void finish(uchar* hash)
```

Mit der ersten Funktion werden beliebig lange Informationssequenzen verarbeitet, die zweite Funktion sorgt für einen Abschluss und ein Rücksetzen der Arbeitsparameter.

Die Hash–Funktion verwenden wir außer für die Signaturberechnung innerhalb der Datei auch für die Generierung eines Kennwortes für die Verschlüsselungsfunktion. Verschlüsselungsfunktionen akzeptieren meist Kennworte von acht bis 32 Byte, wobei die Schlüsselworte einige Anforderungen erfüllen sollen und dann nicht gerade gut merkbar sind. Gut merkbare Kennworte wie „Monika" oder „Putzlappen" stehen meist in Wörterbüchern und sollten nicht verwendet werden, da die Prüfung des Wortschatzes eines Wörterbuchs eine Kleinigkeit für einen Rechner ist. Auf diese Weise ist schon so manches Verschlüsselungssystem mit einer rechnerischen Sicherheit von Milliarden von Arbeitsjahren nach wenigen Stunden geknackt worden. Wenn man mal von kaum noch im Gedächtnis zu behaltenden

Konstruktionen wie „xXo91hA93k;!8hH“ absieht, ist aber auch folgende Vorgehensweise einigermaßen sicher: merken Sie sich einen Satz wie „bezugnehmend auf Ihre Passwortanfrage teilen wir Ihnen mit:“ und verschlüsseln Sie diesen (*zugegebenermaßen bei der Eingabe unangenehm langen Satz*) mit einer Hashfunktion. Benutzen Sie das Ergebnis als Schlüssel für den Verschlüsselungsalgorithmus.[22]

Auch den Code für den Verschlüsselungsalgorithmus laden Sie aus dem Internet, beispielsweise den „AES Advanced Encryption Standard “. Die Benutzung ist meist weniger komplex als bei den Hashfunktionen: Der Schlüssel wird zu Beginn gesetzt und anschließend die Information jeweils 8–, 16– oder 32–byteweise sequentiell verschlüsselt (*einige Algorithmen können mehr, die Blockchiffrierung ist aber meist die Standardeinstellung*). Für die Verschlüsselung eines Datenblocks implementieren wir ein Rückkopplungsverfahren:

(a) Vor den Datenblock wird zunächst eine Zufallzahl der angegebenen Länge geschrieben. Anschließend wird das erste Datensegment aus Zufallzahl und einigen Datenbyte verschlüsselt.
(b) Die verschlüsselten Daten eines Segmentes werden durch die XOR-Operation mit den unverschlüsselten Daten des nächsten Segments verknüpft. Anschließend wird das Segment verschlüsselt.
(c) Ist das letzte Segment kleiner als die Verschlüsselungsbreite des Algorithmus, so werden die Daten mit der unverschlüsselten Zufallzahl aus der Startoperation mit XOR verknüpft. Eine Entschlüsselung ist dann nur nach Entschlüsselung des Startbereiches möglich.

```
/* buf=Datenpuffer mit freien ersten Byte für die
   Zufallzahl,
   doff=Breite der Zufallzahl,
   b1,b2,be,hdec: char*-Zeiger und Hilfspuffer
   encry: Verschlüsselungsalgorithmus (Objekt)
*/
GetRandom(buf,doff);
memmove(hdec,buf,doff);
b1=buf;
b2=b1+encry->BlockSize();
```

[22] Wichtig! Verwenden Sie den Satz möglichst nur für eine Sicherheitsanwendung und denken Sie sich für andere kennwortgeschützte Systems etwas Neues aus! Bei einem Versuch mussten sich Nutzer auf einer Internetseite ein Login erstellen. Nach einigen Wochen besaßen die Experimentatoren eine Reihe von root-Kennworten von Unternehmensrechnern, da aus Bequemlichkeitsgründen selbst hoch angesiedelte Administratoren auf ihre Standardkennworte zurückgriffen. Verwenden Sie möglichst lange Sätze oder ungewöhnliche Wortkombinationen. In der Sprache stecken nämlich so viele grammatische Regeln, dass ein Satz wesentlich weniger Informationen enthält, als seine Länge vorspiegelt. Um beispielsweise einer 128-Bit Zufallzahl zu entsprechen, muss eine Wortkette schon eine Länge von mehr als 1000 Bit aufweisen, als Satz unter Einhaltung grammatischer Regeln noch wesentlich mehr.

```
m=len/encry->BlockSize();
encry->ProcessBlock(b1);
for(i=1;i<m;i++){
    for(be=b2;b1!=be;++b,++b1)
        *b2 ^= *b1;
    encry->ProcessBlock(b1);
}//endfor*/
for(be=buf+len,i=0;b2!=be;++b2)
    *b2 ^= hdec[i%doff];
```

Bei der Entschlüsselung müssen wir nun nur noch die verschlüsselten Daten festhalten, um die XOR–Verknüpfung rückgängig zu machen. Dafür wird ein Datenpuffer von der doppelten Breite eines Chriffratsegmentes benötigt. Am Ende des Kodes befindet sich jeweils die Verschlüsselung der Überhangbytes.

```
m=len/decry->BlockSize();
b1=buf;
memset(hdec,0,decry->BlockSize());
for(i=0;i<m;i++){
   memmove(&hdec[decry->BlockSize()],b1,
                                  decry->BlockSize());
   decry->ProcessBlock(b1);
   for(be=b1+decry->BlockSize(),b2=hdec;b1!=be;
           ++b1,++b2)
      *b1 ^= *b2;
   memmove(hdec,&hdec[decry->BlockSize()],
           decry->BlockSize());
}//endfor
for(be=buf+len,i=0;b1!=be;++b1])
   *b1 ^= buf[i%doff];
```

Eine Arbeitsklasse für verschlüsselte Dateien kann nun wahlweise von der C–Struktur FILE oder der C++–Klasse fstream abgeleitet werden. In der C++–Lösung ist folgende Klassendefinition ein guter Startpunkt:

```
class SecFILE: private fstream {
public:
    SecFILE();
    ~SecFILE();
    int open(string fname, string fkey);
    int close();
    SecFILE& read(char* b, int len);
    SecFILE& write(const char* b, int len);
    int gcount() const;
    SecFILE& remove(int len);
    SecFILE& truncp();
```

```
    int tellp();
    int tellg();
    SecFILE& seekp(int pos, ios::seek_dir dir);
    SecFILE& seekg(int pos, ios::seek_dir dir);
    SecFILE& seekp(int pos);
    SecFILE& seekg(int pos);
    bool good();
    bool fail();
    bool eof();
private:
    char * key;             // Schlüssel
    int ppos, gpos, eofp;   // Schreib/Leseposition
    int doff, dsize;        // Datenoffset und Länge
    int clrpre,encpre;      // Längen Klartextbeginn
    int eofpos, randpos, crcpos;// Positionen
    int gcnt;
    char * gbuf;            // Lesepuffer
    char * pbuf;            // Schreibpuffer
    BlockTransformation * encry;
                            //Verschlüsselungsobjekt
    BlockTransformation * decry;
                            //Entschlüsselungsobjekt
    HashModule * hash;      // Hash-Algorithmus
    Parameter p;            // Parametrierung Datei
public:
    bool encrypt(char * buf, int len);
    bool decrypt(char * buf, int len);
private:
    void put(int pos);
    void get(char* buf, int pos);
    void rewrite(string s); // Neue Datei anlegen
    bool syncbuf();
};//end class
```

Bevor ich Ihnen die Aufgabe übertrage, die Methoden zu implementieren, möchte ich die Schnittstelle im Detail begründen. Die Klasse erbt von der C++–Filestreamklasse, die `fstream`–Funktionen sind aber nicht direkt verwendbar, da alles umgeleitet werden muss. Um Fehlbedienungen zu verhindern, ist die `private`–Deklaration der Vorgängerklasse notwendig. In der Attributliste sind Zeiger auf Objekte zum Verschlüsseln, Entschlüsseln und zur Berechnung des Hashwertes deklariert. Sofern Sie nur jeweils eine Methode verwenden, können Sie die Objekte bereits im Konstruktor erzeugen; der Kopfteil der Datei sieht aber auch den variablen Einsatz von Algorithmen vor, so dass die Initialisierung der Objekte auch

in der `open`–Funktion vorgenommen werden kann, wenn nach Öffnen der Datei klar ist, was benötigt wird. Die Funktion `open(..)` erhält als zusätzliche Parameter den Schlüssel (*Klartext; Hash–Überschlüsselung wird intern vorgenommen*) und öffnet eine vorhandene Datei beziehungsweise erzeugt eine neue, falls keine Datei vorhanden ist, im Schreib- und Lesemodus für binäre Daten. Weitere Öffnungsmodi sind nicht vorgesehen (*und auch nicht sonderlich sinnvoll bei Sicherheitsdateien*). Integritäts- und Schlüsselprüfung werden in der Funktion ebenfalls vorgenommen und bei negativem Ausgang die Datei nicht geöffnet.

> **Aufgabe.** Die Implementation der Funktion `open(..)` ist länglich und daher nervig, aber letzten Endes trivial. Sie sollte zusammen mit der privaten Methode `rewrite()` und der Methode `close()` bearbeitet werden, da die beiden letzten das erzeugen, was die erste verstehen muss. Implementieren Sie den Konstruktor, den Destruktor und diesen Methodensatz nach den Dateibeschreibungen. Eine neue Datei besteht zunächst nur aus dem Kopfblock. Warum ist ein Textmodus für diese Art der verschlüsselten Dateien nicht sinnvoll?

Die anderen Funktionen entsprechen den Stream–Funktionen und sind gemäß den Schnittstellenbeschreibungen im C++–Handbuch zu implementieren. Wie in C++–Streams üblich existieren unterschiedliche Positionen zum Lesen und Schreiben.[23] Da die Sicherheitsdatei blockweise bearbeitet wird, stimmen die Positionszeiger der Basisklasse nicht mit den aktuellen Dateizeigern überein. Aus diesem Grund sind eigene Zeigerattribute und Pufferattribute zum Lesen und zum Schreiben deklariert.

Neu sind lediglich `remove(..)` und `truncp(..)`, die das Löschen von Daten beziehungsweise das Abschneiden erlauben. Wie der Leser leicht überprüfen kann, sind sie ohne Rückgriff auf Stream–Funktionen problemlos implementierbar. Sie Löschen ab der aktuellen Schreibposition (!) die angegebene Anzahl von Bytes (*was unter Umständen einige Zeit in Anspruch nehmen kann*) oder den Rest der Datei.

Weitere öffentliche Methoden sind `decrypt(..)` und `encrypt(..)`, da die Verschlüsselung mit Zufallzahl und Rückkopplung möglicherweise auch an anderer Stelle interessant sind (*wenn auch die Unterbringung in einer Klasse für Dateiverarbeitung nicht gerade ein geeigneter Ort ist*).

> **Aufgabe.** Implementieren Sie die Schreib-, Lese- und Positioniermethoden. Sie benötigen hier auch die restlichen privaten Methoden.

Die Dopplung der Schreib- und Leseattribute macht die Methode `syncbuf()` notwendig. Wird im gleichen Pufferbereich geschrieben, in dem auch der Lesezeiger steht, so muss bei der nächste Leseoperation natürlich der zuletzt geschriebene

---

[23] Falls Sie aus irgendwelchen Gründen die Klasse nicht von fstream erben lassen, sondern intern mit C–Files arbeiten, müssen Sie dies berücksichtigen und vor jeder Lese- oder Schreibaktion den Filepointer positionieren! C kennt ja nur einen Zeiger, der für beide Operationen zuständig ist.

Inhalt zurückgegeben werden und nicht ein alter Pufferinhalt. Beide Puffer sind in diesem Fall zu synchronisieren:

```
bool SecFILE::syncbuf(){
    if(ppos/dsize==gpos/dsize){
        memmove(gbuf,pbuf,doff+dsize);
        return true;
    }else{
        return false;
    }//endif
}//end function
```

Die Methode ist beim Schreiben in die Datei sowie beim Lesen eines neuen Dateisegmentes aufzurufen.

Aufgabe. Das Bearbeiten von Textdateien macht, wie bereits dargelegt, mit den hier diskutierten Arbeitsmethoden wenig Sinn. Aber auch in strukturierten Dateien können einzelne Informationen in Textform abgelegt werden (*die Satzlängen sind dann allerdings meist keine Konstanten mehr*). Implementieren Sie die folgenden Methoden zum teilweisen verarbeiten von Strings:

```
SecFILE& operator>>(string& s);
SecFILE& getline(char* c, int len, char delim);
SecFILE& operator<<(const string s);
```

Falls Sie weitere Operatoren für andere Datentypen zulassen, sollten diese allerdings Binärdaten in die Datei schreiben, also zum Beispiel

```
SecFILE& operator<<(int& i){
    write(&i,sizeof(int));
    return *this;
}//end method
```

3.9.5 Bemerkungen zur Verschlüsselung

Der Einsatz von Verschlüsselungsmethoden beschränkt sich nicht auf das Verdecken von Informationen, und die Algorithmen sind derart vielfältig, dass Sie noch die meisten Kapitel dieses Buches durcharbeiten müssen, um das notwendige Handwerkszeug für eigene Implementationen zu besitzen. Wenn Ihr Interesse geweckt ist, können Sie einige kleinere Versuche aber auch schon hier beginnen.

Gute Verschlüsselungsalgorithmen erzeugen zufällige Bitmuster ohne erkennbare Struktur. Das gibt uns eine erste Testmöglichkeit an die Hand: Nach Verschlüsseln einer nicht zu kleinen Datenmenge werden Original und Chiffrat komprimiert. Selbst bei gut komprimierbarem Original (*beispielsweise einem Text*) sollte das Chiffrat nicht mehr komprimierbar sein. Dies ist zwar nur ein erster, aber mit bereits vorhandenen Mitteln durchführbarer Test.

Ein weiteres Kriterium ist die Änderungsrate des Chiffrats bei der Veränderung des Original. Im Idealfall ändert sich im Mittel die Hälfte aller Bits im Chiffrat bei Änderung eines Bits im Original. Um das zu erreichen, müssen die Bits miteinander korreliert werden. Dazu wird in einem Verschlüsselungsschritt das vorhandenen Bitmuster um eine Reihe von Positionen zyklisch verschoben und dann mit dem unverschobenen Bitmuster verknüpft, zum Beispiel

```
muster[k+1] = muster[k] | (muster[k] << p[k])
```

Durch mehrere solcher Schritte wird jedes Bit des Endergebnisses von vielen Bits des Original beeinflusst.

Hash–Algorithmen sind nicht informationserhaltend. Zu ihrer Konstruktion können deshalb die Operationen AND und OR eingesetzt werden, nach deren Ausführung Unsicherheiten entstehen, wie das Originalbitmuster war (*bei* AND *kann ein Nullbit durch die Kombinationen 0&0, 0&1 oder 1&0 entstanden sein*). Bei genügend vielen Durchläufen sind die möglichen Ausgangsmuster praktisch nicht mehr rekonstruierbar, da deren Anzahl potentiell steigt. Der Einsatz der beiden Operationen muss jedoch ausgewogen sein, um nicht zu viele Bits im Ergebnis durch OR zu setzen oder durch AND zu Löschen.

Sie können sich unter Verwendung der Operationen XOR, OR, AND und SHIFT an der Entwicklung einer Hashfunktion versuchen. Das Ergebnis können Sie auch zur Verschlüsselung von Daten einsetzen. Hashen Sie dazu zunächst einen beliebig gewählten Schlüssel. Von dem Ergebnis nutzen Sie die Hälfte der Bits zum Verknüpfen mit den zu chiffrierenden Daten. Ein erneutes Hashen des gesamten Ergebnisses liefert einen neuen Teilschlüssel. Sie können so iterativ das gesamte Original verschlüsseln. Die Entschlüsselung erfolgt in gleicher Weise. Sinngemäß ergibt das folgenden Code:

```
H[20] = Hash(key)

for(i=0;i<dataLen;i=i+10)
    data[i..i+9]=data[i..i+9] xor H[0..9]
    H[20]=Hash(H[20])
```

Ver- und Entschlüsselung werden in diesem einfachen Beispiel durch ein nicht informationserhaltendes Verfahren erreicht. Echte informationserhaltende Algorithmen verzichten darauf, in dem die Operationen AND und OR durch Tabellen ersetzt werden, die ein Bitmuster durch ein anderes ersetzen. Hier geeignete Tabellen und Strategien zu finden, die eine gute Verschlüsselung bewirken und obendrein auch noch umkehrbar sind, ist nicht ganz einfach, so dass ich aus Frustgründen an dieser Stelle vor solchen Versuchen erst einmal abraten möchte. Für nach anderen Kriterien entworfenen und vielleicht einfacher durchschaubaren Algorithmen ist das Studium weiterer Kapitel notwendig, so dass Sie sich noch etwas gedulden müssen.

3.10 Textdateien und Verzeichnisse

Die STL umfasst zwar eine beträchtliche Menge an Klassen und Algorithmen, ist aber dennoch relativ beschränkt, wenn es komplexer wird. Als weitere „Standardbibliothek“ kann dem C++ Programmierer die boost++ Bibliothek ans Herz gelegt werden, die mit etwa 5.000 weiteren Dateien und einem ausführlichen Dokumentationsteil daherkommt.

Die Verwendung der boost++ Bibliothek ist in der Regel relativ einfach zu bewerkstelligen. Wie bei der STL und anderen Bibliotheken bindet man die benötigten Header-Dateien, die in einem der im Unterverzeichnis `boost` zu findenden anwendungsspezifischen Verzeichnissen abgelegt sind, in die eigenen Anwendung ein. In vielen Fällen verwendet die boost++ Bibliothek wie die STL Templatemethoden, so dass mit der Einbindung der Header-Dateien bereits alles geschehen ist. Für einige Teile werden jedoch auch Objektdateien benötigt, die im Unterverzeichnis `libs` in Verzeichnissen zu finden sind, die den gleichen Namen wie die Header-Datei-Verzeichnisse aufweisen. Je nach Vorliebe kann man diese Dateien nun ebenfalls in seine Anwendung einbinden oder daraus separat eine dynamische oder statische Bibliothek erzeugen.
Wir beschränken die Nutzung der boost++ Bibliothek an dieser Stelle auf einige Dateioperationen. Zunächst implementieren wir zwei einfache Methoden zum Einlesen und Schreiben von Textdateien:

```
bool textfile_to_buffer(path const& filename,
                        deque<string>& buffer){
    string s;
    boost::filesystem::ifstream fs(filename);
    buffer.clear();
    if(!fs.is_open()) return false;
    while(!fs.eof()){
        getline(fs,s);
        buffer.push_back(s);
    }//endwhile
    return true;
}//end function

bool buffer_to_textfile(path const& filename,
                       deque<string> const& buffer){
    deque<string>::const_iterator it;
    boost::filesystem::ofstream fs(filename);
    if(!fs.is_open()) return false;
    for(it=buffer.begin();it!=buffer.end();it++)
        fs << *it << endl;
    return true;
}//end function
```

Benötigt wird dazu der boost++ Bibliotheksteil `filesystem`, der den Datentyp `path` definiert:

```
template<class String, class Traits>
  class basic_path;
```

Hierbei handelt es sich um einen speziellen Stringtyp, der um eine Reihe von Funktionen erweitert ist, um mit Verzeichnispfaden umgehen zu können. Er erbt allerdings nicht vom Standard-String-Typ, sondern ist ein völlig neuer eigenständiger Typ, der den STL-String (oder irgeneine andere Stringimplementation) nur als Templateparameter enthält. Eine Reihe von Operatoren erlauben zwar eine weitgehend mit einem String kompatible Verwendung, an kritischen Stellen muss allerdings über die Methode `string()` auf den inneren Stringtypen zurückgegriffen werden. Der neue Typ macht dann auch die Verwendung adaptierter Streamklassen sinnvoll, wenn man Konstrukte wie

```
std::ofstream fs(filename.string().c_str());
```

vermeiden will.

> **Aufgabe.** Installieren Sie die boost++ Bibliothek und implementieren Sie die obigen Beispiele. Um sie lauffähig zu machen, müssen Sie auch die Objektteile der `filesystem`-Bibliothek sowie der `system`-Bibliothek einbinden.

Die beiden Methoden bieten eine einfache Möglichkeit, Textdateien zu manipulieren, wären aber auch mit STL-Mitteln relativ leicht zu implementieren gewesen. Der Vorteil der boost++ Bibliothek wird erst sichtbar, wenn man tiefer in das Verzeichnishandling eindringt. Die C Bibliothek weist eine Reihe von Methoden auf, Verzeichnisinhalte zu bearbeiten, ist aber äußerst umständlich zu bedienen, was durch die boost++ Bibliothek behoben wird. Diese definiert Iteratortypen, mit denen Verzeichnisse auch rekursiv durchsucht werden können. Die folgende Methode sammelt beispielsweise alle Dateinamen mit bestimmten Dateierweiterungen in einem verzeichnisbaum, sortiert nach Dateinamen:

```
struct less_special {
 inline bool operator()(path const& p1,
                       path const& p2)const{
    return basename(p1.leaf())<basename(p2.leaf());
 }//end function
};//end struct

typedef multiset<path,less_special> file_set;

void read_filenames_recursiv(
         path const& start_dir,
         string_set const& ext_list,
         file_set& fs){
    recursive_directory_iterator end_it;
    for(recursive_directory_iterator
              it(start_dir);it!=end_it;++it){
        if(!is_directory(*it)){
            if(ext_list.size()==0){
                fs.insert(*it);
```

```
            }else{
                if(ext_list.find(extension(*it))
                            !=ext_list.end()){
                    fs.insert(*it);
                }//endif
            }//endif
        }//endif
    }//endfor
}//end function
```

Das Startverzeichnis für die Suche `start_dir` kann den Startpunkt absolut oder relativ setzen, die zulässigen Dateierweiterungen sind im Set `ext_list` hinterlegt. Der Container für die Aufnahme der gefundenen Dateien ist als `multiset` definiert, was notwendig ist, da er nach Dateinamen sortiert werden soll und gleiche Dateinamen in verschiedenen Verzeichnissen zulässig sind. Wie dem Code zu entnehmen ist, ist der Wert eines Verzeichnisiterators offenbar vom Typ `path`, aus dem mit den Methoden `leaf()` der Dateiname (ohne Verzeichnisanteil), mit `extension` auch die Dateierweiterung extrahiert werden kann.

Darüber hinaus hält der Verzeichnisiterator eine Fülle weiterer Methoden bereit, um beispielsweise Dateieigenschaften auszulesen, Verzeichnisse zu Erzeugen oder zu Löschen oder Links von Dateien in anderen Verzeichnissen anzulegen und anderes. Benötigt man solche Methoden, ist zwar in der Regel etwas Einarbeitung und Probieren notwendig, aber der Aufwand ist allemal geringer, als mittels der C Bibliothek das Problem selbst anzugehen.

Aufgabe. Implementieren Sie eine Methode

```
bool full_path(path const& start_dir,
               path& fname, bool exact=true);
```

zum bestimmen des Standortes einer Datei. In `start_dir` soll ein Startverzeichnis vorgegeben werden, ab dem der Verzeichnisbaum nach dem in `fname` angegebenen Dateinamen durchsucht wird. `exact` definiert, ob bei der Suche auf die Groß/Kleinschreibung des Dateinamens geachtet werden soll. Die Suche endet beim ersten Auftreten des Dateinamens, der komplette Pfad wird in `fname` ausgegeben.

Aufgabe. Implementieren Sie einen rudimentären Verzeichnisiterator mit Hilfe von C-Funktionen aus der C-Bibliothek. Der Iterator soll über den Inhalt eines Verzeichnisses iterieren (das ist der Pflichtteil, die Rekursion können Sie als Kür betrachten. Die Aufgabe soll Ihnen nur einen Eindruck vom möglichen Innenleben der boost-Methoden geben).

3.11 Laufwerksimulation

Die Organisation von Plattenlaufwerken gehört eigentlich eher in ein Buch über Betriebssysteme, und wer sich mit diesem Thema detaillierter auch hinsichtlich der Implementierung auseinander setzen will, kann beispielsweise auf die Standardwerke von Andrew Tanenbaum zurückgreifen.

Dass hier trotzdem ein kurzes Kapitel zu diesem Thema eingefügt ist, hat auch praktische Gründe. Eine Dateiverschlüsselung kann man nämlich nicht so ohne weiteres in bereits bestehenden Anwendungen unterbringen, eine normale Datei in einer verschlüsselten Festplatte schon eher. Dazu muss man aber wissen, wie eine Festplatte organisiert werden kann.[24] Wir beschränken uns hier allerdings auf ein veraltetes, aber eben auch recht einfaches Verwaltungsmodell, das für die heutigen Plattengrößen aber nicht geeignet ist.

3.11.1 Die „File Allocation Table“ FAT

In älteren System mit noch kleinen Festplattengrößen hatten die CPUs auch die Ansteuerung der Festplatten zu übernehmen. Eine Festplatte unterteilt sich in einzelne Platteneinheiten (*magnetisierbare Scheiben*), die jeweils eine bestimmte Anzahl von Spuren (*kreisförmigen geschlossenen Bahnen, die abgetastet werden*) besitzen, die jeweils wieder in eine bestimmte Anzahl von Sektoren (*auf die Platte gesehen gewissermaßen Kuchenscheiben*) unterteilt werden. Jeder Sektor kann eine bestimmte Anzahl Nachrichtenbytes aufnehmen. Die Sektoren werden bei der Grundformatierung softwaremäßig angelegt (*spezielle Magnetisierung, die im Datenteil nicht auftritt, sowie eine weitere Spezialisierung für die Kennzeichnung des 1. Sektors*), der Rest ist Hardware. Eine Plattenkonfektionierung kann folgendermaßen aussehen:

```
Platten               4
Spuren               64
Sektoren             32
Bytes/Sektor      1.024

Total Bytes   8.388.608
```

Mit dieser Information kann nun die Position eines bestimmten Bytes exakt berechnet werden.

> **Aufgabe.** Implementieren Sie eine Adressierungsklasse, die die Position eines Bytes auf einer konfigurierbaren virtuellen Platte berechnet und Schreib-/Leseoperationen simulieren kann. Sie können dazu auf Techniken zurückgreifen, die bei der Verschlüsselung erarbeitet wurden.

[24] Auch dazu existieren natürlich Werkzeuge wie TrueCrypt, so dass man, wie bei fast allem anderen auch, nicht selbst Hand anlegen muss. Eine Übung ist aber allemal sinnvoll.

Werden Dateien auf der Platte gespeichert, so wird der Speicherplatz unabhängig von der tatsächlichen Größe immer in Sektoren, in diesem Fall also 1 kB-Blöcken zugewiesen. Um zu kennzeichnen, welche Sektoren einer bestimmten Datei zugewiesen sind, werden die Sektoren auf ein `int`-Feld abgebildet. Im obigen Beispiel benötigt man dafür ein Feld der Größe

```
int FAT[8192];
```

Wird der Sektor mit der Indexnummer `k` gelesen, so ist in `FAT[k]` die Indexnummer des Folgesektors eingetragen; handelt es sich um den letzten zugewiesen Sektor, enthält die Tabelle den Wert `-1`.

3.11.2 Verzeichnisse

Um die Verwaltung von Dateien auf der Platte zu organisieren, wird der spezielle Dateityp „Verzeichnis“ eingeführt, der eine baumartige Verwaltung ermöglicht. Das Konzept selbst ist wohl hinreichend bekannt und muss hier nicht vorgestellt werden. Verzeichnisse (Verzeichnisdateien) enthalten die Namen der von ihnen verwalteten Dateien (und Unterverzeichnisse), Zugriffsrechte und Statusinformationen (die wir hier aber nicht berücksichtigen werden), die Nummer des ersten Sektors einer Datei sowie die Dateigröße. Sie können nur vom Dateisystem gelesen werden und werden beim Öffnen eines Verzeichnisses vollständig eingelesen.

```
struct Laufwerk::Directory {

    struct DirEntry{
        char name[256];
        int  sektor;
        int  size;
        bool dir;
    };//end struct

    bool changed;
    list<DirEntry> entries;
};//end struct
```

Ändert sich die Größe einer Datei, muss dies im Heimatverzeichnis eingetragen werden. Bei dem speziellen Dateityp `Directory` führt das zu Problemen: die Größen müssten unter Umständen rekursiv nachkorrigiert werden, was bei größeren Verzeichnisbäumen einigen Aufwand verursachen kann, und andere Anforderungen wie etwa nach der Größe aller in einem Verzeichnis verwalteten Dateneinheiten führen zu Redundanzen. Wir legen zur Vermeidung von Problemen deshalb zusätzlich fest:

Die Verzeichniseinträge werden in einer verketteten Liste verwaltet.[25] Diese enthält immer zwei Verzeichnis-Standardeinträge:

```
fname == "."      // kennzeichnet das aktuelle Verzeichnis
fname == ".."     // kennzeichnet das Elternverzeichnis
```

Diese Einträge werden in dieser Reihenfolge immer als erste auf der Platte gesichert. Der erste Eintrag enthält die Größe der Verzeichnisdatei selbst. Nach seinem Einlesen erlaubt er das Einlesen des kompletten Verzeichnisses. Bei allen anderen Einträgen vom Typ `Directory` ist das Attribut `size` funktionslos, d.h. wird nicht ausgewertet.

Der zweite Eintrag erlaubt einen Wechsel zum Elternverzeichnis, ohne dass von der Wurzel des Verzeichnisbaums eine Suche durchgeführt werden müsste. Der Verzeichnisbaum kann daher von einem beliebigen aktuellen Punkt in beide Richtungen durchlaufen werden.

Beim Ändern des Verzeichnisinhalts wird das Attribut `changed=true` gesetzt, so dass vor dem Einlesen eines anderen Verzeichnisses die Änderungen abgespeichert werden können. Dies ist z.B. schon dann der Fall, wenn sich die Größe einer Datei (*aber nicht einer Unterverzeichnisdatei!*) im Verzeichnis ändert.

Die Dateinamen waren im ursprünglichen FAT-System auf 8 Zeichen für den Hauptnamen und 3 Zeichen für die Erweiterung beschränkt; die hier verwendete Struktur erlaubt Dateinamen bis 255 Zeichen, die Sie aber auch nach Bedarf anders vergeben können. Wesentlich ist allerdings die feste Zuordnung der Speichergröße. Jeder Verzeichniseintrag besitzt damit eine feste Position, und bei Änderungen eines Eintrags braucht nur dieser Eintrag geändert werden, was bei den langsamen Plattenzugriffen von wesentlicher Bedeutung ist. Eine Freigabe der Größe durch Verwendung der Klasse `string` hat hier also nichts zu suchen.

3.11.3 Dateideskriptor

Eine Datei ist eine Folge von Sektoren. Um Daten korrekt lesen und schreiben zu können, ist die Kenntnis der Schreib-/Leseposition sowie die Kenntnis der Gesamtgröße der Datei notwendig. Für die Verwaltung wird zusätzlich noch der Startindex des Verzeichnisses benötigt, in der die Datei verwaltet wird.

```
struct Laufwerk::Filehandle {
private:
    int sektor;    // Startsektor des Elternverzeichnisses
    int pos;       // laufende Bytenummer in der Datei
    int eof;       // Größe der Datei
```

[25] Verzeichnisse sind nicht notwendigerweise sortiert, so dass ohnehin eine serielle Suche durchgeführt wird. Außerdem kann nach unterschiedlichen Größen gesucht werden. Optimierungen in verschiedene Richtungen sind zwar denkbar, aber für eine Vorstellung des Funktionsprinzips wie hier unnötig.

```
    bool changed; // Inhalt des aktuellen Blocks geändert
    bool eof_chgd;      // Dateigröße verändert
    char buf[1024];     // aktueller Datenpuffer
    vector<int> chain;  // Blockkette in der FAT
};//end struct
```

Beim Öffnen einer Datei wird ein Filehandle erzeugt, in den aus dem Verzeichnis der Startsektor des Verzeichnisses und die Dateigröße übernommen wird. Der FAT-Inhalt wird in die Kette `chain` übernommen.

FAT (die Datei beginnt in Sektor 0):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	..
4		14		7			11				2			-1		

Kette der Sektoren:

0	4	7	11	2	14											

Hier stehen die Indizes in der Reihenfolge, in der sie verwendet werden. Bei einem beliebigen Sprung kann so gezielt der benötigt Sektor geladen werden, ohne erst die FAT-Liste durchsuchen zu müssen.

Das Schreiben und Lesen in einer Datei erfolgt über den Datenpuffer, in den jeweils ein Sektor eingelesen wird. Beim Schreiben wird das Attribut `b_changed=true` gesetzt, beim Überschreiten der Sektorgrenze wird vor dem Lesen des neuen Sektors der alte geänderte auf die Platte zurückgeschrieben. Ändert sich die Dateigröße durch Schreiben weiterer Daten hinter das Dateiende (`eof_chgd==true`), muss beim Schließen der Datei auch das Verzeichnis geändert werden. Da das nur bei Anwenderdateien, nicht aber bei Verzeichnisdateien passieren darf, legen wir zusätzlich fest, dass `sektor==-1` für Verzeichnisse gilt. Anhand dieser Zusatzbedingung kann ein Dateideskriptor erkennen, ob er eine Datei oder ein Verzeichnis verwaltet.

3.11.4 Simulation eines Laufwerks

Die Simulation eines (virtuellen) FAT-Laufwerks erfolgt mit der Klasse `Laufwerk`. Die beiden Datenstrukturen `Filehandle` und `Directory` werden als Unterstrukturen in der Klasse Laufwerk erzeugt. `Filehandle` ist im `public` – Bereich, enthält aber nur Attribute und Methoden von Typ `private`, `Directory` ist im `private` – Bereich definiert, also für den Anwender nicht zugänglich. Der Anwender erhält beim Öffnen einer Datei ähnlich wie in C eine Zeigervariable des Typs `Filehandle`, mit der er aber selbst nichts anfangen kann. Das Laufwerk wird über eine Datei simuliert, die FAT als Tabellenattribut in der Klasse `Laufwerk` deklariert.

```
class Laufwerk {
public:
    struct Filehandle;
    ...
    bool create_dir(string dname);
    bool change_dir(string dname);
    bool erase_dir(string dname);
    string list_dir();
    string dir();
    ...
    Filehandle* open(string fname);
    void read(Filehandle* handle, char* buf, int bytes);
    void write(Filehandle* handle, char const* buf,
                               int bytes);
    bool seek(Filehandle* handle, int pos);
    int  tell(Filehandle* handle);
    bool eof(Filehandle* handle);
    void close(Filehandle* handle);
    bool erase(string fname);
private:
    struct Directory;
    fstream f;     // Datei des Laufwerks
    string lwname;
    int fblocks;   // Größe des virtuellen Laufwerks
    vector<int> ftable; // FAT
    Directory* act_dir; // aktuelles Verzeichnis
    ...
};//end class
```

`act_dir` ist das aktuell verwendete Verzeichnis. Um die Handhabung der Verzeichnisstruktur möglichst einfach zu gestalten, erhält die Struktur `Directory` zusätzlich eine Attribut des Typs `Filehandle`. Beide müssen zudem auf Attribute und Methoden des Mutterobjeks `Laufwerk` zugreifen können, was zirkulare Abhängigkeiten schafft – programmiertheoretisch ein ziemlicher Fauxpas, aber wir haben es hier ja mit Betriebssystemstrukturen zu tun, und in der Betriebssystemprogrammierung sind ja Dinge an der Tagesordnung, die anderswo vermieden werden. Die erweiterten Strukturen sind somit

```
struct Laufwerk::Filehandle {
private:
    Filehandle(Laufwerk& l):
               lw(l), pos(0), sektor(0), eof(0),
               changed(false), eof_chgd(false) {}
    Laufwerk& lw;
    ...
    friend class Laufwerk;
```

```
    friend class Laufwerk::Directory;
};//end struct

struct Laufwerk::Directory {
    Directory(Laufwerk& l): lw(l), changed(false) {}
    Laufwerk& lw;
    Laufwerk::Filehandle* fh; // Handle zum Lesen der
                              // Verzeichnisdaten
    ...
};//end struct
```

Weitere Methoden in den einzelnen Klassen werden wir an verschiedenen Einsatzszenarien studieren.

3.11.5 Freie Sektoren und Zuordnung zu Dateien

Das erste Feld der FAT enthält immer den Startindex der Kette der freien Sektoren (*der erste Sektor bleibt damit ungenutzt*), der zweite Sektor ist der erste des Wurzelverzeichnisses. Sowohl die Liste freier Sektoren als auch alle in Verzeichnisse und Dateien umgelinkten Einträge der FAT sind damit eindeutig erreichbar.

Einen freien Sektor zur Vergößerung einer Datei erhält man durch Herausnahme des ersten Eintrags aus der Liste der freien Sektoren und Ersatz dieses Eintrags duch den dort angegebenen Index:

```
int Laufwerk::get_free(int last){
    int neu=ftable[0];
    ftable[0]=ftable[neu];
    ftable[neu]=-1;
    if(last>0){
        ftable[last]=neu;
    }//endif
    return neu;
}//end function
```

Der neue Sektor wird an das Ende der vorhandenen Sektorenkette der Datei angehängt. Der Index wird als Parameter übergeben und die -1 durch den Index des neuen Sektors ersetzt. Neue Sektoren werden somit einzeln abgerufen.

Datensektoren werden durch Löschen einer Datei frei, was aufgrund der Beziehung zwischen FAT und `Filehandle::chain` durch zwei Zuweisungen zu einer Rückgabe der Dateisektoren an die FAT führt (*eine Einzelrückgabe der Sektoren tritt im Prinzip nicht auf*).

```
void Laufwerk::make_free(vector<int>& b){
    ftable[b.back()]=ftable[0];
    ftable[0]=b.front();
}//end function
```

3.11.6 Initialisierung eines Laufwerks

Zum Kennenlernen der Funktionsweise der wesentlichen Systemmethoden betrachten wir zunächst die Einrichtung eines neuen Laufwerks. Die Funktionsabläufe beschränken wir auf das für diese Operation notwendige (*weitere Funktionalitäten der einzelnen Methoden werden bei weiteren Abläufen erläutert*).

`Laufwerk::Laufwerk(..)`. Zunächst wird die FAT durch

```
fat[i]=i+1
```

eingerichtet und in eine Datei geschrieben sowie eine weitere Datei für die Daten des Laufwerks erzeugt. Die gesamte FAT besteht zunächst aus einer Kette freier Sektoren. Mit Hilfe der statischen Methode `Directory::create_dir`, die ein Zeigerobjekt des Typs `Directory` zurückgibt, wird das Wurzelverzeichnis erzeugt und auf `act_dir` abgelegt.

```
act_dir=Directory::create_dir(*this,1);
Directory* Directory::create_dir(Laufwerk&,
                                 int parent_dir_sektor).
```

Die statische Methode erzeugt ein neues leeres Verzeichnis mit den beiden Standardeinträgen und liefert dieses als Zeigerobjekt zurück. Der Sektor des Elternverzeichnisses – hier der des Wurzelverzeichnisses, also des Verzeichnisses selbst – wird als Parameter übergeben und in den zweiten Verzeichniseintrage geschrieben.

```
dir->fh=Laufwerk::Filehandle::create_file(lw,-1);
strcpy(entry.name,".");
entry.s_block=dir->fh->chain.front();
entry.size=2*sizeof(DirEntry);
entry.dir=true;
dir->entries.push_back(entry);
strcpy(entry.name,"..");
entry.sektor=parent_block;
entry.size=0;
dir->entries.push_back(entry);
dir->changed=true;...
return dir;
```

Durch Aufruf der statischen Methode `Filehandle::create_file` wird zunächst eine neue Datei für das Verzeichnis angelegt. Als Heimatverzeichnissektor wird `-1` übergeben, da ein Verzeichnis erzeugt wird (*der Heimatsektor ist im Verzeichnis selbst abgelegt, die -1 verhindert die oben beschriebenen Aktualisierungsprobleme*). Der im Dateideskriptor belegte erste Index in `chain` wird für den Selbstbezug, der durch den ersten Verzeichniseintrag. hergestellt wird, benötigt. Bei Erzeugen des Wurzelverzeichnisses ist dies (korrekterweise) Sektor 1.

```
Filehandle* Filehandle::create_file(Laufwerk&,
                                    int home_dir_sektor).
```

Die statische Methode erzeugt ein neues Dateiobjekt für Daten- und Verzeichnisdateien und liefert es als Zeigerobjekt zurück.

```
fh->sektor=parent;
fh->changed=true;
fh->eof_chgd=true;
fh->chain.push_back(lw.get_free(-1));
return fh;
```

Positionszeiger und Größe werden mit Null initialisiert (Constructor), für die Belegungskette wird ein freier Sektor angefordert, wobei es sich hier um den Sektor Eins des Wurzelverzeichnisses handelt (*neue Dateien besitzen noch keine Ketten, so dass -1 als letzter zugewiesener Dateisektor in der Anforderng übergeben wird*). Das Sektorattribut hat den Inhalt -1 und zeigt damit ein Verzeichnis an. Nach Markieren durch `changed=true` und `eof_chgd=true` (*hiermit wird das Schreiben auf die Datei beim Schließen des Objekts erzwungen*) wird das Dateiobjekt zurückgegeben.

Mit Abschluss dieser Arbeitskette ist ein neues Laufwerk mit Wurzelverzeichnis erzeugt. Mit Abbau des Laufwerkobjekts wird es nun in die Datei geschrieben.

`Laufwerk::~Laufwerk().` Das Verzeichnisobjekt wird durch `delete act_dir` zerstört und hierdurch auf die Platte geschrieben. Alle im Anwenderbereich gegebenenfalls noch offenen Dateien werden nicht geschlossen, da im Laufwerk keine Angaben über exportierte Zeigerobjekte vorhanden sind. Dies kann zu Fehlern im Datenbestand führen!

Nach Schließen des aktuellen Arbeitsverzeichnis – hier: des neuen Wurzelverzeichnisses – wird die FAT in eine zweite Datei geschrieben.

`Directory::~Directory()`. Sofern das Verzeichnis als geändert markiert ist (*ist hier der Fall*), wird für der Dateideskriptor auf die Position Null gestellt und mit Hilfe der Laufwerksschreibmethoden der gesamte Verzeichnisinhalt in die Datei geschrieben. Anschließend wird die Datei geschlossen.

```
Laufwerk::Directory::~Directory(){
    int i;
    if(changed){
        list<DirEntry>::iterator it;
        fh->seek(0);
        entries.front().size=entries.size()*
                                    sizeof(DirEntry);
        for(it=entries.begin();it!=entries.end();it++){
            lw.write(fh,(char*)&(*it),sizeof(DirEntry));
        }//endfor
    }//endif
    delete fh;
}//end function
```

Vor dem Schreiben auf die Platte wird im ersten Verzeichniseintrag die Größe der gesamten Verzeichnisstruktur für das spätere Rücklesen von der Platte eingetragen.

`Filehandle::seek(int)`. Hier muss zunächst geprüft werden, ob sich durch das Positionieren der Datenpuffer ändert. Die Position darf die Dateigröße auch nicht überschreiten.

```
bool res;
    if(npos<=pos){
        res=true;
    }else{
        npos=eof;
        res=false;
    }//endif
    if(pos/1024 != npos/1024){
        if(changed){
            lw.f.seekp(chain[pos/1024]*1024,ios::beg);
            lw.f.write(buf,1024);
            changed=false;
        }//endif
        lw.f.seekg(chain[npos/1024]*1024,ios::beg);
        lw.f.read(buf,1024);
    }//endif
    pos=npos;
    return res;
```

Ändert sich der Inhalt des Datenpuffers durch die Positionierung (Positionierung auf ein anderes Segment der Platte), so muss ggf. der alter Sektor (falls verändert) gesichert und der neue eingelesen werden. Die Position auf der virtuellen Platte wird durch die Sektornummer in der Dateikette indiziert.

`Laufwerk::write(..)`. Der Methodenaufruf entspricht in der Syntax etwa dem als bekannt vorausgesetzten C-Aufruf. Die Methode besteht aus einem Schleifenkonstrukt, das byteweise die Daten an die Dateiobjektmethode `put` übergibt.

`Filehandle::put(..)`. Die Methode legt das übergebene Zeichen an der aktuellen Position im Puffer (ermittelt durch `pos%1024`) ab und erhöht den Positionszeiger. Gleichzeitig wird überprüft, ob die Dateigröße verändert wurde. Ist der Puffer gefüllt, wird er auf die Platte geschrieben (ermittelt durch `pos/1024-1`) und der nächste Pufferblock eingelesen. Dies kann auch ein neuer Sektor sein, der erst von der FAT angefordert und an das Ende der Sektorenkette der Datei angehängt werden muss. Hierbei ist aus der FAT-Kette jeweils die tatsächliche Position des Sektors zu ermitteln. Jede Schreiboperation führt außerdem zu `changed=true`, so dass beim Schließen der Datei noch nicht gesicherte Restdaten auf die Platte geschrieben werden.

```
buf[(pos++)%1024]=c;
changed=true;
if(pos%1024==0){
    lw.f.seekp(chain[pos/1024-1]*1024,ios::beg);
    lw.f.write(buf,1024);
    if(pos>eof){
        chain.push_back(lw.get_free(chain.back()));
    }//end function
    lw.f.seekg(chain[pos/1024]*1024,ios::beg);
    lw.f.read(buf,1024);
    changed=false;
}//endif

if(eof<pos){
    eof=pos;
    eof_chgd=true;
}//endif
```

`Filehandle::~Filehandle()`. Falls der aktuelle Sektor verändert wurde, wird er nun in die Datei geschrieben. Falls die Größe der Datei verändert wurde und die Datei kein Verzeichnis ist, wird das zugehörige Verzeichnis geändert. Da wir hier das Wurzelverzeichnis sicher, trifft dies nicht zu (sektor des Heimatverzeichnisses -1).

```
if(changed){
    lw.f.seekp(chain[pos/1024]*1024,ios::beg);
    lw.f.write(buf,1024);
}//endif

if(eof_chgd && sektor>0){
    ...
```

Damit ist die Inbetriebnahme des neuen virtuellen Laufwerks abgeschlossen.

> **Aufgabe.** Implementieren Sie nun alles und ergänzen Sie die nur angerissenen Kodeteile.

3.11.7 Laufwerk öffnen

Beim Öffnen eines vorhandenen Laufwerks wird nach Einlesen der FAT das Wurzelverzeichnis von Sektor 1 mittels der statischen Funktion

```
Directory* Directory::read_dir(Laufwerk&,
                               int sektor)
```

eingelesen. Diese öffnet zunächst die Datei, die das Wurzelverzeichnis enthält, und liest dann in zwei Schritten den Verzeichnisinhalt ein.

```
Directory* dir=new Directory(lw);
dir->fh=Laufwerk::Filehandle::open_file(lw,fb,
                                sizeof(DirEntry),-1);
DirEntry entry;
lw.read(dir->fh,(char*)&entry,sizeof(DirEntry));
dir->fh->eof=entry.size;
dir->entries.push_back(entry);
while(dir->fh->pos < dir->fh->eof){
    lw.read(dir->fh,(char*)&entry,sizeof(DirEntry));
    dir->entries.push_back(entry);
}//endwhile
return dir;
```

Die zwei Schritte sind notwendig, da die Größe der Datei und damit die Anzahl der Einträge im ersten Dateieintrag selbst gespeichert ist. Dieser wird zunächst eingelesen und anschließend die Dateigröße neu gesetzt. Das Einlesen erfolgt über die Dateideskriptormethode get, die eine umgekehrte Version von `put` ist. Beim Übergang zu einem neuen Sektor wird der alte gesichert, falls er verändert wurde.

```
Filehandle* Filehandle::open_file(Laufwerk&,
                  int sek, int eof, int rsek).
```

Mit dieser Methode wird eine existierende Datei geöffnet. Der erste Übergabeparameter gibt den Startsektor der Datei an, und mit Hilfe der FAT wird die `chain`-Kette gebildet. Der zweite Parameter ist die Dateigröße, im Fall eines Verzeichnis zunächst die Größe eines Verzeichniseintrags (*wird später korrigiert, siehe oben*). Der dritte Parameter ist der Startsektor des Heimatverzeichnisses, in diesem Fall -1, da Verzeichnisse selbst ja keine Heimatverzeichnisse benötigen.

```
Filehandle* fh=new Filehandle(lw);
fh->eof=eof;
fh->chain.push_back(fb);
fh->sektor=sektor;
while(lw.ftable[fb]!=-1){
    fb=lw.ftable[fb];
    fh->chain.push_back(fb);
}//endwhile
lw.f.seekg(fb*1024,ios::beg);
lw.f.read(fh->buf,1024);
return fh;
```

Das Wurzelverzeichnis ist nun geöffnet. Im Weiteren spielt sich gegenüber dem ersten Vorgang nichts Neues ab.

`char Filehandle::get()`. Das Einlesen der Daten auf Filehandleebene erfolgt ähnlich wir das Schreiben byteweise. Auch hier ist zu überprüfen, ob beim

Überschreiten der Segmentgrenze das alte Segment gesichert werden muss, weil neben den Lese- auch Schreiboperationen stattgefunden haben.

```
char c;
c=buf[(pos++)%1024];
pos=min(eof,pos);
if(pos%1024==0){
    if(changed){
        lw.f.seekp(chain[pos/1024-1]*1024,ios::beg);
        lw.f.write(buf,1024);
        changed=false;
    }//endif

    lw.f.seekp(chain[pos/1024]*1024,ios::beg);
    lw.f.read(buf,1024);
}//endif
return c;
```

3.11.8 Dateien öffnen

In Verzeichnissen können nun Dateien erzeugt und Daten in sie geschrieben werden. Dabei wird immer das aktuelle Verzeichnis verwendet. Beim Öffnen ist zu prüfen, ob die Datei bereits im Verzeichnis angelegt ist oder neu eingerichtet werden muss.

```
Laufwerk::Filehandle* Laufwerk::open(string fname){
    list<Directory::DirEntry>::iterator it;
    Filehandle* fh;
    Directory::DirEntry entry;
    for(it=act_dir->entries.begin();
                   it!=act_dir->entries.end();it++){
        if(fname==string(it->name) && !it->dir){
            return Filehandle::open_file(*this,
                           it->sektor,it->size,
                           act_dir->entries.front().sektor);
        }//endif
    }//endfor
    strcpy(entry.name,fname.c_str());
    entry.size=0;
    entry.dir=false;
    fh=Filehandle::create_file(*this,
                  act_dir->entries.front().sektor);
    entry.sektor=fh->chain.front();
    act_dir->entries.push_back(entry);
    act_dir->changed=true;
    return fh;
}//end function
```

Die Methoden zum Einrichten einer neuen Datei oder zum Öffnen einer vorhandenen sind oben bereits beschrieben. Bei einer neuen Datei wird ein neuer Eintrag in die Liste eingefügt und das Verzeichnis als geändert markiert, um beim Verzeichniswechsel ein Schreiben der neuen Daten auf die Platte zu erzwingen.

3.11.9 Verzeichnisse erzeugen

Auch hier muss zunächst kontrolliert werden, ob ein Verzeichnis dieses Namens bereits existiert. Die Erzeugung des neuen Verzeichnisses erfolgt wie beim Wurzelverzeichnis beschrieben. Da aber nicht in das neue Verzeichnis gewechselt wird, wird es anschließend als Objekt wieder gelöscht und damit auf die Platte geschrieben.

```
bool Laufwerk::create_dir(string name){
    list<Directory::DirEntry>::iterator it;
    Directory::DirEntry entry;
    Directory* dir;
    for(it=act_dir->entries.begin();
                    it!=act_dir->entries.end();it++){
        if(name==string(it->name)){
            return false;
        }//endif
    }//endfor
    dir=Directory::create_dir(*this,
                    act_dir->entries.front().sektor);
    strcpy(entry.name,name.c_str());
    entry.dir=true;
    entry.size=0;
    entry.sektor=dir->entries.front().sektor;
    act_dir->entries.push_back(entry);
    act_dir->changed=true;
    delete dir;
    return true;
}//end function
```

3.11.10 Verzeichnis wechseln

Sofern ein Verzeichnis des angegebenen Namens vorhanden ist, wird es eingelesen und das aktuelle Verzeichnis durch das neu eingelesene ersetzt.

```
bool Laufwerk::change_dir(string name){
    list<Directory::DirEntry>::iterator it;
    Directory* dir;
```

```
    for(it=act_dir->entries.begin();
                    it!=act_dir->entries.end();it++){
        if(name==string(it->name) && it->dir){
            dir=Directory::read_dir(*this,it->sektor);
            delete act_dir;
            act_dir=dir;
            return true;
        }//endif
    }//endfor
    return false;
}//end function
```

3.11.11 Löschen von Dateien

Dateien müssen beim Löschen geschlossen sein (was allerdings in dieser einfachen Version nicht überprüft wird). Das Löschen besteht im Umhängen der Sektorenliste in den Freibereich der FAT, wobei aber durch Setzen der Änderungsmerker auf den Wert `false` dafür gesorgt werden muss, dass beim Schließen des Dateiobjektes dieses nicht versucht, noch irgendetwas auf die Platte zu schreiben.

```
bool Laufwerk::erase(string fname){
    list<Directory::DirEntry>::iterator it;
    Filehandle* fh;
    for(it=act_dir->entries.begin();
                    it!=act_dir->entries.end();it++){
        if(fname==string(it->name) && !it->dir){
            fh=Filehandle::open_file(*this,
                    it->sektor,it->size,-1);
            make_free(fh->chain);
            fh->changed=fh->eof_chgd=false;
            delete fh;
            act_dir->entries.erase(it);
            act_dir->changed=true;
            return true;
        }//endif
    }//endfor
    return false;
}//end function
```

Das Verzeichnis wird nach Entnahme des Dateieintrags als geändert markiert und kann beim Sichern auf die Platte auch weniger Sektoren also vorher belegen. Im System ist jedoch keine Freigabe einzelner Sektoren vorgesehen. Verzeichnisse können wie normale Dateien nur wachsen, aber nicht den belegten Platz dynamisch anpassen.

3.11.12 *Löschen von Verzeichnissen*

Verzeichnisse werden rekursiv gelöscht, d.h. zunächst werden alle Dateien gelöscht, dann alle ggf. enthaltene Unterverzeichnisse. Die beiden ersten Einträge (Selbstbezug und Elternverzeichnis) können nicht gelöscht werden. Um die Rekursion zu ermöglichen, wird `act_dir` nach Zwischensicherung mit dem zu löschenden Unterverzeichniss überschieben und dessen Einträge schrittweise gelöscht.

```
bool Laufwerk::erase_dir(string dname){
    list<Directory::DirEntry>::iterator it,jt;
    Directory* dir;
    if(dname=="." || dname=="..") return false;
    for(it=act_dir->entries.begin();
                    it!=act_dir->entries.end();it++){
        if(dname==string(it->name) && it->dir){
            dir=act_dir;
            act_dir=Directory::read_dir(*this,
                                        it->sektor);
            jt=act_dir->entries.begin();
            while(jt!=act_dir->entries.end()){
                if(jt->dir){
                    erase_dir(jt->name);
                    jt=act_dir->entries.erase(jt);
                }else{
                    erase(jt->name);
                    jt=act_dir->entries.begin();
                }//endif
            }//endwhile
            make_free(act_dir->fh->chain);
            act_dir->changed=act_dir->fh->changed=
                   act_dir->fh->eof_chgd=false;
            delete act_dir;
            act_dir=dir;
            act_dir->entries.erase(it);
            act_dir->changed=true;
            return true;
        }//endif
    }//endfor
    return false;
}//end function
```

Nach leeren des Verzeichnisses wird dessen Sektorenkette in den Freibereich der FAT eingefügt und das Objekt gelöscht. Durch Löschen aller Veränderungsmerker

wird dabei verhindert, dass die gelöschten Daten erneut auf die Platte geschrieben werden. Abschließend wird das gesicherte Hauptverzeichnis wieder aktiviert.

Ähnlich werden die Auflistungsmethoden implementiert, die dem Leser als Aufgabe überlassen seien.

3.11.13 Abschlussbemerkungen

Wir haben zwar nun ein komplettes Laufwerk implementiert, das wir zusammen mit einer Dateiverschlüsselung zu einem verschlüsselten Laufwerk ausbauen können, jedoch können wir es zunächst nur innerhalb einer Anwendung als simuliertes Laufwerk verwenden.

> **Aufgabe**. Entwerfen Sie ein Testprogramm zur Untersuchung des Laufzeitverhaltens eines Laufwerkes. Wenn die Schnittstellen korrekt eingehalten wurden, sollte es mit dem simulierten Laufwerk ebenso arbeiten können wie mit dem echten Laufwerk (ggf. müssen Sie noch einige Anpassungsmethoden schreiben).
>
> Testen Sie das Laufzeitverhalten des normalen Laufwerks, des simulierten Laufwerks (das sich ja eines normalen Laufwerks bedient), eines verschlüsselten normalen Laufwerks (z.B. mit TrueCrypt) und eines verschlüsselten simulierten Laufwerks.
>
> **Aufgabe**. Bei kleinen Laufwerksgrößen ($\sim$ 100-300 MB, je nach Hauptspeicher) kann man auch ein RAM-Laufwerk simulieren, das nur Speicher im Heap verwendet (ggf. kann das komplette Laufwerk auf der Platte gesichert und wieder geladen werden). Ändern Sie die Gesamtimplementation für diese Verwendung und führen Sie Laufzeittests durch.

Um es in bestehenden Anwendungen einsetzen zu können, müssen Sie nun allerdings Ihrem Betriebssystem klarmachen, dass es als Laufwerk akzeptiert und in die Laufwerksliste eingebunden wird. An der Stelle müssen wir jedoch in diesem Buch abbrechen, und falls Sie weitermachen wollen, müssen Sie nun auf die passende Betriebssystemliteratur zurückgreifen.

Das FAT-System ist zudem veraltet und nicht bzw. nur bedingt für größere Laufwerke geeignet. Neuere Dateiverwaltungssysteme verwenden Strukturen, die auf Bäumen basieren (B+ Bäume). Falls Sie in dieses Gebiet ebenfalls vorstoßen wollen, muss hier ebenfalls auf speziellere Betriebssystemliteratur verwiesen werden.