Kapitel 10
Speicherverwaltung (und ein wenig mehr)

10.1 Die Laufzeitproblematik

Das Überschreiben beliebiger Operatoren in C++ ist zwar für die Programmerstellung recht angenehm, bringt jedoch auch Probleme mit sich. Betrachten wir die Anweisungen

```
string s1,s2,s3;
s1="erster Teilstring"
s2="zweiter Teilstring"
s3=s1+s2;
```

so laufen bei der Bearbeitung der letzten Anweisungszeile folgende Prozesse ab:

(a) Die Strings `s1` und `s2` sollen während der Operation konstant bleiben. `s1` stellt die Methode des Addionsoperators zur Verfügung. Diese erzeugt einen neuen String und kopiert in diesen hintereinander die Inhalte von `s1` und `s2`.

```
string operator+(const string& t) const {
    string t;
    ...
    return t;
}//end function
```

(b) Der String mit dem Ergebnis befindet sich irgendwo im Speicher, der der Additionsmethode zur Verfügung steht. Die Position kennt der rufende Programmteil nicht, kann also das Ergebnis nicht auf `s3` kopieren. Die Position bei Rückgabe mitzuteilen ist keine gute Lösung, da bei weiteren Funktionsaufrufen aus dem Hauptprogramm den aufgerufenen Methoden mitgeteilt werden müsste, dass bestimmte Speicherbereiche gesperrt sind und die Additionsfunktion obendrein noch zwischen Variablen, die Rückgabewerte sind, und solchen, die es nicht sind, unterscheiden müsste, da sie bei ersteren die Destruktoren nicht ausführen darf. Bei Verlassen der Methode wird deshalb ein weiterer String erzeugt, ohne dass der Programmierer dies im Code bemerkt und dessen Position dem Hauptprogramm bekannt ist und der Zuweisungsmethode (*bereitgestellt*

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_10, © Springer-Verlag Berlin Heidelberg 2010

von `s3`) mitgeteilt werden kann. In diesen wird der Inhalt des ersten Strings übernommen, der anschließend völlig normal mit seinem Destruktor zerstört wird.

Eine Position, die das Hauptprogramm kennt, ist die Adresse des Stacks bei Aufruf der Unterfunktion. An dieser Stelle wird die Übergabevariable erstellt. Da sie nur bis zum Ende der Anweisungszeile benötigt wird (*dann ist ihr Inhalt kopiert oder insgesamt uninteressant*), ist sie eine temporäre Variable ohne eigenen Namen im Programm.

```
STACK .....................
          |      |
          |      bekannte Position im Hauptprogramm
         Position von „t"
         -------->  Kopie von „t" nach „tt"
```

(c) Nach Bearbeitung der „Addition“ wird in der Zuweisungsmethode der Inhalt des zweiten temporären Strings auf `s3` übernommen. Damit ist die Anweisungszeile vollständig abgearbeitet und der temporäre String wird nun durch Aufruf des Destruktors zerstört.

Da die Stringlänge nicht generell festgelegt werden kann, sind die Datenspeicher von Strings dynamische Objekte, die nach Bedarf erzeugt und angepasst werden. Würde nun ein String einfach in der Form

```
class string {
private:
   int len;
   char* buffer;
   ...
};//end class
```

definiert, so wäre im Laufe der Bearbeitung ein Datenpuffer zweimal anzufordern und zu zerstören, der Inhalt zweimal zu kopieren. Bei noch komplexeren Abläufen kann der Aufwand weiter wachsen und viel Zeit für Erzeugen, Kopieren und Vernichten aufgewendet werden. Wir untersuchen nun Methoden, die diesen Aufwand vermeiden.

10.2 Das einfache Referenzkonzept

Zunächst verlagern wir die datentragenden Attribute in eine separate Struktur, von der Zeigerobjekte erzeugt werden. Das Arbeitsobjekt in der Anwendung enthält nicht mehr die Attribute direkt, sondern nur noch einen Zeiger auf ein Datenobjekt. Das enthebt uns der Notwendigkeit, die Attribute bei Erzeugung oder Freigabe eines „Trägerobjektes“ ebenfalls zu erzeugen oder zu freizugeben. Wir können die Zeiger auf die eigentlichen Dateninhalte an andere Objekte weitergeben, müssen

nun jedoch auf die Lebensdauer achten und Mechanismen bereitstellen, die selektives Verändern von zu einem bestimmten Trägerobjekt gehörenden Inhalten ermöglichen.

Das erste Problem haben wir bei den Objektfabriken schon einmal in einem anderen Zusammenhang bearbeitet und können die Erfahrungen hier übernehmen: betrachten wir das Konstruktor/Destruktor–Geschehen genauer, so ist nicht auszuschließen, dass ein Attribut gleichzeitig zu mehreren noch lebenden Objekten gehört. Wir müssen daher eine Buchführung implementieren, in dem wir nachhalten, wie viele Objekte ein Attribut benutzen.

Wir demonstrieren dies an einer Vorlagenklasse für Polynome. Aus einer ersten Implementierung

```
template <class T> class Polynom {
    ...
    int grad;       // Polynomgrad
    int reserved;  // reservierter Speicherplatz,
    T * value;    // Koeffizienten des Polynoms
```

wird durch Ausgliederung des Datenteils:

```
template <class T> class Polynom {
    ...
    struct DATA {
        int grad;          // Polynomgrad
        int reserved;      // reservierter Speicher
        T * value;         // Koeffizienten
        int ref_count;     // Objekt-Anzahl
    } * data ;
```

Die Datenstruktur ähnelt derjenigen der STL–Vorlagenklasse `vector`. Der Speicherplatz wird in größeren Einheiten reserviert. Hierdurch wird eine Reserve gebildet, die ständige Neukonfiguration des Platzes unnötig macht. An dieser Stelle sind weitere Optimierungen möglich, auf die wir später kommen. Diese sowie die mathematischen Eigenschaften von Polynomen sind auch der Grund, weswegen wir die Klasse `vector` nicht direkt verwenden. Wir begründen auch das später noch genauer.

Schauen wir uns die ablaufenden Operationen bei Addition zweier Polynome nun parallel an:

Die Lebensdauer der inneren Datenstruktur wird durch das zusätzliche Attribut `ref_count` gesteuert. Soll der Dateninhalt in eine weitere Variable übernommen werden, so wird er nicht kopiert, sondern einfach der Zeiger auf die bereits vorhandenen Daten übernommen und die Zählvariable erhöht. An ihr ist abzulesen, wie viele Objekte gleichzeitig auf die Datenstruktur verweisen. Die Ausführungszeit von des Kopierkonstruktors verkürzt sich auf diese Weise erheblich.

Wird ein Trägerobjekt freigegeben, so muss die zugehörende Datenstruktur nur dann ebenfalls freigegeben werden, wenn kein weiteres Objekt mehr darauf verweist. Die Operationen im Destruktor reduzieren sich daher in vielen Fällen auf

Schritt	Ohne Referenz	Mit Referenz

Konstruktor	/*Initialisierung*/	data = new DATA();
		data->ref_count=0;
		/*Initialisierung*/

Kopier-Konstruktor	/*Initialisierung*/ /*Kopie des Dateninhaltes*/	data=p.data; data->ref_count++;

DestruKtor	/*Speicherfreigabe*/	data->ref_count--;
		if(data->ref_count<0){
		
		/*Speicherfreigabe*/
		delete data;
		}//endif

Zuweisungs-Operator	/*Kopie des Inhaltes*/	data->ref_count--;
		if(data->ref_count<0){
		/*Speicherfreigabe*/
		delete data;
		{//endif
		data=p.data;
		data->ref_count++;

Destruktor	Siehe c)	Siehe c)

die Dekrementierung der Zählvariablen, und ein Aufräumen des Speicherplatzes entfällt. In der Zuweisungsoperation wird das Kopieren von Inhalten durch das Aufräumen des vorhandenen Datenspeicherplatzes und die Übernahme des Zeigers anderen Trägerobjektes ersetzt. Zusammengefasst erreichen wir bei drei Vorgängen eine Verbesserung des Aufwands bei neutralem Verhalten der restlichen.

Die Betrachtung ist allerdings so noch nicht korrekt, da wir den zweiten Teil unserer Aufgabenliste noch nicht berücksichtigt haben. Besitzen zwei Trägerobjekte einen Zeiger auf das gleiche Datenobjekt, so darf das natürlich nicht dazu führen, dass bei Zuweisung eines neuen Wertes zu einem Trägerobjekt sich der Inhalt des anderen ebenfalls ändert. Konkret: verweist etwa bei Ausführung der Anweisungszeile `a+=b` auch ein Objekt C auf den gleichen Speicherplatz wie `a`, so wird ohne entsprechende Maßnahmen sein Inhalt fälschlicherweise ebenfalls geändert. Dies verhindern wir durch die gezielte Erstellung einer Kopie des Datenbereiches, sobald dies aufgrund einer Operation notwendig wird

```
Polynom<T>& Polynom::operator+=(
                          const Polynom<T>& p){
    DATA * help;
    if(data->ref_count>0){
        help=new DATA();
        /*Kopieren data -> help */
        data.ref_count--;
```

```
        data=help;
  }//endif
...
```

Zählen wir nun die im Hintergrund ablaufenden Operationen ab, so erhöht sich der Aufwand nun, da bei Ausführung des Zuweisungsoperators durch Übernahme des Datenzeigers eine Speichereinheit freigegeben wird, zur Erstellung einer Kopie in einer späteren Operation aber wieder eine neue Einheit erzeugt werden muss. Die Verbesserung in einem Teilbereich wird also durch eine kleine Verschlechterung in einem anderen erkauft. Gegenstrategien sind zwar theoretisch möglich, aber nicht empfehlenswert, da hierdurch unsere Regeln für sauberes Programmieren verletzt werden. Der Gewinn des Referenzmodells liegt ganz eindeutig in der schnelleren Abwicklung von Operationen mit temporäre Variablen bei (*fast*) Aufwandneutralität bei anderen Operationen.

Das Referenzkonzept kann durch eine eigene Vorlagenklasse implementiert werden. Die eigentlichen Speicherstrukturen werden als eigene Klasse definiert und als Typparameter an die Vorlagenklasse übergeben:

```
// Referenzzählung durch Template-Klasse
template <class T> class SimpleReference {
public:
    SimpleReference(){count=0;};
    ~SimpleReference(){};
    ...
    int count;
    T obj;
};//end class

// Arbeitsklasse
class Work {
    SimpleReference<Data> * dat;
```

Um eine weitere Variable auf einen Datenbereich zeigen zu lassen, implementieren wir die Methode `NewRef(..)`, die den Zähler des Quellbereiches erhöht und den nicht mehr benötigten Zielbereich freigibt.

```
// Methoden in „SimpleReference"
inline SimpleReference<T> *
    NewRef(SimpleReference<T> * dest){
        count++;
        dest->Delete();
        return this;
};//end function
inline void Delete(){
    count--;
    if(count<0)
```

```
        delete this;
};//end function
// Arbeitsklasse
Work& operator=(const Work& w){
    dat=w->NewRef(dat);
    return *this;
};//end method
```

Aufgabe. Die Reihenfolge von `count++` und `dest->Delete()` in der Methode `NewRef(..)` ist wichtig und darf nicht geändert werden. Warum?

Wird in einer Methode eine Veränderung der Daten durchgeführt, so muss der Datenbereich zunächst auf eine eigene Variable kopiert werden, sofern mehrere Variable auf ihn zeigen:

```
// Kopiermethode
inline SimpleReference<T> * Copy(){
    SimpleReference<T> * n;
    if(count==0)
        return this;
    count--;
    n=new SimpleReference<T>;
    n->tobj=tobj;
    return n;
};//end function
// Arbeitsklasse
Work& operator+=(const Work& w){
    dat=dat->Copy();
    ...
```

Aufgabe. Wie viele Kopieroperationen werden in der Anweisung

```
a = b * (c + d);
```

durch das Referenzkonzept eingespart? Überprüfen Sie Ihre Lösung mittels eines Testprogramms. Könnte theoretisch noch mehr eingespart werden?

10.3 Referenzen mit temporärer Zwischenspeicherung

Das einfache Referenzkonzept beinhaltet die Freigabe der Speicherobjekte, sobald das letzte darauf zeigende Trägerobjekt ungültig wird. Wenn auch die Anzahl der Freigabe- und Erzeugungsoperationen durch das Referenzkonzept erniedrigt wird, bleiben doch in Abhängigkeit vom Algorithmus möglicherweise erhebliche Mengen von Erzeugungs- und Freigabeoperationen übrig, die durch die Speicherverwaltung der Laufzeitumgebung meist nicht sonderlich effektiv bearbeitet werden und die

eigentlich auch unnötig sind, wenn im weiteren Verlauf der Anwendung an anderer Stelle neue Trägerobjekte erzeugt werden, die ja ebenfalls Bedarf an Datenobjekten haben.

Hinzu treten weitere Effekte, die zu zusätzlichen Laufzeitverlusten führen: Stellen Sie sich Polynome vor, deren Koeffizientenfeld jeweils mit der Grundgröße 10 initialisiert wird, im weiteren Verlauf der Rechnung aber auf 50 oder mehr Koeffizienten schrittweise erweitert werden muss. Erlischt die letzte Referenz auf diesen erweiterten Speicherbereich, so geht natürlich auch die Kenntnis der notwendigen Reserve verloren, und die Speicheranpassung wiederholt sich ständig aufs Neue.[1]

10.3.1 Die Strategie

Eine Ausweichstrategie besteht in der Zwischenspeicherung nicht mehr benötigter Datenobjekte in einer für den Anwender unsichtbaren Kellerliste anstelle einer Rückgabe des Speicherplatzes an das Betriebssystem und einer Wiederverwendung dieser zwischen gesicherten Datenobjekte, sobald ein neues Hauptobjekt einen Datenbereich anfordert. Die Arbeit kann so beschrieben werden:

- Die Objektverwaltung legt beim Start der Anwendung oder spätestens bei der ersten Erzeugung eines Trägerobjektes eine leere Kellerliste hinreichender Größe für Zeigervariablen an.
- Wird ein Datenelement freigegeben, so wird der Zeiger auf das Datenelement in die Kellerliste eingefügt wird. Wird die vorgegeben Größe erreicht, so wird die Liste in vorgegebenen nicht zu kleinen Schritten vergrößert.
- Wird ein Datenelement angefordert, so wird das oberste Zeigerobjekt aus der Kellerliste entnommen und der Anwendung das Datenobjekt nach Reinitialisierung zur Verfügung gestellt. Ist die Kellerliste leer, so wird ein neues Datenobjekt durch Anforderung von Speicherplatz vom Betriebssystem erzeugt.
- Zum Programmende werden alle in der Liste vorhandenen Zeigerobjekte gelöscht (*für ein Löschen der Objekte vor Beenden der Anwendung besteht in der Regel keine Notwendigkeit, so dass wir diese Strategie hier außer Acht lassen*)..

Zusätzlich zu den Einsparungen gegenüber dem Aufwand der allgemeinen Freispeicherverwaltung der Laufzeitumgebung haben die Datenobjekte nun auch ein „Gedächtnis“ über die benötigten Speicheranpassungen: Nach einigen Rechenschritten verfügen die gespeicherten Datenobjekte bereits die notwendige Reserve, so dass weitere Anpassungen nur noch selten notwendig sind.

---

[1] Man könnte natürlich an eine Buchführung über die mittlere oder maximale Länge denken und die Initialisierungsgröße während der Lebenszeit der Anwendung anpassen. Aufwandsmäßig ist das kein großer Kostenfaktor. Wir gehen hier aber einen anderen Weg.

10.3.2 Die Basisklasse

Für die Kellerliste verwenden wir ein Zeigerfeld. Außerdem benötigen wir ein Iteratorattribut, das auf den letzten und den nächsten freien Platz in der Liste verweist. Template–Parameter ist wieder die Datenklasse, von der wir voraussetzen, dass sie ein zugängliches Attribut `ref_cnt` besitzt. Die Liste wird in sinnvollen Schritten vergrößert, zum Beispiel jeweils um 1.024 Speicherplätze für Zeiger. Im Destruktor müssen wir die mit Datenobjekten belegten Zeiger natürlich freigeben. Das Grundverwaltungsgerüst erhält damit die Form:

```
template <class S> class RefAllocator {
private:
    typedef S** PS;
    PS array, end, it;

    RefAllocator(){
        array=(PS)malloc(1024*sizeof(PS));
        end=array+1024;
        it=array;
    };//end constructor

    RefAllocator(const RefAllocator<S>& s);
    RefAllocator<S>& operator=();

    inline void Resize(){
        int d;
        d=distance(array,end);
        array=(PS)realloc(
                      array,(d+1024)*sizeof(PS));
        end=array+(d+1024);
        it=array+d;
    };//end function

public:
    ~RefAllocator(){
        PS pi;
        for(pi=array;pi!=it;++pi)
            delete *pi;
        free(array)
    };// end destructor
    ...
```

Für die Erzeugung, Freigabe und Kopie der Datenobjekte implementieren wir die in einen früheren Kapitel bereits entwickelten Methoden nun unter Nutzung der Kellerliste im `public`–Bereich:

```
inline S * New(){
    S * s;
    if (it!=array) {
        s=*(--it);
        s->Init();
    } else {
        s= new S();
    }//endif
    s->ref_cnt=1;
    return s;
};//end function

inline S* Reference(S* ziel, S* quelle){
    quelle->ref_cnt++;
    if (ziel)
        Delete(ziel);
    return quelle;
};//end function
inline S* Copy(S* obj){
    if (obj->ref_cnt > 1) {
        obj->ref_cnt--;
        S* result = New();
        *result = *obj;
        result->ref_cnt=1;
        return result;
    } else {
        return obj;
    }//endif
};//end function
inline void Delete(S * obj){
    if (--obj->ref_cnt == 0) {
        if(it==end)
            Resize();
        *it=obj;
        ++it;
    }//endif
};//end function
```

Die einzige zusätzliche Anforderung an die Datenobjekte gegenüber der einfachen Referenzzählung ist die Existenz der Methode `Init()`. Bei Freigabe der Datenobjekte sind diese in einem nicht näher festgelegten Zustand. Bei einem Polynom kann dies beispielsweise ein höheres Polynom sein. Für ein neu erzeugtes Hauptobjekt ist

jedoch ein Standardzustand wie ein Nullpolynom sinnvoll, der normalerweise durch den Konstruktor hergestellt wird. Die Rolle des Konstruktors übernimmt die Methode `Init()` bei wiedergenutzten Objekten, die beispielsweise das Gradattribut auf Null zurücksetzt.

10.3.3 Die Ankerobjekte der Speicherverwaltung

Für jeden Datenobjekttyp benötigen wir eine Kellerliste und für jede Kellerliste ein Ankerobjekt. Für diese Objekte ist zu fordern:

- In einer Anwendung sollte für einen bestimmten Vorlagentyp nur eine einzige Instanz existieren.
- Dem Anwendungsprogrammierer sollte kein zusätzlicher Aufwand durch die Listenverwaltung entstehen.

Als ersten Schritt zur Gewährleistung dieser Bedingungen sind die Konstruktoren (*Basiskonstruktor, Kopierkonstruktor*) sowie der Zuweisungsoperator im privaten Bereich der Klasse deklariert, das heißt eine frei nutzbare Variable kann nicht (*versehentlich*) erfolgreich deklariert werden. Die Deklaration einer Variablen kann nun nur statisch innerhalb der Klasse erfolgen, womit die anwendungsweite Existenz nur einer Instanz gewährleistet ist. Für den Zugriff auf diese Variable benötigen wir eine statische Funktion in der Klasse. Die Definition einer statischen Variablen in der Klasse führt jedoch auf ein Problem: sie muss irgendwo im Code auch deklariert werden, beispielsweise:

```
class Foo {
    static double d;
    ...
};
...
double Foo::d=4.025;
```

Das ist bei normalen Trägerklassen kein Problem, denn die Ankervariable kann hier im Implementationsteil der Klasse deklariert werden, bei Template–Trägerklassen wie Polynomen ist das aber unangenehm, da erst zur Compilezeit des entgültigen Programms feststeht, was benötigt wird. Konkret: Geben wir dem statischen Attribut/Ankerobjekt den Namen `allocator` und kürzen wir die aufwendige Typbezeichnung eines komplexeren zusammengesetzten Typs durch eine `typedef`-Anweisung ab, so sind Anweisungen wie

```
typedef RefAllocator<
        Polynom<Restklasse<int> >::Data>  PRD;
...
PRD PRD::allocator;
```

auf verschiedene Dateien verteilt, die im Laufe der Anwendungserstellung entstehen. Hinzu tritt ein weiteres generelles Problem: Statische Variablen werden vor dem eigentlichen Programmstart erzeugt. Ist darunter auch eine Variable der Trägerklasse, so stellt sich die Frage, ob das Ankerobjekt zu dem Zeitpunkt, zu dem es benötigt wird, bereits erzeugt ist. Dieses Problem der dynamischen Erzeugung lässt sich durch eine statische Initialisierung beseitigen:

```
template <class S> class RefAllocator {
private:
    static RefAllocator<S>* allocator;
public:
    inline static RefAllocator<S>& Allocator(){
        if(allocator==0)
            allocator= new RefAllocator<S>();
        return *allocator;
    };//end function
};//end class
...
RefAllocator<MyClass>*
            RefAllocator<MyClass>::allocator=0;
```

Das beseitigt zwar das Reihenfolgeproblem der Erzeugung, aber nicht das der Deklaration der Ankervariablen und führt zusätzlich auf ein weiteres: Die Laufzeitumgebung baut beim Beenden keine Zeigervariablen automatisch ab. Das sieht hier zwar unkritisch aus, wenn jedoch die gleichen Verfahren auf eine Netzwerkanwendung angewandt werden, werden Ressourcen auf entfernten Rechnern nicht wieder ordnungsgemäß freigegeben. Man sollte mit solchen Nachlässigkeiten daher gar nicht erst anfangen, auch wenn es im Einzelfall unkritisch aussieht.

Das Freigabeproblem lässt sich durch eine weitere statische Methode und die Verwendung eines Betriebssystemaufrufs beseitigen:

```
static void DeleteAllocator() { ... };
inline static RefAllocator<S>& Allocator(){
    if(allocator==0){
        allocator= new RefAllocator<S>();
        atexit(&RefAllocator<S>::DeleteAllocator);
    }//endif
    return *allocator;
};//end function
```

`atexit` übernimmt den Zeiger auf die Funktion `DeleteAllocator()` auf einen Stack, der bei Beenden des Programm abgearbeitet wird, so dass das Allocator–Objekt ordnungsgemäß freigegeben werden kann. Bei gegenseitigen Abhängigkeiten sind dann zwar immer noch nicht alle Reihenfolgeprobleme beseitigt, hier aber schon.

Mit einer weiteren Modifikation können wir für unseren Anwendungsfall alle Anforderungen erfüllen:

```
inline static RefAllocator<S>& Allocator(){
    static RefAllocator<S> allocator;
    return allocator;
};//end function
```

Als statisches Objekt in der statischen Funktion `Allocator()` ist das Ankerobjekt deklariert und die externe Deklaration ist nicht mehr notwendig, das heißt der Anwendungsentwickler kann sich auf die Typdefinition beschränken. Der Compiler beziehungsweise die Laufzeitumgebung ist nun auch zwangsverpflichtet, das Ankerobjekt spätestens vor der `return`–Anweisung zu erzeugen, und als statisches Objekt ist der Abbau bei Programmende ebenfalls garantiert.

Allerdings ist das Objekt nur über diese Funktion zugänglich und wird im Programm erzeugt, auch wenn es vielleicht gar nicht benötigt wird. Das spielt für das hier untersuchte Speicherkonzept keine Rolle, aber möglicherweise in anderen Anwendungen, weshalb ich hier auch anderen Lösungen Raum gegeben habe.

> **Aufgabe.** Führen Sie mit den verschiedenen Modellen Laufzeituntersuchungen durch. Überlegen Sie, wie die Testfälle gestaltet werden müssen, um die Einflüsse der internen Verwaltung temporär nicht benötigter Objekte und des Gedächtnisses der Objektgröße auch messen zu können. Zur Erinnerung: in Kap. 3 wurden bereits Werkzeuge zum Thema „Objektstatistiken“ vorgestellt.

10.4 Ein universeller Datenpuffer

Zur Übung implementieren wir nun eine Erweiterung der STL–Stringklasse für die Pufferung von Binärdaten, die wir in den folgenden Kapiteln einsetzen werden. Die STL–Klasse `string` ist ihrerseits eine Spezialisierung der allgemeineren STL–Klasse `basic_string<..>` für den Datentyp `char` mit einer Reihe von Zeichenkettenfunktionen, die denen in C entsprechen. Auch wenn man meist Objekte der Klasse `string` für die Arbeit mit lesbaren Zeichenketten verwendet (*siehe beispielsweise die Parameterstrings*), so ist auch eine Verarbeitung von Binärdaten möglich, wobei mittels der Methoden `resize(..)/reserve(..)` die Größe des Datenbereiches und des Reservepuffers kontrolliert wird. Allerdings ist das einfache Anfügen und Entfernen von Daten ist nur am hinteren Ende des Datenpuffers vorgesehen. Müssen Daten vor die vorhandenen eingefügt werden, so werden aufwendige Kopier- und Verschiebeoperationen notwendig, da der Beginn des Datenbereiches immer mit dem Beginn des physikalischen Puffers zusammenfällt.

```
string s1, s2;
s1="12345" ;  s2="abcd"
s1=s2+s1;
```

```
// Der Inhalt von s1 ist "abcd12345", die
// Pufferlänge 10.
// Bei Verwendung des „+"-Operators werden die
// Daten auf eine temporäre Variable kopiert. In
// einer optimierten Version werden die Daten von
// s1  nach hinten verschoben (sofern Reserven
// vorhanden sind) und anschließend der Inhalt von
// s2  an den Beginn kopiert.
```

Auch das Entfernen von Informationen vom Pufferbeginn erfolgt durch Verschieben der gesamten nachfolgenden Informationen nach vorne. In vielen Anwendungen werden aber gerade Anfüge- oder Löschoperationen an beiden Enden des Datenpuffers benötigt. Die STL–Containerklasse `deque` besitzt zwar diese Eigenschaften, uneingeschränkt geeignet ist sie jedoch auch nicht, da zum einen der Datenbereich nicht zusammenhängend ist, zum anderen das Anfügen mehrerer Datenelemente gleichzeitig nicht ganz einfach ist (*siehe Algorithmen zur STL in Kap. 4.6*). Fazit: Wir sollten hier selbst tätig werden.

Um problemlos Daten vor den vorhandenen Anfügen zu können, müssen wir vorne und hinten Reservepuffer in einem zusammenhängenden Speicherbereich haben, das heißt der Speicherbeginn darf (*in der Regel*) nicht mit dem Datenbeginn übereinstimmen. Dazu definieren wir ein vier–Zeiger–Modell mit Methoden zur Längenberechnung der einzelnen Bereiche. Unter Rückgriff auf die STL folgt:[2]

```
struct BufferBase {
    char * start_buf;
    char * start_dat;
    char * end_dat;
    char * end_buf;
    .. // Attribute und Methoden für Referenzmodell

    inline int bufLen()const{
        return distance(start_buf,end_buf);
    };//end function

    inline int datLen()const{...}
    inline int frontFree()const{...}
    inline int backFree()const{...}
```

[2] Natürlich ist auch eine direkte Zeigerarithmetik möglich. Die Gesamtlänge könnte auch durch `reinterpret_cast<int>(end_buf) - reinterpret_cast <int> (start_buf)` berechnet werden. Wie schon mehrfach dargelegt (*Sie können das ruhig noch einmal kontrollieren*), macht die STL aber auch nichts anderes und man geht Problemen bei einem Systemwechsel aus dem Weg.

Die Größe des Puffers und die Position der Datenzeiger kann anwendungsspezifisch projektiert werden, beispielsweise durch[3]

```
start_dat=end_dat=start_buf+datLen()/4;
```

Das Einfügen von neuen Daten vor oder hinter den vorhandenen ist einfach zu erledigen. Wenn der Platz nicht mehr ausreicht, wird zunächst der vorhandene Inhalt verschoben, und erst wenn dies nicht ausreicht, an der jeweiligen Seite eine entsprechende Reserve hinzugefügt.

```
inline void resizeFront(int len){
    if(len<=frontFree()) return;
    if(len<=frontFree()+backFree()){
        len-=frontFree();
        memmove((start_dat+len),start_dat,len);
        start_dat+=len;
        end_dat+=len;
        return;
    }//endif
    len=(len/DBufferBase_RES_SIZE+1)
                    *DBufferBase_RES_SIZE;
    char * nb=(char*)malloc(len+bufLen());
    memmove(nb+(len+frontFree()),start_dat,
                    datLen());
    end_dat=nb+(len+frontFree()+datLen());
    start_dat=nb+(len+frontFree());
    len+=bufLen();
    free(start_buf);
    start_buf=nb;
    end_buf=nb+len;
}//end function
```

Die Vorgehensweise lässt sich leicht begründen: Würde bei einem nicht ausreichenden Platz der Puffer sofort vergrößert, so hätten wir einen Speicherfresser konstruiert. Nehmen wir dazu an, die Anwendung reinitialisiert den Datenbereich in der oben vorgeschlagenen Art und die Anwendung fügt Daten von einem kByte im vorderen Bereich an. Dann ergibt sich folgender Ablauf der Speicherbelegung:

Erst bei einer Speicherbelegung > 4.000 Byte ist keine Umspeicherung mehr notwendig, und dieser Wert wird nur asymptotisch langsam angesteuert, so dass eines der Ziele unserer Maßnahmen nicht erreicht wird. Außerdem wird viermal so viel Speicher als notwendig angefordert.

Für die Verschiebung der Daten auf dem vorhandenen Platz kann die Strategie in Abhängigkeit von der Anwendung modifiziert werden. Ist beispielsweise bekannt,

[3] Hier ist die direkte Zeigerarithmetik eindeutig. Für die Verwendung von `advance(..)` besteht keine Notwendigkeit.

Pufferlänge	*Datenstart*	*zusätzlich*
1000	250	750
1750	438	562
2312	578	422
2734	683	317
...		

dass Anfügeoperationen mehrfach hintereinander an der gleichen Seite erfolgen, so kann der vorhandene Bereich bei Bedarf auch vollständig an das andere Ende verschoben werden. Wie schon häufiger erinnere ich daran, dass Sie sich vergewissern sollten, hier nicht zu kompliziert zu denken und Arbeit in eine Optimierung zu investieren, die dann eine Leistungsverbesserung von einem Prozent bewirkt.

Formal betrachten wir den Datenbereich als unstrukturierten Binäredatenpuffer . Anwendungen können dem Bereich natürlich beliebige Strukturen aufprägen, wie Parameterstrings (*dann ist der Bereich natürlich kein reiner Binärdatenpuffer mehr*), Klassen mittels spezieller Allokatoren (*siehe Kap. 4.3*) oder ASN.1–Datenbeschreibungen, die wir in einem späteren Kapitel vorstellen werden. Aber der Transporteur muss sich natürlich nicht dafür interessieren, was er transportiert.

Die Klasse `BufferBase` wird in die Trägerklasse `DBuffer` eingebunden, die eine interne Referenzzählung durchführt und über folgende Attribute und Methoden verfügt:

```
class DBuffer{
public:
    Dbuffer::DBuffer(){..}
    DBuffer::DBuffer(const DBuffer& p){..}
    Dbuffer::~DBuffer(){..}

    inline int size() const {return buf->datLen();}
    inline char     front(){..}
    inline char     back(){..}
    inline char*    begin(){..}
    inline char*    end(){..}
    inline const char* cbegin(){..}
    inline const char* cend(){..}
    inline const char* c_str(){..}
    inline DBuffer& add_back(const char c){..}
    inline DBuffer& add_back(const char* c,
                             int len){..}
    inline DBuffer& add_back(const DBuffer& db){..}
    inline DBuffer& add_front(const char c){..}
    ...
    inline DBuffer& insert(int pos,
                           const char c){..}
```

```
    ...
    inline DBuffer& erase(int pos, int len){..}
    inline DBuffer& erase_front(){..}
    inline DBuffer& erase_front(int len){..}
    inline DBuffer& erase_back(){..}
    inline DBuffer& erase_back(int len){..}
    inline DBuffer& clear(){..}
    inline DBuffer& operator=
                    (const DBuffer& db){..}
    inline char&    operator[](int i){..}

protected:
    mutable struct BufferBase * buf;
};//end class
```

Aufgabe. Die Referenzzählung mit Hilfe der Klasse `RefAllocator<..>` haben Sie ja in diesem Kapitel schon geübt. Implementieren Sie die Methoden von `DBuffer` nach diesem Konzept.

10.5 Speicherklasse für kleine Objekte

Wenn viele Objekte zur Laufzeit benötigt werden, die genaue Anzahl aber erst zur Laufzeit ermittelt werden kann, greift man im allgemeinen auf Zeigerobjekte im Heap zurück. Das Speichermanagement, das dazu vom Betriebssystem zur Verfügung gestellt wird, ist aber vorzugsweise auf größere Objekte und relativ wenig Bewegung optimiert, so dass bei kleinen Objekten mit kurzen Lebenszyklen deren Verwaltung einen erheblichen Anteil an der Laufzeit in Anspruch nehmen kann.

Nun mag der Leser zunächst auf die Idee kommen, hier mit Vektoren oder anderen Speichermodellen zu arbeiten, und meinen, das Problem damit gelöst zu haben, doch auch diese Strukturen greifen auf das normale Speichermanagement zurück. Um die Probleme zu beheben, bleibt daher nur der Griff in eine Ebene darunter und die Speicherverwaltung selbst zu übernehmen.

Die Strategie ist schnell entworfen: man hole sich vom Betriebssystem einen großen Speicherblock, der eine bestimmte Anzahl kleiner Objekte aufnehmen kann, und weise aus diesem Bereich Speicherplatz zu. Wie groß der Speicherblock werden muss, kann man mit Hilfe von Werkzeugen, die im Kapitel mit eben diesem Namen entwickelt wurden, feststellen.

Allerdings ist mit einer generellen Strategie an dieser Stelle bereits Schluss. Vieles hängt nämlich davon ab, wie im Weiteren tatsächlich mit dem Speicherplatz umgegangen wird, und eine generell beste Strategie existiert nicht. Wir entwerfen hier ein mehrschichtiges Modell, das auch für den Einsatz in Vererbungshierarchien geeignet ist.

10.5.1 Basis einer Speicherbank

In einem einzelnen Bankobjekt wird eine begrenzte Anzahl von Speicherstellen für Objekt mit konstanter Größe angelegt. Wird der Speicherplatz nicht von einem Objekt verwendet, nutzen wir ihn, um eine verlinkte Liste freier Speicherplätze zu verwalten. Wir ordnen die Datenblöcke linear auf einem Feld an und geben zu Beginn eines Feldes jeweils den Index des nächsten ungenutzten Feldes an. Da die kleinste Objektgröße ein einzelnes Byte darstellt, können nur 256 Indizes miteinander verknüpft werden, d.h. die Bank enthält maximal 256 Objekte. Ist beispielsweise die Blockgröße eines Objektes vier Byte, so ist der Inhalt der Bank nach der Initialisierung

1			
2			
3			
...			
...			
253			
254			
255			
0			

Wären beispielsweise sämtliche Datenblöcke zwischen dem zweiten und dem 252. belegt, würde in der dritten Zeile der Wert 3 gegen den Wert 252 ausgetauscht.

Diese Speicherbank kapseln wir in einer Struktur, die einen Zeiger auf den ersten freien Platz und die Anzahl der noch freien Plätze enthält.

```
struct Buffer {
    void  init(unsigned int);
    void* alloc();
    void  dealloc(void*);
    unsigned char* buffer;
    unsigned int   first_free;
    unsigned int   free;
    unsigned int   size;
};
```

Wie zu bemerken ist, ist die Struktur weder mit einem Konstruktor noch einem Destruktor versehen. Wie wir noch sehen werden, würden uns diese Standardkonstrukte einigen Ärger verursachen. Die Initialisierung erfolgt mit der Funktion `init`:

```
void Buffer::init(unsigned int siz){
    unsigned char* p;  unsigned int   i;
```

```
    buffer = (unsigned char*) malloc(256*siz);
    for(p=buffer,i=1 ; i<256 ; i++,p+=siz) *p=i;
    first_free=0; free=256; size=siz;
}//endfunction
```

Um der Bank nun den Platz für ein Objekt zu entnehmen, muss nur die Adresse mit dem Index `first_free` ausgegeben werden. Der dort hinterlegte Wert übernimmt die nächste Ausgabeposition, durch Mitzählen der ausgegebenen Speicherplätze weiß man, wann man aufhören muss.

```
void* Buffer::alloc(){
    if(free==0) return 0;
    unsigned char* p;
    p=buffer+first_free*size;
    first_free=*p;
    free--;
    return p;
}//end function
```

Die Zurückgabe ist etwas komplizierter, da man prüfen muss, ob der zurückgegebene Platz vor oder hinter der aktuellen Topposition liegt. Letzteres verlangt eine kleine Suche, um den freigewordenen Block wieder in die Liste einzuhängen.

```
void  Buffer::dealloc(void* ptr){
    unsigned char* p;
    unsigned int ofs;
    p=static_cast<unsigned char*>(ptr);
    ofs=static_cast<unsigned int>(p-buffer)/size;
    free++;
    if(ofs<first_free){
        *p=first_free;
        first_free=ofs;
    }else{
        p=buffer+first_free*size;
        while(*p<ofs) p+=size;
        *(static_cast<unsigned char*>(ptr))=*p;
        *p=ofs;
    }//endif
}//end function
```

Ab der Topposition muss die Position in der verlinkten Liste gesucht werden, an der über die freiwerdende Stelle gesprungen wird. Der alte Link wird gegen den freiwerdenden ausgetauscht, dieser übernimmt den Inhalt des alten Links.

Das Anfordern von Speicherplatz erfolgt außerordentlich effektiv in konstanter Laufzeit, die Zurückgabe ist ebenfalls in konstanter Zeit abwickelbar, wenn die Objekte in der umgekehrten Reihenfolge ihrer Ausgabe zerstört werden, besitzt aber die Laufzeitordnung $O(n)$ bei Rückgabe in umgekehrter Reihenfolge.

Die Datenstruktur enthält keinerlei Prüfmechanismen, ob mit den Zeigern verantwortlich umgegangen wird. Bei einer Geschwindigkeitsoptimierung ist dies aber auch nicht das primäre Kriterium.

10.5.2 Objekte fester Größe

Die Basis-Objekte werden in ein übergeordnetes Objekt eingepackt, dass die Speicherung beliebig vieler Objekte einer festen Größe erlaubt. Hierzu verwaltet es die notwendige Anzahl von Basisobjekten in einem Vektorcontainer.

```
class FixedSpaceAllocator {
public:
    FixedSpaceAllocator(unsigned int);

    void destroy();

    void* alloc();
    void  dealloc(void*);
private:
    vector<Buffer> v;
    unsigned char** stb, **ste, **stp;
    unsigned int dsize;
    unsigned int last_bank;
};
```

Ist die erste Bank verbraucht, wird bei Anforderung weiteren Speicherplatzes eine weitere Bank hinzugefügt. Diese Vorgehensweise führt jedoch zu einem Problem bei der Rückgabe des Speichers. Da nur ein Zeiger zurückgegeben wird, muss jede Bank einzeln überprüft werden, ob er in ihren Bereich entfällt. Wir implementieren daher noch einen Garbage-Collector, der den Speicherplatz nicht sofort zurück gibt, sondern ihn auf einem Stack zwischenspeichert und bei Bedarf wieder ausliefert. Ist dieser Speicher leer, sorgt last_bank für eine erste Suche nach Speicherplatz in der zuletzt angesprochenen Bank.

```
void* FixedSpaceAllocator::alloc(){
   void* ptr;
   if(stp!=stb)  return stp--;
   ptr=v[last_bank].alloc();
   if(ptr!=0) return ptr;
   for(last_bank=0;last_bank<v.size();last_bank++){
      ptr=v[last_bank].alloc();
      if(ptr!=0) return ptr;
   }//endfor
   v.push_back(Buffer());
   v.back().init(dsize);
   ptr=v.back().alloc();
   return ptr;
}//end function
```

```
void FixedSpaceAllocator::dealloc(void* ptr){
    vector<Buffer>::iterator it;
    unsigned int ofs;
    *(stp++)=static_cast<unsigned char*>(ptr);
    if(stp!=ste) return;
    for(stp=stb;stp!=ste;stp++){
        for(it=v.begin();it!=v.end();it++){
            ofs=static_cast<unsigned int>
                        (*stp-it->buffer)/dsize;
            if(ofs<256){
                it->dealloc(*stp);
                break;
            }//endif
        }//endfor
    }//endfor
}//end function
```

Sofern auf dem Stack oder in der zuletzt angesprochenen Bank noch Speicherplatz zur Verfügung steht, erfolgt die Zuteilung in konstanter Zeit; lediglich bei sehr vielen gleichzeitig existierenden Objekten kommt es sporadisch zu einem größeren Zeitaufwand, wenn eine neue Bank eingerichtet werden muss.

Wie bei der ersten Struktur sind auch hier keine Konstruktoren oder Destruktoren definiert. Auch diese Klasse ist eine Zwischenklasse, die jedoch mit der Funktion `destroy` den Abbauf beider Strukturen übernimmt.

```
void FixedSpaceAllocator::destroy(){
    vector<Buffer>::iterator it;
    for(it=v.begin();it!=v.end();it++)
        free(it->buffer);
    free(stb);
}
```

10.5.3 Die allgemeine Allokatorklasse

Die allgemeine Allokatorklasse muss in der Lage sein, Speicherplatz für Objekte unterschiedlicher Größen bereit zu stellen. Sie erledigt dies, indem sie für jede benötigte Größe ein Objekt des Typs `FixedSpaceAllocator` einrichtet und in einer Map verwaltet.

```
class SmallObjectAllocator {
public:
    static SmallObjectAllocator& allocator();
    void* alloc(unsigned int);
    void  dealloc(void*,unsigned int);
```

```
private:
    SmallObjectAllocator();
    ~SmallObjectAllocator();
    map<unsigned int,FixedSpaceAllocator> stores;
};
```

Damit die Speicherverwaltung eindeutig ist, wird ein Singleton dieses Objektes in der üblichen Weise als statische Variable in einer statischen Methode definiert. Allozierung und Deallozierung sind sehr einfach gestrickt:

```
void* SmallObjectAllocator::alloc
 (unsigned int size){
   map<unsigned int,FixedSpaceAllocator>::iterator
                              it;
    it=stores.find(size);
    if(it!=stores.end()){
        return it->second.alloc();
    }else{
        stores.insert(pair<unsigned int,
                           FixedSpaceAllocator>
                (size,FixedSpaceAllocator(size)));
        return stores.find(size)->second.alloc();
    }//endif
}
void  SmallObjectAllocator::dealloc
 (void* ptr,unsigned int size){
    map<unsigned int,FixedSpaceAllocator>::iterator
                              it;
    it=stores.find(size);
    if(it!=stores.end()) it->second.dealloc(ptr);
}
```

Der Destruktor sorgt für den Aufruf von `destroy` in allen Objekten der Map.

Mit diesem Objekt ist die Speicherverwaltung komplett, und diese Klasse ist auch die einzige, die tatsächlich `public` deklariert sein sollte. Die internen Strukturen lassen sich in einem Objektmodul kapseln und durch inline-Deklaration auch laufzeitschnell implementieren.

Aus Sicht der Endklasse wird auch deutlich, weshalb Konstruktoren und Destruktoren in den unteren Klassen fehlen. Wir definieren hier verschiedentlich Zeigervariablen mit vom Betriebssystem zugeordneten Speicherbereichen. Bei der Übergabe an die Container müssten diese im Falle eines normalen Konstruktor/Destruktor-Geschehens mühsam behandelt werden, während die gewählte Implementation einfach einen Standardkopierkonstruktor unterstellt, der die Objektinhalte kopiert, aber nicht in Allozierungsverfahren eingreift. Dies erledigen wir daher von einer höheren Stuffe aus auch für die unteren.

Die Speicherzuordnung nach diesem Mechanismus wird in der Regel sehr schnell funktionieren, doch man muss auch im Auge behalten, dass bei der Anforderung und der Rückgabe auch unvorhersehbar größere Verwaltungsarbeit notwendig ist. Gegebenenfalls muss man dies in Anwendungen mit engen Zeitrahmen berücksichtigen.[4]

10.5.4 Eine Basisklasse für die Allokatornutzung

Bislang haben wir allerdings nur die Verwendungsmöglichkeiten erreicht, die die C-Funktionen `malloc` und `free` zur Verfügung stellen, nicht jedoch die C++ Form mit `new` und `delete`. Da auch diese Operatoren überschreibbar sind, können wir die Speicheranforderung auf die Allokatorklasse umleiten:

```
class SmallObject {
public:
  inline void* operator new(unsigned int size){
    return
     SmallObjectAllocator::allocator().alloc(size);
  }
  inline void  operator delete(void* ptr,
                               unsigned int size){
    SmallObjectAllocator::allocator().
                          dealloc(ptr,size);
    }
};
```

Wichtig für die Gesamtfunktion ist die Eigenschaft des Operators `delete`, neben dem zu zerstörenden Zeiger auch dessen Speicherplatz als Übergabeparameter zu enthalten. Dies erlaubt die Weitergabe an die Allokatorklasse, die hiermit wiederum die zuständig Unterklasse ermitteln kann.

Um mehr muss man sich nicht kümmern. Alle von `SmallObject` erbenden Klassen, auch entfernte Verwandte, melden sich beim Aufruf des `new`-Operators mit ihrem konkreten Platzbedarf an, so dass immer der benötigte Speicherplatz geliefert werden kann.

[4] Effekte dieser Art sind u.a. verantwortlich für die Nicht-Echtzeitfähigkeit mancher Programmierumgebungen wie Java.