6 Allokatoren

In diesem Kapitel werden die Allokatoren eingehender betrachtet. Als Beispiel wird die Implementation des Standard-Allokators besprochen.

Allokatoren sind ein wesentlicher Bestandteil der STL. Kein Container kommt ohne sie aus, und jeder Konstruktor widmet ihnen einen Parameter.

Allokatoren sind – genau wie die Iteratoren – ein abstraktes Konzept. So, wie Iteratoren den Zugriff auf Container kapseln, damit sich die Algorithmen nicht um den direkten Zugriff kümmern müssen, so dienen die Allokatoren dem Kapseln des Speicherzugriffs.

Ein Container soll sich nicht selbst um die Reservierung von Speicher bemühen, sondern er verwendet einen Allokator, der eine definierte Schnittstelle besitzt.

Auf diese Weise kann die Art der Speicherverwaltung für einen Container geändert werden, ohne daß davon die Implementationsdetails des Containers betroffen sind.

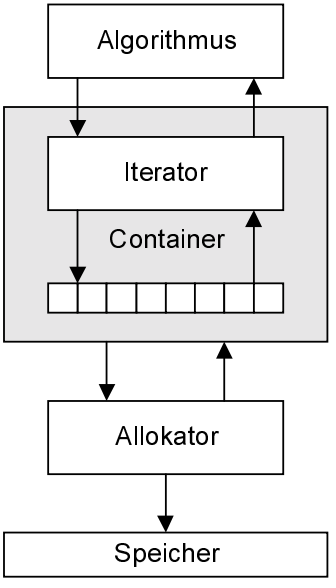

Abbildung 6.1 Der Aufbau der STL

Abbildung 6.1 zeigt die STL und die Zusammenhänge ihrer Komponenten. Es ist sehr schön zu erkennen, daß bis auf den Iterator alle Komponenten unabhängig voneinander implementiert werden können – solange die entsprechenden Schnittstellen eingehalten werden.

Lediglich der Iterator, der für den Zugriff auf die Datenelemente seines Containers dessen Implementation kennen muß, ist bis auf die Ausnahmen der unabhängigen Iteratoren[1] an seinen Container gebunden.

6.1 Der Standard-Allokator

Sehen wir uns nun den Standard-Allokator, der allen vordefinierten STL-Containern genügt, an.

Da wären als erstes das Definitionsskelett mitsamt der im Allokator definierten Datentypen:

Definition

```
template<class Typ>
class allocator
{
  public:
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef Typ* pointer;
    typedef const Typ* const_pointer;
    typedef Typ& reference;
    typedef const Typ& const_reference;
    typedef Typ value_type;
};
```

Der Allokator definiert anhand des Datentypen der zu verwaltenden Elemente einige neue Datentypen, die später von den Containern übernommen werden:

Datentyp	Beschreibung
const_pointer	Typ für konstante Zeiger.
const_reference	Typ für konstante Referenzen.

Tabelle 6.1 Die Datentypen von allocator

1 Zu diesen Iteratoren zählen Stream-oder Insert-Iteratoren.

Datentyp	Beschreibung
difference_type	Typ für die Angabe von Abständen. Normalerweise ein signed-Typ.
pointer	Zeiger-Typ
reference	Referenz-Typ
size_type	Der Typ, mit dem Größen ausgedrückt werden. Im allgemeinen ein unsigned-Typ.
value_type	Der Typ der zu verwaltenden Elemente.

Tabelle 6.1 Die Datentypen von allocator

Der Allokator teilt das Anlegen eines Elements in zwei Stufen. Die erste Stufe ist das bloße Anfordern des Speichers, den das Element flach kopiert benötigt.

Dann kann in diesem Speicherbereich ein Element konstruiert werden. Das heißt, der Konstruktor des Elements initialisiert den Speicher mit den entsprechenden Werten und fordert eventuell vom Element benötigten dynamischen Speicher an.

Die Freigabe des Speichers läuft umgekehrt. Zuerst wird das Element zerstört, indem sein Destruktor aufgerufen wird, der alle vom Element belegten Ressourcen freigeben sollte. Danach kann der Speicherbereich vom Allokator freigegeben werden.

6.1.1 Speicher reservieren und freigeben

Zum Reservieren und Freigeben von Speicher besitzt der Allokator zwei Methoden:

allocate

```
pointer allocate(size_type anz, void* info=0){
  return (static_cast<pointer>
         ( operator new(anz*sizeof (value_type)) ) );
}
```

Die Funktion reserviert Speicher für **anz** Elemente des verwalteten Typs. Der Parameter **info** hat beim Standard-Allokator keine Funktion, kann aber bei selbstimplementierten Allokatoren dazu verwendet werden, eine Information zu übermitteln, die bei der Vergabe des Speichers berücksichtigt wird.

Zum Beispiel könnte bei einem Allokator, der den Speicher in großen Blöcken verwaltet, über **info** der Wunsch geäußert werden, Speicherplatz in einem bestimmten Block zu bekommen.

Zum Reservieren wurde **operator new()** verwendet. Dieser Operator wird von **new** dazu verwendet, um uninitialisierten Speicherplatz anzufordern. Und genau der wird benötigt.

allocate reserviert den Speicher, initialisiert ihn aber nicht.

deallocate

```
void deallocate(pointer ptr, size_type anz){
  operator delete(ptr);
}
```

Die Funktion gibt den für **anz** Elemente reservierten Speicher an Adresse **ptr** frei. Als Umkehrfunktion zu **operator new()** gibt **operator delete()** den Speicher nur frei, ohne vorher das Element zu zerstören.

deallocate gibt nur Speicher frei, ohne vorher das Element zu zerstören.

6.1.2 Elemente konstruieren und zerstören

Nachdem Speicher reserviert wurde, muß er irgendwann initialisiert werden. Dies geschieht im allgemeinen durch Konstruktion eines Objektes im reservierten Speicher. Umgekehrt muß ein konstruiertes Objekt zerstört werden, bevor sein Speicher freigegeben wird.

Sollte das Objekt nicht vor der Freigabe des Speichers zerstört worden sein, werden eventuell vom Objekt belegte Ressourcen nicht freigegeben. Es entsteht ein Ressourcenleck.

Für das Konstruieren und Zerstören von Objekten stehen zwei Methoden zur Verfügung:

construct

```
void construct(pointer ptr, const Typ& obj){
  new ((void*)ptr) Typ(obj);
}
```

Initialisiert den Speicher ab Adresse **ptr** mit dem Objekt **obj**. Dazu wird der Copy-Konstruktor von **Typ** verwendet.

Wenn der Speicher bei **ptr** mit dem entsprechenden Allokator reserviert wurde, stimmt die Größe des Speicherbereichs mit der Größe eines Objektes vom Typ **Typ** überein.

construct konstruiert ein Objekt in bereits vorher reserviertem Speicher.

destroy

```
void destroy(pointer ptr){
  ptr->~Typ();
}
```

Die Methode ruft den Destruktor des an der Adresse **ptr** befindlichen Objektes auf. Der Speicher wird aber nicht freigegeben.

destroy zerstört ein Objekt, ohne den Speicher freizugeben.

6.1.3 Adressen

Es wurde eine Methode implementiert, um die Adresse eines übergebenen Objektes zu bekommen. Diese Methode – **address** – existiert für variable und konstante Allokatoren:

```
pointer address(reference r) const{
  return (&r);
}
const_pointer address(const_reference r) const{
  return (&r);
}
```

6.1.4 max_size

```
size_type max_size() const{
  size_type s=static_cast<size_type>(-1)/sizeof(Typ);
  return ((0<s)?s:1);
}
```

Die Funktion ermittelt, wie viele Elemente des zu verwaltenden Typs maximal verwaltet werden können.

Da die Basis von Größen der Datentyp **size_type** ist, können nicht mehr Bytes verwaltet werden, als die größte Zahl, die mit einem **size_type** dargestellt werden kann. Wenn diese maximale Anzahl an Bytes noch durch die Größe des zu verwaltenden Datentyps dividiert wird, erhält man den gewünschten Wert.

6.1.5 Vergleichsoperatoren

```
template<class Typ>
bool operator==(const allocator<Typ> &a1,
                const allocator<Typ> &a2){
  return(true);
}

template<class Typ>
bool operator!=(const allocator<Typ> &a1,
                const allocator<Typ> &a2){
  return(false);
}
```

Die Vergleichsoperatoren vergleichen nicht wirklich zwei Allokator-Instanzen. Da der Allokator nur aus Methoden besteht, müssen zwei Allokator-Instanzen vom selben Typ auch gleich sein. Deswegen wird vom ==-Operator **true** und vom !=-Operator **false** zurückgegeben.

6.2 allocator<void>

Für den Datentypen **void** wurde eine Spezialisierung von **allocator** implementiert:

```
template<>
class allocator<void>
{
```

```
  public:
     typedef void* pointer;
     typedef const void* const_pointer;
     typedef void value_type;
};
```

6.3 Ein eigener Allokator

Wir wollen als Anwendungsbeispiel einen eigenen Allokator implementieren. Und zwar soll dieser Allokator nicht für jedes Element einzeln Speicher anfordern, sondern einen größeren Speicherblock, der dann happenweise vergeben wird.

Es muß nur eine Lösung für den Fall gefunden werden, daß der Speicherblock voll ist. Einen größeren Block anfordern und den Inhalt des alten kopieren wäre auf der einen Seite äußerst ineffizient und auf der anderen Seite nicht durchführbar, weil die Verweise auf die Speicherbereiche, die vergeben wurden, nicht nachträglich verändert werden können.

Wir implementieren daher eine simple, einfach verkettete Liste, mit der beliebig viele Speicherblöcke aneinander gehängt werden können.

Abbildung 6.2 zeigt die Strategie.

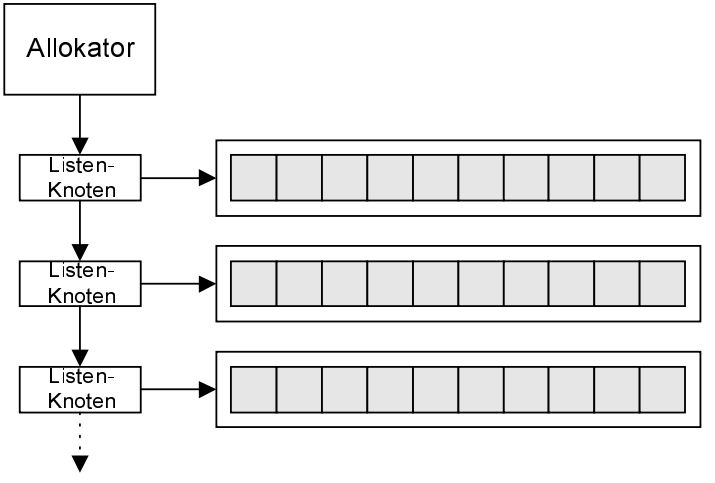

Abbildung 6.2 Die Strategie unseres eigenen Allokators

Diese Lösung besitzt einige Hürden, die nicht unbedingt offensichtlich sind, die aber genommen werden müssen.

- Es muß eine Möglichkeit der Markierung gefunden werden, denn sollte ein Teil eines Speicherblockes zurückgegeben werden, dann soll dieser bei einer späteren Allokation wieder zur Verfügung stehen.
- Wenn ein Speicherbereich freigegeben wird, dann wird dem Allokator lediglich ein Zeiger auf diesen Bereich übergeben. Es muß dann aber noch ermittelt werden, zu welchem unserer Speicherblöcke dieser Bereich gehört.
- Es ist zu berücksichtigen, daß ein Speicherbereich den Zuständigkeitsbereich wechseln kann. Sollte zum Beispiel bei einer Liste ein Knoten durch Splicing einer anderen Liste zugewiesen werden, dann liegt der Speicherbereich immer noch im Allokator der alten Liste, obwohl die neue Liste versuchen wird, mit ihrem Allokator diesen Bereich freizugeben. Dieser Allokator findet den Speicherbereich natürlich nicht in seiner Liste.

Den letzten Punkt lösen wir geschickt damit, daß wir den Zeiger auf die Liste als statisch markieren. Damit verwalten Allokatoren, die Elemente desselben Typs speichern, auch dieselbe Speicherliste.

Um den Speicherblock zu finden, in dem ein freigegebener Speicherbereich liegt, muß notgedrungen die Liste durchforstet werden. Das ist nicht sonderlich elegant, könnte aber durch Wahl einer anderen Datenstruktur verbessert werden.

Um belegte und freie Teile eines Speicherblocks zu markieren, speichern wir mit jedem Listenelement einen Wert, dessen Bits angeben, welcher Bereich frei, und welcher belegt ist.

Sehen wir uns das Definitionsskelett an:

Definition

```
template<class Typ>
class block_allocator
{
  public:

    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef Typ* pointer;
```

```
    typedef const Typ* const_pointer;
    typedef Typ& reference;
    typedef const Typ& const_reference;
    typedef Typ value_type;

    typedef block_allocator<Typ> MeinTyp;

  private:
    struct Knoten {
      Knoten *nach;
      size_type bmaske;
      char *speicher;
    };

    static size_type maxelanz,voll,insanz;
    static Knoten *anfang;
};
```

Im **private**-Bereich ist die Knoten-Struktur definiert, die einen Verweis auf den Nachfolgeknoten, eine Maske für die Speicherbelegung und einen Zeiger auf den speicherbereich besitzt.

Die Attribute des Allokators stehen für die Anzahl der Elemente eines Speicherblocks (**maxelanz**), den Wert einer Bitmaske für einen vollen Bereich (**voll**), die Anzahl der aktiven Instanzen des entsprechenden Allokator-Typs (**insanz**) und einen Verweis auf den ersten Listenknoten (**anfang**).

Das Wissen um die Anzahl der aktiven Instanzen ist wichtig, weil nur bei der Destruktion der letzten Instanz die Speicherbereiche freigegeben werden dürfen.

Die Konstruktoren bedienen sich einer Hilfsfunktion:

Konstruktoren

```
block_allocator() {
  Construct();
}

block_allocator(const MeinTyp &a){
  Construct();
}
```

Die Hilfsfunktion **Construct** berechnet für den Fall der Erstinstanz die notwendigen Werte:

Construct

```
void Construct() {
  if(!insanz){
    size_type bits=voll=static_cast<size_type>(-1);
    maxelanz=0;
    for(;bits;++maxelanz, bits>>=1);
  }
  insanz++;
}
```

Dabei wird der größtmögliche Wert ermittelt (wie bei **max_size**) und die Anzahl der Bits gezählt. Diesen Wert benutzen wir als Anzahl der Elemente pro Speicherbereich.

Destruktor

Der Destruktor löscht für den Fall, daß es sich um die letzte Instanz handelt, die Liste:

```
~block_allocator() {
  if(!(--insanz)) {
    while(anfang)
      AnfangLoeschen();
  }
}
```

NeuerBlock

Eine weitere Hilfsfunktion ist **NeuerBlock**, die einen neuen Speicherbereich anlegt und die Liste einfügt.

```
void NeuerBlock() {
  Knoten *tmp=anfang;
  anfang=new Knoten();
  anfang->nach=tmp;
  anfang->bmaske=0;
  anfang->speicher=reinterpret_cast<char*>
          (operator new(maxelanz*sizeof(value_type)));
    }
```

Der Nachteil einer einfach verketteten Liste liegt im Unwissen um den Vorgänger eines Knotens. Die einzige Ausnahme ist der erste Knoten, denn auf ihn zeigt **anfang**. Deswegen können wir auf einfache Weise einen Knoten am Anfang der Liste einfügen. Da die Liste keine Sortierung besitzt, spielt die Einfügeposition keine Rolle.

Es wird Speicher für **maxelanz** Elemente der Größe **sizeof(value_type)** reserviert.

ZumAnfang

Wenn in einem Speicherblock ein Speicherbereich freigegeben wurde, dann könnte dieser bei der nächsten Reservierung direkt verwendet werden. Deshalb verschieben wir den Speicherblock an den Anfang der Liste. Die Funktion **ZumAnfang** erledigt dies:

```
void ZumAnfang( Knoten *k) {
  swap(k->bmaske,anfang->bmaske);
  swap(k->speicher,anfang->speicher);
}
```

Ein Verschieben des Knotens wäre aufwendig, weil wir zuerst den Vorgänger ermitteln müßten, was nur durch Traversieren der Liste möglich ist. Wir tauschen daher einfach die Bitmasken und die Verweise auf die Speicherblöcke aus.

AnfangLoeschen

Wenn ein Speicherblock komplett leer ist, dann kann er freigegeben werden. Am einfachsten ist ein Löschen am Anfang der Liste:

```
void AnfangLoeschen() {
  Knoten *tmp=anfang->nach;
  delete(anfang->speicher);
  delete(anfang);
  anfang=tmp;
}
```

Da bei der Freigabe eines Speicherbereiches der Knoten mitsamt Speicherblock an den Anfang der Liste gesetzt wird (**ZumAnfang**), wird auch ein Block, der durch eine Freigabe komplett geleert wurde, am Anfang der Liste stehen.

allocate

Kommen wir nun zu einem Herzstück des Allokators, der allocate-Funktion:

```
pointer allocate(size_type anz, void *info){
  if(anz!=1)
    return(static_cast<pointer>(0));
```

Sollte über **allocate** versucht werden, gleichzeitig Speicher für mehr als ein Element zu reservieren, dann bricht die Funktion ab, da dies nicht unterstützt wird[2].

```
Knoten *akt=anfang;
while((akt) && (akt->bmaske==voll))
  akt=akt->nach;
```

Die Schleife läuft so lange, bis entweder das Ende der Schleife oder ein Block mit freiem Bereich gefunden wurde.

```
if(!akt){
  NeuerBlock();
  akt=anfang;
}
```

Sollte das Ende der Liste erreicht worden sein, war kein Speicherbereich mehr frei. Es muß ein komplett neuer Block angelegt werden.

```
size_type bit=1;
char *mem=akt->speicher;
while(akt->bmaske&bit) {
  bit<<=1;
  mem+=(sizeof(value_type));
}
```

Die Schleife bestimmt anhand der Bitmaske den ersten freien Speicherbereich im Block.

```
  akt->bmaske|=bit;
  return(reinterpret_cast<pointer>(mem));
}
```

Das Bit für den neu reservierten Bereich wird gesetzt und ein Zeiger auf den Bereich zurückgegeben.

construct

Die Konstruktion eines Elementes in einem reservierten Bereich ist wieder trivial:

```
void construct(pointer ptr, const Typ& obj){
  new ((void*)ptr) Typ(obj);
}
```

2 Die Unterstützung einer Reservierung für mehrere Elemente kann mit wenig Aufwand nachträglich implementiert werden. Wir gehen hier jedoch nicht darauf ein.

Die Zerstörung des Elementes kann auch kommentarlos bleiben:

destroy

```
void destroy(pointer ptr){
  ptr->~Typ();
}
```

Interessant wird wieder die Funktion **deallocate**, die einen Speicherbereich freigibt:

deallocate

```
void deallocate(pointer ptr, size_type anz){
  Knoten *akt=anfang;
  while(akt&&(( reinterpret_cast<char*>(ptr) <
    akt->speicher)||((( reinterpret_cast<char*>(ptr) -
    akt->speicher)/sizeof(value_type))>=maxelanz)))
        akt=akt->nach;
```

Der freizugebende Bereich kann nur in einem Block liegen, dessen Adresse kleiner oder gleich der Adresse des Bereiches ist.

Andererseits passen nur **maxelanz** Elemente in einen Block, weswegen die Adresse des freizugebenden Bereiches außerhalb des Blockes liegen muß, wenn **(Speicherbereichadresse – Speicherblockadresse)/sizeof(value_type)** größer oder gleich **maxelanz** ist.

```
      akt->bmaske-=1<<((reinterpret_cast<char*>(ptr) - akt-
>speicher)/sizeof(value_type));
      ZumAnfang(akt);
      if(!anfang->bmaske)
        AnfangLoeschen();
    }
```

Nachdem der freizugebende Bereich in der Bitmaske als unbelegt gekennzeichnet wurde, wird der Bereich an den Anfang der Liste gesetzt. Anschließend wird überprüft, ob der Bereich komplett leer ist und gegebenenfalls freigegeben.

Zum Schluß darf nicht vergessen werden, die statischen Attribute außerhalb des Templates zu initialisieren:

Initialisierung der statischen Attribute

```
template<class Typ>
block_allocator<Typ>::size_type
block_allocator<Typ>::maxelanz=0;
```

```
template<class Typ>
block_allocator<Typ>::size_type
block_allocator<Typ>::voll=0;

template<class Typ>
block_allocator<Typ>::size_type
block_allocator<Typ>::insanz=0;
```

Alle anderen Methoden sind mit dem Standard-Allokator identisch und werden hier nicht aufgeführt.

Der hier entwickelte Allokator dient hauptsächlich der Demonstration einer eigenen Implementation. Wenn er auch für kleine Mengen von kleinen Elementen recht effizient seinen Dienst versieht, ist er doch für eine uneingeschränkte Nutzung nicht zu empfehlen.

Den Quellcode des **block_allocator**-Templates finden Sie auf der CD unter \BUCH\KAP06\PRG01.CPP.