Kapitel 6: Dynamische Speicherverwaltung

Dieses Kapitel beschreibt die Operatoren `new` und `delete`, mit denen dynamisch Speicher reserviert und freigegeben wird. Außerdem wird im Detail eingegangen auf:

- die Implementierung von Klassen mit dynamischen Elementen
- die Fehlerbehandlung mit New-Handlern

6.1 Die neuen Operatoren new und delete

Beispielaufruf der Funktion malloc()

```
#include <stdlib.h>

long *ptr;

ptr = (long *)malloc( sizeof(long));

if( ptr == NULL)
{
   // Fehler: Nicht genug Speicher verfuegbar.
}
```

Beispielaufruf des Operators new

```
long *ptr;

ptr = new long;   // Falls nicht genug Speicherplatz vorhanden
                  // ist, wird ein New-Handler aufgerufen, der
                  // die Fehlersituation zentral behandelt.
                  // Der standardmaessig installierte New-Handler
                  // loest eine Exception aus.
```

Hinweis

1. Eine Exception zeigt eine Fehlersituation an. Zur Behandlung des Fehlers kann die Exception vom Programm aufgefangen werden. Eine nicht aufgefangene Exception bewirkt, daß das Programm beendet wird. Mehr dazu in Kapitel 16, »Exception-Handling«.
2. Ältere Compiler setzen bei der dynamischen Speicherverwaltung kein Exception-Handling ein. Im Fehlerfall liefert dann `new` den NULL-Zeiger zurück.

Speicherverwaltung mit new und delete

Mit der dynamischen Speicherverwaltung können Programme auf einen großen freien Speicherbereich zugreifen, den sogenannten *Heap* oder *Freispeicher*. Je nach Betriebssystem und dessen Konfigurierung, kann die Größe des Freispeichers durch Auslagerung (engl. *swapping*) die Größe des freien Speichers auf der Festplatte annehmen.

Der von einem Programm dynamisch reservierte Speicher läßt sich ständig dem aktuellen Bedarf anpassen. Das bringt ein hohes Maß an Flexibilität mit sich, weshalb die dynamische Speicherverwaltung oft eingesetzt wird.

In C erfolgt die dynamische Speicherverwaltung mit Hilfe der Funktionen `malloc()`, `free()` und anderen. Zwar sind diese Funktionen weiterhin verfügbar, doch genügt es in C++ nicht, nur Speicherplatz zu reservieren und wieder freizugeben. Für Objekte müssen auch die entsprechenden Konstruktoren und Destruktoren aufgerufen werden. Damit das möglich ist, wurden die Operatoren `new` und `delete` eingeführt.

Die nebenstehenden Beispielaufrufe zeigen, wie im Unterschied zur traditionellen Speicherreservierung mit `malloc()` dieselbe Aufgabe mit `new` gelöst wird.

Vorteile

Aus der Tatsache, daß `new` und `delete` Operatoren sind, ergeben sich verschiedene Vorteile:

- Dem Operator `new` kann als Operand direkt der Typ des anzulegenden Objekts angegeben werden. Damit ist keine `sizeof`-Angabe notwendig.
- Da der Compiler den angegebenen Typ kennt, wird das Objekt korrekt über einen Konstruktor erzeugt. Insbesondere kann auch ein dynamisch erzeugtes Objekt initialisiert werden.
- `new` liefert immer einen Zeiger mit dem richtigen Typ zurück. Somit ist keine `cast`-Konstruktion notwendig.
- Der Operator `delete` ruft für Objekte den Destruktor auf. Objekte werden also wieder »ordnungsgemäß« zerstört.
- Falls nicht genug Speicherplatz vorhanden ist, wird ein »New-Handler« aufgerufen. Das ist eine Funktion, welche die Fehlersituation zentral behandelt. Damit entfällt die Notwendigkeit, bei jedem Reservieren vo Speicher einen Rückgabewert abzufragen.

Es kann auch ein eigener New-Handler installiert werden.

- Da `new` und `delete` Operatoren sind, können sie überladen werden. So ist für eigene Klassen eine Optimierung der Speicherverwaltung möglich.

Mit dem Einsatz von `new` und `delete` ist die dynamische Speicherreservierung flexibler handhabbar und sicherer geworden.

6.2 Speicherreservierung für Standard-Datentypen

Beispielaufrufe von new

```
// ------------------------------------------------------------
// Neue Objekte vom Typ int und float
// ------------------------------------------------------------

   int *ptrInt;
   ptrInt = new int;             // Ohne Initialisierung.
   *ptrInt = 7;                  // Wert 7 zuweisen.

   float *ptrFloat, x = 10.5;
   ptrFloat = new float(x);      // Mit Initialisierung.

   ptrFloat = new int(10);       // Fehler!
                                 // Compiler passt auf.


// Neues Objekt vom Typ int* (Zeiger auf int)

   int **ptrPtrInt, intVar;
   ptrPtrInt = new int *;        // Auch moeglich: new (int *)
                                 // ohne Initialisierung.
   *ptrPtrInt = &intVar;         // Eine Adresse zuweisen.
   **ptrPtrInt = 77;             // intVar = 77;


// Referenzen sind keine Objekte, koennen also auch
// nicht mit new erzeugt werden.
// Aber Referenzen auf ein mit new erzeugtes Objekt
// sind moeglich.

   int& refInt = *(new int);     // refInt ist der Name
                                 // der neuen int-Variablen.
   refInt = 5;
```

Der Operator new

Der Operator `new` reserviert dynamisch Speicherplatz für Objekte beliebigen Typs, insbesondere auch für Objekte mit einem elementaren Datentyp, wie z. B. `char` oder `double`. Diese Situation wollen wir zunächst betrachten.

`new` ist ein unärer Operator, der einen Datentyp als Operand erwartet. Im einfachsten Fall hat ein Aufruf von `new` die folgende Form:

Syntax: `Zeiger_auf_Typ = new Typ;`

Der Operator `new` legt ein Objekt des angegebenen Typs an und liefert seine Adresse. Diese wird gewöhnlich einer passenden Zeigervariablen (hier `Zeiger_auf_Typ`) zugewiesen.

Beispiel:
```
double *px;
px = new double;
```

Hier wird Speicherplatz für ein Objekt vom Typ `double` reserviert (also 8 Byte) und seine Adresse dem Zeiger `px` zugewiesen. Somit ist `*px` das neue `double`-Objekt.

Initialisierung

Für Datentypen, die keine Klassen sind, ist der neue Speicherplatz nach dem obigen Aufruf von `new` nicht initialisiert. Zur Initialisierung kann aber zusätzlich zum Typ ein Anfangswert in Klammern angegeben werden.

Beispiel: `px = new double(0.07);`

Hier erhält das neue `double`-Objekt `*px` den Anfangswert `0.07`.

Der Operator new [ ]

Mit dem Operator `new[]` wird Speicherplatz für Vektoren dynamisch reserviert. Neben dem Typ der Vektorelemente ist auch die Anzahl der Vektorelemente anzugeben.

Syntax: `Zeiger_auf_Typ = new Typ[anzahl];`

`Zeiger_auf_Typ` adressiert dann das erste von insgesamt `anzahl` Vektorelementen. Die Angabe von Initialisierungswerten ist nicht möglich.

Beispiel:
```
int *pv, anzahl = 100;
pv = new int[anzahl];
```

Mit dieser Anweisung wird der Speicherplatz für 100 Vektorelemente vom Typ `int` reserviert, nämlich:

```
pv[0] ≡ *pv,  pv[1] ≡ *(pv+1), . . . ,  pv[99] ≡ *(pv+99)
```

6.3 Speicher freigeben

Beispielprogramm

```
// ------------------------------------------------------
//  Mit dynamisch reserviertem Speicherplatz arbeiten.
// ------------------------------------------------------

#include <iostream>
using namespace std;

int main()
{
    double x, *zahl;
    unsigned int max, anzahl = 0;

    cout << "Wieviele Zahlen wollen Sie maximal eingeben:  ";
    cin  >> max;

    // ----  Speicher reservieren:
    zahl = new double[max];

    // ----  Mit dem dynamischen Vektor arbeiten:
    cout << "Zahlen eingeben: (Ende mit einem Buchstaben)\n";
    int i;                    // Bis zu max Zahlen einlesen:
    for( i = 0; i < max  &&  cin >> x; ++i )
      zahl[i] = x;
    anzahl = i;

    // und etwas damit tun, z.B.
    cout << "Die eingelesenen Zahlen:  \n";
    for( i = 0; i < anzahl; ++i )   // Zahlen wieder ausgeben:
    {
       cout.width(10);              // mit Feldbreite 10
       cout << zahl[i];
    }

    // Speicher wieder freigeben:
    delete [] zahl;

    return 0;
}
```

Der Operator delete

Sobald dynamisch reservierter Speicher nicht mehr benutzt wird, sollte er wieder freigegeben werden. Dann steht er späteren Aufrufen von `new` wieder zur Verfügung. Wird der Speicherplatz nicht explizit freigegeben, so bleibt er bis zum Programmende reserviert.

Die Freigabe eines mit `new` reservierten Speicherplatzes geschieht mit dem unären Operator `delete`. Ein Aufruf von `delete` für Objekte, die keine Vektoren sind, hat folgende Form:

Syntax: `delete Zeiger_auf_Typ;`

Der Operand `Zeiger_auf_Typ` ist ein Zeiger auf das Objekt, das zuvor mit `new` dynamisch erzeugt wurde! Ein `delete`-Ausdruck hat den Typ `void`.

Beispiel:
```
double *px = new double(2.3);
. . .
delete px;
```

Der Operator delete [ ]

Ein Zeiger, der von `new` zurückgegeben wurde, kann auf ein einzelnes Objekt zeigen oder auf das erste Element eines Vektors. Der Programmierer muß selbst auf die richtige Verwendung des Zeigers achten. Das gilt auch für die Freigabe des Speichers.

Wurde mit dem Operator `new[]` Speicherplatz für einen Vektor reserviert, so ist zur Freigabe des Speichers der Operator `delete[]` aufzurufen. Hierbei wird durch die Angabe der Vektorklammern dem Compiler mitgeteilt, daß nicht nur ein einzelnes Objekt, sondern ein ganzer Vektor freigegeben werden soll.

Beispiel:
```
int *pv = new int[100];
. . .
delete [ ] pv;          // pv zeigt auf einen Vektor.
```

Die Unterscheidung zwischen `delete` und `delete[]` ist wichtig, wenn Speicherplatz für Objekte einer Klasse freigegeben wird: Bei Vektoren muß für jedes Vektorelement der Destruktor aufgerufen werden.

Richtiger Aufruf

Die falsche Verwendung von `delete` bzw. `delete[]` kann fatale Folgen haben. Es ist folgendes zu beachten:

- `delete` darf nicht zur Freigabe von statisch reserviertem Speicher verwendet werden.
- `delete` darf nicht zur Freigabe von Speicher aufgerufen werden, der mit `malloc()` reserviert wurde.
- `delete` darf nicht zweimal für dasselbe Objekt aufgerufen werden.

Es ist aber erlaubt und sicher, wenn `delete` mit dem NULL-Zeiger aufgerufen wird. In dem Fall kehrt `delete` sofort zurück. Beim »Aufräumen« muß dann nicht extra getestet werden, ob ein NULL-Zeiger vorliegt.

6.4 Speicherverwaltung für Klassen

Beispielprogramm

```
// ------------------------------------------------
// new und delete fuer Objekte vom Typ Artikel
// ------------------------------------------------

#include "artikel.h"

#include <iostream>
using namespace std;

int main()
{
  // Dynamisch ein Artikel-Objekt anlegen
  Artikel *einArtikel;
  einArtikel = new Artikel("Radio", 1234L, 198.98);
  einArtikel->print();

  // Dynamisch einen Vektor mit 3 Artikel-Objekten
  // anlegen. Default-Konstruktor ist notwendig.
  const int anzahl = 3;
  Artikel *elektroTab = new Artikel[anzahl];

  elektroTab[0] = *einArtikel;
  delete einArtikel;       // Wird nicht mehr benoetigt

  elektroTab[1] = Artikel("CD-Player", 2345L, 333.33);
  elektroTab[2] = Artikel("Fernseher", 3456L);

  elektroTab[2].setPreis(2222.22);

  cout << "\nDie Artikel in der Tabelle:\n\n";
  for( int i = 0; i < anzahl;  ++i)
  {
    elektroTab[i].print();
  }

  delete [] elektroTab;

  return 0;
}
```

Wie schon erwähnt, sind die Operatoren `new` und `delete` entwickelt worden, um auch Instanzen einer Klasse dynamisch zu erzeugen. Dazu genügt es nämlich nicht nur, den notwendigen Speicherplatz zu reservieren. Es muß auch ein geeigneter Konstruktor ausgeführt werden. Vor der Freigabe des Speicherplatzes sind darüber hinaus »Aufräumarbeiten« mit Hilfe des Destruktors vorzunehmen. Dies ist bei den Operatoren `new` und `delete` sichergestellt.

new für Klassen

Ein Aufruf von `new` für Klassen hat im Prinzip dieselbe Syntax wie ein Aufruf für Standard-Typen, also

```
new Typ
```

oder

```
new Typ( Initialisierungsliste )
```

`Typ` ist jetzt die Klasse, für die ein Objekt erzeugt wird.

Im ersten Fall wird für das neue Objekt der Default-Konstruktor aufgerufen. Im zweiten Fall sind in der Initialisierungsliste Werte anzugeben, die einem entsprechenden Konstruktor als Argumente übergeben werden können. Findet der Compiler keinen passenden Konstruktor, so quittiert er dies mit einer Fehlermeldung.

Beispiel: `new Datum`

Mit diesem Ausdruck wird der Speicherplatz für ein Objekt der Klasse `Datum` angefordert. Sofern der Speicherplatz verfügbar ist, wird der Default-Konstruktor von `Datum` aufgerufen und ein Zeiger auf das neue Objekt zurückgegeben. Mit der Anweisung

Beispiel: `Datum *ptrDate = new Datum(1,1,2000);`

erhält der Zeiger `ptrDate` die Adresse eines Objekts der Klasse `Datum`. Das Objekt `*ptrDate` wird zugleich mit den angegebenen Werten initialisiert. Über `ptrDate->` können dann alle `public`-Elemente von `Datum` angesprochen werden.

Der Speicher für einen Vektor von Objekten wird mit dem Operator `new[]` reserviert. Für jedes einzelne Vektorelement wird dann der Default-Konstruktor ausgeführt. Die Angabe einer Initialisierungsliste ist nicht möglich.

Speicher freigeben

Bei der Freigabe des Speicherplatzes sorgt der Operator `delete` dafür, daß das Objekt »ordnungsgemäß« zerstört wird. Das heißt, er ruft implizit den Destruktor des Objekts auf.

Beispiel: `delete ptrDate;  // Objekt *ptrDate entfernen`

Wurde für einen Vektor Speicherplatz dynamisch reserviert, so ist dieser mit dem Operator `delete[]` freizugeben. Für jedes Vektorelement wird dann der Destruktor aufgerufen.

Beispiel: `delete[] elektroTab;  // Vektor elektroTab entfernen`

6.5 Fehlerbehandlung

Beispielprogramm

```
// ------------------------------------------------------
// Installation eines eigenen New-Handlers.
// ------------------------------------------------------

#include <stdlib.h>                     // Fuer exit()
#include <iostream>
#include <new>                          // Fuer den New-Handler
using namespace std;

void my_new_handler(void);              // Prototyp

int main()
{
   // Eigenen New-Handler installieren
   set_new_handler( my_new_handler);

   // Immer wieder Speicherplatz anfordern
   unsigned int block = 10000;
   char *p;
   for( long i = 1;  ; ++i)             // "Endlosschleife"
   {
     p = new char[block];
     cout << "Neue Speicheradresse: " << (void*)p
          << "\nBisher belegter Speicher: " << i * block
          << " Bytes" << endl;
   }
   cerr << "Diese Meldung sollten Sie nicht sehen!\n";
   return 0;
}

void my_new_handler(void)              // Eigener New-Handler
{
   // Meldung ueber Standard-Fehlerkanal ausgeben
   cerr << "\nNicht genug freier Speicher vorhanden!!"
           "\nProgramm wird beendet!!\n";

   // Programm mit Exit-Status != 0 beenden
   exit(1);
}
```

Hinweis New-Handler gemäß dem ANSI-Standard werden noch nicht von allen Compilern unterstützt. Im Fehlerfall liefert dann `new` den Wert NULL.

Aktionen im Fehlerfall

Ein Programm, das dynamisch Speicher reserviert, muß damit rechnen, daß irgendwann nicht mehr genug Freispeicher zur Verfügung steht. In einem C-Programm wird diese Situation abgefangen, indem bei *jeder* Speicheranforderung der Return-Wert auf `NULL` getestet wird. Falls nicht genug Speicher vorhanden ist, muß eine geeignete Aktion ausgeführt werden, wie z. B.:

- Senden einer Fehlermeldung an die Standard-Fehlerausgabe
- Sichern von Daten und/oder das frühzeitige Beenden des Programms
- Freigeben von nicht mehr benötigtem Speicher und ein erneuter Versuch, den erforderlichen Speicher zu reservieren

New-Handler

C++ bietet einen bequemen Mechanismus, Aktionen festzulegen, die nach einem erfolglosen Aufruf von `new` ausgeführt werden sollen. Dies geschieht durch die Definition einer Funktion, dem sogenannten *New-Handler*. Kann der Operator `new` den angeforderten Speicherplatz nicht reservieren, so wird automatisch der New-Handler aufgerufen.

Der standardmäßig installierte New-Handler löst eine Exception aus. Wird diese nicht aufgefangen, so endet das Programm.

Der New-Handler kann als globale Funktion selbst definiert werden. Diese Funktion besitzt keinen Parameter und auch keinen Return-Wert. Vor dem ersten Aufruf von `new` muß der New-Handler mit der Funktion `set_new_handler()` installiert werden.

Die Funktion `set_new_handler()` ist in der Header-Datei `new.h` deklariert und erwartet als Argument die Adresse des New-Handlers, d. h. den Funktionsnamen.

Beispiel: `set_new_handler(my_handler);`

Damit ersetzt die selbstdefinierte Funktion `my_handler` den Default-New-Handler.

Temporäre New-Handler

Die Funktion `set_new_handler()` liefert die Adresse eines zuvor installierten New-Handlers. Dies kann dazu benutzt werden, in bestimmten Programmabschnitten verschiedene New-Handler zu installieren.

```
void (*ex_handler)();   // Zeiger auf alten New-Handler
ex_handler = set_new_handler(my_handler);
. . .                   // Mit my_handler() arbeiten,
. . .                   // anschliessend wieder
                        // ex_handler()installieren:
set_new_handler(ex_handler);
```

6.6 Dynamische Elemente (1)

Die Datenelemente der Klasse String

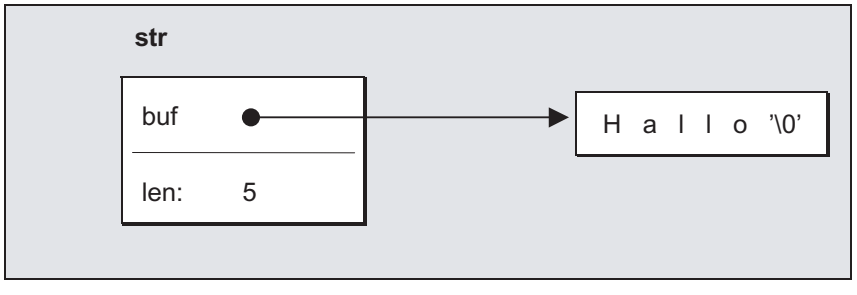
```
// ------------------------------------------------
//  Eine String-Klasse (Version 1)
// ------------------------------------------------

class String
{
  private:
    // private Datenelemente
    char *buf;
    int len;

public:
    // Hier folgen die Deklarationen der
    // oeffentlichen Methoden
    . . .
};
```

Ein Objekt der Klasse String im Speicher

```
String str("Hallo");
```



Die Flexibilität der dynamischen Speicherverwaltung kann auch für Klassen genutzt werden, um Datenelemente »variabler Länge« zu bilden: Die Klasse besitzt dann einen Zeiger auf dynamisch reservierten Speicher, der die eigentlichen Daten enthält. Solche Datenelemente heißen auch *dynamische Elemente*.

Mit der Verwendung dynamischer Elemente sind jedoch einige Besonderheiten verbunden, die anhand des Beispiels einer Klasse `String` erläutert werden sollen.

C-Strings

In C gibt es keinen elementaren Datentyp für Strings. Vielmehr werden Strings wie Zeichenfolgen behandelt, die in `char`-Vektoren gespeichert sind. Ein C-String ist daher ein Zeiger auf das erste Element eines `char`-Vektors.

Bei einer Zuweisung oder einem Vergleich von C-Strings werden nur die beteiligten Zeiger, nicht die Zeichen zugewiesen oder verglichen. Deshalb müssen spezielle Funktionen wie etwa `strcpy()` und `strcmp()` aufgerufen werden, die Strings zeichenweise verarbeiten.

Beispiel:
```
char str[20];
strcpy(str, "Ein Beispiel");
```

Hierbei muß der Programmierer sicherstellen, daß der Zielvektor groß genug ist.

Anforderungen an eine Klasse String

Das Ziel beim Entwurf einer Klasse `String` ist es, Strings so einfach wie elementare Datentypen handhaben zu können. Es sollte beispielsweise anstelle eines Aufrufs der Standard-Funktion `strcpy()` eine einfache Zuweisung möglich sein:

Beispiel:
```
String str;
str = "Ein Beispiel";
```

Nicht der Programmierer, sondern das Objekt selbst sollte garantieren, daß der durch den Zeiger `str` adressierte Speicherbereich genügend groß ist, um den Quellstring aufzunehmen.

Datenelemente der Klasse String

Jedes Objekt der Klasse `String` stellt eine Zeichenfolge dar. Da Strings unterschiedlich lang sein können, wäre es nicht effizient, Zeichenfolgen fester Länge zu verwenden. Deshalb enthält ein Objekt ein dynamisches Element, also einen Zeiger auf die eigentliche Zeichenfolge. Diese ist ein gewöhnlicher `'\0'`-terminierter C-String. Der dafür tatsächlich benötigte Speicherplatz wird dynamisch mit `new` und `delete` verwaltet. Außerdem wird noch ein Datenelement für die oft benötigte Länge des Strings bereitgestellt.

Die Standard-Bibliothek stellt eine wesentlich komplexere Klasse `string` zur Darstellung von Strings zur Verfügung. Die hier schrittweise entwickelte Klasse `String` zeigt das prinzipielle Vorgehen bei der Implementierung einer solchen Klasse.