Kapitel 6

Speicherverwaltung

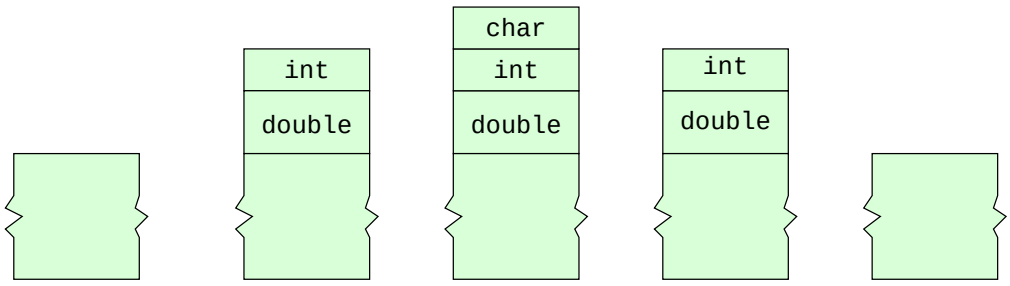
Wie bereits mehrfach angekündigt, hat man mit C++ eine größere Kontrolle über die Speicherverwaltung – aber damit auch eine größere Verantwortung. Man unterscheidet in C++ zwei Arten von Speicher: den Stapel (Stack) und den Haufen (Heap). Während ersterer einfach zu handhaben ist, schließlich ist ein Stapel ein geordnetes Ding, ist es schon schwieriger einen Haufen von Speicherbereichen zu verwalten. Im Zuge dessen werden wir auch mit den vielgefürchteten Zeigern (Pointern) in Berührung kommen. Es hat sich aber gezeigt, dass, wenn man diese ausschließlich dann einsetzt, wenn es nötig ist, und die Datentypen schön kapselt, man auch mit Zeigern gut auskommen kann, ohne sich ständig an ihren Spitzen zu stechen.

6.1 Stapel

Bevor wir uns in die Tiefen der Speicherverwaltung wagen, wollen wir uns dem angenehmeren Teil widmen, mit dem man es zum Glück in den meisten Fällen zu tun hat: dem Stapel. Sobald eine Variable definiert wird, wird vom System dafür Speicher angefordert:

```
void Person::erhoeheEinkommen(double Wert, int )
{
  char c;
      ⋮
}
```

Man kann sich das so vorstellen, dass der angeforderte Speicher wie ein Stapel verwaltet wird. Dabei wird er in der Reihenfolge wieder zurückgegeben, wie er angefordert wurde.

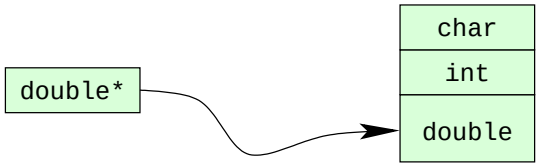


Genauer gesagt wird der Speicher wieder freigegeben, sobald der Gültigkeitsbereich (Scope) der Variable verlassen wird. Im Unterschied zu Java wird der Speicher für die Variable *sofort* freigegeben und nicht zu einem undefinierten Zeitpunkt später. Insbesondere ist es daher nicht nötig, dass eine Garbagge-Collection ausgeführt wird.

Falls vorhanden, wird automatisch vor der Freigabe des Speichers der Destruktor ausgeführt. Die Rolle des Destruktors rückt daher eher in den Mittelpunkt, da genau bekannt ist, wann er ausgeführt wird.

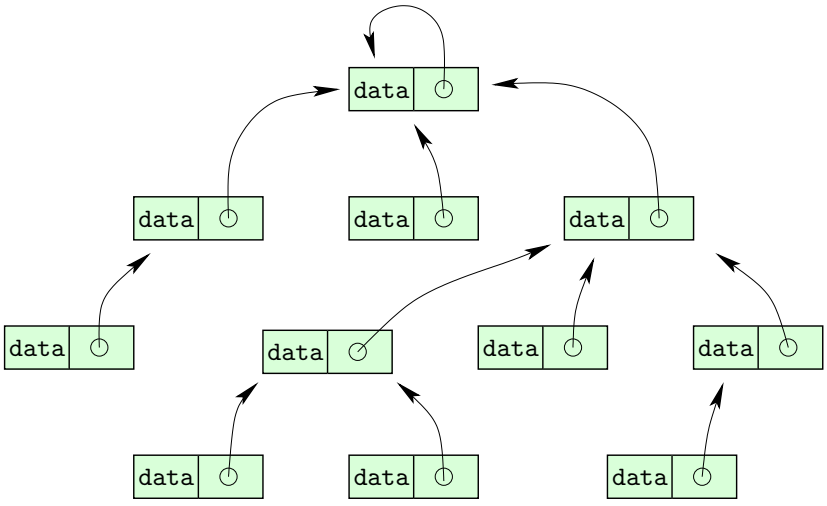
6.2 Zeiger

Die Speicherverwaltung im vorherigen Abschnitt ist einfach und gut, aber was machen wir, wenn wir einen Speicherbereich länger brauchen? Es wäre nicht möglich, eine Funktion zu schreiben, die Speicher anfordert um darin das Ergebnis zurückzugeben. Um dies zu realisieren, benötigen wir aber erst eine Konstruktion, die uns erlaubt auf den Speicherbereich zu *zeigen*, den wir zurückgeben. Die Idee hinter Zeigern ist, dass wir eine Variable haben, die auf eine andere „zeigt“:



Typische Anwendungen von Zeigern sind dynamische Datenstrukturen, wo Zeiger zwischen Teilen der Datenstruktur hin- und herzeigen. Sie sind immer dann sinnvoll, wenn man die Struktur lokal ändern möchte ohne sie komplett neu aufbauen zu müssen. Prominente Beispiele sind:

- verkettete Listen
- dynamische Arrays
- Bäume:



Die Verwendung von Zeigern erläutert man ab besten wieder an einem Beispiel:

```
int a = 5;
int b = 7;
int *p;
p = &a;
int *q;
q = &5;// Fehler
int *r;
r = &b;
*p = 9;// a ist nun 9
p = r;// p zeigt nun auf b
*p = 8;// b ist nun 8
```

Bei der Deklaration eines Zeigers setzt man einen Stern vor den Variablennamen, wie dies bei `p` geschieht. Um nun einem Zeiger einen Wert zuweisen zu können brauchen wir noch die Möglichkeit, die Adresse einer Variable zu bekommen. Dies geschieht mit einem vorgestellten `&`, wie man bei der Zuweisung von p sehen kann. Die Zuweisung für q ist deshalb fehlerhaft, da 5 eine Konstante ist.

Letztendlich benötigen wir noch die Möglichkeit, den Inhalt der Variable zu bekommen, auf die ein Zeiger zeigt. Unglücklicherweise geschieht dies wieder mit einem Stern, so dass dieser je nach Kontext eine andere Bedeutung hat. Diese Verwendung sieht man in der drittletzten und in der letzten Zeile. Man sollte sich unbedingt den Unterschied zur vorletzten Zeile klar machen, wo der Zeiger selbst einen neuen Wert bekommt, also auf etwas anderes zeigt.

Zusammenfassend haben wir folgende Symbole benutzt:

`*`	Deklaration von “Pointer auf”
`(&`	Deklaration einer Referenz)
`*`	Inhalt auf den der Pointer zeigt
`&`	Adresse von einer Variablen

Ein Stück zu recht sind Zeiger ein gefürchtetes Werkzeug. Die am schwierigsten zu findenden Fehler haben meist mit Zeigern zu tun. Wenn man einen Zeiger benutzt, der auf einen

ungültigen Bereich zeigt, kann man sich schon mal den Tag versauen. Nichts desto trotz können Zeiger sehr hilfreich sein, wenn man sie gekonnt einsetzt.

Wenn man nun das Verhalten von Zeigern mit dem von Instanzen von Klassen in Java vergleicht, so kann man bis zu einem gewissen Grad sagen, dass in Java alle Variablen Zeiger sind.

6.3 Speicher anfordern: `new`

Nun soll die Möglichkeit vorgestellt werden, wie man Speicher vom System anfordern kann, der nicht automatisch zurückgegeben wird. Der Befehl dazu ist ähnlich wie in Java `new`:

Format
```
type* new type;
```

Damit ist gemeint, dass die Funktion `new` bei einem Parameter `type` einen Zeiger auf diesen Typ zurückgibt. Dabei reserviert es den Speicherbereich, auf den der Zeiger zeigt (wenn noch so viel Speicher verfügbar ist).

Die Benutzung erläutern wir wieder an mehreren Beispielen:

Benutzung
```
#include <string>
int main()
{
  int* MeinZeiger = new int;
  std::string * ZweiterZeiger = new std::string;
  short int * ZeigerMitInitialisierung = new short int(42);
  double const* ZeigerAufKonstante =
    new double const(3.1415926535897);
  return 0;
}
```

Im ersten Fall wird Speicher für ein `int` geholt und der Zeiger `MeinZeiger` damit initialisiert. Ähnlich im zweiten Fall für eine Klasse vom Typ `string`. Die dritte Variable erhält einen Zeiger auf einen Speicherbereich von Type `short int`, der bereits mit 42 initialisiert wurde. Im letzten Fall wird die Technik mit `const` derart kombiniert, dass wir einen Zeiger auf eine Konstante haben. (Wir erinnern uns: Von der Variable ausgehend nach links: „*“ für Zeiger und `const` für Konstante.) Den Wert `*ZeigerAufKonstante` also 3.1415926535897 kann daher nicht verändern. Den `ZeigerAufKonstante` jedoch schon!

6.4 Speicher zurückgeben: `delete`

Wenn wir Speicher mit `new` anfordern, dann sind wir selbst dafür verantwortlich, diesen zurückzugeben. Dies geschieht mit `delete`:

Format:
```
delete PointerToType;
```

Auch hier ein kleines Beispiel, das hoffentlich weitgehend selbsterklärend ist:

```
int main()
{
  int* MeinZeiger = new int;
  double * ZweiterZeiger = new double;
      :
  delete MeinZeiger;
  double * AuchZweiterZeiger = ZweiterZeiger;
  *AuchZweiterZeiger = 1.41;
  delete AuchZweiterZeiger;
  *ZweiterZeiger = 1.73;// Fehler zur Laufzeit!!!
  return 0;
}
```

Die letzte Zeile zeigt eine der Gefahren, die Zeiger so eklig machen: Der Speicher, auf den `ZweiterZeiger` zeigt, ist bereits wieder freigegeben worden. Allerdings existiert der Zeiger noch. Wenn wir nun versuchen darauf zuzugreifen und z.B. wie im gegebenen Fall in den Speicherbereich, auf den der Zeiger zeigt, zu schreiben, ist dies falsch. Das Problem ist, dass dieser Fehler erst zur Laufzeit auftritt und es daher sein kann, dass er erst spät entdeckt wird – z.B. wenn das Programm schon ausgeliefert ist. Was die Sache noch gemeiner macht ist, dass nicht klar ist, wo der Wert nun hingeschrieben wird. Es kann ja sein, dass der Speicherbereich inzwischen anderweitig vergeben wurde. Dann ändert sich urplötzlich eine andere Variable im Programm. Es kann sogar passieren, dass der Speicherblock garnicht mehr diesem Programm zugewiesen wurde. Wenn das Betriebssystem das merkt, wird das Programm meist gnadenlos abgebrochen (Schutzverletzung oder Segmentation fault).

6.5 Felder und Zeiger

Leider sind Zeiger und Felder in C/C++ nah verwandt, was leicht für Verwirrung sorgt. Das geht soweit, dass Felder und Zeiger in vielen Situationen austauschbar sind. Man kann sich unter einem Feld in etwa einen Zeiger auf das erste Element vorstellen.

Das hat zur Folge, dass Felder einen Sonderfall von `delete` sind: Wenn man Speicher von Feldern zurückgibt, muss man nicht `delete`, sondern `delete[]` verwenden.

`Beispiel:`

```
int main()
{
  int* MeinArray = new int[1000];
      :
  delete[] MeinArray;

  return 0;
}
```

Da Zeiger und durch ihre Verwandtschaft damit auch Felder gefährlich sind, sollte man besser die gekapselte Version von Feldern verwenden, wenn es vertretbar ist. Diese Klasse heißt `std::vector` und wird im Abschnitt 9.4 vorgestellt werden.

6.6 Die dunkle Seite der Macht

Da man es nicht oft genug sagen kann, hier noch einmal eine Warnung vor Zeigern. Verwenden Sie Zeiger nur, wenn es nicht anders (sinnvoll) geht, z.B.

- wenn ansonsten massives Kopieren von Speicher notwendig ist,
- wenn man keine Referenz verwenden kann,
- wenn man keine Standard-Container verwenden kann.

Einen Integer oder Double kann man prima kopieren. Auch bei einer handvoll Zahlen lohnt es sich im Allgemeinen nicht, deswegen Zeiger zu bemühen.

Wenn es nur darum geht, eine Abkürzung für einen langen Ausdruck zu haben, dass tut man häufig besser besser daran, eine Referenz zu benutzen. Dann kann es auf jeden Fall nicht passieren, dass man aus Versehen den Zeiger ändert an Stelle des Wertes der Variable, auf die der Zeiger zeigt.

Für Dinge wie Listen, dynamische Arrays oder Bäume gibt es in C++ vorgefertigte Klassen, die man in vielen Fällen prima zu seinen eigenen Datenstrukturen zusammenstecken kann.

Wenn man doch einmal Zeiger verwenden muss, dann sollte man ihn

- in einer Klasse einkapseln und
- diese Klasse austesten, bevor man sie verwendet.

6.7 Aufgaben

1. Schreiben Sie ein Programm, das
   - Speicher mit `new` anfordert
   - dem Speicher etwas zuweist
   - dem Zeiger etwas zuweist
   - allen Speicher wieder zurückgibt
2. Erweitern Sie das Programm, so dass mindestens drei Kombinationen von „Zeiger„, „const“ und „Referenz“ verwendet werden. Weisen Sie dabei der Variablen jeweils etwas zu. Generieren Sie für jede ihrer Variablen mindestens einen Fehler.
3. Schreiben Sie eine Klasse, die die Verwendung eines Zeigers kapselt. Ein Beispiel wäre ein Array, das eine Funktion „GrößeAndern“ hat. Sorgen Sie dafür, dass der Destruktor den Speicher wieder frei gibt, wenn er am Ende des Geltungsbereichs automatisch aufgerufen wird.