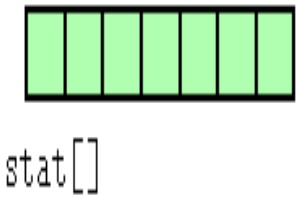
# 15. Dynamische Speicherverwaltung

Bisher waren Datenobjekte (Variable), die in einem Programm definiert werden statische Objekte. Das heißt, dass der C-Compiler für die Bereitstellung von Speicher für diese Objekte sorgt und dass ihre Anzahl und Größe (also der Speicherbedarf) zum Zeitpunkt der Übersetzung festliegen muss. Bei manchen Algorithmen ergibt sich das Problem, dass die Größe eines Objektes (z. B. Arrays) bzw. die Anzahl der Objekte erst zur Programmlaufzeit angegeben werden kann.

Mit den bisher bekannten Datenstrukturen der Sprache C bleibt nur die Notlösung: Definition statisch angelegter Objekte mit Maximalgröße. Zum einen wird hier oft Speicher verschwendet und zum anderen reicht mitunter der verfügbare Speicher nicht für die Maximalwerte aus und das Programm wird nicht gestartet, obwohl in der aktuellen Situation wesentlich weniger Speicher gebraucht würde. Das Ganze ist also wenig flexibel.

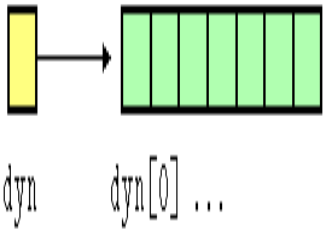
Die Lösung liegt in einer dynamischen Speicherallokation. Diese Lösung ist am Speicherbedarf ausgerichtet und der belegte Platz kann sogar wieder dynamisch freigegeben werden. Dynamisch allokierte Objekte können nicht wie statische Objekte definiert werden. Sie werden auch nicht über einen Namen, sondern nur über ihre Adresse angesprochen. Diese Adresse entsteht bei der Speicherbelegung ("Allokation") und wird normalerweise einer Zeiger-Variablen zugewiesen.

Eine statische Variable, etwa ein Array, hat einfach die Form:




Bei dynamischer Allokation geht man folgendermaßen vor:

- Es wird also zuerst eine Zeigervariable entsprechenden Typs definiert, die aber noch keinen sinnvollen Wert besitzt.
- Dann wird für das Objekt, auf das die Zeigervariable verweisen soll, ausreichend Speicher allokiert und nun der Zeigervariablen die Adresse dieses Speichers zugewiesen.




Es ist wichtig, sich den Unterschied zum äquivalenten statischen Array klarzumachen. Während `dyn` eine *Variable* ist, deren Speicherplatz sich aus `&dyn` ergibt, ist `stat` eine *konstante Adresse*. Interessant ist die Frage, was denn dann `&stat` ist. Intern und formal wird dazu folgender Trick verwendet: `&stat` entspricht `stat`, die Adressen sind also identisch.

DI Harald Sander

Die dynamische Speicherbelegung erfolgt mittels spezieller, in der Standardbibliothek enthaltenen, Speicherallokationsfunktionen. `size_t` ist dabei ein maschinenabhängig definierter Datentyp `unsigned int`, definiert in `<stdlib.h>`, `<stdio.h.h>`, `<stddef.h>`, etc.).

- `void *malloc(size_t size)`
  belegt einen `size` Bytes großen zusammenhängenden Bereich und liefert die Anfangsadresse davon zurück.
- `void *calloc(size_t nobj, size_t size)`
  liefert die Anfangsadresse zu `nobj*size` Bytes großem Bereich zurück; der Inhalt dieses Bereiches ist mit dem Wert 0 initialisiert.
- `void * realloc(void *ptr, size_t size)`
  Veränderung der Größe eines allokierten Speicherblocks.
- `void free(void *)`
  wird schließlich verwendet, um nicht mehr benötigten dynamisch belegten Speicherplatz wieder freizugeben.
- `size_t sizeof(something)`
  ist ein Operator, wobei `something` nicht nur eine einfache oder strukturierte Variable sein kann, sondern auch ein Datentyp. Der Operator liefert die Länge des Übergabeparameters in Bytes zurück. Es ist ratsam, statt einer konstanten Größe eine Datenelementes anzugeben, die Größe mit `sizeof()` zu bestimmen. Erstens kann man sich nicht vertun und zweites sind soche Programme portabler. Also z. B. statt `calloc(2,100)` besser `calloc(sizeof(int),100)` verwenden.

Die Speicherallokations-Funktionen liefern die Anfangsadresse des allokierten Blocks als void-Pointer `(void *)` es ist daher kein Type-Cast bei Zuweisung an Pointer-Variable erforderlich. Um dem Programm mehr Klarheit zu geben, schadet es aber auch nicht, Type-Cast zu verwenden.

Die Funktionen `malloc` und `calloc` liefern bei Fehler (z. B. wenn der angeforderte Speicherplatz nicht zur Verfügung steht) den Nullpointer `NULL` zurück. Nach jedem Aufruf sollte deshalb deren Rückgabewert getestet werden! Dies kann mit `void assert(int)` geschehen. `assert()` ist ein Makro und benötigt das Headerfile `<assert.h>` und u. U. `<stdio.h>`. Ist das Argument NULL, wird das Programm abgebrochen, der Modulname und die Programmzeilennummer des `assert`-Aufrufs ausgegeben.

Für `malloc`, `calloc` und `free` wird das Headerfile `<stdlib.h>` oder `<alloc.h>` benötigt.

Der für die dynamische Speicherverwaltung zur Verfügung stehende Speicherbereich wird als **Heap** bezeichnet. Die Lebensdauer dynamisch allokierten Speichers ist nicht an die Ausführungszeit eines Blocks gebunden. Nicht mehr benötigter dynamisch allokierter Speicher ist explizit freizugeben (`free()`).

Ein erstes Beispiel:

```
int *ip;
ip = (int *) malloc(n*sizeof(int));
free(ip);
```

Im folgenden Beispielprogramm wird dynamisch Speicherplatz für Objekte bereitgestellt, die einfachen Variablen der Datentypen `int`, `float` und `double` entsprechen. Nach dem Ablegen von eingelesenen Zahlenwerten auf diesen Objekten und dem anschließenden Ausdrucken wird der zugeteilte Speicherplatz wieder freigegeben.

```
#include <stdio.h>
#include <stdlib.h>
main()
  {
  int *ip; float *fp; double *dp;

  /* Speicher anfordern */
  ip = (int *) malloc (sizeof(int));
  fp = (float *) malloc (sizeof(float));
  dp = (double *) malloc (sizeof(double));

  printf("Drei Zahlen eingeben:\n");
  scanf("%d %f %lf", ip, fp, dp);
  printf("%d, %f, %f\n", *ip, *fp, *dp);

  /* Speicher freigeben */
  free(ip);
  free(fp);
  free(dp);
  }
```