Übersicht

- Speichertypen
- Speicherverwaltung und -nutzung

Speichertypen

Beim Laden eines Programms in den Speicher (Programmausführung) kommen 3 verschiedene Speicherbereiche zum Einsatz:

- Text Segment (Code Segment): Kompilierter Code

Zur Datenspeicherung kann der Compiler zugreifen auf:

- Stack Segment: Lokale (automatische) Variablen der Funktionen, als Stapel realisiert
- Heap Segment: Speicher unabhängig von Funktionsaufrufen

[Heap] Speichertypen

- Externe Variablen und interne Variablen mit Attribut *static:*
  Der benötigte Speicherplatz wird vom Compiler 'bereitgestellt' (adressiert), der dann während der Ausführung des Programms zur Verfügung steht.

- Automatische Variablen
  Speicherplatz wird dynamisch bereitgestellt, d.h. während der Ausführung des Programms und jedes Mal, wenn ein Programmblock betreten wird, der die Variablendefinition enthält. Beim Verlassen des Blocks wird der Speicher wieder freigegeben.
  Realisierung als Stack, damit erfolgt Speicherfreigabe in umgekehrter Richtung der Speicheranforderung (und erst, wenn Speicher oberhalb auf dem Stack wieder frei ist).

[Heap] Speichertypen : Heap

- Dynamische Speicherallokation während der Laufzeit des Programms.
  Der bereitgestellte Speicher steht zeitlich unbegrenzt zur Verfügung, bis er auf ausdrückliche Aufforderung hin wieder freigegeben wird.

- Dieser Speicher wird Heap genannt und hat eine eigene Speicherverwaltung durch das Betriebssystem

- Die Verfügbarkeit des Heap ist nicht an Programmblöcke gebunden.

[Heap] Variablen und Konstanten im Speicher

```
char testc[]="hallowelt!";
int testi[10];
int testi2;

void testfkt()
{
	static int j = 3;
	int i;
	printf("auto int          : %p\n",(void*)&i);
	printf("static   int          : %p\n",(void*)&j);
	printf("Adresse der Funktion: %p\n",(void*)&testfkt);
}

int main(int anzahl,char *args[])
{
	char *string="hallo";
	char string2[]="welt";
	int test2 = 1;
	int *i;
	printf("char pointer              : %p\n",(void*)string);
	printf("lokal char array          : %p\n",(void*)string2);
	printf("global char array         : %p\n",(void*)testc);
	printf("global int variable       : %p\n",(void*)&testi2);
	i = (int*)malloc(sizeof(int));
	printf("int variable              : %p\n",(void*)&test2);
	printf("dynamisch allozierter int: %p\n",(void*)i);
	printf("main funktion             : %p\n",(void*)&main);
	testfkt();
	return 0;
}
```

Funktionen zur Heapverwaltung

```
void* malloc(size_t groesse);

void free(void* zeiger);

void* calloc(size_t anzahl, size_t groesse)
```

```
float* v;
int laenge;

v = (float*) malloc(laenge * sizeof(float));

if(v == NULL)
{
    /*  Der Nullpointer bedeutet eine Fehlermeldung
        von malloc, es ist also etwas schiefgegangen */
}
```

```
v = malloc(laenge * sizeof(float));
```

[Heap] Funktionen: Speicher anfordern

- calloc stellt einen Speicherbereich zur Verfügung für Anzahl n Objekte der Größe m und beschreibt diesen Speicherbereich mit 0. Nicht immer sinnvoll, besser mit
- malloc: stellt Speicherbereich der Größe m zur Verfügung

... beide liefern einen void-Pointer auf diesen Speicherbereich zurück.

... beide können fehlschlagen, müssen also getestet werden dahingehend, ob sie NULL zurückgegeben haben (=Fehler)

[Heap] Funktionen zur Heapverwaltung

```
void testfunktion(int laenge)
{
    float* v;

    v = (float*) malloc(laenge * sizeof(float));
    if(v == NULL)
    {
        /*  Der Nullpointer bedeutet eine Fehlermeldung
            von malloc, es ist also etwas schiefgegangen */
    }
}
```

[Heap] Funktionen: Speicher freigeben

- Genauso wichtig wie die Anforderung von Speicher ist es, den Speicher wieder freizugeben, da sonst dieser Speicher belegt bleibt

  (und ein Programm schließlich wegen unzureichendem Heapspace abstürzen kann)

- Daher wird die Funktion free() verwendet, welcher ein Zeiger auf einen Speicherbereich übergeben wird (welcher zuvor mit malloc/calloc alloziert wurde).

[Heap] Funktionen zur Heapverwaltung

```
void bessere_testfunktion(int laenge)
{
    float* v;

    v = (float*) malloc(laenge * sizeof(float));
    if(v == NULL)
    {
        /*  Der Nullpointer bedeutet eine Fehlermeldung
            von malloc, es ist also etwas schiefgegangen */
    }

    /*
        Hier passiert irgendetwas, was den Speicher
        tatsächlich nutzt
    */

    free(v);
}
```

[Heap] Nutzen von dynamischem Speicher

- Der Bedarf an Speicher hängt oft vom Benutzer-verhalten ab, d.h. der tatsächliche Speicherbedarf ist bei der Code-Erstellung nicht formulierbar (und soll nicht an Programmblöcke gebunden sein)
- Möglichkeit 1: Man dimensioniert z.B. Felder so groß, daß sie die größtmögliche zu erwartende Eingabe auch gespeichert werden kann.

  Nachteil: Speicherbedarf orientiert sich immer am 'worst case'.

  Zudem ist der Speicherbedarf oft gar nicht abzuschätzen.
- Möglichkeit 2: Speicher je nach Bedarf selbst anfordern, und die zu speichernden Informationen dort ablegen.

[Heap] Nutzen von dynamischem Speicher

Vorgehen:

- Berechnung des benötigten Arbeitsspeichers
  z.B. Feld von n int-Werten: n * sizeof(int)

- Anfordern des Speichers:
  malloc(größe-des-Speichers)
- Entgegennahme der Adresse, an welcher der zur Verfügung gestellte Speicher beginnt.

- Über einen Pointer kann an dieser Adresse nun Information abgelegt werden.

[Heap] Beispiel

```
#define DEFLEN 10

void main(int anzahl, char* argument[])
{
	unsigned int maxlen, j, i=0;
	float* v;
	if(anzahl < 2 || sscanf(argument[1], "-%u",&maxlen) != 1)
		maxlen = DEFLEN;
	v = (float *) malloc(maxlen * sizeof (float));
	if (v == NULL)
	{
		printf("Speicherallokation fehlgeschlagen!\n");
		return 1;
	}
	printf("Zahlen eingeben: ");
	while (i < maxlen)
	{
		if(scanf("%f", &v[i]) == EOF)
			break;
		i++;
	}
	for (j = 0 ; j < i ; j++)
		printf("%f\n",v[j]);
	free(v);
}
```

[Heap] Beispiel, ergänzt

```
void* memmove(void* s1, const void* s2, size_t N);
```

```
{
        float z;
        ...
        while (i < maxlen)
        {
                if(scanf("%f", &v[i]) == EOF)
                        break;
                i++;
        }
        while (scanf("%f",&z) != EOF)
        {
                memmove(&v[0],&v[1],(maxlen-1) * sizeof (float));
                v[maxlen-1] = z;
        }

}
```

[Heap] Matrizen auf dem Heap

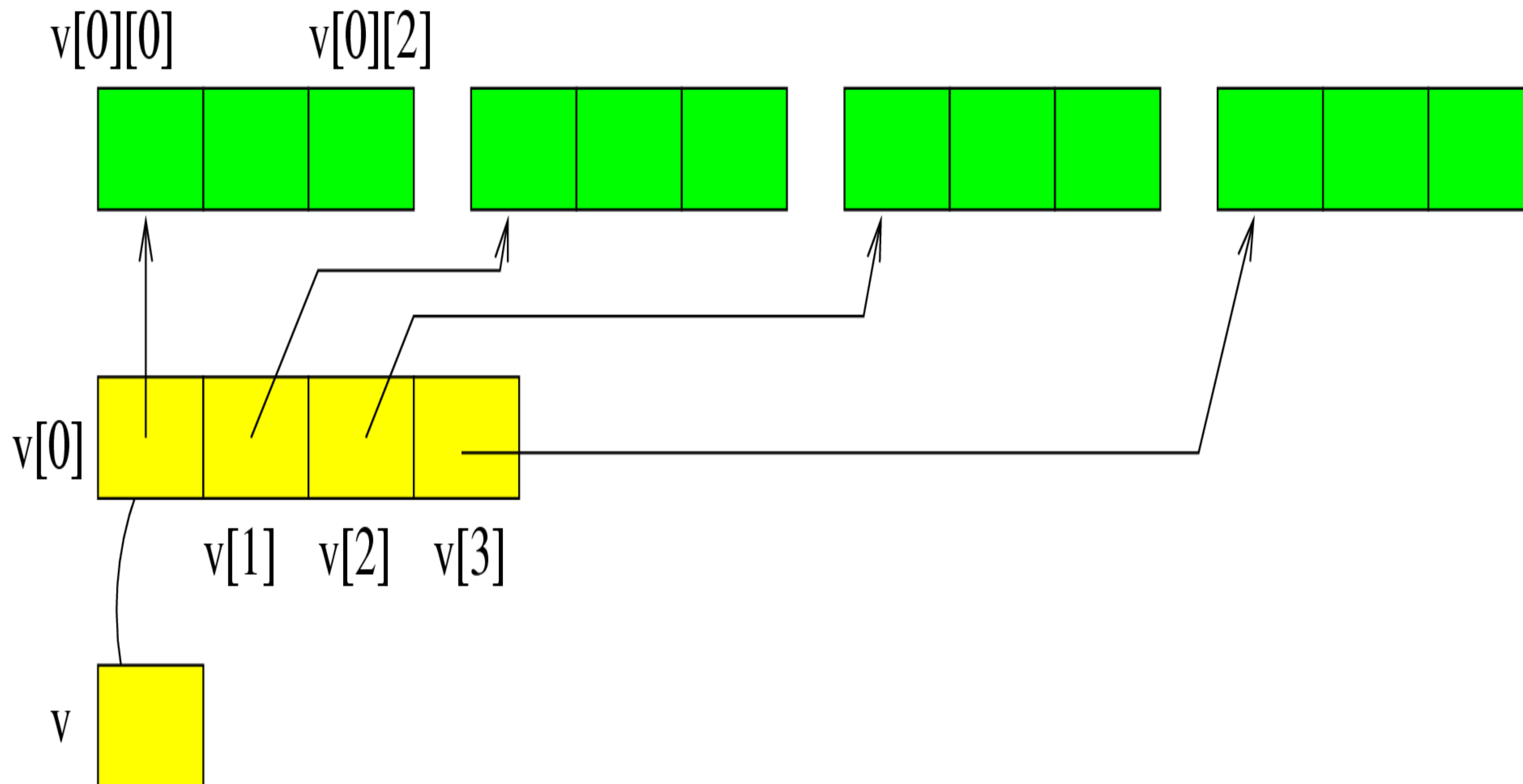
Matrix, formal:

- m[i][j]

- (Paare) eckiger Klammern sind Operatoren, aufeinanderfolgende Paare werden von links nach rechts abgearbeitet

- Der Name einer Matrix repräsentiert einen Zeiger auf den Vektor von Zeigern auf die Zeilen (mit den Werten)

Matrizen auf dem Heap



Matrizen auf dem Heap

```
/* Beispiel einer n x m Matrix */

int n,m;
double** a;
int i;

a = (double**) malloc(n * sizeof (double*));
if ( a == NULL)
    /* Fehlerbehandlung */

for (i = 0 ; i < n ; i++)
{
    a[i] = (double*) malloc(m * sizeof(double));
    if (a[i] == NULL)
            /* Fehlerbehandlung */
}
```

[Heap] *volatile* und *register*

Der Vollständigkeit halber:

- volatile: unterbindet die Optimierung des Zugriffs auf Variablen, sodass die Variable bei jedem Zugriff neu aus dem Speicher geladen werden muss (und nicht beispielsweise in einem Prozessorregister vorgehalten werden darf).
  Kann bei Hardwaresteuerung notwendig sein.

- register: Damit soll bewirkt werden, dass Variablen mit diesem Schlüsselwort versehen möglichst in Prozessorregistern gehalten werden. Ein guter Compiler sollte das aber von vorneherein versuchen.