

Entwicklung/Erweiterung einer Applikation in C

Die zu erstellende C-Applikation benötigt drei Dateien, nämlich *dynVector.h*, *dynVector.c* und *dynVectorTest.c*. Diese Dateien erhalten Sie weitgehend vorbereitet. Die `#includes` ergänzen Sie selber, so wie dies Ihre Projektumgebung verlangt. Am Schluss geben Sie **nur** die beiden Files *dynVector.c* und *dynVectorTest.c* ab - achten Sie also darauf, dass diese stets mit dem gegebenen Headerfile kompilierbar sind.

Aufgabenstellung:

Die *dynVector*-Dateien sollen einen dynamischen Vektor realisieren und testen. Dazu wird die Datenstruktur

```
typedef struct _DynVector {
    int slotSize;           // amount of bytes which makes up a slot
    int totalSize;          // amount of bytes of storage area
    int nextSlot;           // index of next empty slot
    unsigned char *storage; // pointer to dynamically allocated memory
    int slotIncr;           // amount of slots to add dynamically if we
} DynVector;              // need more space yet have run out of it
```

bereit gestellt. Mit Anlegen einer Variablen dieses Typs ist vorerst kein Speicherplatz für Vektorelemente vorhanden. Erst mit Hinzufügen einzelner Elemente wird allenfalls Speicher angefordert. Wie häufig Speicherplatz neu anzufordern ist, wird beim Anlegen der Variablen festgelegt. Vorausgesetzt, die Struktur `Point` sei anderswo deklariert, so wird z.B. in der Codezeile unten ein dynamischer Vektor kreiert, der ausschliesslich `Point`-Elemente aufnimmt. Steht kein (weiterer) Platz zur Aufnahme von `Point`-Elementen zur Verfügung, so wird vorerst Platz für 10 weitere Elemente geschaffen.

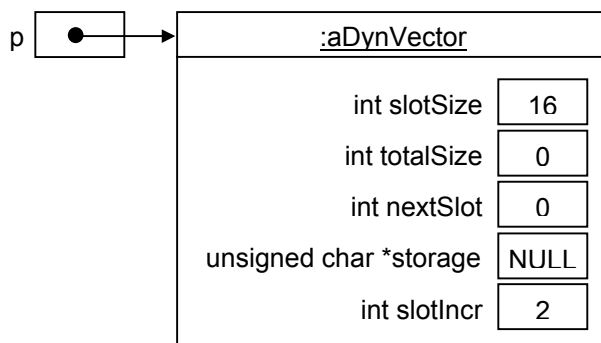
```
DynVector *p = createDynVector(sizeof(Point), 10);
```

Damit der Vektor für alle Typen zu verwenden ist, wird hinter den Kulissen mit dem Typ `char` gearbeitet (`char` von Ihnen als 1 Byte anzunehmen). Dies lässt dann z.B. auch einen Vektor wie folgt zu:

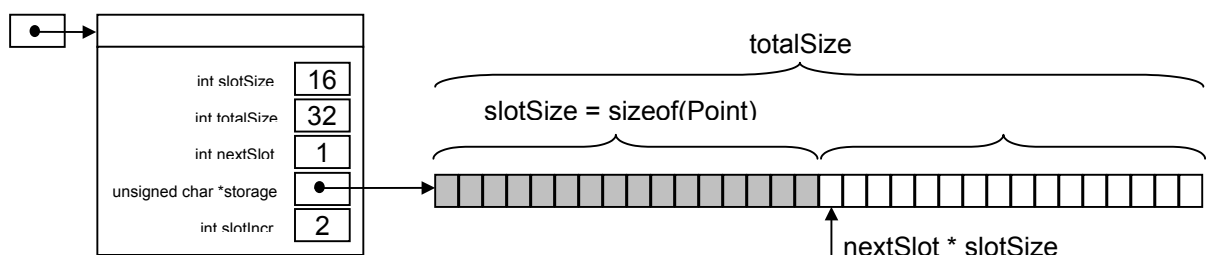
```
DynVector *q = createDynVector(sizeof(long), 25);
```

- 0) Nicht benotet, aber nützlich: Notieren Sie in einer Skizze, wie der Vektor vergrössert wird. Bezeichnen Sie die Skizze mit `slotSize` etc. wie durch die Membernamen und deren Bedeutung gegeben.

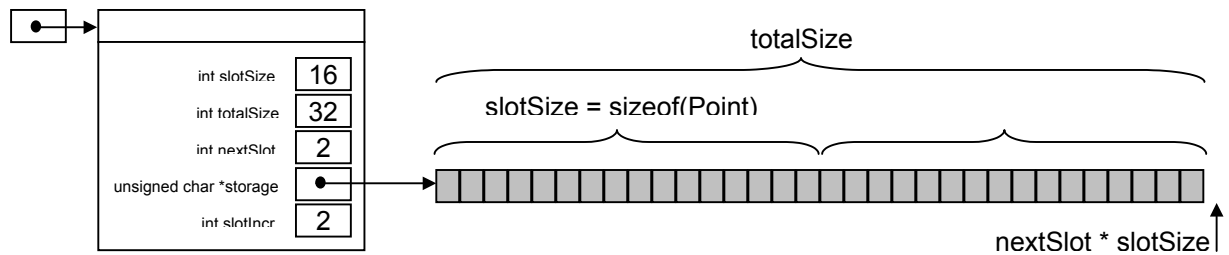
Situation nach `DynVector *p = createDynVector(sizeof(Point), 2);`:



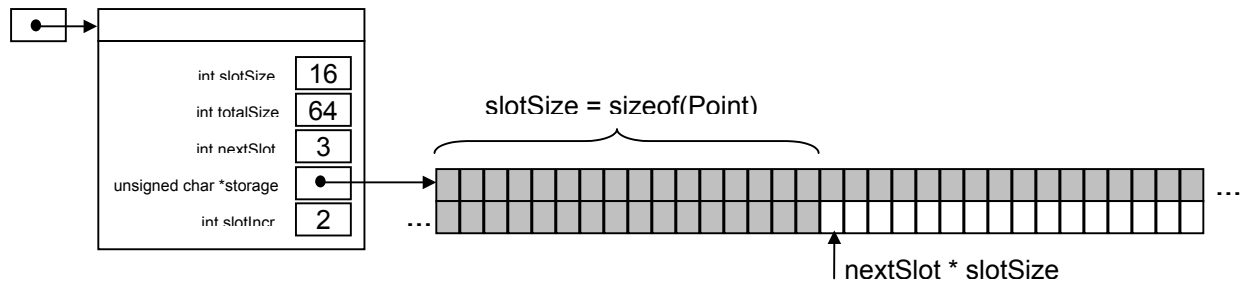
Situation nach `addElDynVector(p, pointP);`:



Situation nach weiterem Hinzufügen mit `addElDynVector(p, pointP); :`



Situation nach drittem Hinzufügen mit `addElDynVector(p, pointP); :`



a) Implementieren Sie die Funktion `createDynVector`. Vgl. die Beschreibung im File `dynVector.c`.

Merke:

- Die 3 abgegebenen Dateien sind kompilierbar und können ausgeführt werden.
- Mit Implementieren einzelner Funktionen lohnt es sich, Teile im Testfile auszukommentieren und erst später mit fortschreitender Lösung wieder zu berücksichtigen.

1 P. Allokation für ein Vektor-Element

4 P. Initialisierung der Member des Vektors (storage nicht berücksichtigt)

b) Ergänzen Sie die Funktion `printDynVector` so, dass Sie auch Informationen zu Vektorelementen an `cout` ausgegeben werden. Vgl. `printDynVector` in der Datei `dynVector.c`.

Vorerst sollte die Ausgabe so aussehen:

```
DynVector:
  slotSize: 16
  totalSize: 0
  nextSlot: 0
  storage: 0
  slotIncr: 2
Point: NULL
```

Nach Implementation von c) und Hinzufügen eines `Points` sollte die Ausgabe etwa so aussehen:

```
DynVector:
  slotSize: 16
  totalSize: 32
  nextSlot: 1
  storage: 167840808
  slotIncr: 2
Point:
  x: 12345678.00  y: -9876543.00
```

1 P. Korrekter Aufruf der Funktion via den Funktionspointer `fp`.

1 P. Korrektes Argument in der Argumentenliste.

- c) Ergänzen Sie die Funktion `addEIDynVector`. Wählen Sie eine Lösung, bei der das Vektorelement (das selbst keine Pointer enthalten darf) in den Vektor-Slot kopiert wird.

Tipps:

- Sollte das korrekte Allokieren nicht gelingen, versuchen Sie allenfalls mit Anforderung von zuviel Speicherplatz eine (nicht-korrekte, aber funktionierende) Lösung zu erarbeiten, damit Sie auch weitere Aufgaben lösen können.
- Speicherbereich kann mit `memcpy` kopiert werden.

- 1 P. Anfordern von Speicher, wenn kein unbenutzter Slot mehr vorhanden
- 1 P. Allokation für storage
- 1 P. Korrekte Anzahl Bytes für Allokation angefordert
- 1 P. Erhalten des alten Werts in storage, falls Reallokation misslingt
- 1 P. Kopieren des Elements (hier des Point) in den Slot mit `memcpy`.
- 1 P. Updaten der Verwaltungs-Member (`totalSize`, `nextSlot`)

- d) Implementieren Sie die beiden getter-Funktionen `getEIPtrDynVector` und `getSizeDynVector`.

- 1 P. `getEIPtrDynVector` Abfangen zu grosser Indizes und NULL-Rückgabe
- 1 P. `getEIPtrDynVector` Rückgabe des korrekten Pointers auf das storage-Feld
- 1 P. `getSizeDynVector` Rückgabe der Anzahl belegter Slots

- e) Realisieren Sie `deleteDynVector`.

- 1 P. Freigabe von storage
- 1 P. Freigabe des Vektors selbst

- f) Bei korrekter Implementation funktioniert auch bereits der Vektor für das Speichern von `int`-Elementen. Es fehlt aber noch die Implementation der Funktion `printInteger` (vgl. Datei `dynVectorTest.c`). Realisieren Sie diese Funktion im Sinne der Aufgabe b). Es sind Änderungen in der Parameterliste und im Funktionsblock nötig.

- 1 P. Korrekter Parameter in `printInteger`
- 1 P. Korrekter Funktionsblock

- g) Zusätzliche Punkte:

- 1 P. Überall getestet, ob Allokation auch erfolgreich verlief
- 1 P. Lösung in 3 Dateien realisiert
- 1 P. Kompilierbar und läuft.

Häufige Fehler:

- `sizeof` wird während Compile Time ausgewertet. Folgendes Code Snippet ist z.B. falsch:

```
void xyz(void *myType) {
    int i = sizeof(*myType);
}
```

- Ähnlich falsch:

```
void xyz(int i) {
    char arr[i];
}
```

- Kein Test, ob Allokation erfolgreich war.

- `printDynVector` kann nicht wissen, wie ein Vektor- Element bzw. seine Member darzustellen sind. Daher muss ein Pointer auf eine Funktion übergeben werden, womit ein Callback via Funktionspointer möglich wird.

- Bei Freigabe von dynamischem Speicher muss - falls vorhanden - zuerst aller verschachtelt angelegte Speicher freigegeben werden. Hier also muss zuerst `storage`-Speicher freigegeben werden und dann der Vektor selbst.

- Wenn alleine `realloc` verwendet wird (statt auch `malloc`), so muss darauf geachtet werden, dass für den Fall, wo `realloc` wie `malloc` funktionieren soll, der erste Parameter tatsächlich `NULL` ist. Daher ist dann beim Kreieren des Vektors dafür zu sorgen, dass `storage` ebenfalls zu `NULL` initialisiert ist. Vgl.:
`char * temp = realloc(dv->storage, ...);`

- Wurde vorgängig dynamischer Speicherplatz erhalten und der Pointer z.B. in die Variable `p` abgelegt, so darf beim Reallozieren nicht direkt an die Variable `p` zugewiesen werden Sollte `realloc` keinen Speicherplatz mehr erhalten, so wird die alte Adresse mit `NULL` überschrieben und der Memory Leak ist perfekt. Stets über eine temporäre Variable vorgehen und diese testen. Erst bei Erfolg den alten Pointer überschreiben.

- Vermeiden Sie `malloc(0)`. Wenn nötig fangen Sie eine mögliche 0 ab, bevor diese als Argument an `malloc` gereicht wird. Der ANSI-Standard sagt nicht genau, was in einem solchen Fall als Rückgabewert zurückgegeben wird.

Überhaupt: Konsultieren Sie stets die Referenz für solche Spezialfälle, wenn Sie nicht sicher sind, wie sich C verhält. Andernfalls umschiffen Sie den Spezialfall: dann ist ev. der Code etwas komplizierter, aber (hoffentlich) sicherer.

- Was sagen Sie dazu:

```
char []f() {
    char arr[12];
    return arr;
}
?
```