Wieso brauchen wir Doppelreferenzen?

Übungsblatt 6

Eric Kunze — 10. September 2021

Die Ausführungen basieren auf einem OPAL-Forum-Post aus dem Wintersemester 2020/21, siehe https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/26663256067/CourseNode/101767297876002/Message/29317267462.

Keine Garantie auf Vollständigkeit und/oder Korrektheit!

Wir betrachten eine einfach-verkettete Liste, definiert durch folgenden Datentyp:

```
typedef struct element *list;
struct element {
    int value;
    list next;
};
```

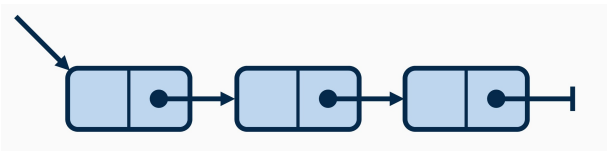
Wir wollen nun beispielhaft das Anhängen an das Ende der Liste ansehen. Dazu sei zunächst eine einfache Hilfsfunktion zum Ausgeben einer Liste gegeben:

```
void print(list lp) {
    printf("[");

    while(lp != NULL) {
        printf("%d, ", lp->value);
        lp = lp->next;
    }

    printf("]\n");
}
```

Man beachte dabei, dass dieses `print` nichts mit der „Standard-Ausgabe“ `printf` zu tun hat und insbesondere nicht standardmäßig implementiert ist.

1 Einfache Referenzen

Wir versuchen zunächst dieses Problem mit einfachen Referenzen zu lösen, d.h. der Kopf unserer Funktion soll so aussehen: `void append_p(list lp, int n)`.



Zunächst gehen wir bis zum Ende der Liste, indem wir in einer `while`-Schleife den Pointer immer weiter setzen bis wir schließlich den `NULL`-Pointer am Ende erreichen. Dann können wir mit `malloc` ein neues `struct element` erzeugen, dieses entsprechend befüllen und abschließend die Verknüpfung zur bisherigen Liste herstellen.

```
void append_p(list lp, int n) {
    while(lp->next != NULL)
        lp = lp->next;

    struct element *c = malloc(sizeof(struct element));
    c->value = n;
    c->next = NULL;

    lp->next = c;
}
```

Testen wir diese Funktion mithilfe der folgenden `main`:

```
int main(void) {
    list l = malloc(sizeof *l);
    l->value = 2;
    l->next = NULL;

    append_p(l, 4);
    print(l);
}
```

so erhalten wir die Ausgabe `[2, 4]`; es scheint also alles geklappt zu haben. Nun kommen wir aber zum Problemfall: was passiert, wenn unsere Liste nicht initial mit einer `2` gefüllt ist, sonder initial leer ist?

```
int main(void) {
    list l = NULL;

    append_p(l, 4);
    append_p(l, 2);

    print(l);
}
```

Dieses Programm wird nicht wie erhofft funktionieren, da wir in der `append_p`-Funktion versuchen, einen `NULL-Pointer` zu derefenzieren, wenn wir die Schleifenbedingung `lp->next != NULL` prüfen. Genauer entsteht ein zufälliger Speicherzugriff; ein „segmentation fault“ oder „access violation“.

2 Ein Ausweg mit einfachen Referenzen?

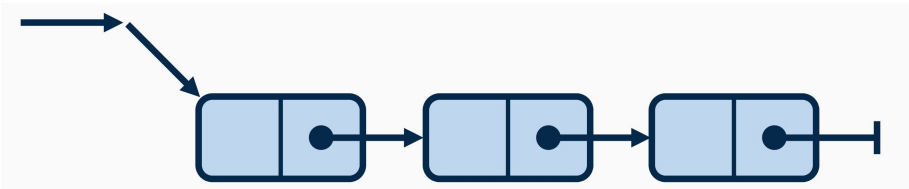
Nun versucht man natürlich erst einmal mit einfachen Mitteln (d.h. einfachen Referenzen) auch den Fall der leeren Liste einzubeziehen. Dabei könnte folgende Funktion entstehen:

```
void append_p(list lp, int n) {
    while(lp != NULL)
        lp = lp->next;

    lp = malloc(sizeof(struct element));
    lp->value = n;
    lp->next = NULL;
}
```

Damit wird der `NULL`-Pointer nicht mehr dereferenziert, sondern direkt geprüft (*ohne* Dereferenzierung in der Schleifenbedingung). Nun bekommt man allerdings ein neues Problem. Nutzen wir obige `main`-Funktion, so erhalten wir die Ausgabe `[]` — ein merkwürdiges Verhalten, das sich aber mithilfe des pulsierenden Speichers erklären lässt. Beim Aufruf von `append_p(l, 4)` wird der Pointer `l` *kopiert* und in `lp` gespeichert. Ändert man nun innerhalb von `append_p` den `lp`-Pointer, so ändert sich der `l`-Pointer in der `main`-Funktion *nicht*. Beim Rücksprung geht `lp` dann vollständig verloren, der bearbeitete Speicher ist nicht mehr erreichbar und wir halten nur noch den unveränderten `l`-Pointer der `main`. Somit gilt auch nach den beiden Aufrufen `append_p(l, 4)` und `append_p(l, 2)` noch `l == NULL` und eine leere Liste wird ausgegeben.

Somit ist dieser „Ausweg“ mit einfachen Referenzen offensichtlich nicht zielführend.

3 Ausweg: doppelte Referenzen



Nun zeigen wir abschließend noch eine funktionierende Variante, die auch leere Listen behandeln kann und ihre Arbeit beim Rücksprung nicht verwirft. Dies erreichen wir, indem beim Funktionsaufruf nicht eine Kopie des Pointers übergeben wird, sondern wiederum eine Referenz auf den Pointer (so entsteht eine Doppelreferenz). Damit manipulieren wir innerhalb der `append_pp`-Funktion die originalen Pointer und Daten und nicht eine Kopie davon.

```
void append_pp(list *lp, int n) {
    while(*lp != NULL)
        lp = &(*lp)->next;

    struct element *c = malloc(sizeof(struct element));
    c->value = n;
    c->next = NULL;

    *lp = c;
}
```

Mit der zugehörigen `main`

```
int main(void) {
    list l = NULL;

    append_pp(&l, 4);
    append_pp(&l, 2);

    print(l);
}
```

erhalten wir schließlich die erhoffte Ausgabe `[2, 4, ]`.