Einführung in die Programmierung für Physiker

SoSe 2024 – Marc Wagner

Michael Eichberg: eichberg@itp.uni-frankfurt.de

Exercise sheet 9

To be handed in on 21.06.2024.

Exercise 1 [*Recursion counter*] (3 pts.)

In the lecture, the merge sort algorithm was presented as an example of a recursive algorithm. Moreover, the complexity of the algorithm was discussed. Your task is now to verify this complexity by explicit counting. Use `static` variables to count and print the number of recursion levels N_{lvl} as well as the number of operations $N_{\text{ops}}^{(i)}$ at recursion level i (since prefactors are irrelevant for the complexity of an algorithm, it is sufficient to count the number of iterations of the loop responsible for merging two sorted lists). E.g. for an array of size $N = 1024$, you should count $N_{\text{lvl}} = 10$ recursion levels, with $\mathcal{O}(N)$ operations at the ith level.

Hint: It is sufficient to count the operations of the first three iterations, i.e. you can define a static array with three elements for $N_{\text{ops}}^{(i)}$.

Exercise 2 [*Inclusion guards*] (3 pts.)

(i) Compile the `.c` file below, where two `.h` files are included, and run the program.

- `ex2i.c`:

```
#include "a.h"
#include "b.h"

int main() {
    f_a();
    f_b();

    return 0;
}
```

- `a.h`:

```
#ifndef __A_H__
#define __A_H__

#include <stdio.h>

void f_a() {
    printf("f_a wird ausgefuehrt.\n");
}

#endif
```

- `b.h`:

```
#ifndef __B_H__
#define __B_H__

#include <stdio.h>
#include "a.h"
```

```
void f_b() {
    f_a();
    printf("f_b wird ausgefuehrt.\n");
}

#endif
```

(ii) Remove from both `.h` files the preprocessor directives `#ifndef ...`, `#define ...` and `#endif`. Try to compile the `.c` file. Why do you get an error?

(iii) Read about "include guards" (e.g. `https://de.wikipedia.org/wiki/Include-Guard`) and summarize in your own words the basic concept. Refer in your summary to the example from task (i) and task (ii).

Exercise 3 [*Symbolic constants*] (1 + 2 + 1 = 4 pts.)

(i) Compile the following code and run the program.

```
#include <stdio.h>

#define N 4

struct a {
    int b[N];
};

int main() {
    struct a a1;
    printf("%lu\n", sizeof(a1.b) / sizeof(int));

#define N 3

    struct a a2;
    printf("%lu\n", sizeof(a2.b) / sizeof(int));
}
```

Explain, why `#define N 3` has no effect on the size of the array `a2.b`.

(ii) Try to compile the following code.

```
#include <stdio.h>

struct a {
    int b[N];
};

int main() {
    struct a a1;
    printf("%lu\n", sizeof(a1) / sizeof(int));
}
```

Why do you get an error? Try to compile the code again with the option `-DN=4` using the `gcc` compiler. If you are not familiar with `gcc`'s `-D` option, you can read about it in the documentation:

https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html.

Why does the code compile now? Run the program. What is the effect of the option `-DN=10`?

(iii) Compile the following code with the option `-DN=10` and run the program.

```
#include <stdio.h>

#define N 4

struct a {
    int b[N];
};

int main() {
    struct a a1;
    printf("%lu\n", sizeof(a1) / sizeof(int));
}
```

The option `-DN=10` does not have an effect anymore. Explain, why. Try to find a simple way to extend the code with two lines of preprocessor directives such, that the option `-DN=10` leads to arrays `b` in `struct a` with 10 elements, but that it is also possible to compile the code without the `-D` option (in that case arrays `b` in `struct a` should have 4 elements according to `#define N 4`).

Exercise 4 [*Linked lists*] (1 + 1 + 3 + 1 + 3 + 1 = 10 pts.)

You have learned, how you can define a `struct` to bundle multiple variables. One example from the lecture is a linked list, where each element contains a name and a pointer to the next element. In the following, we will explore this example in more detail.

(i) Consider a loop of the form

```
for (/* expr1 */; /* expr2 */; /* expr3 */) {
    /* ... */
}
```

How can you iterate over the elements of a linked list, using only the expressions controlling the `for` loop?

(ii) Accessing the ith element of an array of size N has complexity of $\mathcal{O}(1)$, since the memory address is known to be `start + i * element_size`. What complexity do you expect for access of the ith element of a linked list of size N?

Now your task is to further develop the linked list example by adding new features and test them. Remember to free memory and set pointers to `NULL` when list elements are not needed anymore. Moreover, check for `NULL` pointers in your functions and exit with appropriate error messages.

(iii) The linked list in the lecture is a so-called *singly linked list*, i.e. each element has just one pointer to the next element. Expand this `struct` to a doubly linked list, which has an additional pointer to the previous element. Modify the functions `l_delete` and `l_append` accordingly.

(iv) Write a function, which inserts a new element at a position, defined by an index, into the list.

(v) Write a function, which inverts the order of list elements. What is the complexity of your function? Design an efficient algorithm for list reversal, which has complexity $\mathcal{O}(N)$, and a less efficient algorithm with complexity $\mathcal{O}(N^2)$.

(vi) Write a function, which copies an entire list.

Time to Test! Use the `main` function provided in the lecture and suitable tests for your newly added features.