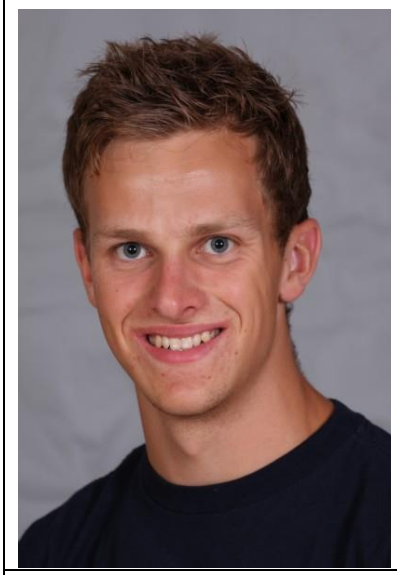
Mads Bornebusch, s1233627
Kristian Sloth Lauszus, s123808
Simon Piotr Almind Patrzalek, s123401

12. maj, 2013

Afleveringsopgave 5

02102 Indledende Programmering

Mads Bornebusch s1233627



Kristian Sloth Lauszus s123808

Simon Patrzalek s123401

Rapporten og øvelsen har været et samarbejde mellem Mads Friis Bornebusch, Kristian Sloth Lauszus og Simon Patrzalek.
For at opnå en vidtrækkende forståelse for opgaverne og deres natur, har hvert gruppemedlem lavet alle opgaverne. Derefter har vi i samråd besluttet hvilke løsninger der var de bedste og gengivet disse i denne rapport. Idet hver opgave indeholdte forskellige dele, kan den endelige løsning til en enkelt opgave, bestå af dele som er udarbejdet af forskellige gruppemedlemmer.

Mads Bornebusch, s1233627
Kristian Sloth Lauszus, s123808
Simon Piotr Almind Patrzalek, s123401

12. maj, 2013

Opgave 1

Denne opgave gik ud på at løse en simpel opgave fra et eksamenssæt fra år 2012. Man havde fået udleveret en ukomplet fil, hvor der var givet et struct, en hjælpefunktion og en main-funktion. Desuden var der en udskrift, som der skulle printes hvis man havde opnået den rigtige calculate- funktion. Herunder ses vores løsning, som så gav den korrekte udskrift i konsollen:

```
void calculate(combo_t combo) {
    helper(combo.one,combo.two,combo.one+combo.two);
}
```

Figure 1 - Calculate-funktionen

Opgave 2

Denne opgave gik ud på at kunne behandle en liste af tal. Selve listen blev defineret i et struct, kaldet stack_t, hvor parametrene var et array, til lagring af de diverse heltal, og to heltal, som henholdsvis viste antallet af heltal i arrayet og som viste længden af arrayet. Dette var den enkleste del, idet vi skulle udvikle adskillige funktioner, som kunne behandle listen. Blandt disse funktioner var der nogle som skulle kunne oprette nye lister og poppe/slette heltal fra listen. Dette betød at vi skulle arbejde med hukommelsesallokering, dvs. brug af malloc, realloc, osv. Det skal siges at struct'et ikke blev kaldt stack_t i vores opgaver, da en af de computere der blev brugt til programmeringen var unix-baseret og stack_t i denne allerede var optaget af en anden, integreret funktion. Vi benytter derfor stack_t2.

De forskellige funktioner som man skal kunne behandle struct'et med, var følgende:

"pop", som skulle kunne fjerne et heltal fra listen, uden at ændre på listens længde, dvs. capacity-parameteren. Struct-parameteren "size" vil selvfølgelig blive ændret.

"newStack" skulle kunne skabe en ny liste/ stack. Hertil blev der brugt malloc.

"push" skulle kunne indsætte et heltal i listen. Hvis listen ikke var lang nok, skulle denne funktion være i stand til at lave en ny liste med den dobbelte længde af den gamle. Til dette brugte vi realloc.

"top" havde ikke en defineret funktion i opgaveteksten. Derfor har vi bare givet den basale funktion at vise hvilket heltal der ligger øverst i stakken.

Nedenunder ses koden for funktionerne:

```
int pop(stack_t2 * stack_p) {
    if (stack_p->size > 0) { // Check if there is actually something in the array
        stack_p->size--;
        return stack_p->array[stack_p->size];
    }
    return -1;
}
```

Figure 2 - Koden for pop-funktionen

```
void push(stack_t2 * stack_p, int value) {
    if (stack_p->size >= stack_p->capacity) { // It's already full
        stack_p->capacity *= 2;
```

```
        stack_p->array = (int*)realloc(stack_p->array, stack_p->capacity*sizeof(int));
    }
    stack_p->array[stack_p->size] = value;
    stack_p->size++;
}
```

Figure 3 - koden for push-funktionen

```
stack_t2 * newStack(void) {
    stack_t2 * stack_p = (stack_t2*)malloc(sizeof(stack_t2));

    stack_p->capacity = 1;
    stack_p->array = (int*)malloc(sizeof(int));
    stack_p->size = 0;

    return stack_p;
}
```

Figure 4 Koden for newStack-funktionen

For at kunne bekræfte at vores programmering var udført tilstrækkeligt, skulle vi desuden have prøveudskrifter. Dette gjorde vi ved at ændre i filen Opg2.c som er filen der tester stack'en. Filen og udskriften fra testen er vist nedenfor.

```
int main() {
    stack_t2 * myStack = newStack();
    if (empty(myStack))
        printf("Stack is empty\n");
    else
        printf("Stack is not empty\n");

    printf("popped: %d\n", pop(myStack));
    push(myStack, 123);
    push(myStack, 99);
    printf("The numbers 123 and 99 have
been pushed\ntop: %d\n\n", top(myStack));
    push(myStack, 4444);
    printf("Three values have been
pushed\n");
    while (!empty(myStack)) {
        int value;
        value = pop(myStack);
        printf("popped: %d\n", value);
    }
    printf("top: %d", top(myStack));
    free(myStack);
    return 0;
}
```

Figure 5 koden til at teste funktionerne i stack.c

```
Stack is empty
popped: -1
The numbers 123 and 99 have been pushed
top: 99

Three values have been pushed
popped: 4444
popped: 99
popped: 123
top: -1
```

Figure 6 udskriften fra programmet

Det kan ses at vi først testede empty(myStack)-funktionen på den nyoprettede stack. Udskriften viser at stack'en er tom. Herefter prøvede vi at poppe fra stack'en og får værdien -1 som funktionen pop(myStack) i stack.c giver når stack'en er tom. Herefter pushede vi to tal til stack'en og brugte funktionen top(myStack) som returnerede den øverste værdi. Dette gav som forventet det seneste tal der er blevet pushet, nemlig 99. Vi tilføjede nu endnu et tal og poppede så i et while-loop tallene fra stack'en indtil den er tom. Herefter testede vi funktionen top(myStack) på en tom stack hvilket gav det forventede resultat -1. Programmet

frigjorde så, med free(myStack), den hukommelse der havde været allokeret til stack'en og terminerede. Det er vigtigt at man husker at frigøre hukommelsen når man er færdig med at bruge den. Hvis man bliver ved med at allokere hukommelse og ikke frigør noget af det igen får man et såkaldt Memory Leak. Når et program på denne måde bliver ved med at allokere mere og mere hukommelse, kan det i sidste ende føre til et system crash.

Opgave 3

I denne opgave skulle man oprette en database over studerende der kunne håndtere navn, år for startet uddannelse, semester og antallet af GPA. Det specielle ved denne øvelse var dog ikke kun brugen af funktioner på den foreskrevne struct student_t, men at informationen angående startår, startsemester og karaktergennemsnit alle skulle gemmes i den samme int.

De fem første bits skulle således indeholde startåret, mens det femte bit angav semesteråret, hvor 0 og 1 angav hhv. efterår og forår. Bit 6-13 skulle indeholde karaktergennemsnittet, som var en værdi mellem 0-255.

Koden til indlæsningen af de forskellige data kan ses nedenfor. Bemærk at vi tjekker om den indtastede værdi er indenfor det ønskede interval samt bevirker at den ikke kærer kører i et uendeligt loop hvis man indtaster en streng i stedet.

```
do {
        puts("\nEnter start year (2009-2040):");
        scanf("%d", &temp);
        while(getchar() != '\n');
    } while(temp < 2009 || temp > 2040);

    (students+studentsSize)->data = (temp-2009) & 0x1F;

    do {
        puts("\nEnter start semester (0=Autumn/1=Spring):");
        scanf("%d", &temp);
        while(getchar() != '\n');
    } while(temp < 0 || temp > 1);

    (students+studentsSize)->data |= (temp & 0x1) << 5;

    do {
        puts("\nEnter GPA (0-255):");
        scanf("%d", &temp);
        while(getchar() != '\n');
    } while(temp < 0 || temp > 255);

    (students+studentsSize)->data |= (temp & 0xFF) << 6;
```

Figure 7 - koden der muliggøre indlæsningen af information om studerende

Navnet på den studerende skulle gemmes som en streng med længden 5, men da der også skal være plads til null-karakteren '\0', var den maksimale længde af et navn 4 bogstaver. Dette løste vi på følgende måde:

```
puts("\nEnter name (4 characters only):");
        scanf("%4s", (students+studentsSize)->name);
```

```
        while(getchar() != '\n');
```

Bemærk at den nederste linje sørger for at læse resten af strengen hvis der bliver indtastet mere end 4 bogstaver.

For at kunne printe alle informationerne lavede vi følgende funktion:

```
void printStudents(student_t * students, int size) {
    int i, avg = 0;
    for (i = 0; i < size; i++) {
        printf("s%04d: %s %d ", i, (students+i)->name, ((students+i)->data & 0x1F) +
2009);
        if ((students+i)->data & 0x20)
            printf("Spring");
        else
            printf("Autumn");
        printf(" %d\n", ((students+i)->data >> 6) & 0xFF);
        avg += ((students+i)->data >> 6) & 0xFF;
    }
    printf("\nAverage GPA = %.2f\n", (double)avg/(double)size);
}
```

Figure 8 - printStudent-koden

Derudover lavede vi en udvidelse til programmet, så dataene blev gemt i en tekst-fil ved navn ”database.txt”, så man dermed kunne gemme tidligere indtastninger, ligesom hvis det var en rigtig database. Man ville måske dog ikke have overskrevet filen hver gang hvis det var en meget stor database, som vi gør i vores program, men i dette tilfælde var det ikke noget problem, da databasen var forholdsvis lille. Bemærk at indlæsningen og det at skrive til en fil ikke fungerer på alle Windows computere.

Programmet er testet ved at tilføje de to studerende John og Mary til databasen og derefter at udskrive alle studerende. Udskriften af dette er vist på **Fejl! Henvisningskilde ikke fundet.**. Her giver programmet (udover vores egen tilføjelse) den udskrift der er specificeret i opgaveteksten. Vi har tilføjet at programmet skriver databasen til en fil og programmet giver derfor følgende information til brugeren før det terminerer ”Writing students to database”. Når programmet starter op, indlæser det databasen fra filen database.txt. Hvis denne fil eksisterer, udskriver det indholdet af filen. Dette er vist på **Fejl! Henvisningskilde ikke fundet.**, øverst. Her er 10 studerende i filen. På **Fejl! Henvisningskilde ikke fundet.**, nederst er der vist en test hvor diverse ugyldige input afprøves. Hvis navnet er længere end værdien af NAME_SIZE-1 er det kun de fire første bogstaver der bliver gemt. Ved år, semester og GPA sørger et while-loop for at programmet ikke går videre før brugeren har indtastet en gyldig værdi. Udskriften nederst på **Fejl! Henvisningskilde ikke fundet.** viser at programmet har forkastet de ugyldige værdier der var forsøgt indtastet.

```
Welcome to CUDB - The C University Data
Base
0: Halt
1: List all students
2: Add a new student
Enter action:2

Enter name (4 characters only):John

Enter start year (2009-2040):2009

Enter start semester (0=Autumn/1=Spring):0

Enter GPA (0-255):200

Enter action:2

Enter name (4 characters only):Mary

Enter start year (2009-2040):2040

Enter start semester (0=Autumn/1=Spring):1

Enter GPA (0-255):250

Enter action:1

s0000: John 2009 Autumn 200
s0001: Mary 2040 Spring 250

Average GPA = 225.00

Enter action:0
Writing students to database
Bye
```

Figure 9 - Test af CUDB

```
Existing students from database:
s0000: John 2009 Autumn 123
s0001: Mary 2010 Spring 234
s0002: Ben 2011 Autumn 128
s0003: Glen 2012 Autumn 234
s0004: Rob 2013 Spring 255
s0005: Lil 2014 Autumn 1
s0006: Marc 2015 Spring 128
s0007: Bo 2016 Autumn 99
s0008: Kris 2017 Spring 255
s0009: Bill 2018 Autumn 1

Average GPA = 145.80
```

```
Welcome to CUDB - The C University Data
Base
0: Halt
1: List all students
2: Add a new student
Enter action:2

Enter name (4 characters only):JamesBond

Enter start year (2009-2040):2050

Enter start year (2009-2040):2020

Enter start semester (0=Autumn/1=Spring):2

Enter start semester (0=Autumn/1=Spring):1

Enter GPA (0-255):512

Enter GPA (0-255):255

Enter action:1

s0000: Jame 2020 Spring 255

Average GPA = 255.00
```

Figure 10 - To forskellige test. Øverst en test af læsning fra en fil og nederst en test af ugyldige input