Opdrachten kwartaal 3 week 3 – Dynamisch geheugenallocatie

In de week 1 heb je een FIFO-buffer geïmplementeerd waarin een aantal getallen opgeslagen kan worden. Dit buffer heb je in week 2 in een embedded IoT-applicatie gebruikt om meetdata op te slaan als de wifi-verbinding tijdelijk verbroken is. De code voor de buffer is eerst, in week 1, op de pc ontwikkeld en getest voordat deze code, in week 2, in de embedded applicatie is gebruikt. Doordat de code voor de buffer in een aparte module (een `.h` en een `.c` bestand) is opgenomen en beheerd wordt in een Git submodule, is het eenvoudig om deze code in verschillende applicaties te gebruiken. Zolang de interface van de module (in `buffer.h`) niet veranderd wordt, is het mogelijk om de implementatie van de module (in `buffer.c`) te wijzigen zonder dat de applicaties aangepast hoeven te worden.

In de buffer die je de week 1 hebt geïmplementeerd kan een maximaal aantal getallen (8) opgeslagen worden. Als je meer getallen in de buffer wilt opslaan, dan moet je een (hopelijk kleine) aanpassing maken in `buffer.c` en het programma opnieuw compileren. In week 2 heb je de buffer aangepast zodat er maximaal 100 getallen in opgeslagen kunnen worden. Omdat het maximaal aantal getallen dat in de buffer opgeslagen kan worden, vast ligt na het compileren van de code, noemen we dit een *statisch* buffer. Als we te weinig ruimte voor de buffer reserveren, dan kan er data verloren gaan. Als we echter heel veel ruimte voor de buffer reserveren, hebben we veel RAM-geheugen nodig. Op de pc zal dit geen probleem zijn, maar op het embedded systeem zal de hoeveelheid RAM beperkt zijn.

Het zou natuurlijk veel mooier zijn als de buffer automatisch, tijdens het uitvoeren van de code, zou kunnen groeien en krimpen afhankelijk van de situatie. Dit kan gerealiseerd worden door gebruik te maken van *dynamische geheugenallocatie*. In de programmeertaal C kun je dynamisch geheugen alloceren met behulp van de standaardfunctie `malloc`[1]. Dynamisch gealloceerd geheugen kun je weer vrijgeven (voor hergebruik) met behulp van de standaardfunctie `free`[2].

Je leert deze les hoe je:

- dynamisch geheugen kunt alloceren en weer vrij kunt geven;
- een FIFO-buffer kunt implementeren die dynamisch groeit en krimpt met behulp van een *singly linked list*;
- *stubs* kunt gebruiken om te testen of jouw code goed werkt als het geheugen vol is en om te testen of jouw code het gealloceerde geheugen netjes vrijgeeft.

[1] Zie: https://en.cppreference.com/w/c/memory/malloc.

[2] Zie: https://en.cppreference.com/w/c/memory/free.

Om een dynamisch FIFO buffer te implementeren gaan we gebruik maken van een *singly linked list*. Dit is een datastructuur waarbij een aantal elementen (**struct**s) met behulp van pointers aan elkaar gekoppeld zijn.

We maken dus gebruik van de kennis over **struct**s[3] en pointers in C die je in EMS20 hebt opgedaan. Als je jouw kennis van deze onderwerpen wilt opfrissen, dan kun je de volgende EMS20 lessen terugkijken:

- week 1 les 2 “Pointers in C”;
- week 2 les 1 “Structs en enums”.

Je kunt een dynamische *singly linked list* aanmaken door elementen van het hieronder gegeven type `buffer_element` dynamisch aan te maken en via de `next` pointers aan elkaar te rijgen.

```
typedef struct buffer_element_tag {
    int value;
    struct buffer_element_tag *next;
} buffer_element;
```

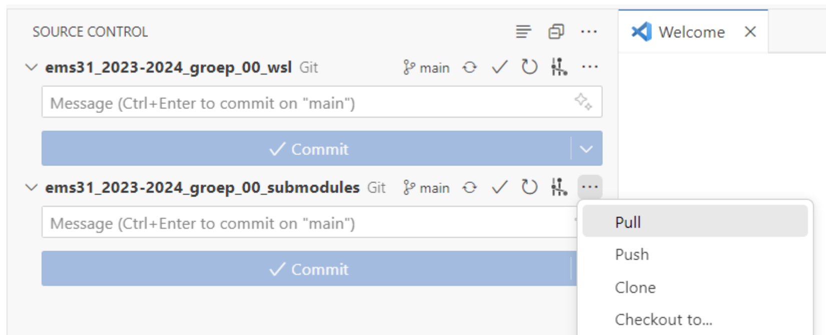
Als je dan een pointer naar het eerste element van de lijst genaamd `first` en een pointer naar het laatste element van de lijst genaamd `last` bijhoudt, dan kun je eenvoudig een FIFO-buffer implementeren. Bij een `buffer_put`-operatie moet een nieuw `buffer_element` aangemaakt worden met `malloc` en dit element moet achter het laatste element in de lijst geplaatst worden. De pointer `last` moet daarbij gebruikt en aangepast worden. Bij een `buffer_get`-operatie moet het eerste `buffer_element` losgemaakt worden uit de lijst en worden vrijgegeven met `free`. De pointer `first` moet daarbij gebruikt en aangepast worden. Bedenk dat er speciale acties nodig zijn als een element toegevoegd moet worden aan een lege lijst en ook als het allerlaatste element uit een lijst wordt verwijderd. De buffer is vol als `malloc` geen geheugen meer kan reserveren. De functie `malloc` geeft dan de waarde `NULL` terug.

3.3.1 Start Visual Studio Code en druk op F1 om het commando venster te openen. Type “WSL” en kies voor “WSL: Connect to WSL in New Window”. Klik op “Open Folder...” en open het directory `ems31_2023-2024_groep_XX_wsl`[4]. Open nu het Source Control window (druk op ctrl + ⇧ + G) en pull de veranderingen die je in week 2 hebt

[3] Zie voor een uitgebreide uitleg over **struct**s: https://bytebucket.org/HR_ELEKTRO/cprog/wiki/Dictaat-C_ebook.pdf#section.8.2.

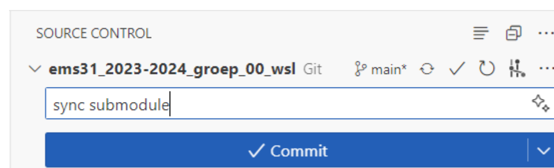
[4] Vul in plaats van `XX` het nummer van je groep in.

gemaakt in het repository ems31_2023-2024_groep_XX_submodules door op de "..." te klikken en te kiezen voor "Pull". Zie figuur 1.



Figuur 1: Pull de veranderingen die je in week 2 hebt gemaakt in het repository ems31_2023-2024_groep_XX_submodules.

3.3.2 Commit deze aanpassing nu ook in het repository ems31_2023-2024_groep_XX_wsl, zie figuur 2.



Figuur 2: Commit (en Sync) het repository ems31_2023-2024_groep_XX_wsl.

Push deze wijzigingen naar de remote repository door op "Sync Changes 1" te klikken.

3.3.3 Open een terminal window in Visual Studio Code door op ctrl + ' te drukken. Ga naar het directory ems31_2023-2024_groep_XX_submodules en kopieer het subdirectory buffer_8 naar een nieuwe directory buffer_dyn door de volgende commando's in te typen:

```
cd ems31_2023-2024_groep_XX_submodules/
cp -r buffer_8 buffer_dyn
cd ..
```

Kopieer nu het directory buffer_8_test naar het directory buffer_dyn, verwijder het directory build uit het nieuwe directory buffer_dyn en start Visual Studio Code in dit directory door de volgende commando's in te typen:

```
cp -r buffer_8_test/ buffer_dyn_test
cd buffer_dyn_test
rm -r build
code -r .
```

Hernoem het bestand `test_buffer_8.cpp` naar `test_buffer_dyn.cpp` en open dit bestand door de volgende commando's in te typen:

```
mv test_buffer_8.cpp test_buffer_dyn.cpp
code test_buffer_dyn.cpp
```

Verander op regel 5 `buffer_8` in `buffer_dyn` zodat de header file uit de module `buffer_dyn` gebruikt wordt en sla het bestand op.

Open nu het bestand `CMakeLists.txt` en verander elke `buffer_8` in `buffer_dyn`. Dit kun je het eenvoudigst doen met behulp van de menuoptie Edit > Replace of door op ctrl + H te drukken. Sla het gewijzigde bestand op. CMake zal nu automatisch het project builden.

Als je de testen runt, zal er nu waarschijnlijk een falende test zijn omdat je het maximaal aantal getallen dat in de buffer opgeslagen kan worden, hebt veranderd van 8 naar 100. Pas de testcode aan zodat alle testen weer slagen.

3.3.4 Commit en Sync alle veranderingen in beide repositories.

3.3.5 Breid de testcode in het bestand `test_buffer_dyn.cpp` uit met een test waarmee er honderdtien getallen in de buffer opgeslagen worden. Voer deze test uit. De nieuw toegevoegde test zal uiteraard falen omdat het huidige buffer, na de aanpassing in week 2, slechts maximaal honderd integers kan bevatten.

Pas de testcode in het bestand `test_buffer_dyn.cpp` aan, zodat de test die het hele buffer volschrijft met de waarde 13 niet meer uitgevoerd wordt. Dit om te voorkomen dat het hele RAM en virtuele geheugen[5] van je pc volledig wordt opgevuld als je de dynamische buffer hebt geïmplementeerd.

3.3.6 Pas nu de code in het bestand `../ems31_2023-2024_groep_XX_submodules/buffer_dyn/buffer.c` aan zodat er een onbepaald aantal getallen in de buffer opgeslagen

[5] Zie eventueel https://nl.wikipedia.org/wiki/Virtueel_geheugen.

kunnen worden. Het geheugen moet dynamisch worden aangemaakt als een getal in de buffer wordt geplaatst en ook weer netjes vrijgegeven worden als een getal uit de buffer wordt verwijderd. De buffer moet werken als een FIFO-buffer (First In First Out). De buffer is pas vol als er geen geheugen meer beschikbaar is. Maak daarbij gebruik van een singly linked list zoals hierboven besproken. Het is niet de bedoeling dat het bestand `buffer.h` wordt aangepast, de interface van de buffer hoeft namelijk niet gewijzigd te worden. Door de interface niet te wijzigen zorg je ervoor dat de applicatiecode die gebruik maakt van deze buffer (bijvoorbeeld `temperatuur_logger`) niet gewijzigd hoeft te worden. Vergeet niet om alle wijzigingen (regelmatig) te commiten en te pushen naar de remote repositories.

3.3.7 Je wilt natuurlijk ook testen of jouw implementatie goed werkt als het geheugen vol is. Dit is op de pc niet zo gemakkelijk omdat er zoveel RAM-geheugen beschikbaar is. Bovendien wordt op de pc zogenoemd virtueel geheugen gebruikt als het RAM-vol is. Een deel van de inhoud van het RAM wordt dan verplaatst naar het achtergrondgeheugen (SSD[6] of HDD[7]) als het RAM vol is. Dit wordt dan later weer teruggezet als dit nodig is, waarbij ruimte wordt gemaakt door de inhoud van een deel van het RAM dat momenteel niet nodig is naar het achtergrondgeheugen te kopiëren.

Om toch te kunnen testen of jouw code goed werkt als het geheugen vol is, gaan we een zogenoemde '*stub*' gebruiken. Een stub[8] is een functie die bij het testen gebruikt wordt om een bepaalde functie te vervangen zodat de code die gebruikt maakt van deze bepaalde functie beter getest kan worden. Een functie die nog niet geïmplementeerd is, kan bijvoorbeeld vervangen worden door een stub. Maar een stub kan ook gebruikt worden om een functie een bepaalde waarde terug te laten geven die je nodig hebt om een bepaalde test uit te voeren.

Om te testen of je code goed werkt als het geheugen vol is, gaan we een functie `test_malloc` definiëren die bijvoorbeeld de eerste drie keer gewoon `malloc` aanroept, maar de vierde keer `NULL` teruggeeft. We willen de code in het directory `buffer_dyn` niet aanpassen want dat is de juiste implementatie van een dynamisch buffer.

[6] Zie eventueel https://nl.wikipedia.org/wiki/Solid_state_drive.

[7] Zie eventueel https://nl.wikipedia.org/wiki/Harde_schijf.

[8] Zie eventueel https://nl.wikipedia.org/wiki/Stub.

We gaan daarom een nieuwe directory buffer_dyn_test_out_of_memory aanmaken waarin we een kopie van het bestand buffer.c plaatsen en die aanpassen zodat de functie test_malloc wordt gebruikt in plaats van malloc.

Voer de volgende commando's uit in de WSL terminal[9]:

```
cd ~/ems31_2023-2024_groep_XX_wsl
cp -r buffer_dyn_test/ buffer_dyn_test_out_of_memory
cp ems31_2023-2024_groep_00_submodules/buffer_dyn/buffer.* ↩
  ↪ buffer_dyn_test_out_of_memory/
cd buffer_dyn_test_out_of_memory/
rm -r build
code -r .
```

Voeg de volgende code toe aan het bestand buffer.c:

```
static int malloc_count = 0;

// test_malloc returns NULL after three successful allocations
// test_malloc counts the number of calls
void* test_malloc(size_t size)
{
    if (malloc_count == 3)
    {
        return NULL;
    }
    malloc_count++;
    return malloc(size);
}
```

Vervang elke aanroep naar malloc met een aanroep naar test_malloc.

Vervang alle code in het bestand test_buffer_dyn met de volgende testcode:

```
#include <gtest/gtest.h>
// Include C code in C++ bestand
extern "C"
{
    #include "buffer.h"
}
```

[9] Vervang XX door het nummer van je groep.

```
TEST(buffer, out_of_memory)
{
    // write 0, 1, and 2 into the buffer
    for (int i = 0; i < 3; i++)
    {
        EXPECT_TRUE(buffer_put(i));
    }
    // buffer should be full now
    EXPECT_TRUE(buffer_is_full());
    // try to write 3 into the buffer
    EXPECT_FALSE(buffer_put(3));
    // retrieve all values from the buffer
    int retrieved_value;
    for (int i = 0; i < 3; i++)
    {
        EXPECT_TRUE(buffer_get(&retrieved_value));
        EXPECT_EQ(retrieved_value, i);
    }
    // buffer should be empty now
    EXPECT_TRUE(buffer_is_empty());
    // try to retrieve a value from the empty buffer
    EXPECT_FALSE(buffer_get(&retrieved_value));
}
```

Build het project, voer de test uit en zorg ervoor dat de test slaagt.

Als je fouten gevonden hebt in jouw implementatie van het dynamisch buffer, verbeter dan ook de code in het directory `buffer_dyn`.

Vergeet niet om alle wijzigingen (regelmatig) te commiten en te pushen naar de remote repositories.

3.3.8 Je wilt natuurlijk ook testen dat het dynamisch gealloceerd geheugen ook weer correct wordt vrijgegeven. Dit is nog niet zo eenvoudig. Om te testen of je code het gealloceerde geheugen netjes vrijgeeft, gaan we een stub `test_free` definiëren

De functie `test_malloc` houdt al bij hoe vaak die wordt aangeroepen. Als `test_free` dat ook doet, dan kunnen we checken of, als de buffer weer leeg is, `test_free` net zo vaak is aangeroepen als `test_malloc`.

Voeg de volgende code toe aan het bestand buffer.c:

```
static int free_count = 0;

// test_free counts the number of calls
static void test_free(void *p)
{
    free_count++;
    free(p);
}

// returns true if the number of calls to malloc and free are ↩
  ↪ equal
bool frees_equals_mallocs(void)
{
    return malloc_count == free_count;
}
```

Vervang elke aanroep naar free met een aanroep naar test_free.

Voeg onderin het bestand test_buffer_dyn de volgende testcode toe:

```
extern "C"
{
    bool frees_equals_mallocs(void);
}

TEST(buffer, no_memory_leak)
{
    // check for memory leaks
    EXPECT_TRUE(frees_equals_mallocs());
}
```

Build het project, voer de testen uit en zorg ervoor dat alle testen slagen.

Als je fouten gevonden hebt in jouw implementatie van het dynamisch buffer, verbeter dan ook de code in het directory buffer_dyn.

Vergeet niet om alle wijzigingen (regelmatig) te commiten en te pushen naar de remote repositories.

3.3.9 Test de code in het directory `buffer_dyn` ook op de CC3220S LaunchPad. Door in het project `temperatuur_logger_6_10` het statische buffer te vervangen door een dynamisch buffer.

Begin met het pullen van de Git submodule en pas daarna de verwijzingen naar `buffer.h` en `buffer.c` aan.

Vergeet niet om alle wijzigingen (regelmatig) te commiten en te pushen naar de remote repositories.