Wiki »

Exercise 8 - Resource Handling ¶

Exercise 8 - Resource Handling
- Motivation
- Exercise 1 - Ensuring garbage collection on a dynamically allocated std::string
  - Implementing
  - Test
  - Diskussion
- Exercise 2 - The Counted Pointer
  - Implementing
  - Test
  - Diskussion
- Exercise 4 - Discarding our solution in favor of boost::shared_ptr<>
  - Implementing
  - Test
  - Diskussion
- Exercise 5 - Resource Handling

Motivation

Ved større programmings projekter hvor kompleksiteten og mængden af kode stiger, kan overblikket for måske simple ting nemt forsinde. En af de ting kan nemt være, at glemme er at "frigive" eller ihvertfald holde styr alle de ressourcer man tidligere har brugt. Det kan også være at release-funktionen aldrig bliver kaldt, hvis programmet feks. forlade scope ved en return eller exception. Og det er her Resource Handling kommer ind i billedet, for at gøre livet nemmere for de hårdetarbejdende programmører, som netop sikrer at alt den allokerede resource og bliver frigivet når det ikke skal bruges mere.

Exercise 1 - Ensuring garbage collection on a dynamically allocated std::string

Til at starte bruges en kombiantion af RAII og SmartPointer idioms til at færdiggøre klassen *SmartString*.

Implementing

SmartString.hpp

```
#ifndef SMART_STRING_HPP
#define SMART_STRING_HPP

#include <string>

class SmartString
{
public:
    explicit SmartString(std::string* str):str_(str) {}
    ~SmartString(){delete str_;}
    std::string* get(){return str_;}
    std::string* operator->(){return str_;}
    std::string& operator*(){return *str_;}

private:
```

```
    SmartString(const SmartString&);
    SmartString& operator=(const SmartString&);
    std::string* str_;
};

#endif
```

Test

Til at teste ovenstående implementering *SmartString.hpp* bruges udleverede tesprogram, og der opnås følgende resultat:

```
stud@stud-virtual-machine:~/Desktop/Courses/i3isu_f2020_beany_business/exe8$ ./bin/host/prog
String length: 11
The 'ss' string does contain 'world'
```

Diskussion

Resultatet fra testprogemmet viser at implementering virker efter hensigten.

Copy og assignment operatorerne er private i SmartString klassen.
Dette skyldes at vi ikke ønsker at klassen kan kopieres, altså at den eneste måde vi kan oprette et objekt af typen påp
er via. et constructor kald.

Dette skyldes at destructoren sletter den allokerede resource. Hvis man forestillede sig at objektet blev kopieret, og kopien blev slettet(destructor kald).
Dette ville resultere i at resourcen der benyttes i begge objekter blev slettet blot fordi det ene objekt ophørte med at eksistere.
Dette er problematisk, i sær fordi man ikke ville vide at ens resource var slettet i det resterende objekt.

Derfor er en resource knyttet til et enkelt objekt. Og dette kan altså ikke kopieres.

-> og * operatoren er overloadet for smartstring klassen.
Disse er overloadet således, at de ikke returnerer en pointer/derefereret variabel til objektet. I stedet returneres en pointer/dereferet pointer til den resource som objektet indeholder, altså vores string.
Derved kan der kaldes funktioner på vores string vha. -> operatoren. Desuden kan der kaldes funktioner i smartstring klassen ved . operatoren.
Ved denne løsning kan objektet tilgåes som hvis det var en regulær string-pointer, men samtidig fås den sikkerhed for cleanup, der er implementeret i SmartString klassens destructor.

Exercise 2 - The Counted Pointer

I denne opgave udvides klassen *SmartString*, så flere kan have en reference til den.

Implementing

SmartString.hpp

```
#ifndef SMART_STRING_HPP
#define SMART_STRING_HPP

#include <string>

class SmartString
{
```

```
public:
    explicit SmartString(std::string* str):str_(str) {count_=new int(1);}
    ~SmartString(){
        if(--*count_==0)
        {
            delete str_;
            delete count_;
        }
        }
    std::string* get(){return str_;}
    std::string* operator->(){return str_;}
    std::string& operator*(){return *str_;}
    SmartString(const SmartString& other)
    {
        count_=other.count_;
        str_=other.str_;
        ++*count_;

    }
    SmartString& operator=(const SmartString& other)
    {
        count_=other.count_;
        str_=other.str_;
        ++*count_;
    }
    int getCount()const
    {
        return *count_;
    }

private:
    int *count_;
    std::string* str_;
};

#endif
```

Test

Denne gang udgives testprogrammet så

main.cpp

```
#include "SmartString.hpp"
#include <iostream>

int main(int argc, char* argv[])
{
    SmartString ss(new std::string("Hello world"));
    std::cout << "String length: " << ss->length()<<std::endl;
    if(ss->find("world")!=std::string::npos)
        std::cout<<"The 'ss' string does contain 'world'"<<std::endl;
    else
    {
        std::cout<<"The 'ss' string does NOT contain 'world' "<<std::endl;

    }
```

```
        std::cout<<"antal brugere af resource "<<ss.getCount()<<std::endl;
    {
    SmartString ss2=ss;
    SmartString ss3(ss2);
    std::cout<<*ss2<<std::endl;
    std::cout<<*ss3<<std::endl;
    std::cout<<"antal brugere af resource "<<ss2.getCount()<<std::endl;
    }
    std::cout<<"antal brugere af resource "<<ss.getCount()<<std::endl;
}
```

Resultat af testen:

```
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe8/exe2$ ./bin/host/prog
String length: 11
The 'ss' string does contain 'world'
antal brugere af resource 1
Hello world
Hello world
antal brugere af resource 3
antal brugere af resource 1
```

Diskussion

Copy og assignment operatorerne er private i SmartString klassen.
Dette skyldes at vi ikke ønsker at klassen kan kopieres, altså at den eneste måde vi kan oprette et objekt af typen påp
er via. et constructor kald.

Dette skyldes at destructoren sletter den allokerede resource. Hvis man forestillede sig at objektet blev kopieret, og kopien blev slettet(destructor kald).
Dette ville resultere i at resourcen der benyttes i begge objekter blev slettet blot fordi det ene objekt ophørte med at eksistere.
Dette er problematisk, i sær fordi man ikke ville vide at ens resource var slettet i det resterende objekt.

Derfor er en resource knyttet til et enkelt objekt. Og dette kan altså ikke kopieres.

-> og * operatoren er overloadet for smartstring klassen.
Disse er overloadet således, at de ikke returnerer en pointer/derefereret variabel til objektet. I stedet returneres en pointer/dereferet pointer til den resource som objektet indeholder, altså vores string.
Derved kan der kaldes funktioner på vores string vha. -> operatoren. Desuden kan der kaldes funktioner i smartstring klassen ved . operatoren.
Ved denne løsning kan objektet tilgåes som hvis det var en regulær string-pointer, men samtidig fås den sikkerhed for cleanup, der er implementeret i SmartString klassens destructor.

Grunden til at counteren er dynamisk allokeret i constructeren, skyldes at den skal eksistere ligeså længe som resourcen. Dette implementeres ved at de begge oprettes på heap og gemmes (resourcen inden construktor kald, og counteren i constructor kaldet).
Derefter slettes de så begge i destructoren hvis der ikke er flere objekter der gør krav på den delte resource. (altså hvis counteren er 0).

Copy og assignment operatorerne implementeres således at de hver især kopierer pointeren til count og resourcen, og derefter inkrementerer count. Derved kan det ses at endnu et objekt benytter resourcen.
I destructoren dekrementeres count, og hvis denne når ned på 0, deallokeres både count og resourcen. Da resourcen ikke længere er i brug, og vi derfor skal lave cleanup.

Exercise 4 - Discarding our solution in favor of boost::shared_ptr<>

Implementing

I denne opgave har vi erstattet vores egen smartpointer med den allerede implementerede smart pointer fra boost biblioteket. Herefter har vi testet den, både for at lære den bedre at kende, men også for at teste om den kan anvendes på samme måde som vores egen klasse.
Anvendelsen, bortset fra at man selvfølgelig ikke skal implementere boost klassen selv, er meget ens. Den største forskel er at vores egen klasse kun kunne bruges på strings, hvor boost klassen kan bruges på mange datatyper. Derfor skal datatypen specificeres når denne initialiseres.

main.cpp

```
#include <iostream>
#include <boost/shared_ptr.hpp>

int main(int argc, char* argv[])
{
    boost::shared_ptr<std::string> ss(new std::string("Hello world"));
    std::cout << "String length: " << ss->length()<<std::endl;
    if(ss->find("world")!=std::string::npos)
        std::cout<<"The 'ss' string does contain 'world'"<<std::endl;
    else
    {
        std::cout<<"The 'ss' string does NOT contain 'world' "<<std::endl;

    }
        std::cout<<"antal brugere af resource "<<ss.use_count()<<std::endl;
    {
    boost::shared_ptr<std::string> ss2=ss;
    boost::shared_ptr<std::string> ss3(ss2);
    std::cout<<*ss2<<std::endl;
    std::cout<<*ss3<<std::endl;
    std::cout<<"antal brugere af resource "<<ss2.use_count()<<std::endl;
    }
    std::cout<<"antal brugere af resource "<<ss.use_count()<<std::endl;
}
```

Test

For at teste brugte vi samme program som i opgave 2 og printede derefter resultatet.

```
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe8/exe4$ ./bin/host/prog
String length: 11
The 'ss' string does contain 'world'
antal brugere af resource 1
Hello world
Hello world
antal brugere af resource 3
antal brugere af resource 1
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe8/exe4$
```

Resultatet blev, som forventet, identisk med det resultat vi fik da vi brugte vores egen klasse og dermed også det korrekte resultat.

Diskussion

Når man har lært at bruge smart pointers, f.eks. ved selv at implementere en klasse med funktionaliteten, giver det rigtig god mening at bruge boost biblioteket.
Boost biblioteket tilbyder samme funktionalitet og mere til. Samtidig er boost biblioteket gennemtestet, grundigt supporteret og peer-reviewet. Ift. kodestandarder kan man ikke opnå en højere standard, og derfor skal vi

selvfølgelig bruge dette bibliotek fremfor egen kode. Det er også væsentlig mere effektivt da vi slipper for implementering og debugging.

Exercise 5 - Resource Handling

En ressource kan være stor set alt der kan være begrænset mængde af i IT. Der er de åbenlyse som memory og CPU når man snakke software man skal selvfølgelig også tænker over plads på harddrive og måske kapacitet til grafiske udregninger.
Når man til gengæld bevæger sig lidt 'dybere' og ser bort fra de hardware-afhængige ressourcer, skal der overvejes ting som mutexes og semaphores, måske kan det endda være nødvendigt at tage højde for databaseadgang og om et program skal lukkes ellers holdes åbent.

Ressourcer er ekstremt vigtige at have styr på når vi snakker software. Uanset hvilken platform og til hvilket formål, vil vi altid gerne udnytte vores ressourcer bedst muligt. Der kan selvfølgelig være steder man prioriterer at lave sikker kode, fremfor hurtigt, eller tidspunkter hvor en deadline kan gøre det svært at levere kode der håndterer memory optimalt. Dette er også grunden til at der udvikles koncepter som RAII og biblioteker som boost, på den måde kan vi både hurtigere og mere effektivt udvikle software som levere på ydelse og er optimeret.

Der kan selvfølgelig opstå problemer med ressourcer, f.eks. hvis man gerne vil dynamisk initialisere noget data. Når dette data ikke skal burges mere, skal man huske at bruge funktionen *delete* på dette data. For ellers vil der opstå et memory leak, og bruge unødig plads.

Der kan også ske problemer, hvis man eksplicit vil synkronisere en del af sin kode ved at oprette en *std::mutex* og heri bruger dot-operatoren *.lock()*. Umiddelbart ser det også fint ud og vil fungere, men kun hvis *.unlock()* også vil blive kaldt. Eller hvis der sker en *exception* før at mutex'en bliver låst op igen, for så vil den anden tråd gå hen i deadlock når den prøver at modtage låsen.

I forhold til tåde kan der også opstå problemstillinger, hvis der ønskes at køre noget kode i en anden tråd. Hvilket også går fint så længe man husker at bruge *join* i tråden, for hvis ikke, når tråden går ud af scope vil *terminate* blive kaldt og programmet lukkes.
Alt dette vil RAII kunne hjælpe med og herved holde styr på det for programmørerne, og herved undgår menneskelige fejl, som vil ske.

Når man anvender memory ressourcer kan man undgå at anvende dynamisk allokering hvis man compile-time, ved præcis hvad der skal bruges af ressourcer. Dette gør allokering simplere, da der ikke kan opstå memory leaks i statisk allokerede objekter. Man kan ikke altid undgå dynamisk allokering og det kan være et rigtig godt værktøj, derfor er det vigtigt at vi lærer at bruge det korrekt. Nogle gange er det et must at vi bruges dynamisk allokering da, dette f.eks. kan sikre at memory allokeres på heapen, i stedet for stacken. Når vi har delte ressourcer mellem forskellige tråde eller objekter, skal disse alokeres på heapen, da de ellers ville blive slettet hvis vi går ud af scope. Et godt eksempel er shared_ptr, her skal det objekt der peges på, være dynamisk allokeret da vi gerne vil bevare objektet, selv hvis én ud af flere mulige pointers bliver slettet.

- **Tilføjet af Mikkel Høj Hansen for cirka en måned siden**

  1: Benytte smartpointer idiom / RAII til at rydde op efter brug af strings.

  Gruppen benytter henholdsvis RAII og smartpointer til at rydde op efter der er brugt string. Først er det gjort ved "delete" i destructoren.
  Der er dog ikke ryddet op i overload af assignment operatoren. Her bliver counteren kun talt op. Hvis counteren rammer 0, skal den string hvis værdi ændres slettes, og desuden skal counteren decrementeres i dette tilfælde, da den instans der lå i variablen inden den sættes lig en ny værdi, ikke længere er i brug.

  2: Benytte en counter til at holde styr på hvor mange der benytter en instans af en string.

Samme pointe som delmål 1 - der ryddes ikke op i assignment operatoren.
Derudover benytter gruppen en counter til at holde styr på hvor mange instanser af smartstring der er i brug, og denne incrementeres og decrementeres når der oprettes og slettes strings.

3: Have forståelse til at implementere tidligere opgaver, nu med boost-library.

Gruppen løser opgaven fint ved brug af boost library. Samme test køres, hvor der benyttes scope til at teste om oprydning af strings fungerer som forventet. De korrekte værdier bliver printet.

4: Kunne forklare brug af resource handling.

Der er svaret særdeles fyldestgørende på spørgsmålene vedrørende resource handling, og gruppen udviser fin forståelse af smartstrings og resource handling generelt.

Must haves:

Der er relevante kodefiler i repository.
Der er makefiles til alle opgave i repository.

BOTTOM LINE:

rigtig fin opgave der viser god forståelse for resource handling - blot med en enkelt lille fejl i assignment operatoren.

Opgaven vurderes til at være OK.

opg1.png (11,7 KB) Magnus Nygaard Lund, 2020-04-27 12:18

exe4boost.png (45,3 KB) Joachim Krøyer Leth-Jørgensen, 2020-04-27 14:02

opg2.png (17,5 KB) Magnus Nygaard Lund, 2020-04-28 08:40