Part II: Skriftlig-svar (kortsvar) spørsmål (60 poeng) - LF

Oppgave 1 og 2 - Pekere og referanser (10 poeng totalt) - uttrekk, 2 oppgaver til hver

1. (5 points) Hva er fordelen med `std::unique_ptr` sammenlignet med vanlig peker? Forklar kort.

> **Solution:** Sikrer mot minnelekasje ved at man ikke må allokere og deallokere det dynamiske minnet.
>
> (s. 703 – 704)

2. (5 points) Hva er fordelen med peker sammenlignet med referanse? Forklar kort.

> **Solution:** Kan peke på ingenting og endre hva den peker på. Kan i motsetning til referanser brukes i forbindelse med tabeller.
>
> (s. 610 – 612 + s. 650)

3. (5 points) Hva er fordelen med referanse sammenlignet med peker? Forklar kort.

> **Solution:** Er sikrere og bør derfor velges om man har muligheten. Kan i motsetning til pekere brukes ved operatoroverlasting.
>
> (s. 610 – 612 + s. 1104 – 1105)

4. (5 points) Hvilke fordeler er det med smartpekere? Forklar kort.

> **Solution:** Holder styr på sitt eget minne og rydder opp minneallokeringen automatisk ved å bli ødelagt når de går utenfor skopet slik som objekter, så vi ikke trenger å eksplisitt kalle delete på objektet/variabelen.
>
> (s. 703 – 704 + s. 709)

5. (5 points) Hvilke fordeler er det med konstante referanser? Forklar kort.

> **Solution:** Det beskytter koden mot feilaktige endringer. Dette bidrar til å gjøre koden tryggere/sikrere, og gjør det lettere å fange opp en del feil tidlig.
>
> (Mindre viktig: potensielt raskere og mer lesbar kode, men dette vil også avhenge av en del andre faktorer og er noe mer diskuterbart. I tillegg slipper vi unødvendig kopiering av variabler, akkurat som for vanlige referanser.)

Kommentar: Studenten skal ikke ha trekk for å ha glemt noe av det som her står som mindre viktig. Dersom studenten har glemt det viktige, men har med noe av det mindre viktige, bør studenten få noe uttelling.

(s. 276 – 278)

Oppgave 3 - Minnelekkasje (10 poeng) - uttrekk, 1 oppgave til hver (s. 588 - 600)

6. (10 points) Se på klassedeklarasjonene under.

```
class Node {
    int nodeValue;
    Node * nextNode = nullptr;

public:
    Node(int nodeValue) : nodeValue{nodeValue} {}

    void addNode(int nodeValue) {
        nextNode = new Node(nodeValue);
    }
    Node * getNextNode(){return nextNode;}
    int returnValue(){return nodeValue;}

    ~Node() {
        if(nextNode) delete nextNode;
    }
};
```

(a) (5 points) Hvorfor risikerer koden minnelekkasje?

Solution: Det er mulig å lage en annen `nextNode` til en node som allerede har en `nextNode` (ved hjelp av `addNode()`-funksjonen), slik at den første `nextNode` aldri blir slettet.

(s. 588 – 600 + s. 779 – 782)

(b) (5 points) Hvordan kan problemet løses?

Solution: Her er det flere muligheter. Det er ikke krav om å komme med flere mulige løsninger for å få full uttelling på oppgaven - ett korrekt svar holder! Mulige løsninger:

- Bruke `unique_ptr` for nextNode.
- Frigjøre minnet i addNode() før en ny nextNode lages.
- Kun opprette en nextNode i addNode() dersom noden *ikke* har en nextNode fra før av.
- Lage en setNodeValue()-funksjon, og bruke denne til å oppdatere verdien til nextNode i addNode() i stedet for å opprette en ny node dersom nextNode allerede finnes.
- Godt mulig det finnes andre fornuftige løsninger også.

(s. 588 – 600 + s. 703 – 704 + s. 779 – 782)

7. (10 points) Se på funksjonen `mess` under.

```
1  void mess() {
2      vector<int*> v;
3      for(int i = 0; i < 10; ++i) {
4          auto u = make_unique<int>(i);
5          auto p{u.get()};
6          v.push_back(p);
7
8          for (int j = 0; j < *p; ++j) {
9              v.push_back(new (int)(j+i));
10         }
11     }
12     for(int i = 0; i < v.size(); i++) {
13         cout << *v[i] << endl;
14     }
15 }
```

Hvilke to større problemer har funksjonen?

Solution:

1: Den unike pekeren `u` går ut av skopet ved linje 11 og blir slettet. Dermed blir innholdet av pekeren `p` slettet, og man får udefinert oppførsel når man prøver å skrive innholdet til skjerm på linje 13.

2: Minnelekkasje for elementene som legges til i `v` på linje 9. (Det brukes `new` uten `delete`.)

(s. 588 – 600 + s. 703 – 704)

Oppgave 4 og 5 - Hva er problemet i koden? (20 poeng) - uttrekk, alle får 2 oppgaver

8. (10 points) Se på klassedeklarasjonene under.

```
1  #include <iostream>
2  using namespace std;
3
4  struct Person {
5  private:
6    string name;
7    int age;
8    string address;
9
10 public:
11   Person(string name, int age, string address)
12     : name{name}, age{age}, address{address} {}
13   string getName() { return name; }
```

```
14    const int getAge() { return age; }
15    string getAddress() const { return address; }
16  };
17
18  int main() {
19    const Person ola{"Ola", 50, "ola@nordmann.no"};
20    cout << "Name: " << ola.getName() << endl;
21    cout << "Age: " << ola.getAge() << endl;
22    cout << "Email: " << ola.getAddress() << endl;
23    return 0;
24  }
```

(a) (5 points) Hva er problemet med koden? Forklar kort.

> **Solution:** Medlemsfunksjonene `getName()` og `getAge()` er ikke konstante. Dette er et problem siden de blir forsøkt brukt på det konstante objektet `Ola`.
>
> (s. 330 – 332)

(b) (5 points) Hvilke (en eller flere) endringer må til i klassedeklarasjonen for at koden i `main()` skal kjøre som forventet? Forklar kort og skriv hvilke kodelinjer du ønsker å endre, og hva du vil endre dem til.

> **Solution:** Oppdatere medlemsfunksjonene `getName()` og `getAge()` til å bli `const` ved å skrive på linje 13 og 14:
>
> ```
> string getName() const { return name; }
> const int getAge() const { return age; }
> ```
>
> (s. 330 – 332 + s. 1110)

9. (10 points) Se på klassen under.

```
1  //This code is in the file Banana.h
2  #pragma once
3  #include "std_lib_facilities.h"
4
5  class Banana {
6      int sweetness = 70;
7      int weight = 100;
8      bool isRipe = true;
9  public:
10     friend ostream& operator<<(ostream& os, const Banana & b);
11 };
```

```
1  //This code is in the file Banana.cpp
2  #include "Banana.h"
3  ostream& Banana::operator<<(ostream& os, const Banana & b){
4          os << b.weight << " grams banana with " << b.sweetness  << ".";
5          if(b.isRipe) os << " Ready to be eaten!";
6          return os;
7  }
```

(a) (5 points) Hva er problemet med klassen? Forklar kort.

Solution: Operatoren er ikke en medlemsfunksjon til klassen fordi den er deklarert som en `friend`. Det blir derfor feil å skrive `Banana::` foran `operator<<` på linje 3 i `Banana.cpp`.

(s. 308 – 317 + s. 327 – 330 + s. 346 – 348 + s. 1088 + s. 1111)

(b) (5 points) Hvordan kan problemet med denne koden minimum løses? (Mao: Hva er den enkleste måten å løse dette problemet på?)

Solution: Fjern `Banana::` fra linje 3 i `Banana.cpp`, slik at linje 3 ser slik ut:

```
ostream& operator<<(ostream& os, const Banana & b)
```

Forslag om å fjerne `Banana::` fra linje 3 i `Banana.cpp`, og legge hele operatoroverlastingen utenfor klassen (altså ikke `friend`), for deretter å lage get-funksjoner for å få tak i de private medlemsvariablene bør også gi en del uttelling. Det viktigste her er at studentene har fått med seg at en `friend`-funksjon *ikke* er en medlemsfunksjon, og at `operator<<` **aldri** kan være en medlemsfunksjon siden den tar inn en `ostream&` som første argument og alle medlemsfunksjoner implisitt tar inn et objekt av klassen som første argument.

(s. 308 – 317 + s. 327 – 330 + s. 346 – 348 + s. 1088 + s. 1111)

10. (10 points) Se på klassen under.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  class Person {
6  private:
7      string name;
8      int age;
9  public:
10     Person(string name, int age) : name{name}, age{age} {};
11 };
12
13 int main() {
14     vector<Person> persons(2);
15     persons[0] = Person{"Roar", 35};
16     persons[1] = Person{"Kari", 19};
17     return 0;
18 }
```

(a) (5 points) Hva er problemet med klassen? Forklar kort.

Solution: Standardkonstruktøren/default-konstruktøren, som kalles når man initialiserer en vektor med en fikset størrelse og ingen argumenter, eksisterer ikke.

(s. 327 – 330)

(b) (5 points) Hvordan kan problemet med denne koden minimum løses? Forklar kort, og skriv forslag til kode som fikser dette. NB! Skriv kun hva du vil legge til/fjerne/endre og hvor i koden dette er. Du trenger ikke å skrive inn hele klassen på nytt.

Solution: Lag en standardkonstruktør/default-konstruktør på linje 10. F.eks. ved å skrive:

```
Person() {};
```

NB! Andre varianter av default-konstruktør (f.eks. en som setter et defaultnavn + alder, en som er satt =default etc) godkjennes også.

Alternativt kunne man endret på hvordan vectoren opprettes og brukes i `main()`, slik at det ikke er behov for en default-konstruktør. Det kan da se slik ut:

```
int main() {
    vector<Person> persons;
    persons.push_back(Person{"Roar", 35});
    persons.push_back(Person{"Kari", 19});
}
```

(s. 327 – 330)

Oppgave 6 og 7 - Forklar kort (10 poeng) - uttrekk, 2 oppgaver til hver

11. (5 points) Forklar kort hvorfor det er lurt å bruke vektorfunksjonen `.at()` i stedet for tabellindeksoperatoren, altså `operator[]`, dersom man vil aksessere et element i en vektor.

Solution: Fordi vektorfunksjonen `.at()` fungerer som et sikkerhetsnett ved å gi en kjøretidsfeil hvis vi indekserer utenfor vektorens grenser.

(s. 693 – 694)

12. (5 points) Forklar kort hvorfor det kan være nyttig å returnere en referanse til et objekt i implementasjon av tilordningsoperatoren.

Solution: Å returnere referansen til venstresiden gjør det mulig å kjede/lenke sammen tilordninger (`multi-assignment`), slik som `a = b = c` eller `(a += 1) += 2`.

(s. 179 – 186 + s. 1089 – 1090)

13. (5 points) Forklar kort hvorfor det er lurt å bruke `override` når man overlaster virtuelle funksjoner i subklasser.

Solution: `override` sikrer at funksjonsdeklarasjonen i subklassen er lik funksjonsdeklarasjonen i superklassen. Dersom disse deklarasjonene er ulike vil `override` sørge for at koden *ikke* kompilerer. På den måten sikres det at man faktisk overlaster funksjonen i superklassen, og at skrivefeil el. ikke har sneket seg inn.

(s. 508 – 511)

14. (5 points) Forklar kort hvorfor operatoroverlasting er nyttig.

Solution: Operatoroverlasting gjør det mulig å bruke den overlastede operatoren på en egendefinert type. Dette kan gjøre koden mer oversiktlig og leselig, og bidra til mer generisk programmering.

(Mindre viktig: Bidrar til mer generisk programmering bl.a. ved at STL-algoritmer kan brukes på den egendefinerte typen (dersom operatorene algoritmen bruker er overlastet for typen) og at egenskapene til STL-beholderne (f.eks. sortering i `map`) kan dras nytte av (i `map`-eksempelet vil dette gjelde dersom `operator<` er overlastet).)

Kommentar: Studenten skal ikke ha trekk for å ha glemt noe av det som her står som mindre viktig. Dersom studenten har glemt det viktige, men har med noe av det mindre viktige, bør studenten få noe uttelling.

(s. 321 – 323 + 682 – 693)

Oppgave 8 - Polymorfi (10 poeng) - uttrekk. 1 oppgave til hver

15. (10 points) Se på koden under.

```
#include "std_lib_facilities.h"

class Drink{
public:
    virtual void serveDrink() const {cout << "Here is your drink!\n";}
};
class Tea : public Drink{
public:
    void serveDrink() const override {cout << "Here is your tea!\n";}
};
class HotChocolate : public Drink{
public:
    void serveDrink() const override {cout << "Here is your hot chocolate!\n";}
};

int main(){
    vector<Drink*> drinks;

    Tea earlGrey{};
    HotChocolate cocoa{};
    Drink mysteryDrink{};
    Tea greenTea;

    drinks.push_back(&earlGrey);
    drinks.push_back(&cocoa);
    drinks.push_back(&mysteryDrink);
    drinks.push_back(&greenTea);

    for(auto d : drinks){
```

```
        d->serveDrink();
    }
}
```

Hvorfor lønner det seg å la elementene i vektoren `drinks` i `main()` være av typen `Drink*` og ikke av typen `Drink`?

> **Solution:** Fordi vi ved å bruke pekere får benyttet oss av polymorfi, med andre ord det at et objekt både er et objekt av sin egen klasse og et objekt av klassene den har arvet fra. Dersom vi hadde latt vektoren inneholde `Drink`-kopier i stedet ville alle elementene i vektoren vært av typen `Drink`, og spesialiseringen som følger med subklassene hadde gått tapt. I dette kodeeksempelet gjør polymorfien at vi får utskrift som varierer med typen drikke. Hadde vi latt vektoren være full av kopier hadde vi i stedet fått "Here is your drink!" for alle drikkene i vektoren.
>
> Kommentar: Å nevne polymorfi er et fullverdig svar.
>
> (s. 943 – 954)

16. (10 points) Se på koden under.

```
#include "std_lib_facilities.h"

class Dessert{
public:
    virtual void serveDessert() const {cout << "Here is your dessert!\n";}
};
class IceCream : public Dessert{
public:
    void serveDessert() const override {cout << "Here is your ice cream!\n";}
};
class Cake : public Dessert{
public:
    void serveDessert() const override {cout << "Here is your cake!\n";}
};

int main(){
    vector<Dessert*> desserts;

    IceCream chocolate{};
    Cake carrotCake{};
    Dessert mysteryDessert{};
    IceCream coffee;

    desserts.push_back(&chocolate);
    desserts.push_back(&carrotCake);
    desserts.push_back(&mysteryDessert);
    desserts.push_back(&coffee);

    for(auto d : desserts){
        d->serveDessert();
    }
}
```

Hvorfor lønner det seg å la elementene i vektoren `desserts` i `main()` være av typen `Dessert*` og ikke av typen `Dessert`?

Solution: Fordi vi ved å bruke pekere får benyttet oss av polymorfi, med andre ord det at et objekt både er et objekt av sin egen klasse og et objekt av klassene den har arvet fra. Dersom vi hadde latt vektoren inneholde `Dessert`-kopier i stedet ville alle elementene i vektoren vært av typen `Dessert`, og spesialiseringen som følger med subklassene hadde gått tapt. I dette kodeeksempelet gjør polymorfien at vi får utskrift som varierer med typen dessert. Hadde vi latt vektoren være full av kopier hadde vi i stedet fått "Here is your dessert!" for alle dessertene i vektoren.

Kommentar: Å nevne polymorfi er et fullverdig svar.

(s. 943 – 954)