Opgaver

Monday, 1 October 2018 11.03

Exercise 1 Safety Guarantees

Exercise 1.1 Code inspecting

Consider each of these 4 different code snippets. What do you reckon that they provide as a guarantee? You have to be specific and relate to the different functionalities that actually are in use.

This means that some functionalities may provide one type of guarantee whereas others may provide another.

When designing code you should also strive to ensure that although your code *insides* do as promised, interfaces should promote proper use as well - consider if any of the interfaces fail - e.g. cannot uphold guarantees. Also note any mistakes you may find in your quest.

Use reasoning to substantiate your answer!

Do note that small subtle tricks may have been employed. Do not let yourselves be fooled! :-D.

Exceptions

Exercise 1.1.1 Snippet 1

Listing 1.1: C++ Snippet 1 - Guarantees

```
1  class Test { /* Some code */ };
2
3  template<typename T, int N>
4  class MyArray
5  {
6  public:
7    T& operator[](size_t i)
8    {
9      return data_[i];
10   }
11
12 private:
13    T   data_[N];
14 };
15
16 /* Using */
17 void f()
18 {
19   MyArray<Test, 10> my;
20
21   Test t;
22
23   my[5] = t;
24 }
```

Da der ingen exceptions el tjek af andre art, er koden No exception safety,
Men den er også 'usikker' da der ingen garanti er for hverken allokering el assignments

Det eneste sted der kan gå noget galt er assignment i linje 23
Men hvis der sker en fejl er det ligegyldigt, da der bare kommer til at være én invalid plads i et array der ellers virker
ANTAGET at vi kan anvende en default constructor!

Exercise 1.1.2 Snippet 2

Listing 1.2: C++ Snippet 2 - Guarantees

```
1  class Test { /* Some code */ };
2
3  template<typename T>
4  class MyVector
5  {
6  public:
7    MyVector(size_t capacity)
8    : capacity_(capacity), count_(0), data_(new T[capacity])
9    {}
10
11   bool full() const { return (count_ == capacity_); }
12
13   void push_back(const T& oneMore)
14   {
15     if(full())
16     {
17       capacity_ *= 2;
18       T* newData = new T[capacity_];
19
20       std::copy(data_, data_+count_, newData);
21       std::swap(data_, newData);
22       delete[] newData;
23     }
24
25     data_[count_] = oneMore;
26     ++count_;
27   }
28 private:
29    size_t   capacity_;
30    size_t   count_;
31    T*       data_;
32 };
33
34 /* Using */
35 void f()
36 {
37   MyVector<Test> my(20);
38
39   Test t;
40
41   my.push_back(t);
42 }
```

Hvad kan fejle:

Linje 8	Allokering af data Hvis vi går udenfor memory
Linje 18	Allokering af data
Linj 20	Copy() er ikke en sikker funktion for brugerdefineret typer (incl. Std string)
NOTE Linje 21	swap() er altid sikker - ingen exceptions kastes
NOTE Linje 22	Delete kaster ALDRIG exceptions! = man må ikke throw noget i destructor!!!
Linje 25	Assignment kaster en exception, hvis den fejler

Hvad kan vi gøre:

Linje 8	HVIS der som her kun er kaldt én new(allokering) sørger new for selv at rydde op, og objektet er aldrig blevet lavet. Derfor kan der heller ikke kaldes en destructor HVIS der derimod er 2 allokeringer i constructoren, skal JEG sørger for at try/catche på dem, og selv rydde op efter dem. (ved to kan man nøjes med den sidste, da hvis den fejler på den første vil det være ligesom situationen ovenfor, så ved 2 el flere skal de efterfølgende tjekkes hver for sig!) Her kan heller ikke kaldes Destruct, igen da objektet ikke er oprettet! Lav try/catch til at håndtere bad_alloc for hver efterfølgende new!
Linje 18	Samme som linje 8 her vil man i catch skulle reducere capasity, for ikke at have delvis ændret objektet.
Linj 20	Lav catch til hvis den fejler
NOTE Linje 21	swap() er altid sikker - ingen exceptions kastes
Linje 25	Vi er ligeglade, da det ikke har ændret objektet. Men ved catch af den bør man smide et rethrow for at fortælle client at denne funktion er fejlet! dermed kommer vi aldrig til count_++, og ved næste kald vil count stadig stå på samme værdi/plads

Garanti:
Der er ingen garanti'er da der ingen catch'es/handling er!

Exercise 1.1.3 Snippet 3

Listing 1.3: C++ Snippet 3 - Guarantees

```
1  class String
2  {
3  public:
4    String() : s_(nullptr){}
5
6    String(const char* s) : s_(new char[strlen(s)+1])
7    {
8      std::strcpy(s_, s);
9    }
10
11   String(const String& other)
12   : s_(new char[strlen(other.s_)+1])
13   {
14     std::strcpy(s_, other.s_);
15   }
16
17   String& operator=(const String& other)
18   {
19     delete[] s_;
20     s_ = new char[strlen(other.s_)+1];
21     std::strcpy(s_, other.s_);
22     return *this;
23   }
24
25   ~String()
26   {
27     delete[] s_;
28   }
29 private:
30   char* s_;
31 };
32
33 /* Using */
34 void f()
35 {
36   String s("Hello world");
37
38   String aCopy(s);
39
40   s = "Hello girls";
41 }
```

Hvad kan fejle:

Linje 6	Allokering af data Hvis vi går udenfor memory
Linje 8	Her er den sikker, da allokeringern af s, sker i linje 6.
Linje 12	Allokering af data Hvis vi går udenfor memory
Linje 14	Samme som linje 8
Linje 20	Allokering af data Hvis vi går udenfor memory
Linje 36, 38 og 40	Det kan gå galt, men det er ligegyldigt, da objektet ikke vil blive oprettet

Hvad kan vi gøre:

Linje 6 + 12	HVIS der som her kun er kaldt én new(allokering) sørger new for selv at rydde op, og objektet er aldrig blevet lavet. Derfor kan der heller ikke kaldes en destructor HVIS der derimod er 2 allokeringer i constructoren, skal JEG sørger for at try/catche på dem, og selv rydde op efter dem. (ved to kan man nøjes med den sidste, da hvis den fejler på den første vil det være ligesom situationen ovenfor, så ved 2 el flere skal de efterfølgende tjekkes hver for sig!) Her kan heller ikke kaldes Destruct, igen da objektet ikke er oprettet! Lav try/catch til at håndtere bad_alloc for hver efterfølgende new!
Linje 20	Hele overload'en af assignmentoperatoren burde omskrives til at benytte swap(), da det er en sikker funktion, der udnytter scope-resolution til destruction
Linje 8 + 14	Vil lave et try/catch omkring dem, og rethrow til client, så han ved noget er gået galt!
Linje 36, 38 og 40	Vi er ligeglade, da det ikke har ændret objektet. Men ved catch af den bør man som nævnt smide et rethrow for at fortælle client at denne funktion er fejlet!

Garanti:
Der er ingen garanti'er da der ingen catch'es/handling er!

Exercise 1.1.4 Snippet 4

Listing 1.4: C++ Snippet 4 - Guarantees

```
1  class DataSet
2  {
3  public:
4    DataSet(Key* key, Blob* blob)
5    : key_(key), blob_(blob)
6    {
7      if(!key->isValid())
8        throw InvalidKey(key->id());
9    }
10
11   void overWrite(const Key* key, const Blob* blob)
12   {
13     *key_  = *key;
14     *blob_ = *blob;
15   }
16
17   ~DataSet()
18   {
19     delete key_;
20     delete blob_;
21   }
22 private:
23   Key*  key_;
24   Blob* blob_;
25 };
26
27 /* Using */
28 void f()
29 {
30   DataSet ds(new Key, new Blob);
31
32   {
33     Key k(getKeyValue());
34     Blob b(fetchDBBlobByKey(k));
35
36     ds.overWrite(&k, &b);
37   }
38 }
```

Hvad kan fejle:

Linje 8	Der mangler catch til throw i constructoren
Linje 13+ 14	Giver memory-leak, da den det som key'en pegede på ikke længere kan tilgås, og ikke er blevet slettet inden overwrite

Hvad kan vi gøre:

Linje 8	Vil indsætte en catch i constructoren (da den så er med hver gang en instans af objektet bliver lavet) ELLER på linje 31, da throw'et skal gribes hvis programmet ikke skal crashe!
Linje 13+ 14	Brug swap() for at sikre de overwritede elementer bliver slettet automatisk

Garanti:
Der er ingen garanti'er da der ingen catch er på den hjemmelavede throw funktion!