Vector, part 2: Exception safety and Allocator

The aim of the assignment is to learn how to handle exceptions and ”raw memory”. The STL containers normally use the std::allocator to allocate memory but can use other allocators. We will only use one allocator “template<class T> Dalloc;” (Debug allocator) that will help with the debugging. You find it in the test program. The allocator is stateless so you can handle it any way, having a member “Dalloc<T> _dAlloc;” is one solution.
allocate(0) is permitted and gives a null_ptr (as Visual studio do).
Use placement new to construct objects: `new(pekare) T(värde)`

1 Vector klassen

Vector klassen ska ha. (kolla std::vector om ni undrar hur något ska fungera)
[const] anger att funktionen finns i två versioner, med const och utan const.
Rött anger att de bara krävs för VG, se även 7.2 nedan.

Funktionerna ska fungera som i std::Vector	**Kommentar**	**Ordo**
`value_type, size_type, difference_type, reference, const_reference, pointer, const_pointer, iterator, const_iterator, reverse_iterator, const_reverse_iterator`	typedef som kan användas lokalt också (speciellt iterator varianterna)	
`~Vector() noexcept;`		n
`Vector() noexcept;`		1
`template<class Titer> Vector(size_t newCapacity, Titer begin, Titer end)`	Not a requirement but good to have for implementing the other constructors.	n
`Vector(const char* other);`	For testing	n
`Vector(const Vector& other);`		n
`Vector(Vector&& other) noexcept;`		1
`Vector& operator=(const Vector& other);`	For G it is easiest to make it have strong safety level, for VG it should be fast and have basic safety level.	n
`Vector& operator=(Vector&& other) noexcept;`		n
`[const] T& operator[](size_t i) [const] noexcept;` `[const] T& at(size_t i) [const];`	Indexerar utan och med range check (throws `std::out_of_range`)	1
`[const] T* data() [const] noexcept;`	gives a reference to the internal array holding the Vector	1
`size_t size() const noexcept;`	aktuellt antal element	1
`size_t capacity() const noexcept;`	Hur mycket kan size() bli utan att omallokera	1
`void reserve(size_t cap);`	Öka capacity till >= cap	n
`void shrink_to_fit();`	till skillnad från std så kräver vi att utrymmet krymps maximalt (size()==capacity())	n
`void resize(size_t n)`	Ändrar alltid size() till n, om n>size() så fylls det på med T()	
`void push_back(const T& c);`	lägger till ett element sist	1[1]
`void push_back(T&& c);`	lägger till ett element sist	1[1]

[1] Amorterat O(1), vid omallokering O(n).

Funktionerna ska fungera som i std::Vector	Kommentar	Ordo
template< class... Args > reference emplace_back(Args&&... args);	Konstruerar ett nytt element sist.	1[1]
`friend bool operator==(const Vector& lhs, const Vector& other)`	global funktion	n
och alla de andra (!=,<,>,<=,>=). Observera att om du implementerar == och < så kan de andra uttryckas med dem (t.ex. är x<=y det samma som !(y<x)	globala funktioner Det går att göra med <=> (spaceship)	n
`Alla begin() och end() varianter som krävs (6 av varje)`	Se https://en.cppreference.com/w/cpp/container/vector	1
`friend void swap(Vector& lhs, Vector& rhs );`	global funktion	1
`bool Invariant() const;`	For testing	
`friend std::ostream& operator<<(std::ostream& cout , const Vector& other);`	For testing	
`template<class T>` `void swap(Vector<T>& lhs, Vector<T>& rhs );`	Global function	

2 Iterator klassen

Se https://en.cppreference.com/w/cpp/named_req/RandomAccessIterator för detaljer. Obs. Den uppfyller även BidirectionalIterator som uppfyller …
Iter nedan kan stå för `iterator, const_iterator, reverse_iterator, const_reverse_iterator.` (const) nedan ska vara med för const iteratorer och strykas för de andra.
Enklaste sättet att implementera detta är att definiera:

```
template<class X, int Direction> class VectorItt;
```

där X är "T" eller "const T" och Direction är +1 eller -1 (för reverse_iterator).
Det är tillåtet att använda std::iterator;

Funktioner/typedefs	Kommentar
`iterator_category, value_type, difference_type, pointer, reference`	typdefs needed to make the standard algorithms work.
`Iter(X* p) noexcept;`	
`Iter() noexcept;`	
`Iter(const Iter& other) noexcept;`	
`Iter& operator=(const Iter& other);`	
`const_iterator(iterator&)`	
`const_iterator& operator=(iterator&)`	
`X* operator->() const noexcept;`	
`X& operator[](size_t i) const noexcept;`	Indexering
`Iter& operator++() noexcept;`	++it;
`Iter& operator--() noexcept;`	--it;
`Iter operator++(int) noexcept;`	it++
`Iter operator--(int) noexcept:`	--it;
`VectorItt& operator+=(difference_type i) noexcept;`	
`Iter operator+(difference_type i) const noexcept;`	
`Iter operator-(difference_type i) const noexcept;`	

Funktioner/typedefs	Kommentar
`difference_type operator-(` `  const Iter& other) const noexcept;`	För iterator a och heltal i gäller alltid: ((a+i)-a)==i
`friend bool operator==(const Iter& lhs,` `  const Iter& rhs) noexcept;`	global funktion
och alla de andra (!=,<,>,<=,>=)	globala funktioner går bra att använda <=>

3 Exception Safety

Vi förutsätter att template parametern T har rimlig exception safety, se **Fel! Ogiltig självreferens i bokmärke.** 3 **Fel! Ogiltig självreferens i bokmärke.**.
I tabellen nedan finns medlemsfunktioner som inte är deklarerade noexcept och kravet på deras exception level.

Everything works as std::vector	Exception Safety level	Kommentar

`template<class Titer>`
`Vector(size_t newCapacity,`
`  const Titer& begin,`
`  const Titer& end)`
	Strong	Frivillig hjälp konstruktor

`Vector(const char* other);`	Strong	För testning
`Vector(const Vector& other);`	Strong	
`Vector& operator=(const Vector& other);`	Basic eller Strong	See VG spec
`[const] T& at(size_t i) [const];`	Strong	
`void reserve(size_t cap);`	Strong	
`void shrink_to_fit();`	Strong	
`void resize(size_t n)`	Basic eller Strong	Basic för VG
`void push_back(const T& c);`	Strong	
`void push_back(T&& c);`	Strong	
template< class... Args > reference emplace_back(Args&&... args);	Strong	

4 Krav på T

To make the exception handling feasible we assume that T has:

- Destructor is noexcept
- Move constructor and move assignment is noexcept
- Copy constructor has strong exception safety level (for a constructor the basic and strong safety level is the same!).
- Copy assignment has basic safety level.

5 Safety levels, tolkning i denna labb.

Det finns en viss otydlighet i hur man ska tolka exception levels så här är ett förtydligande som gäller för denna labb – och är en rimlig tolkning i allmänhet.

5.1 Strong exception safety

En operation med Strong safety ger antingen ingen exception, då ger den förväntat normalt resultat, eller så ger den exception och då återställer den allting till läget före operationen och kastar sedan exception.

5.2 Basic exception safety

An operation with Basic exception safety will, after an exception, leave all objects in a consistent state and will not give any memory leaks. An example is `a=b` for std::vector, if an exception is thrown `a` will be in a consistent state but we do not know what it contains.

6 Tips

6.1 Help constructor

Do a constructor taking iterators as arguments.

```
template<class Titer>
Vector(size_t newCapacity,
    const Titer& begin,
    const Titer& end)
```

When you have this it is easy to make the copy constructor and the test constructor as:

```
Vector(const Vector& other) :
    Vector(other.size(),other.begin(), other.end()) {}
```

and

```
Vector(const char* other) :
    Vector(std::strlen(other), other, other + std::strlen(other)) {}
```

7 Skillnaden mellan G och VG.

7.1 Krav för G.

Räcker med Basic safety level i stället för Strong. Observera att för tilldelning (operator=) är det enklare at göra strong!
Det krävs inte heller att man i alla situationer undviker ”onödiga” minnesallokeringar. Men helt onödiga allokeringar tillåts inte.

7.2 Krav för VG.

Viktigast är att kraven på snygg kod som inte krånglar till något i onödan, även att man undviker koduppprepning.

Ska aldrig göra en extra allokering av minne! Dvs. vid allokering så kan extra minne allokeras men det ska inte ske någon allokering alls om det kan undvikas.
Exception Safety levels ska följas.

7.2.1 Emplace_back and push_back(T &&)

7.2.2 Resize ska ha Basic safety level

Basic för att kunna göra den maximalt snabb!

7.2.3 Tre versioner av ”tilldelningsoperatorn”

Gör tre versioner av tilldelningsoperatorn och kommentera för och nackdelar, alla tre ska uppfylla minst Basic Safety level (inga minnesläckor och lämna objektet i ett läge så att man kan fortsätta använda det). De skall också ha högst O(n) i tid och utrymmeskomplexitet där n = this->size() + other.size().
De tre varianterna är:

- Vector& AssSimple(const Vector& other): Den ska vara så enkelt skriven som möjligt.
- Vector& AssFast(const Vector& other): Ska ha minimalt med ny minnesallokering i alla situationer (att det finns extra minne allokerat gör inget, men det ska inte allokeras nytt minne om det går att undvika). Denna är den krångliga!
- Vector& AssStrong(const Vector& other): Den ska ha Strong Safety level: Om exception så ska allt återställas i ursprungsläget. (Obs! den är mycket enkel att skriva!)

Ni ska definiera tilldelningen som AssFast:

```
Vector& operator=(const Vector& other) { return AssFast(other); }
```

8 Testprogrammet

Testprogrammet innehåller 7 filer:

- Dalloc.hpp	Innehåller Dalloc som är en Allocator som spårar vad som allokeras och deallokeras. Tanken är att den ska byggas ut till att generera minnesfel på ”beställning”.
- Dhelper.h	Special klass till testprogrammet
- TestLevel.h	Ska inkluderas först i alla cpp filer. Styr hur testprogrammet fungerar. Börja med att definiera Definiera antingen G_BETYG eller VG_BETYG. Definera sedan LEVEL till 1 och öka succesivt upp till ungefär 22, sätt sen LEVEL till 99 och den riktiga testen börjar.
- VectBasicTest.cpp	Enkel nivåstyrd test av hela labben (se även TestLevel.h)
- VectorIterTest.hpp	Testar iteratorerna, förutsätter att VecotrTest fungerar.
- VectorTest.cpp	Test av själva Vector, oberoende av iteratorer.
- Main.cpp	