1 Constantly thinking about const

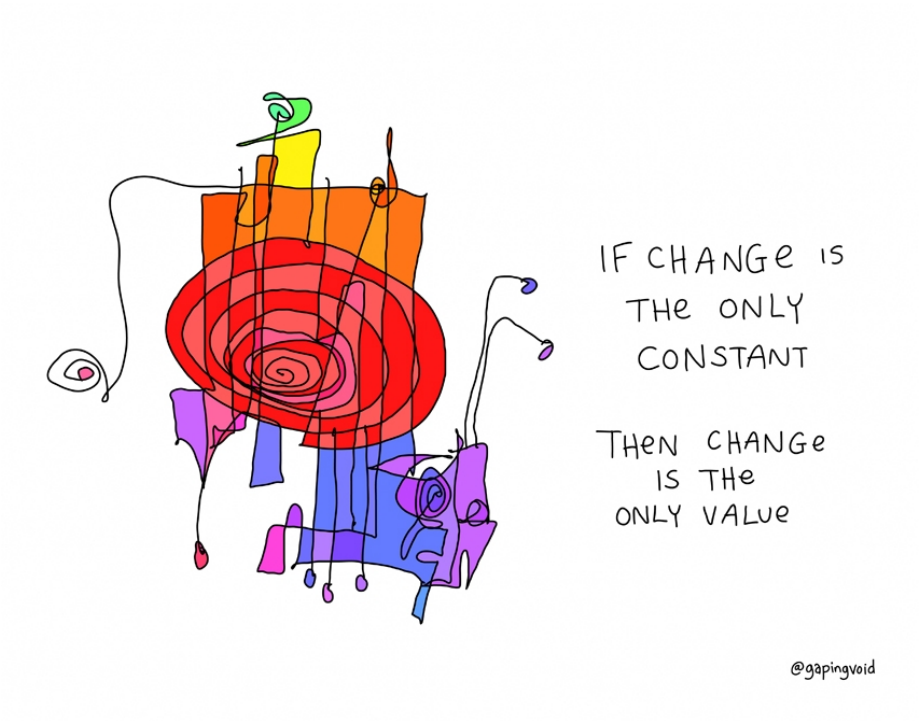

Figure 1: "If Change Is The Only Constant"

De helft van het hoorcollege ging over const. Vanaf nu verwachten we dat al jouw C++-code `const` correct is. Dat wilt zeggen dat je alles wat `const` kan zijn als `const` markeert. Van professionele C++-programmeurs wordt verwacht dat zij `const` correcte code afleveren. In deze opdracht gaan we daarmee oefenen.

Het standaardiseringscommittee heeft een hele FAQ over const correctness[1]. Een van de vragen gaat over wanneer je aan const correctness gaat werken:

> **Should I try to get things const correct "sooner" or "later"?**
>
> At the very, very, *very* beginning.
>
> Back-patching const correctness results in a snowball effect: every const you add "over here" requires four more to be added "over there."
>
> Add const early and often.

[1] Zie volgend URL op de website van het C++-standaardcommittee: https://isocpp.org/wiki/faq/const-correctness

Talen als Java en C# laten toe dat je variabelen van primitieve types als constant markeert. In Java bestaat daarvoor het sleutelwoord `final`, in C# de sleutelwoorden `const` en `readonly`. Je kunt echter geen objecten als constantes aanmerken. De sleutelwoorden `final` en `readonly` slaan namelijk op de referentie. Je kunt een `final` of `readonly` referentie niet naar een ander object laten verwijzen maar het object waar de referentie naar wijst, kun je nog steeds aanpassen:

```
final Person p = new Person();
p.setAge(37);
```

C++ stelt je in staat om objecten van alle types, zowel built-in als user-defined, als constante te kwalificeren. Het is vervolgens aan de programmeur om te bepalen welke member functions van een klasse op een constante aangeroepen kunnen worden.

Hiervoor gebruik je het sleutelwoord `const`. Een member function die als `const` is gekwalificeerd mag de member variables van `this` niet aanpassen. Daarnaast mag een `const` functie alleen andere `const` functies aanroepen op `this`. Zo wordt gegarandeerd dat een `const` functie het object waar `this` naar wijst niet wijzigt. Daarmee is het veilig om een `const` functie aan te roepen op een constant object.

Hieronder zie je een voorbeeld van een kleine `const`-correcte klasse:

```
class Person {
  int _age;
public:
  Person(int newAge);
  int getAge() const;
  void setAge(int newAge);
};
```

In het bovenstaande voorbeeld is `getAge` als `const` gekwalificeerd. Je mag `getAge` daarom aanroepen op constanten van het type `Person`. `setAge` mag je daarentegen niet aanroepen op een constante. Je zou daarmee immers de staat van het object veranderen.

Mocht je toch proberen een object aan te passen in een `const` functie dan zal de compiler een foutmelding geven:

```
int Person::getAge() const {
    _age = 13; // oeps!
    return _age;
}
```

```
person.cpp:12:10: error: cannot assign to non-static data
member within const member function 'getAge'
    _age = 13;
    ~~~~ ^
```

```
person.cpp:11:13: note: member function 'Person::getAge'
is declared const here
int Person::getAge() const {
```

1.1 Opdracht

Ben je er nog? Goed zo!

Bekijk onderstaande code en bepaal welke functies je als `const` kunt kwalificeren. Pas hun declaraties aan en schrijf bijpassende definities. Zorg ervoor dat de definitie compileert en test deze in een klein programma.

```
class Student {
    int _number;
    int _age;
public:
    Student(int number) : _number { number } {};
    int getNumber();
    void setNumber(int newNumber);
    int getAge();
    void setAge(int newAge);
    int ageDifference(const Student &other);
    int futureAge() { return getAge() + 1; }
};
```

2 Constantinople

We kijken naar een kleine klasse genaamd `City`. Deze klasse houdt de naam het inwonersaantal van een stad bij. De klasse bezit getters en setters voor beide member variables. Daarnaast is er een losse functie, `populationDifference` die het verschil in inwonersaantal van twee steden berekend.

De code ziet er als volgt uit:

```
1  #include <iostream>
2  #include <string>
3
4  class City {
5      int _population;
6      std::string _name;
7  public:
8
9      City(int population, std::string &name) :
10         _population { population }, _name { name } {};
```

```
11
12      int getPopulation();
13      void setPopulation(int newPopulation);
14
15      std::string& getName();
16      void setName(std::string& newName);
17  };
18
19  int populationDifference(City& city, City& other) {
20      return city.getPopulation() - other.getPopulation();
21  }
22
23  int City::getPopulation() {
24      return _population;
25  }
26
27  void City::setPopulation(int newPopulation) {
28      _population = newPopulation;
29  }
30
31  std::string& City::getName() {
32      return _name;
33  }
34
35  void City::setName(std::string &newName) {
36      _name = newName;
37  }
38
```

Deze klasse gaan we stap voor stap `const`-correct maken. Dit doen we aan de hand van een drietal vragen. De vragen zijn eigenlijk een stappenplan. Mocht je ooit andere code `const`-correct willen maken dan kun je dezelfde vragen aflopen.

2.1 Opdracht

1. Bepaal welke parameters als const gekwalificeerd kunnen worden.

   Gebruik hiervoor onderstaande tabel om te bepalen wanneer je `const` moet toevoegen. (Snap je de tabel?)

Pass by	voorbeeld	Waarde wordt gewijzigd	Wordt niet gewijzigd
value	`T x`	Blijft `T x`.	Blijft `T x`.
reference	`T &x`	Blijft `T &x`.	Moet zijn `const T &x`.
pointer	`T *x`	Blijft `T *x`.	Moet zijn `const T *x`.

Voor pass by value is `const` nooit nodig. De aangeroepen functie ontvangt namelijk een kopie van de originele waarde. Voor referenties en pointers is het de regel dat je ze standaard als `const` kwalificeert. Als je `const` weglaat, zeg je eigenlijk dat jouw functie het oorspronkelijke object via die referentie zal aanpassen.

2. Bepaal welke resultaattypes als `const` gemarkeerd kunnen worden.
    1. Geeft een functie een waarde terug (dus geen pointer of referentie), dan laat je `const` weg. De ontvanger van het resultaat krijgt namelijk een kopie.
    2. Geef je een referentie of een pointer naar een member variable terug vanuit een `const` member function? Markeer dan het resultaattype als `const`.
3. Bepaal welke member functions als const gekwalificeerd kunnen worden.

    Hiervoor kun je volgende regel gebruiken: Markeer een functie als `const` tenzij deze:

    1. Een member variable wijzigt
    2. Een andere member function aanroept die zelf niet `const` is.

3 Nobody expects the resource acquisition!

In deze opdracht gaan we het RAII-idioom toepassen. RAII houdt in dat je een klasse schrijft om een resource te beheren. Deze RAII-klasse verkrijgt in zijn constructor een bepaalde resource, zoals geheugen of een bestand, en geeft dat in zijn destructor weer vrij.

De standaard bibliotheek van C++ heeft een klasse `vector`. Dit is RAII-klasse voor het beheer van een dynamische array. In de constructor reserveert een `vector` geheugen dat in de destructor weer wordt vrijgegeven.

In deze opdracht gaan we een vereenvoudigde variant van de klasse `vector` ontwikkelen die een dynamische array van integers beheert. De klasse heet `IntVector`. We geven alvast een kleine opzet:

```
#include <cstddef>

class IntVector {
    int *_array;
    std::size_t _capacity;
    std::size_t _used;
public:
};
```

3.1 Opdrachten

Let op: *Ook bij deze opdracht werk je natuurlijk const-correct!*

1. Schrijf een constructor voor `IntVector` die als argument de capaciteit van de array meekrijgt.
2. Schrijf de destructor voor `IntVector`. *Als je een destructor implementeert, moet je eigenlijk ook copy- en move-operaties definiëren. Dit komt volgende week aan bod!*
3. Definier volgende member functions. Kwalificeer ze waar mogelijk als const.
    1. `capacity()` die de maximale capaciteit van de `IntVector` teruggeeft.
    2. `push_back(int value)` die een waarde op de eerste vrije plek in de array zet
    3. `size()` die het aantal elementen in de array teruggeeft.
    4. `get(size_t index)` die het element op de meegegeven index teruggeeft.

Hieronder zie je hoe je een `IntVector` zou kunnen gebruiken:

```
void describe(const IntVector &v) {
    std::cout << "IntVector" << std::endl;
    std::cout << "capacity: " << v.capacity() << std::endl;
    std::cout << "size: " << v.size() << std::endl;
    for(size_t i = 0; i < v.size(); ++i) {
        std::cout << "Element " << i << ": " << v.get(i) << std::endl;
    }
}
IntVector v { 10 };
v.push_back(1);
v.push_back(2);
describe(v);
```