Capítulo 1

Imutabilidade vs mutabilidade

Uma das características mais importantes do paradigma funcional, nomeadamente na linguagem Haskell (existem linguagens funcionais impuras) é a noção de imutabilidade das expressões.
Isto faz com que, por exemplo, não seja possível alterar o valor de variáveis já existentes mas sim, criar novas variáveis com os novos valores.
Analisando o excerto de código a seguir podemos verificar que quando *action* é invocado, o valor mostrado é o primeiro valor de *a=123* pois é o valor avaliado para *a* até então.

```
> a = 123
> action = print a
> a = 456
> action
123
```

A linguagem *c++* tenta também lidar com esta noção de imutabilidade.
A noção de funções puras é dada pela avaliação de *referential transparency*.
Uma função é referencialmente transparente se o programa não se comportar de maneira diferente ao substituír a expressão inteira apenas pelo seu valor de retorno, por exemplo:

```
int g = 0;
int ref_trans(int x) {
    return x + 1;
} // referencialmente transparente
int not_ref_trans(int x) {
    g++;
    return x + g;
} // não referencialmente transparente(cada vez que a função é invocada, tem um valor de retorno difer
```

Se uma expressão é referencialmente transparente, não tem efeitos colaterais observáveis e, portanto, todas as funções usadas nessa expressão são consideradas puras.
A ideia de imutabilidade é particularmente útil em ambientes em que se gera concorrência, pois, existem variáveis partilhadas que podem gerar comportamentos inesperados nos programas se não for devidamente protegida a sua alteração. Em *c++* está disponível a keyword *const* que permite controlar a imutabilidade de uma variável. Ao declarar uma variável *const x* estamos a dizer ao compilador que esta variável é imutável e, qualquer tentativa de alteração à variável irá originar um erro. De seguida analisamos a declaração de uma variável *const* e os possíveis erros que podem ser cometidos ao tentar manipular essa variável.

```
const std::string name{"John Smith"};
1 - std::string name_copy = name;
2 - std::string& name_ref = name; // erro
3 - const std::string& name_constref = name;
4 - std::string* name_ptr = &name; // erro
5 - const std::string* name_constptr = &name;
```

Em 1 não há ocorrências de erros pois apenas se está a associar o valor de name a uma nova variável. Em 2 teremos um erro de compilação pois estamos a passar *name* por referência a uma variável não *const*, situação que é resolvida em 3. Em 4 voltamos a ter um erro de compilação pois estamos a criar um pointer não *const* para name. Este erro é resolvido em 5. O facto de em 2 e 5 ocorrer um erro de compilação deve-se ao facto de $name_{r}ef name_{p}trnãoestaremqualificadoscomconstepoderemseralterados.Noentanto, comoapontamparaumavariávelconst, g$