1-style

June 3, 2024

1 Adopt a functional style in old C ++

For FP in C++, the importance of templates isn't (mainly) in the creation of container classes such as vectors, but in the fact that it allowed creation of STL algorithms (…) Most of these algorithms let you pass custom functions to customize the algorithms' behavior (…) The capability to pass functions as arguments to another function, and to have functions that return new functions, made even the first standardized version of C++ an FP language. *Ivan Cukic*

1.1 A vision of functional programming

Get closer to **mathematical logic**: * a **variable** always means/contains the same thing, * given the same arguments, a **function** always returns the same result, * **high-order functions** can manipulate functions, * a **type algebra** allows types to be recombined.

Compensate for the involved efficiencies: * **lazy evaluation**, * **immutable data structures**, * …

1.2 Hypothetical benefits of functional programming

- **Facilitates program proofs** … on only the pure sub-part of a real application (excluding reading and creating files, displays, interaction with the database, etc.).
- **Makes the code more readable?** Not so obvious in C ++, whose syntax serves so many paradigms…

1.3 Practical benefits of functional programming

- **Strengthen existing good practices** aimed at fighting spaghetti code (elimination of global variables).
- **Reduce unintentional state changes** ("OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts." *Michael Feathers*).
- **Promote the distinction between data structures and algorithms**, already more and more practiced in scientific computing, against a naive object approach.
- **Facilitate parallelization** ("No shared states, no problem").

1.4 A classic code example

[4]:
```
#include <iostream>
#include <vector>
#include <string>
```

[11]:
```
void spaces_in_sentences
 ( std::vector<std::string> & sentences, std::vector<unsigned> & spaces )
 {
  unsigned i, imax = sentences.size() ;
  for ( i = 0 ; i<imax ; ++i )
   {
    spaces[i] = 0 ;
    unsigned j, jmax = sentences[i].size() ;
    for ( j = 0 ; j<jmax ; ++j )
     {
      if (sentences[i][j]==' ')
       { ++spaces[i] ; }
     }
   }
 }
```

[14]:
```
int main()
 {
  unsigned SIZE = 2 ;
  std::vector<std::string> sentences(SIZE) ;
  sentences[0] = "Hello world !" ;
  sentences[1] = "Happy birthday" ;

  std::vector<unsigned> spaces(sentences.size()) ;
  spaces_in_sentences(sentences,spaces) ;

  std::cout<<spaces[0]<<' '<<spaces[1]<<std::endl ;
 }
```

1.5 What we can already do with ancient C ++

- Immutable data => use `const`
- Functions made **pure** =>
  - constant input variables,
  - a single output (possibly composite),
  - no side effects, never use of "context" !
- Eliminate or hide as much as possible `if`,`for` or `while` => use the **standard library algorithms**.

  Thinking functionally means thinking about the input data and the transformations you need to chain in order to get the desired output. *Ivan Cukic*

1.6 Modified example

[4]:
```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
```

[7]:
```
unsigned spaces_in_sentence
 ( std::string const & sentence )
 { return std::count(sentence.begin(),sentence.end(),' ') ; }
```

[8]:
```
std::vector<unsigned> spaces_in_sentences
 ( std::vector<std::string> const & sentences )
 {
  std::vector<unsigned> spaces(sentences.size()) ;
  std::transform(sentences.begin(),sentences.end(),spaces.
 ↪begin(),spaces_in_sentence) ;
  return spaces ;
 }
```

[9]:
```
int main()
 {
  unsigned SIZE = 2 ;
  std::vector<std::string> sentences(SIZE) ;
  sentences[0] = "Hello world !" ;
  sentences[1] = "Happy birthday" ;

  std::vector<unsigned> spaces = spaces_in_sentences(words) ;
  std::cout<<spaces[0]<<' '<<spaces[1]<<std::endl ;
 }
```

2 Questions ?