TP 4 - Problem 2 - I

```cpp
#include <algorithm>
#include <cassert>
#include <iostream>
#include <numeric>
#include <stdexcept>
#include <vector>

#define SHOW(arg) std::cout << "Macro SHOW “" #arg "”: " << (arg) << '\n';

class YourVector {
public:
  // Default ctor: do not allocate memory.
  YourVector() : size_{0}, data_{nullptr}, capacity_{0} {}
  // Parametrized ctor: create a vector of size elements, all elements
  // initialized to 0.
  YourVector(int size) : size_{size}, capacity_{size} {
    // When a exception is throwed from a ctor, the compiler deallocates
    // the memory just previously allocated for size_, data_ and
    // capacity_. There is no memory leaks. It is the standard way to tell
```

TP 4 - Problem 2 - II

```
20        // that the objet can not be constructed.
21        if (size < 0)
22          throw std::out_of_range{"Attempted to create a instance of "
23                                  ""YourVector" with a negative size."};
24        // The requested size is zero: fallback to default ctor.
25        if (size == 0)
26          data_ = nullptr;
27        else {
28          // Allocate memory using the "malloc" function. This function returns
29          // a "void *" pointer which should be converted to a "int *" pointer.
30          data_ = reinterpret_cast<int *>(std::malloc(size * sizeof(int)));
31          // Oops, unable to get memory... The standard way is to throw a
32          // "bad_alloc" exception.
33          if (data_ == nullptr)
34            throw std::bad_alloc{};
35          // Use the fill algorithm from the STL to assign the initial value.
36          std::fill(data_, data_ + size_, 0);
37          // for (int i{}; i < size_; ++i)
38          //   data_[i] = 0;
39        }
```

TP 4 - Problem 2 - III

```
40      }
41      // Copy ctor.
42      YourVector(YourVector const &other)
43          : size_{other.size_}, capacity_{other.size_} {
44        // The source vector is an uninitialized one.
45        if (size_ == 0)
46          data_ = nullptr;
47        else {
48          // Allocate memory using the “malloc” function. This function returns
49          // a “void *” pointer which should be converted to a “int *” pointer.
50          data_ = reinterpret_cast<int *>(std::malloc(size_ * sizeof(int)));
51          // Oops, unable to get memory... The standard way is to throw a
52          // “bad_alloc” exception.
53          if (data_ == nullptr)
54            throw std::bad_alloc{};
55          // Copy the data, using the copy algorithm of the STL.
56          std::copy(other.data_, other.data_ + size_, data_);
57          // for (int i{}; i < size_; ++i)
58          //   data_[i] = other.data_[i];
59        }
```

TP 4 - Problem 2 - IV

```
60      }
61      // Dtor: free the allocated (if any) memory.
62      ~YourVector() {
63        if (data_ != nullptr) {
64          // Sanity check.
65          assert(size_ != 0);
66          std::free(data_);
67        }
68      }
69      // Returns the max of the vector elements.
70      int max() const {
71        if (size_ == 0)
72          throw std::invalid_argument{
73              "Try to compute the max element of an empty vector."};
74        // Compute the max element using the max_element algorithm of the STL.
75        return *std::max_element(data_, data_ + size_);
76        // int max {data_[0]};
77        // for (int i{1}; i < size_; ++i)
78        //   if (data_[i] > max)
79        //     max = data_[i];
```

TP 4 - Problem 2 - V

```
80        // return max;
81      }
82      // Returns the sum of the vector elements.
83      int sum() const {
84        if (size_ == 0)
85          throw std::invalid_argument{
86              "Try to compute the sum of the elements of an empty vector."};
87        // Compute the sum using the accumulate algorithm of the STL.
88        return std::accumulate(data_, data_ + size_, 0);
89        // int sum {};
90        // for (int i{}; i < size_; ++i)
91        //   sum += data_[i];
92        // return sum;
93      }
94      // Returns the actual vector size.
95      int get_size() const { return size_; }
96      // Pushes the element at the vector end.
97      void push_back(int element) {
98        // Oops, need to reallocate...
99        if (size_ == capacity_) {
```

TP 4 - Problem 2 - VI

```
100         // A convenient way to set the new capacity.
101         int new_capacity{static_cast<int>(1.5 * (size_ + 8))};
102         // Assign data_ after, as realloc can return nullptr. If data_ is
103         // equal to nullptr, realloc is like malloc.
104         void *ptr{std::realloc(data_, new_capacity * sizeof(int))};
105         // Oops, unable to get memory... The standard way is to throw a
106         // “bad_alloc” exception.
107         if (ptr == nullptr)
108           throw std::bad_alloc{};
109         // It is time, now, to update data_ and capacity_.
110         data_ = reinterpret_cast<int *>(ptr);
111         capacity_ = new_capacity;
112       }
113       // Do the job.
114       data_[size_++] = element;
115     }
116     // Returns an instance of YourVector containing the elements between i
117     // and j.
118     YourVector extract(int i, int j) {
119       if ((i < 0) || (size_ <= i))
```

TP 4 - Problem 2 - VII

```
120         throw std::out_of_range{"First index out of bounds."};
121       if ((j < 0) || (size_ <= j))
122         throw std::out_of_range{"Second index out of bounds."};
123       if (j > i)
124         throw std::range_error{"Attempt to extract an invalid range."};
125       // Construct the new objet with uninitialized elements.
126       YourVector rvo{j - i + 1, UninitialisedTag{}};
127       // Copy the relevant elements, using the copy algoritm of the STL.
128       std::copy(data_ + i, data_ + j + 1, rvo.data_);
129       // for (int k {}; k <= j-i; ++k)
130       //   rvo.data_[k] = data_[i+k];
131       return rvo;
132     }
133     // Operator=
134     YourVector &operator=(YourVector const &other) {
135       // Protection over autoassignation.
136       if (this != &other) {
137         // Oops, need to allocate memory.
138         if (other.size_ > capacity_) {
139           // First, we try to allocate the necessary memory. This way, if it
```

TP 4 - Problem 2 - VIII

```
140           // fails, we can exit leaving the lhs untouched.
141           void *ptr{malloc(other.size_ * sizeof(int))};
142           // Oops, unable to get memory... The standard way is to throw a
143           // “bad_alloc” exception.
144           if (ptr == nullptr)
145             throw std::bad_alloc{};
146           // Do not forget to free the old memory.
147           std::free(data_);
148           // Set the new values of the parameters.
149           size_ = capacity_ = other.size_;
150           data_ = reinterpret_cast<int *>(ptr);
151         } else {
152           // Enough room: adjust the size.
153           size_ = other.size_;
154         }
155         // Copy the elements of the vector using the copy algorithm of the
156         // STL.
157         std::copy(other.data_, other.data_ + size_, data_);
158         // for (int i{}; i < size_; ++i)
159         //   data_[i] = other.data_[i];
```

TP 4 - Problem 2 - IX

```
160       }
161       return *this;
162     }
163     // Operator +=
164     YourVector &operator+=(int n) {
165       // Increment each element using the for_each algorithm of the STL
166       // "[&](int &i) { i += n; }" is a lambda function which captures the
167       // parameter n by reference.
168       std::for_each(data_, data_ + size_, [&](int &i) { i += n; });
169       // for (int i{}; i < size_; ++i)
170       //   data_[i] += n;
171       return *this;
172     }
173     // Operator *=
174     YourVector &operator*=(double x) {
175       // Increment each element using the for_each algorithm of the STL
176       // "[&](int &i) { i *= n; }" is a lambda function which captures the
177       // parameter x by reference.
178       std::for_each(data_, data_ + size_, [&](int &i) { i *= x; });
179       // for (int i{}; i < size_; ++i)
```

TP 4 - Problem 2 - X

```
180       //   data_[i] *= x;
181       return *this;
182     }
183     // Operator [] const: returns a copy as const int (you can also return a
184     // const reference to the requested element).
185     int const operator[](int i) const {
186       if ((i < 0) || (size_ <= i))
187         throw std::out_of_range{"Index out of bounds."};
188       return data_[i];
189     }
190     // Operator []: returns a reference to the requested element so that the
191     // operator can be used to the left hand side of a assignation.
192     int &operator[](int i) {
193       if ((i < 0) || (size_ <= i))
194         throw std::out_of_range{"Index out of bounds."};
195       return data_[i];
196     }
197     // Operator unary -: negate in place the vector.
198     YourVector &operator-() {
199       if (size_ == 0)
```

TP 4 - Problem 2 - XI

```
200         throw std::invalid_argument{
201             "Try to get the opposite of an empty vector."};
202       // Increment each element using the for_each algorithm of the STL
203       // “[](int &i) { i = -i; }” is a lambda function.
204       std::for_each(data_, data_ + size_, [](int &i) { i = -i; });
205       // for (int i{}; i < size_; ++i)
206       //   data_[i] = -data_[i];
207       return *this;
208     }
209
210   private:
211     // Class used as a tag. This tag flags the ctor with uninitialized
212     // elements.
213     struct UninitialisedTag {};
214     // Ctor with uninitialized elements. Accept narrowing conversion for
215     // size_ and capacity_.
216     YourVector(int size, UninitialisedTag) : size_{size}, capacity_{size} {
217       if (size < 0)
218         throw std::out_of_range{"Attempt to create a instance of "
219                                 "“YourVector” with a negative size."};
```

TP 4 - Problem 2 - XII

```
220       if (size == 0)
221         data_ = nullptr;
222       else {
223         data_ = reinterpret_cast<int *>(std::malloc(size * sizeof(int)));
224         if (data_ == nullptr)
225           throw std::bad_alloc{};
226       }
227     }
228     // Current size of the vector.
229     int size_;
230     // Pointer to the first element of the vector.
231     int *data_;
232     // Current capacity.
233     int capacity_;
234     // The free operator << must be a friend of my user defined class so it
235     // have access to the private data member of this class. It is preferable
236     // to declare the operator as a private one: it is only found via the
237     // argument-dependant lookup.
238     friend std::ostream &operator<<(std::ostream &, YourVector const &);
239     // Give access to the internals of this class.
```

TP 4 - Problem 2 - XIII

```
240     friend YourVector sum2Vectors(YourVector const &, YourVector const &);
241     friend YourVector product2Vectors(YourVector const &,
242                                       YourVector const &);
243     friend bool operator==(YourVector const &, YourVector const &);
244     friend double scalprod2Vectors(YourVector const &, YourVector const &);
245   };
246
247   // Free functions.
248
249   // Extends the free operator << with the user defined class. This operator
250   // returns a reference to the stream object so you can chain stream
251   // operations together.
252   std::ostream &operator<<(std::ostream &os, YourVector const &v) {
253     if (v.size_ == 0)
254       return os << "[]";
255     std::string str{"["};
256     for (int i{}; i < v.size_; ++i)
257       str += std::to_string(v.data_[i]) + ' ';
258     // The last space is crushed.
259     str.back() = ']';
```

TP 4 - Problem 2 - XIV

```
260     return os << str;
261   }
262
263   YourVector sum2Vectors(YourVector const &u, YourVector const &v) {
264     if (u.size_ != v.size_)
265       throw std::invalid_argument{"sum2Vectors: different sizes."};
266     if (u.size_ == 0)
267       throw std::invalid_argument{"Try to sum empty vectors."};
268     // Create a vector with uninitialized elements.
269     YourVector rvo{u.size_, YourVector::UninitialisedTag{}};
270     // Compute and assign the sum using the transform algorithm of the STL.
271     std::transform(u.data_, u.data_ + u.size_, v.data_, rvo.data_,
272                    std::plus<int>());
273     // for (int i{}; i < u.size_; ++i)
274     //   rvo.data_[i] = u.data_[i] + v.data_[i];
275     return rvo;
276   }
277
278   YourVector product2Vectors(YourVector const &u, YourVector const &v) {
279     if (u.size_ != v.size_)
```

TP 4 - Problem 2 - XV

```
280       throw std::invalid_argument{"product2Vectors: different sizes."};
281     if (u.size_ == 0)
282       throw std::invalid_argument{"Try to multiply empty vectors."};
283     // Create a vector with uninitialized elements.
284     YourVector rvo{u.size_, YourVector::UninitialisedTag{}};
285     // Compute and assign the product using the transform algorithm of the
286     // STL.
287     std::transform(u.data_, u.data_ + u.size_, v.data_, rvo.data_,
288                    std::multiplies<int>());
289     // for (int i{}; i < u.size_; ++i)
290     //   rvo.data_[i] = u.data_[i] * v.data_[i];
291     return rvo;
292   }
293
294   bool operator==(YourVector const &u, YourVector const &v) {
295     // Quick return is the user codes u == u.
296     if (&u == &v)
297       return true;
298     // Quick return if sizes differ.
299     if (u.size_ != v.size_)
```

TP 4 - Problem 2 - XVI

```
300       return false;
301     // Do the test using the equal algorithm of the STL.
302     return std::equal(u.data_, u.data_ + u.size_, v.data_);
303     // for (int i{}; i < u.size_; ++i)
304     //   if (u.data_[i] != v.data_[i])
305     //     return false;
306     // return true;
307   }
308   bool operator!=(YourVector const &u, YourVector const &v) {
309     // Reuse the operator ==.
310     return !(u == v);
311   }
312   double scalprod2Vectors(YourVector const &u, YourVector const &v) {
313     if (u.size_ != v.size_)
314       throw std::invalid_argument{"scalprod2Vectors: different sizes."};
315     if (u.size_ == 0)
316       throw std::invalid_argument{
317           "Try to get the scalar product of empty vectors."};
318     // Compute the scalar product using the inner_product algorithm of the
319     // STL.
```

TP 4 - Problem 2 - XVII

```
320     return std::inner_product(u.data_, u.data_ + u.size_, v.data_, 0);
321     // int sum {};
322     // for (int i{}; i < u.size_; ++i)
323     //   sum += u.data_[i] * v.data_[i];
324     // return sum;
325   }
326
327   int main() {
328     SHOW(YourVector{})
329     YourVector v{1};
330     SHOW(v)
331     v.push_back(2);
332     SHOW(v)
333     SHOW(v.max())
334     SHOW(v.sum())
335     SHOW(v.extract(0, 0))
336     SHOW(v.extract(1, 1))
337     YourVector u;
338     u = v;
339     SHOW(u)
```

TP 4 - Problem 2 - XVIII

```
340     SHOW(u += 2)
341     SHOW(u *= 3)
342     SHOW(u[0])
343     SHOW(u[0] = -1);
344     SHOW(-u)
345     SHOW(sum2Vectors(u, v))
346     SHOW(product2Vectors(u, v))
347     v = u;
348     SHOW(u == v)
349     SHOW(u != v)
350     SHOW(scalprod2Vectors(u, v))
351     return 0;
352   }
```

TP 4 - Problem 2 - XIX

Output:

```
1  Macro SHOW “YourVector{}”: []
2  Macro SHOW “v”: [0]
3  Macro SHOW “v”: [0 2]
4  Macro SHOW “v.max()”: 2
5  Macro SHOW “v.sum()”: 2
6  Macro SHOW “v.extract(0, 0)”: [0]
7  Macro SHOW “v.extract(1, 1)”: [2]
8  Macro SHOW “u”: [0 2]
9  Macro SHOW “u += 2”: [2 4]
10 Macro SHOW “u *= 3”: [6 12]
11 Macro SHOW “u[0]”: 6
12 Macro SHOW “u[0] = -1”: -1
13 Macro SHOW “-u”: [1 -12]
14 Macro SHOW “sum2Vectors(u, v)”: [1 -10]
15 Macro SHOW “product2Vectors(u, v)”: [0 -24]
16 Macro SHOW “u == v”: 1
17 Macro SHOW “u != v”: 0
18 Macro SHOW “scalprod2Vectors(u, v)”: 145
```