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

```
    // that the objet can not be constructed.
    if (size < 0)
      throw std::out_of_range{"Attempted to create a instance of "
                              "“YourVector” with a negative size."};
    // The requested size is zero: fallback to default ctor.
    if (size == 0)
      data_ = nullptr;
    else {
      // Allocate memory using the “malloc” function. This function returns
      // a “void *” pointer which should be converted to a “int *” pointer.
      data_ = reinterpret_cast<int *>(std::malloc(size * sizeof(int)));
      // Oops, unable to get memory... The standard way is to throw a
      // “bad_alloc” exception.
      if (data_ == nullptr)
        throw std::bad_alloc{};
      // Use the fill algorithm from the STL to assign the initial value.
      std::fill(data_, data_ + size_, 0);
      // for (int i{}; i < size_; ++i)
      //   data_[i] = 0;
    }
```

TP 4 - Problem 2 - III

```
    }
    // Copy ctor.
    YourVector(YourVector const &other)
        : size_{other.size_}, capacity_{other.size_} {
      // The source vector is an uninitialized one.
      if (size_ == 0)
        data_ = nullptr;
      else {
        // Allocate memory using the “malloc” function. This function returns
        // a “void *” pointer which should be converted to a “int *” pointer.
        data_ = reinterpret_cast<int *>(std::malloc(size_ * sizeof(int)));
        // Oops, unable to get memory... The standard way is to throw a
        // “bad_alloc” exception.
        if (data_ == nullptr)
          throw std::bad_alloc{};
        // Copy the data, using the copy algorithm of the STL.
        std::copy(other.data_, other.data_ + size_, data_);
        // for (int i{}; i < size_; ++i)
        //   data_[i] = other.data_[i];
      }
```

TP 4 - Problem 2 - IV

```cpp
  }
  // Dtor: free the allocated (if any) memory.
  ~YourVector() {
    if (data_ != nullptr) {
      // Sanity check.
      assert(size_ != 0);
      std::free(data_);
    }
  }
  // Returns the max of the vector elements.
  int max() const {
    if (size_ == 0)
      throw std::invalid_argument{
          "Try to compute the max element of an empty vector."};
    // Compute the max element using the max_element algorithm of the STL.
    return *std::max_element(data_, data_ + size_);
    // int max {data_[0]};
    // for (int i{1}; i < size_; ++i)
    //   if (data_[i] > max)
    //     max = data_[i];
```

TP 4 - Problem 2 - V

```
      // return max;
    }
    // Returns the sum of the vector elements.
    int sum() const {
      if (size_ == 0)
        throw std::invalid_argument{
            "Try to compute the sum of the elements of an empty vector."};
      // Compute the sum using the accumulate algorithm of the STL.
      return std::accumulate(data_, data_ + size_, 0);
      // int sum {};
      // for (int i{}; i < size_; ++i)
      //   sum += data_[i];
      // return sum;
    }
    // Returns the actual vector size.
    int get_size() const { return size_; }
    // Pushes the element at the vector end.
    void push_back(int element) {
      // Oops, need to reallocate...
      if (size_ == capacity_) {
```

TP 4 - Problem 2 - VI

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

TP 4 - Problem 2 - VII

```cpp
        throw std::out_of_range{"First index out of bounds."};
      if ((j < 0) || (size_ <= j))
        throw std::out_of_range{"Second index out of bounds."};
      if (j > i)
        throw std::range_error{"Attempt to extract an invalid range."};
      // Construct the new objet with uninitialized elements.
      YourVector rvo{j - i + 1, UninitialisedTag{}};
      // Copy the relevant elements, using the copy algoritm of the STL.
      std::copy(data_ + i, data_ + j + 1, rvo.data_);
      // for (int k {}; k <= j-i; ++k)
      //   rvo.data_[k] = data_[i+k];
      return rvo;
    }
    // Operator=
    YourVector &operator=(YourVector const &other) {
      // Protection over autoassignation.
      if (this != &other) {
        // Oops, need to allocate memory.
        if (other.size_ > capacity_) {
          // First, we try to allocate the necessary memory. This way, if it
```

TP 4 - Problem 2 - VIII

```
          // fails, we can exit leaving the lhs untouched.
          void *ptr{malloc(other.size_ * sizeof(int))};
          // Oops, unable to get memory... The standard way is to throw a
          // “bad_alloc” exception.
          if (ptr == nullptr)
            throw std::bad_alloc{};
          // Do not forget to free the old memory.
          std::free(data_);
          // Set the new values of the parameters.
          size_ = capacity_ = other.size_;
          data_ = reinterpret_cast<int *>(ptr);
        } else {
          // Enough room: adjust the size.
          size_ = other.size_;
        }
        // Copy the elements of the vector using the copy algorithm of the
        // STL.
        std::copy(other.data_, other.data_ + size_, data_);
        // for (int i{}; i < size_; ++i)
        //   data_[i] = other.data_[i];
```

TP 4 - Problem 2 - IX

```
      }
      return *this;
    }
    // Operator +=
    YourVector &operator+=(int n) {
      // Increment each element using the for_each algorithm of the STL
      // “[&](int &i) { i += n; }” is a lambda function which captures the
      // parameter n by reference.
      std::for_each(data_, data_ + size_, [&](int &i) { i += n; });
      // for (int i{}; i < size_; ++i)
      //   data_[i] += n;
      return *this;
    }
    // Operator *=
    YourVector &operator*=(double x) {
      // Increment each element using the for_each algorithm of the STL
      // “[&](int &i) { i *= n; }” is a lambda function which captures the
      // parameter x by reference.
      std::for_each(data_, data_ + size_, [&](int &i) { i *= x; });
      // for (int i{}; i < size_; ++i)
```

TP 4 - Problem 2 - X

```
      //   data_[i] *= x;
      return *this;
    }
    // Operator [] const: returns a copy as const int (you can also return a
    // const reference to the requested element).
    int const operator[](int i) const {
      if ((i < 0) || (size_ <= i))
        throw std::out_of_range{"Index out of bounds."};
      return data_[i];
    }
    // Operator []: returns a reference to the requested element so that the
    // operator can be used to the left hand side of a assignation.
    int &operator[](int i) {
      if ((i < 0) || (size_ <= i))
        throw std::out_of_range{"Index out of bounds."};
      return data_[i];
    }
    // Operator unary -: negate in place the vector.
    YourVector &operator-() {
      if (size_ == 0)
```

TP 4 - Problem 2 - XI

```
      throw std::invalid_argument{
          "Try to get the opposite of an empty vector."};
    // Increment each element using the for_each algorithm of the STL
    // “[](int &i) { i = -i; }” is a lambda function.
    std::for_each(data_, data_ + size_, [](int &i) { i = -i; });
    // for (int i{}; i < size_; ++i)
    //   data_[i] = -data_[i];
    return *this;
  }

private:
  // Class used as a tag. This tag flags the ctor with uninitialized
  // elements.
  struct UninitialisedTag {};
  // Ctor with uninitialized elements. Accept narrowing conversion for
  // size_ and capacity_.
  YourVector(int size, UninitialisedTag) : size_{size}, capacity_{size} {
    if (size < 0)
      throw std::out_of_range{"Attempt to create a instance of "
                              "“YourVector” with a negative size."};
```

TP 4 - Problem 2 - XII

```
      if (size == 0)
        data_ = nullptr;
      else {
        data_ = reinterpret_cast<int *>(std::malloc(size * sizeof(int)));
        if (data_ == nullptr)
          throw std::bad_alloc{};
      }
    }
    // Current size of the vector.
    int size_;
    // Pointer to the first element of the vector.
    int *data_;
    // Current capacity.
    int capacity_;
    // The free operator << must be a friend of my user defined class so it
    // have access to the private data member of this class. It is preferable
    // to declare the operator as a private one: it is only found via the
    // argument-dependant lookup.
    friend std::ostream &operator<<(std::ostream &, YourVector const &);
    // Give access to the internals of this class.
```

TP 4 - Problem 2 - XIII

```cpp
  friend YourVector sum2Vectors(YourVector const &, YourVector const &);
  friend YourVector product2Vectors(YourVector const &,
                                    YourVector const &);
  friend bool operator==(YourVector const &, YourVector const &);
  friend double scalprod2Vectors(YourVector const &, YourVector const &);
};

// Free functions.

// Extends the free operator << with the user defined class. This operator
// returns a reference to the stream object so you can chain stream
// operations together.
std::ostream &operator<<(std::ostream &os, YourVector const &v) {
  if (v.size_ == 0)
    return os << "[]";
  std::string str{"["};
  for (int i{}; i < v.size_; ++i)
    str += std::to_string(v.data_[i]) + ' ';
  // The last space is crushed.
  str.back() = ']';
```

TP 4 - Problem 2 - XIV

```cpp
  return os << str;
}

YourVector sum2Vectors(YourVector const &u, YourVector const &v) {
  if (u.size_ != v.size_)
    throw std::invalid_argument{"sum2Vectors: different sizes."};
  if (u.size_ == 0)
    throw std::invalid_argument{"Try to sum empty vectors."};
  // Create a vector with uninitialized elements.
  YourVector rvo{u.size_, YourVector::UninitialisedTag{}};
  // Compute and assign the sum using the transform algorithm of the STL.
  std::transform(u.data_, u.data_ + u.size_, v.data_, rvo.data_,
                 std::plus<int>());
  // for (int i{}; i < u.size_; ++i)
  //   rvo.data_[i] = u.data_[i] + v.data_[i];
  return rvo;
}

YourVector product2Vectors(YourVector const &u, YourVector const &v) {
  if (u.size_ != v.size_)
```

TP 4 - Problem 2 - XV

```
    throw std::invalid_argument{"product2Vectors: different sizes."};
  if (u.size_ == 0)
    throw std::invalid_argument{"Try to multiply empty vectors."};
  // Create a vector with uninitialized elements.
  YourVector rvo{u.size_, YourVector::UninitialisedTag{}};
  // Compute and assign the product using the transform algorithm of the
  // STL.
  std::transform(u.data_, u.data_ + u.size_, v.data_, rvo.data_,
                 std::multiplies<int>());
  // for (int i{}; i < u.size_; ++i)
  //   rvo.data_[i] = u.data_[i] * v.data_[i];
  return rvo;
}

bool operator==(YourVector const &u, YourVector const &v) {
  // Quick return is the user codes u == u.
  if (&u == &v)
    return true;
  // Quick return if sizes differ.
  if (u.size_ != v.size_)
```

TP 4 - Problem 2 - XVI

```cpp
    return false;
  // Do the test using the equal algorithm of the STL.
  return std::equal(u.data_, u.data_ + u.size_, v.data_);
  // for (int i{}; i < u.size_; ++i)
  //   if (u.data_[i] != v.data_[i])
  //     return false;
  // return true;
}
bool operator!=(YourVector const &u, YourVector const &v) {
  // Reuse the operator ==.
  return !(u == v);
}
double scalprod2Vectors(YourVector const &u, YourVector const &v) {
  if (u.size_ != v.size_)
    throw std::invalid_argument{"scalprod2Vectors: different sizes."};
  if (u.size_ == 0)
    throw std::invalid_argument{
        "Try to get the scalar product of empty vectors."};
  // Compute the scalar product using the inner_product algorithm of the
  // STL.
```

TP 4 - Problem 2 - XVII

```
  return std::inner_product(u.data_, u.data_ + u.size_, v.data_, 0);
  // int sum {};
  // for (int i{}; i < u.size_; ++i)
  //   sum += u.data_[i] * v.data_[i];
  // return sum;
}

int main() {
  SHOW(YourVector{})
  YourVector v{1};
  SHOW(v)
  v.push_back(2);
  SHOW(v)
  SHOW(v.max())
  SHOW(v.sum())
  SHOW(v.extract(0, 0))
  SHOW(v.extract(1, 1))
  YourVector u;
  u = v;
  SHOW(u)
```

TP 4 - Problem 2 - XVIII

```
  SHOW(u += 2)
  SHOW(u *= 3)
  SHOW(u[0])
  SHOW(u[0] = -1);
  SHOW(-u)
  SHOW(sum2Vectors(u, v))
  SHOW(product2Vectors(u, v))
  v = u;
  SHOW(u == v)
  SHOW(u != v)
  SHOW(scalprod2Vectors(u, v))
  return 0;
}
```

TP 4 - Problem 2 - XIX

Output:

```
Macro SHOW “YourVector{}”: []
Macro SHOW “v”: [0]
Macro SHOW “v”: [0 2]
Macro SHOW “v.max()”: 2
Macro SHOW “v.sum()”: 2
Macro SHOW “v.extract(0, 0)”: [0]
Macro SHOW “v.extract(1, 1)”: [2]
Macro SHOW “u”: [0 2]
Macro SHOW “u += 2”: [2 4]
Macro SHOW “u *= 3”: [6 12]
Macro SHOW “u[0]”: 6
Macro SHOW “u[0] = -1”: -1
Macro SHOW “-u”: [1 -12]
Macro SHOW “sum2Vectors(u, v)”: [1 -10]
Macro SHOW “product2Vectors(u, v)”: [0 -24]
Macro SHOW “u == v”: 1
Macro SHOW “u != v”: 0
Macro SHOW “scalprod2Vectors(u, v)”: 145
```