Реализовать класс, представляющий собой одномерный массив. Должен содержать:

1. конструктор, деструктор;
2. функцию получения элемента по индексу;
3. функцию добавления элемента в конец;
4. функцию получения размера массива.

Размер массива должен увеличиваться автоматически если не хватает места для вставки очередного элемента. Должны возбуждаться исключения, если:

1. на вход функции, возвращающей элемент по индексу передано некорректное значение;
2. не удалось выделить нужное количество памяти

Работу с памятью желательно реализовать через функции malloc/free, т.к. в этом случае для увеличения размера массива можно использовать более эффективный realloc.

Весь нужный вам функционал есть в стандартном классе vector, приведенный ниже код можно использовать лишь в учебных целях (чтобы потренироваться), во всех остальных случаях используйте надежные классы стандартной библиотеки.  
Объявим класс Array:

1. **template**<**class** ElementType>
2. **class** Array {
3. **public**:
4. Array();
5. ~Array();
6. ElementType& **operator** [] (**unsigned** **int** index);
7. **void** push\_back(**const** ElementType &item);
8. ElementType\* get();
9. **unsigned** **int** size();
10. **struct** bad\_allocation {};
11. **struct** bad\_index {};
12. **private**:
13. ElementType \*m\_array;
14. **unsigned** **int** m\_size;
15. **unsigned** **int** m\_realSize;
16. **const** **static** **int** Step = 128;
17. };

Тут функция получения элемента массива (оператор []) возвращает ссылку на элемент чтобы пользователь мог изменить принятое значение (изменить элемент массива). Функция добавления элемента также принимает константную ссылку на значение. Более [подробно про передачу по ссылке](https://pro-prof.com/forums/topic/pointers_as_args_cplusplus).

Класс помимо указателя на массив содержит его реальный размер (количество выделенной по указателю памяти) и количество записанных в массив элементов (меньше или равное реального размера). Константа Step хранит количество элементов, на которое увеличивается реальный размер всякий раз, когда очередной элемент не входит в существующий массив.

Кроме того, описываются исключения, которые могут вырабатывать функции класса. На самом деле, более правильно — наследовать классы исключений от std::exception.

Конструктор класса вызывает malloc чтобы выделить память. Если при этом он получить NULL — значит нужный блок памяти выделить не удалось, поэтому вырабатывается исключение:

1. **template** <**class** ElementType>
2. Array<Elementtype>::Array() : m\_realSize(Step), m\_size(0), m\_array(0) {
3. m\_array = (ElementType\*)malloc(**sizeof**(ElementType)\*m\_realSize);
4. **if** (0 == m\_array) {
5. **throw** bad\_allocation();
6. }
7. }

Вырабатывать исключения в конструкторе иногда опасно, однако в этом случае это допустимо, т.к. стандарт языка гарантирует при этом корректное освобождение памяти из под членов класса, созданных на стеке (m\_realSize, m\_size и указатель m\_array). Утечки памяти, выделенной динамически (вызовом malloc) также не произойдет — ведь мы вырабатываем исключение только если память не удалось выделить. Подробнее: «[Обработка исключений в конструкторе](https://pro-prof.com/forums/topic/constructor_destructor_exceptions)«.

Деструктор убеждается что массив был создан (указатель не равен нулю) и освобождает память:

1. **template** <**class** ElementType>
2. Array<Elementtype>::~Array() {
3. **if** (m\_array)
4. free(m\_array);
5. }

Строго говоря, в этом коде можно обойтись без проверки указателя, т.к. если мы добрались до деструктора, то паять однозначно была успешно выделена.

Функции получения элементов и параметров массива:

1. **template** <**class** ElementType>
2. ElementType\* Array<elementtype>::get() {
3. **return** m\_array;
4. }
5. **template** <**class** ElementType>
6. **unsigned** **int** Array<elementtype>::size() {
7. **return** m\_size;
8. }
9. **template** <**class** ElementType>
10. ElementType& Array<Elementtype>::**operator** [] (**unsigned** **int** index) {
11. **if** (index < 0 || index > m\_size)
12. **throw** bad\_index();
13. **return** m\_array[index];
14. }

Наибольший интерес может представлять функция добавления элемента, т.к. она содержит код увеличения размера массива с помощью realloc:

1. **template** <**class** ElementType>
2. **void** Array<Elementtype>::push\_back(**const** ElementType &item) {
3. **if** (++m\_size > m\_realSize) {
4. m\_realSize += Step;
5. ElementType \*biggest\_array;
6. biggest\_array = (ElementType \*)realloc(m\_array, **sizeof**(ElementType)\*m\_realSize);
7. **if** (biggest\_array == NULL)
8. **throw** bad\_allocation();
9. m\_array = biggest\_array;
10. }
11. m\_array[m\_size-1] = item;
12. }

В случае ошибки выделения памяти функция realloc вернет нулевой указатель, а начиная с *С++11* она не освобождает память по принятому указателю. Поэтому нельзя обойтись без вспомогательного указателя, в который требуется сохранить результат. Только в случае успешного выделения памяти содержимое m\_array обновляется, в противном случае бросается исключение и память из под массива будет освобождена в деструкторе.

Проверить работу функции можно следующим кодом:

1. **int** main() {
2. Array<**int**> m\_array;
3. **for**(**unsigned** **int** i = 0; i < 15; i++)
4. m\_array.push\_back(rand() % 100);
5. **for**(**unsigned** **int** i = 0; i < m\_array.size(); i++)
6. cout << m\_array[i] << " ";
7. cout << endl;
8. **for**(**unsigned** **int** i = 0; i < m\_array.size(); i++)
9. m\_array[i] = i;
10. **for**(**unsigned** **int** i = 0; i < m\_array.size(); i++)
11. cout << m\_array[i] << " ";
12. cout << endl;
13. }

