Лекция VIII

Повторения мало не бывает: базовые возможности классов в C++

Возможности классов в C++

Вспоминаем (**пятая лекция**), что класс сам по себе - это *тип данных*. При этом, ещё и *составной* - может включать произвольное количество полей разных других типов. Доступ к полям получаем с помощью *объектов класса* (они же - *переменные класса*), который можно регулировать с помощью модификаторов доступа **public** и **private** в определении класса.

Далее понадобится конкретный пример: реализуем *динамический массив действительных чисел* с **безопасной операцией индексации**: можно передавать отрицательные индексы, можно выходящие за границы массива - метод класса будет безопасно выводить нужный индекс (с помощью кольцевого прохода) и возвращать требуемый элемент.

Возможности классов в C++

Первое, что нужно понять при написании собственного класса (пользовательский тип, не забываем) - какие поля он будет содержать в себе? В C++ при реализации подобного контейнера обычно оперируют тремя ключевыми полями:

1. требуется указатель на данные, в нашем примере это динамический массив;
2. желательно хранить текущий размер массива - сколько элементов лежит в нём в конкретный момент времени;
3. полезно будет хранить и **ёмкость** массива (также употребляется термин *вместимость*, англ. *capacity*). Это поле позволит сократить количество перевыделений памяти в определённых случаях.

Таким образом, в C++ нам понадобятся три поля следующих типов данных

```
1 double *arr;     // указатель на массив чисел
2 size_t length;   // текущая длина массива
3 size_t capacity;// текущее количество элементов в массиве,
4                  // на который указывает указатель *arr*
```

Возможности классов в C++

Можно начать реализовывать сам класс. Назовём новый тип как **SafetyArray**

```
1 class SafetyArray
2 {
3   double *_arr;
4   size_t _length;
5   size_t _capacity;
6 };
```

По умолчанию, без указания модификаторов доступа, все поля класса - **закрытые**. Для повторения, это означает невозможность следующего кода:

```
1 SafetyArray arr; // создали объект нашего класса
2
3 // и ничего из следующих строк не смогли сделать:
4 arr._arr = new double[100]; // !! Ошибка компиляции
5 arr._length = 4;            // !! Ошибка компиляции
6 arr._capacity = 10;         // !! Ошибка компиляции
```

Для явного обозначения **закрытых** полей класса, на предыдущем слайде к названию каждого поля был добавлен *символ нижнего подчёркивания*. Это не требование языка, прото так захотелось.

Более явно определение нашего класса можно записать в виде:

```
1 class SafetyArray
2 {
3 private:
4   double *_arr;
5   size_t _length;
6   size_t _capacity;
7 };
```

На одну строчку больше, зато всё очевиднее некуда - все три поля являются **закрытыми**.

Возможности классов в C++

Небольшое отступление от реализации поставленной задачи: в C++ при определении класса можно ставить любое количество модификаторов доступа. Например, мы могли бы следать так:

```
1 class SafetyArray
2 {
3 private:
4   double *_arr;
5 private:
6   size_t _length;
7 private:
8   size_t _capacity;
9 };
```

Ничего не изменилось, все поля по прежнему закрытые. Это приводит к другой особенности: можно чередовать *открытые* и *закрытые* поля:

```
1 class SafetyArray
2 {
3 public:
4   double *_arr;
5 private:
6   size_t _length;
7 public:
8   size_t _capacity;
9 };
```

Можно, но лучше так не делать, без веских обоснований.

Продолжаем. С полями определились - переходим к *методам*. Повторим ключевые факты о методах:

- **метод** - это специальная функция, которую можно вызывать только для *объектов* конкретного класса;
- *метод класса* всегда имеет доступ к полям того объекта, для которого он был вызван;
- аналогично полям, *методы класса* могут быть как **открытыми**, так и **закрытыми**. Доступ контролируется всё теми же модификаторами **public** и **private**;
- *закрытые методы* невозможно вызвать через объект класса (его переменную);
- **методы класса** не только имеют доступ к закрытым полям (и методам) того объекта, для которого они вызваны, но и к закрытым полям (и методам) любого другого объекта того же класса.

Добавление методов в класс **SafetyArray** начнём с группы специальных методов, называем**конструкторами**.
Более-менее строгое определение уже было в пятой лекции, здесь же кратко вспомнем суть этих спец. методов.
На данный момент наш класс выглядит как

```
1 class SafetyArray
2 {
3 private:
4   double *_arr;
5   size_t _length, _capacity;
6 };
```

И его можно использовать в программе таким образом:

```
8
9 SafetyArray my_arr1, my_arr2;
```

Что же происходит в строке под номером 9?

А происходит вот что:

- Были объявлены две переменные - **my_arr1** и **my_arr2** (с точки зрения терминологии класса - 2 объекта)
- Поскольку в сам класс не добавлено ни одного конструктора, C++ сделал это сам. В итоге наш класс получил *стандратный конструктор по умолчанию*, который был вызван по **одному** разу для каждого объекта
- Работа конструктора по умолчанию проста: он выделяет блок памяти под все поля класса и ассоциирует его(блок) с конкретным объектом. Память под каждое поле располагается в порядке их объявления
- В примере для **my_arr1** и **my_arr2** блок состоит из трёх подблоков: **первый** - для указателя на **double**, **второй** и **третий** - для полей типа **size_t**
- Никакие конкретные значения ни для указателя, ни для целых чисел каждого объекта не были заданы

Возможности классов в C++

Последний пункт предыдущего слайда представляет собой реальную проблему, но не с точки зрения кода или работы C++, а со смысловой точки зрения. Проектируя динамический массив, в данный момент мы получили следующую логическую проблему: создаются два массива, у которых непонятно куда указывает указатель на данные (**_arr**) и значения полей **_length** и **_capacity** зависят от момента запуска программы в ОС (напомним, по умолчанию фундаментальные типы данных не получают никаких конкретных значений).

Правило хорошего тона

При программировании на C++ эту неопределённость в значениях **всегда следует** исключать.

Простых вариантов решения проблемы два:

1. Задать значения полей по умолчанию
2. Заменить предоставляемый по умолчанию конструктор

Рассмотрим оба способа

1. Задать значения полей по умолчанию

   Немного модифицируем класс

```
1 class SafetyArray
2 {
3 private:
4   double *_arr      = nullptr;
5   size_t _length    = 0;
6   size_t _capacity = 0;
7 };
```

   Теперь при работе кода:

```
1 SafetyArray my_arr1, my_arr2;
```

   точно известно, что поля **_arr1**, **_length** и **_capacity** равны соответственно **nullptr**, **0** и **0** для обоих объектов **my_arr1** и **my_arr2**.

Второй вариант

2. Заменить предоставляемый по умолчанию конструктор
   Для замены конструктора по умолчанию на собственный вариант, нужно реализовать *открытый* конструктор, который не принимает ни одного параметра:

```
1  class SafetyArray
2  {
3  public:
4    SafetyArray() : _arr{nullptr}, _length{0}, _capacity{0}
5    {}
6
7  private:
8    double *_arr;
9    size_t _length, _capacity;
10 };
```

   Тестовый код работает без изменений:

```
1  SafetyArray my_arr1, my_arr2;
```

   точно известны значения всех полей у каждого объекта.

Оба варианта решают поставленную задачу. Стоит отметить, что в каждом их них сначало выделяется блок памяти под все поля, и затем каждому полю присваиваются указанные значения. Во втором варианте конструктор является специальным методом класса, который вызывается только один раз (**см. пятую лекцию**)

Размышления о конструкторе без параметров

При создании подобного контейнера не совсем ясно, нужны ли нам объекты массива, которые вместо реального блока памяти ссылаются на *нулевой указатель*. Это частный вопрос более общей задачи: при добавлении одного элемента - должны ли мы расширять наш блок динамической памяти тоже только на одну ячейку? Конечно, если програма предполагает работу в условиях жёсткой экономии каждого байта памяти - ответ: да, должны.

Размышления о конструкторе без параметров и расширении массива

В другом случае, предполагая использование нашего массива в системах, где памяти достаточно, обговорим следующие условия расширения массива:

- При использовании конструктора по умолчанию будет выделять динамический массив на **четыре** элемента
- Добавим конструктор для выделения **n** элементов по желанию пользователя (значения не устанавливаются)
- Добавим конструктор для выделения **n** элементов и присвоению каждому элементу конкретного значения
- В случае нехватки выделенного блока памяти при добавлении нового элемента, будет выделятся новый блок удвоенного размера

Возможности классов в C++

Итак, приступим к реализации трёх конструкторов. Помним, что констуктуры - это спец. методы, а методы - это функции, поэтому к ним применимы правила перегрузки. Добавляем:

```
1  class SafetyArray
2  {
3  public:
4    SafetyArray() :
5      _arr{new double[4]}, _length{0}, _capacity{4}
6    {}
7
8    SafetyArray(size_t sz) :
9      _arr{new double[sz]}, _length{0}, _capacity{sz}
10   {}
11
12   SafetyArray(size_t sz, double value) :
13     _arr{new double[sz]}, _length{sz}, _capacity{sz}
14   {
15     for (size_t i = 0; i < sz; ++i) {
16       _arr[i] = value;
17     }
18   }
19
20 private:
```

Возможности классов в C++

Для конструктора без аргументов строка

```
5     _arr{new double[4]}, _length{0}, _capacity{4}
```

работает следующим образом:

- динамически выделяется массив на 4 элемента типа **double** и адрес первого элемента массива записывается в поле **_arr**;
- поля **_length** и **_capacity** присваиваются значения **нуль** и **четыре**, соответсвенно;
- стоит обратить внимание, что поля **_length**(длина массива) и **_capacity**(ёмкость) не совпадают.

Второй конструктор (**строки 8-10**) работает аналогично первому, только размер зависит от переданного в него аргумента **sz**.

Третий конструктор (**строки 12-18**) работает аналогично второму, но в дополнении каждому элементу созданного массива присваивается значение **value**. И его размер уже не нулевой, так как значения добавляются.

Возможности классов в C++

На 16-ом слайде реализации всех конструкторов написаны внутри определения класса. В тоже время, для наглядности ознакомления с классом и для разделения объявления и реализации, методы класса (в том числе и конструкторы) определяются вне фигурных скобок, окамляющих конкретный класс. В нашем случае текущее объявление класса будет выглядит так:

```
1  class SafetyArray
2  {
3  public:
4    SafetyArray();
5    SafetyArray(size_t sz);
6    SafetyArray(size_t sz, double value);
7
8  private:
9    double *_arr;
10   size_t _length;
11   size_t _capacity;
12 };
```

Для методов написаны только *объявления* в смысле C++.

Определения конструкторов пишутся отдельно

```
13
14 SafetyArray::SafetyArray() :
15   _arr{new double[4]}, _length{0}, _capacity{4}
16 {}
17
18 SafetyArray::SafetyArray(size_t sz) :
19   _arr{new double[sz]}, _length{0}, _capacity{sz}
20 {}
21
22 SafetyArray::SafetyArray(size_t sz, double value) :
23   _arr{new double[sz]}, _length{sz}, _capacity{sz}
24 {
25   for (size_t i = 0; i < sz; ++i) {
26     _arr[i] = value;
27   }
28 }
```

Как видно, всё, что нужно для определения метода вне объявления класса, это одобавить к его(метода) названию имя класса и два двоеточия.

Как правило, рекомендуется выносить определения конструкторов и методов из объявления класса.

Возможности классов в C++

Теперь объекты класса можно создавать тремя способами:

```
1  SafetyArray default_arr;
2  // Для объекта default_arr:
3  //   поле _arr указывает на массив из 4-х элементов,
4  //   _length == 0, _capacity == 4
5
6  // Вызываем конструктор с одним аргементом
7  SafetyArray preallocated_arr{16};
8  // Для объекта preallocated_arr:
9  //   поле _arr указывает на массив из 16-х элементов,
10 //   _length == 0, _capacity == 16
11
12 // Вызываем конструктор с двумя аргементами
13 SafetyArray arr_with_values{10, 3.25};
14 // Для объекта arr_with_values:
15 //   поле _arr указывает на массив из 10-х элементов,
16 //   _length == 10, _capacity == 10
17 //   каждый элемент массива равен 3.25
```

Отлично, создание объектов работает.

Но остаётся вопрос - что происходит в тот момент, когда переменная класса выходит из области видимости?

Возможности классов в C++

В C++ для переменных любых типов можно сформулировать общее правило: **как только переменная выходит** из области видимости, считаем, что выделенная под неё память **освобождена** и может быть переиспользована в следующих блоках программы.
И тут же вспоминаем - что такое "*выделенная память*"под объект конкретного класса? А это всего лишь память под каждое его поле. C++ каждому составному типу (классы и структуры) предоставляет специальный метод - **деструктор** по умолчанию - который отвечает за автоматическое освобождение памяти при выходе переменных этого типа из области видимости.
В примере с объектами **SafetyArray** при выходе из области видимости будет удалена память под указатель на **double** и два поля типа **size_t**. При этом, вся память выделенная оператором **new** останется занятой до конца работы программы.

Возможности классов в C++

Оставление массивов в памяти, до которых невозможно добраться, является очень плохой практикой при программировании на C++ (и приводит к утечкам ресурсов в ходе работы программы). Но для нашего класса это легко исправить, заменив *деструктор по умолчанию* на свой собственный.
Главные особенности пользовательского деструктора для любого класса в C++:

- деструктор - это специальный метод, который в каждом классе может быть только в одном экземпляре (в отличии от конструкторов);
- имя деструктора строго определено: это знак "тильда"(~) + название класса;
- деструктор как метод не принимает никаких параметров.

Возможности классов в C++

Добавляем пользовательский деструктор в наш класс.

Объявление класса теперь выглядит так:

```
1  class SafetyArray
2  {
3  public:
4    SafetyArray();
5    SafetyArray(size_t sz);
6    SafetyArray(size_t sz, double value);
7
8    ~SafetyArray(); // <- вот добавили
9
10 private:
11   double *_arr;
12   size_t _length;
13   size_t _capacity;
14 };
```

И реализация деструктора:

```
15
16 SafetyArray::~SafetyArray()
17 {
18   delete[] _arr;
19 }
```

Теперь при выходе объектов класса **SafetyArray**, динамически выделенная память будет корректно возвращаться в ОС.

. . . . . .

Пришло время для обдумывания методов для нашего класса. Технически уже разобрались - что такое методы. Осталось ответить на вопрос - чем являются методы с логической точки зрения?

Ответ можно сформулировать так: методы - это **действия** (операции), которые мы хотим выполнять над конкретными объектами.

Для того, чтобы понять чего же хочется, нужно порассуждать логически над объектом, который мы выражаем в программе.

Возможности классов в C++

Мы реализуем динамический массив. Забывая на время конкретику языка C++, пробуем представить массив как объект-абстракцию. Сам массив является набором элементов, каждый из которых доступен по индексу. Аналогий в реальном мире можно подобрать достаточно, подумайте, для примера, об автопарковке, где все места пронумерованы. Машины на конкретном месте меняются, само место с присвоенным ему индексом - остаётся.

Итак, абстрактно представляя массив как некий объект, какие бы действия (или операции) с ним мы бы могли делать? Для начала список может быть такой:

- узнавать длину массива;
- посмотреть что за предмет находится на конкретном месте (по конкретному индексу);
- положить предмет на конкретное место;
- раз говорим о динамических (или саморасширяющихся) массивах, то хотелось бы иметь возможность добавления новых элементов в него.

Возможности классов в C++

Сформулировав подобный список, должно приходить понимание, что за методы будут нужны в программе.

Стоит отметить, что подобные рассуждения происходят и при переносе других объектов в программы (причём не только на языке C++, но и любом другом, поддерживающим в той или иной степени ООП). Кроме массива и подобных ему контейнеров, программа может работать с файлами, словарями, реализовывать пространственный вектор или управлять светодиодом. Во всех этих случаях использование классов позволяет скрыть техническую реализацию (даже термин имеется - *инкапсуляция*) в описании класса и использовать конкретные объекты для выполнения действий с ними. Так, в примере с RGB-светодиодом, при наличии хорошо продуманного класса, для программного изменения цвета устройства достаточно для объекта вызвать метод, который принимает требуемый цвет. А уже где-то внутри метода отправится сигнал на контролер для изменения напряжения, цвет поменяется, все довольны.

Возможности классов в C++

Возращаемся к разработке класса **SafetyArray**. Добавим объявление всех методов с 25 слайда.

```
1  class SafetyArray
2  {
3  public:
4    SafetyArray();
5    SafetyArray(size_t sz);
6    SafetyArray(size_t sz, double value);
7
8    ~SafetyArray(); // деструктор
9
10   double elem_at(int index);
11   size_t length();
12
13   SafetyArray& push_to_end(double new_value);
14   SafetyArray& set_at(int index, double value);
15
16 private:
17   double *_arr;
18   size_t _length;
19   size_t _capacity;
20 };
```

Добавленные методы предназначены для:

1. Строка

```
10  double elem_at(int index);
```

добавляет метод для получения значения элемента массива по заданному индексу;

2. Строка

```
11  size_t length();
```

добавляет метод, возвращающий текущую длину массива;

3. Метод

```
13  SafetyArray& push_to_end(double new_value);
```

добавляет новый элемент в конец массива. У него несколько необычное возвращаемое значение - ссылка на экземпляр класса. Это сделано для того, чтобы был возможен код вида:

```
1 arr_obj.push_to_end(5.6).push_to_end(105.25);
```

4. Метод

```
13  SafetyArray& set_at(int index, double value);
```

установит новое значение **value** элементу массива с индексом **index**.

Возможности классов в C++

У первых двух методов предыдущего слайда есть особенность: они не меняют внутренних полей объекта. Что взятие значения конкретного элемента, что получение длины массива - означают одинаковое действие: взять некоторую область памяти и считать оттуда значения. Такие методы являются **константными** (или неизменяемыми) по отношению к тому объекту, для которого они были вызваны. Более того, они являются *неизменяемыми* и по своему смыслу. Современный C++ позволяет добавить ключевое слово **const** после списка аргументов для того, чтобы компилятор понял, что мы объявили константный метод, и проверял, что внутри него ни одно поле объекта не меняется. Поэтому,

Правило хорошего тона

При написании собственных классов **все методы**, которые не меняют полей объекта, объявляйте **константными**.

Кроме проверки на этапе компиляции, константные методы помогают современным компиляторам в ряде случаев создать более эффективный машинный код.

Объявление класса чуть изменяется

```
1  class SafetyArray
2  {
3  public:
4    SafetyArray();
5    SafetyArray(size_t sz);
6    SafetyArray(size_t sz, double value);
7
8    ~SafetyArray(); // деструктор
9
10   double elem_at(int index) const; // константный метод раз
11   size_t length() const;           // константный метод два
12
13   SafetyArray& push_to_end(double new_value);
14   SafetyArray& set_at(int index, double value);
15
16 private:
17   double *_arr;
18   size_t _length;
19   size_t _capacity;
20 };
```

Реализуем метод добавления элемента в массив (**push_to_end**).

```
1  SafetyArray& SafetyArray::push_to_end(double new_value)
2  {
3    if ( _length < _capacity ) {
4      _arr[_length] = new_value;
5    } else {
6      _capacity *= 2;
7      double *new_arr = new double[_capacity];
8      for (size_t i = 0; i < _length; ++i) {
9        new_arr[i] = _arr[i];
10     }
11     new_arr[_length] = new_value;
12     delete[] _arr;
13     _arr = new_arr;
14   }
15
16   ++_length;
17   return *this;
18 }
```

Разбор метода **push_to_end**:
строка 3: проверяем, хватает ли заранее выделенной памяти для записи нового значения на последнее место в массиве. За текущее количество элементов в памяти отвечает поле **_capacity**;
строка 4: если хватает, помещаем новое значение в конец массива;
строки 6-13: если не хватает, будем расширять имеющийся массив. Для этого: увеличиваем ёмкость в два раза и выделяем новый динамический массив (**new_arr**) (**строки 6-7**). Копируем в него элементы из текущего дин. массива (поле **_arr**) плюс добавляем новый элемент (**8-11**). Удаляем текущий массив из памяти и приваиваем полю **_arr** новый массив увеличенной ёмкости (**12-13**);
строка 16: увеличиваем длину массива на единицу;
строка 17: встретилось новое ключевое слово C++ - **this**

Разбор метода **push_to_end**:
Что за **this** такой?
C++ неявно добавляет к каждому классу указатель на этот самый тип данных. Этот указатель доступен в каждом объекте и всегда содержит адрес блока памяти, в котором располагаются его (объекта) поля. В явном виде схематично показать **this** можно следующим кодом:

```
1 class SafetyArray
2 {
3 // ...
4 private:
5   SafetyArray *this;
6 };
```

, что сильно похоже на добавление указателя на структуру в качестве её поля (лекция 5). Главная черта **this** - невозможно поменять его значение, этот указатель всегда будет ссылаться на тот объект, внури которого он создан.

Разбор метода **push_to_end**:
А поскольку **this** - указатель, внутри любого метода класса он может использоваться для доступа к любым полям (и методам) с помощью оператора разыменования **->**

```
1  /*
2    Как пример, в каждом методе класса SafetyArray
3    вместо прямого обращения к полям, мы могли бы
4    писать следующие строки:
5  */
6    this->_arr;
7    this->_capacity;
8    this->_length;
9
10 /*
11   А также внутри одного метода можно вызывать любой
12   другой через *this*
13 */
14   this->length();
```

Явное использование **this** полезно в случаях, когда название поля и аргумента, передаваемого в метод, совпадают.

Разбор метода **push_to_end**:
В **строке 17** 31-го слайда происходит разыменование **this** и ***this** означает тот самый объект, к которому применяется метод **push_to_end**. И этот объект возращается по ссылке в соответствии с сигнатурой метода.

.....

Метод **elem_at**: получение значения элемента по индексу.
По задумке для обращения к элементу массива можно будет передавать как положительный , так и отрицательный индексы.
Разберём как они выглядят на примере (при этом реализуем "компьютерную"индексацию: первый элемент начинается с нуля):

```
массив, 4 элемента:     [4.5, 3.78, -6.78, 4.5]
положительные индексы:    0     1      2     3
отрицательные индексы:   -4    -3     -2    -1
```

Хочется верить, что идея отрицательной индексации понятна из примера.

Метод **elem_at**: получение значения элемента по индексу.
Вторая идея, которая будет реализована - *кольцевой индекс*. В случае, если переданный индекс будет превышать длину массива (как при отступе от нуля, так и с конца массива), он будет "закольцовываться": к переданному индексу условно вычитается или прибавляется длина до тех пор, пока получаемый индекс не попадёт в диапазон $[0; \text{length} - 1]$. Функция расчёта внутреннего индекса выглядит так:

$$\text{right\_index} = \begin{cases} 0, & \text{index} == 0 \\ \text{index} \ \% \ \text{length}, & \text{index} > 0 \\ \text{length} - (|\text{index}| \ \% \ \text{length}), & \begin{cases} \text{index} < 0, \\ |\text{index}| \% \text{length}! = 0 \end{cases} \\ 0, & \begin{cases} \text{index} < 0, \\ |\text{index}| \% \text{length} == 0 \end{cases} \end{cases}$$

Здесь **length** - длина массива, **index** - переданный в метод индекс.

Возможности классов в C++

Метод **elem_at**: получение значения элемента по индексу.
Для расчёта внутреннего индекса добавим *закрытый метод*. Не то, чтобы без него прям никак, но надо же на примере ознакомится с возможностью помещать методы под модификатор **private**.
Объявление класса становится:

```
1  class SafetyArray
2  {
3  public:
4    //... конструкторы и деструктор — без изменений, слайд 30
5
6    double elem_at(int index) const; // константный метод раз
7    size_t length() const;           // константный метод два
8
9    SafetyArray& push_to_end(double new_value);
10   SafetyArray& set_at(int index, double value);
11
12 private:
13   double *_arr;
14   size_t _length;
15   size_t _capacity;
16
17   size_t compute_index(int index) const; // закрытый метод
18 };
```

Метод **elem_at**

Реализацию *закрытого метода* также можно выносить из объявления класса:

```
1  size_t SafetyArray::compute_index(int index) const
2  {
3    if (index == 0) {
4      return 0;
5    } else if (index > 0) {
6      return std::abs(index) % _length;
7    }
8
9    size_t rel_index = std::abs(index) % _length;
10   if (rel_index == 0) {
11     return 0;
12   }
13
14   return std::abs(_length - rel_index);
15 }
```

Метод **elem_at**

Вычисление внутреннего индекса готово, можно приступать к реализации метода **elem_at**. Но и здесь осталось решить ещё один важный вопрос - что метод доступа к конкретному элементу будет возвращать, если в массив не было добавлено ни одного числа? Можно было бы возвращать нуль по умолчанию (у нас же массив действительных чисел), но луше будет возвращать специальное значение - **NaN**. Говорящее о том, что нечего пользоваться пустым массивом.

Сама реализация метода достаточно проста:

```
1 double SafetyArray::elem_at(int index) const
2 {
3   if (_length == 0) {
4     return std::nan("");
5   }
6
7   return _arr[ compute_index(index) ];
8 }
```

Индикатор константности (**const**) здесь важен: без него C++ считает, что метод имеет другую сигнатуру (хотя список аргументов не меняется).

И осталась реализация методов **set_at** и **length**

```
1  size_t SafetyArray::length() const
2  {
3    return _length;
4  }
5
6  SafetyArray& SafetyArray::set_at(int index, double value)
7  {
8    if (_length != 0) {
9      _arr[ compute_index(index) ] = value;
10   }
11
12   return *this;
13 }
```

Если массив не содержит элементов (имеет нулевой логический размер), операция установки значения конкретному элементу просто ничего не будет делать.

Не прошло и 40 слайдов, а получилось нечто рабочее

```
1  SafetyArray my_test_arr; // создаём массив
2
3  // заполняем первые 4 элемента
4  my_test_arr.push_to_end(4.5).push_to_end(7.89)
5             .push_to_end(0.555).push_to_end(-8.4);
6
7  cout << "Длина массива: " << my_test_arr.length()
8       << "\nЕго элементы:\n[";
9  for (size_t i = 0; i < my_test_arr.length(); ++i) {
10   if (i != 0) {
11     cout << ", ";
12   }
13
14   cout << my_test_arr.elem_at(i);
15 }
16 cout << "]\n";
```

И почти всё хорошо.

Что же может пойти не так? Рассмотрим следующий пример

```
1  SafetyArray arr1, arr2;
2
3  arr1.push_to_end(555.5);
4  arr1.push_to_end(9.992);
5
6  arr2 = arr1;
7  arr2.set_at(1, 6.78); // устанавливаем второй элемент
8
9  cout << "2-ой элемент из arr1: " << arr1.elem_at(1) << "\n";
10 cout << "2-ой элемент из arr2: " << arr2.elem_at(1) << "\n";
```

Что здесь произошло: изменение второго элемента массива **arr2**, привело и к изменению второго элемента в первом массиве. Более того, программа вылетела с ошибкой двойного освобождения одного блока динамической памяти. Нехорошо.

Проблема вызвана тем, что для каждого класса C++ в дополнению к деструктору по уполчанию, всегда добавляет **оператор присваивания** по умолчанию. Он же - оператор "=". Его стандартное поведение достаточно прямолинейно. Встречая строку с объектами пользовательского типа вида:

```
1 arr2 = arr1;
```

оператор по умолчанию просто берёт все значения полей из правого объекта (в данном случае - **arr1**) и записывает их в соответствующие поля объекта слева (**arr2**). В нашем случае получилось, что адрес из одного указателя был скопирован в другой и два объекта стали указывать на один и тот же динамический массив. А доступ до другого динамического массива, на который указывало поле **_arr** в объекте **arr2** до присваивания, просто пропал: очередная утечка в ходе работы программы. Надо устранять.

Возможности классов в C++

И свойство языка C++, позволяющее устранить возникшую проблему, называется **перегрузкой операторов**.
Как известно (например, https://github.com/posgen/OmsuMaterials/wiki/Operators-priority), в C++ достаточно много операторов: это и математические операторы, и операторы круглых/квадратных скобок, операторы сравнения, оператор приведения типов и прочие. И язык позволяет практически все из них добавлять в *пользовательские типы данных*. Исключения составляют, например, оператор области видимости (**::**) и операторы обращения к полям/методам объектов класса и структур.
Перегрузка операторов делается по общей схеме:

1. обдумываем, хотим ли мы объекты своего класса преобразовывать с помощью какого-либо оператора;
2. если всё-таки хотим, ищем в книгах/интернете сигнатуру нужного нам оператора (возвращаемое значение, количество аргументов);
3. сами по себе операторы - это либо свободные функции, либо методы конкретного класса. Разве что, на их имена налагается строгое ограничение (имя состоит из ключевого слова **operator** и **знака оператора**). Соответственно, реализуем нужную функцию/метод;

Хоть и старую, но хорошую статью по основам перегрузки операторов можно найти тут: https://habr.com/post/132014/

Для класса **SafetyArray** нас интересует практическая сторона - что делать с некоректным присвоением по умолчанию. Один из вариантов - просто запретить присваивание разных объектов:

```
1  class SafetyArray
2  {
3  public:
4    //... конструкторы и деструктор — без изменений, слайд 30
5    //... методы тоже без изменений — слайд 37
6
7    SafetyArray& operator=(const SafetyArray& rhs) = delete;
8
9  private:
10   //... закрытые поля и методы — слайд 37
11 };
```

В таком случае никакие присвоения разных объектов компилятор не доспустит. Но иметь присвоение массивов всё-таки хочется. Идём к следующему варианту.

Возможности классов в C++

Объявление оператора становится таким:

```
1  class SafetyArray
2  {
3  public:
4    //... конструкторы и деструктор — без изменений, слайд 30
5    //... методы тоже без изменений — слайд 37
6
7    SafetyArray& operator=(const SafetyArray& rhs);
8
9  private:
10   //... закрытые поля и методы — слайд 37
11 };
```

Здесь мы объявили оператор присвоения как **метод** нашего класса. Его сигнатура достаточно стандатна:

- возращаемое значение - ссылка на тот объект, который стоит слева от присваивания;
- аргумент - константная ссылка на объект, который стоит справа от оператора присванивания;
- название метода - определено правилами C++.

Реализация оператора присвоения:

```
1  SafetyArray& SafetyArray::operator=(const SafetyArray& rhs)
2  {
3    if (this != &rhs) {
4      delete[] _arr;
5      _length   = rhs._length;
6      _capacity = rhs._capacity;
7
8      _arr = new double[_length];
9      std::copy_n(rhs._arr, _length, _arr);
10   }
11
12   return *this;
13 }
```

В строке 3 проверяем. не пытается ли кто-нибудь присвоить объект самому себе.

В строке 9 используем функцию копирования элементов из одного массива в другой из библиотеки <**algorithm**>.

Теперь код с 42-го слайда работает без проблем.

Возможности классов в C++

С минимумом комментариев приведём перегрузку **оператора вывода в поток** для нашего класса

```
1  class SafetyArray
2  {
3  public:
4    //... конструкторы и деструктор — без изменений, слайд 30
5    //... методы тоже без изменений — слайд 37
6
7    SafetyArray& operator=(const SafetyArray& rhs);
8    friend std::ostream& operator<<(std::ostream&, const
         SafetyArray&);
9  private:
10   //... закрытые поля и методы — слайд 37
11 };
```

- Что за **friend** - читаем хотя бы здесь: https://github.com/posgen/OmsuMaterials/wiki/OOP:-access-to-class-members
- Здесь объявлена свободная дружественная функция. При объявлении, напомним, имена аргументов можно и упускать.
- Название оператора и его сигнатура - соответствует общим реккомендациям C++.

Реализация оператора вывода:

```
1  std::ostream& operator<<(std::ostream& os, const SafetyArray& ↩
       rhs)
2  {
3    os << "{Длина: " << rhs.length() << ", ёмкость: "
4       << rhs._capacity << "\n";
5
6    os << "[";
7    for (size_t i = 0; i < rhs.length(); ++i) {
8      if (i != 0) { os << ", "; }
9
10     os << rhs.elem_at(i);
11   }
12   os << "]}";
13
14   return os;
15 }
```

И становится возможным печать объекта массива хоть в консоль, хоть в файл. Пример:

```
1 SafetyArray arr3, arr4{16}, arr5{20, 4.4};
2
3 std::cout << arr3 << "\n\n"
4           << arr4 << "\n\n"
5           << arr5 << "\n\n";
```

Полный пример в виде программы доступен тут:
https://github.com/posgen/OmsuMaterials/tree/master/2course/Programming/docs/lectures/2017_2018/FP/примеры_с_практик