V

Объекты, классы, все дела

C++: подопытный класс

```
1  class FP003Array
2  {
3  public:
4    FP003Array() = default;
5    FP003Array(size_t arr_sz);
6    FP003Array(const FP003Array& other);
7    ~FP003Array();
8
9    FP003Array& operator=(const FP003Array& rhs);
10
11   int& operator[](size_t index);
12   const int& operator[](size_t index) const;
13
14   size_t length() const;
15   void push(int new_elem);
16
17 private:
18   ...
19 };
```

C++: подопытный класс

Неофициальное определение

В некоторых кругах в C++, открытые методы класса называют его **интерфейсом**.
Подразумевая, что только через открытые методы происходит использование объектов типа.

Исправление копирующего присваивания

```
1  FP003Array&
2  FP003Array::operator=(const FP003Array& rhs)
3  {
4    if (this == &rhs) { return *this; } // <- here
5
6    delete[] data_;
7
8    length_ = rhs.length_;
9    capacity_ = rhs.capacity_;
10
11   data_ = new int[capacity_];
12   for (size_t i = 0; i < rhs.length_; i++) {
13     data_[i] = rhs.data_[i];
14   }
15
16   return *this;
17 }
```

C++: подопытный класс

Добавление тройки методов

```
1  class FP003Array
2  {
3  public:
4    // constructors
5
6    FP003Array& operator=(const FP003Array& rhs);
7    int& operator[](size_t index);
8    const int& operator[](size_t index) const;
9
10   int back() const;              // new
11   int front() const;             // new
12   size_t length() const;
13   int pop();                     // new
14   void push(int new_elem);
15
16 private:
17   // private part
18 };
```

C++: подопытный класс

back – получение значения последнего элемента массива

```
1 int FP003Array::back()
2 {
3   const int idx = length_ == 0 ? 0 : length_ - 1;
4   return data_[idx];
5 }
```

C++: подопытный класс

front – получение значения первого элемента массива

```
1 int FP003Array::front()
2 {
3   return data_[0];
4 }
```

C++: подопытный класс

pop – удаление последнего элемента. Элемент логически удаляется из массива (длина уменьшается на единицу). Метод возвращает значение удалённого элемента.

```
1  int FP003Array::pop()
2  {
3    const int idx = length_ == 0 ? 0 : length_ - 1;
4    const int deleted = data_[idx];
5
6    if (length_ != 0) {
7      length_ -= 1;
8    }
9    return deleted;
10 }
```

C++: копирование объектов, воспоминание о былом

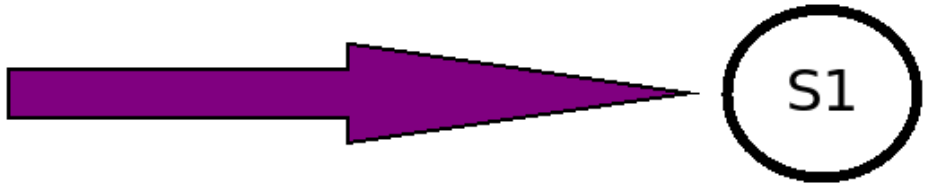
Конструктору копий и *оператору копирующего присваивания* следует всегда иметь одинаковый тип у передаваемого в них параметра. Хотя C++ не запрещает возможность убирать в любом из методов квалификатор **const**.

```
1 class FP003Array
2 {
3   ...
4   FP003Array(const FP003Array& other);
5   FP003Array& operator=(const FP003Array& rhs);
6   ...
7 };
```

C++: копирование объектов, воспоминание о былом

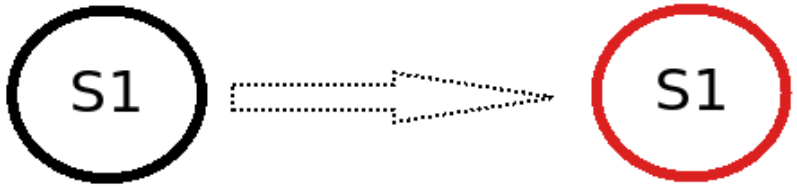
```
class A
{
public:
  ...
private:
  ...
};
```

Создаётся объект с внутренним состоянием S1



1) До копирования

2) После копирования



Два разных объекта, но с одинаковым внутренним состоянием

C++: перемещение объектов

Концепция **перемещения**: возможность определелить *специальные* конструктор и оператор присваивания для устранения копирования. Как правило необходимо в случаях, когда в создаваемом классе происходит управление каким-либо ресурсом. Самые встречающиеся типы ресурсов – *динамическая память* и различные *файловые потоки*.

Сигнатуры перемещающих методов:

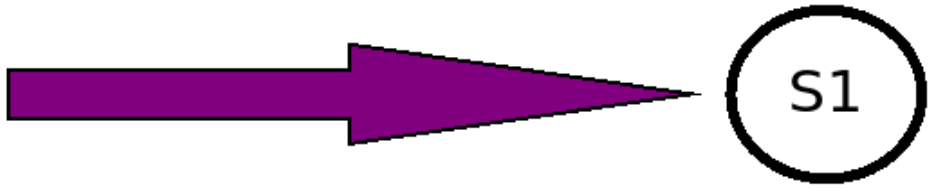
```
1 class FP003Array
2 {
3   ...
4   FP003Array(FP003Array&& other);
5   FP003Array& operator=(FP003Array&& rhs);
6   ...
7 };
```

Параметрами методов выступают *ссылки на временные значения* (они же – ссылки на **rvalue**).

C++: перемещение объектов

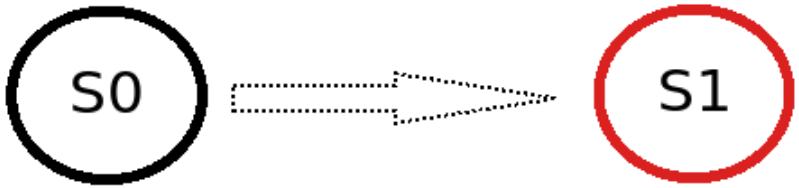
```
class A
{
public:
  ...
private:
  ...
};
```

Создаётся объект с внутренним состоянием S1



1) До перемещения

2) После перемещения



Два разных объекта, и состояние первого "перешло" во второй.
И первый объект получил новое состояние.

Перемещающий конструктор для типа **FP003Array**

```
1 FP003Array::FP003Array(FP003Array&& other) :
2   data_{other.data_}, length_{other.length_},
3   capacity_{other.capacity_}
4 {
5   other.data_ = nullptr;
6   other.length_ = other.capacity_ = 0;
7 }
```

Новый создаваемый объект просто копирует все поля объекта **other**. В свою очерень, **other** теряет связь с тем массивом, который ему принадлежал (*строка 5*) При этом не происходит никакого нового выделения памяти и поэлементного копирования. Сравните с конструктором копий.

Перемещающее присваивание для типа **FP003Array**

```
1  FP003Array&
2  FP003Array::operator=(FP003Array&& rhs)
3  {
4    delete[] data_;
5
6    data_ = rhs.data_;
7    length_ = rhs.length_;
8    capacity_ = rhs.capacity_;
9
10   rhs.data_ = nullptr;
11   rhs.length_ = rhs.capacity_ = 0;
12
13   return *this;
14 }
```

На данный момент пока используем явное удаление данных того объекта, куда перемещается новое состояние (*строка 6*).

C++: использование перемещающих методов

Первый пример: когда используются **временные объекты** класса.

```
1 FP003Array obj = FP003Array(30);
2 //...
3 obj = FP003Array(10);
```

C++: использование перемещающих методов

Второй пример: возвращение объекта класса по значению из функции. Определим функцию, которая создаёт массив заданного размера и заполняет его значениями с заданным шагом

```
1  FP003Array
2  make_array(size_t array_size, int start,
3             int step)
4  {
5    FP003Array new_obj{array_size};
6
7    for (size_t i = 0; i < new_obj.length(); i++) {
8      new_obj[i] = start + i * step;
9    }
10
11   return new_obj;
12 }
```

В обоих примерах – компилятор имеет всё необходимое для исключения копирования.

C++: использование перемещающих методов

Третий пример: как перемещать состояния переменных? Перемещающие методы принимают в качестве параметра – временный объект. Но иногда случаются ситуации, когда хотелось бы применить перемещение к обычным переменным. Для обеспечения такой возможности C++ предоставляет стандартную функцию **std::move**, определённую в заголовочном файле <**utility**>. Хотя, если подключается практически любой заголовочный файл стандартной библиотеки C++ (не C-часть), то <**utility**> будет подключён где-то внутри и **std::move** становится доступным.

C++: использование перемещающих методов

Третий пример: как перемещать состояния переменных?

Базовое использование **std::move**:

```
1 FP003Array obj1{10}, obj2{25};
2 obj1 = std::move(obj2);
3 cout << obj1.length() << endl;
```

- динамический массив из **obj2** перешёл в **obj1** бех копирования;
- объект **obj2** остался в состоянии «массива нулевой длины»;
- вообще говоря, есть логическое ограничение: после того, как применили перемещение к объекту (**obj2** в данном случае), его дальнейшее использование **запрещено**. Это не ограничение C++, но хороший совет избежать неожиданных проблем.

C++: использование перемещающих методов

Третий пример: когда **std::move** полезен?

Мы могли бы организовать функцию обмена значениями двух объектов-массивов следующим образом:

```
1 void swap_array(FP003Array& lhs, FP003Array& rhs)
2 {
3   FP003Array tmp = lhs;
4   lhs = rhs;
5   rhs = tmp;
6 }
```

Данный код рабочий, но в отсутствие перемещающих методов, он приводит к трём копированиям массивов (строки 3, 4, 5). Для мелких массивов можно пережить. При крупных – ненужное замедление операций.

C++: использование перемещающих методов

Третий пример: когда **std::move** полезен?

Тут и помогает перемещение:

```
1 void swap_array(FP003Array& lhs, FP003Array& rhs)
2 {
3   FP003Array tmp = std::move(lhs);
4   lhs = std::move(rhs);
5   rhs = std::move(tmp);
6 }
```

Небольшая модернизация – и копирование устранено.

Более того, именно так работает стандартная функция **std::swap** из <**algorithm**>.

C++: использование перемещающих методов

С учётом знаний про **std::swap**, перемещающий оператор присваивания может быть модернизирован как :

```
1 FP003Array& FP003Array::operator=(FP003Array&& ↩
      rhs)
2 {
3   std::swap(data_, rhs.data_);
4   std::swap(length_, rhs.length_);
5   std::swap(capacity_, rhs.capacity_);
6
7   return *this;
8 }
```

Тут устраняем и ненужное явное удаление. Предполагается, что удаление будет сделано автоматически при уничтожении временного объекта **rhs**.

C++: резюме по созданию объектов

- Что C++ может создавать неявно для каждого типа?
  1. конструктор по умолчанию;
     Создаёт объект класса путём применения *конструктора по умолчанию* к каждому полю в порядке **их объявления**.
  2. копирующий конструктор
  3. копирующее присваивание
     Обе операции копируют значение каждого поля также в порядке **объявления полей**.
  4. перемещающий конструктор
  5. перемещающее присваивание
     Операции перемещают значение каждого поля снова в порядке **объявления полей**.
  6. деструктор
     Вызывает деструктор каждого поля в порядке, **обратном объявлению полей**.

C++: резюме по созданию объектов

- Как явно использовать указанные выше реализации конструкторов?
  Добавление или удаление указанных реализаций происходит с использованием ключевых слов **delete** и **default**.

```
1 class DefaultTest
2 {
3 public:
4   DefaultTest() = default;
5   DefaultTest(const DefaultTest& other) = ↩
        delete;
6 };
```

Данный класс явно сообщает о том, что *конструктор по умолчанию* (строка 4)) использует стандартную реализацию (предыдущий слайд, пункт 1). А копирующий конструктор (строка 5)) явно удалён.

C++: резюме по созданию объектов

И набор правил, касающийся неявного создания перечисленных методов.

1. Добавление неявных конструкторов к типу

C++ автоматически добавит к типу любой из пяти вышеперечисленных *конструкторов* **тогда и только тогда**, когда соответствующие конструкторы **не определены** в самом типе и **в любом** из его полей.

2. Неявный деструктор

Если для составного типа деструктор не определён явно, его создаст компилятор неявно. Всегда.

3. Конструктор по умолчанию

Вспомним и его: как только в типе определён **любой** конструктор, конструктор по умолчанию **никогда** не создаётся.
Здесь есть нюанс только с явным использованием или запретом реализаций по умолчанию (слайд 24). Согласно правилам C++, если тип использует явные реализации или их запрет, то это считается ситуацией, когда происходит явное определение конструктора.

3. Конструктор по умолчанию

На примере, пусть есть определение класса:

```
1 class NoCopyType
2 {
3 public:
4   NoCopyType(const NoCopyType& other) = delete;
5   NoCopyType&
6    operator=(const NoCopyType& rhs) = delete;
7 };
```

С точки зрения C++, даже удаление копирующего конструктора означает, что мы определили свой конкструтор самостоятельно, поэтому никакого *конструктора по умолчанию* не создаётся. Аналогичная ситуация и с использованием реализаций по умолчанию с помощью **default**.

4. Копирующие методы

Копирующие конструктор и оператор присваивания создаются неявно в двух случаях (при условии, что они не определены явно в типе):

1. когда явно определён деструктор класса;
2. когда определён только один из двух копирующих методов (или конструктор, или оператор присваивания). В этом случае, недостающая операция будет создана C++ **неявно**.

В дополнении, неявного добавления копирующих методов не происходит, если (*нестатические*) поля составного типа –

- – являются **ссылками**;
- – являются **константными**.

4. Копирующие методы

Для примера,

```
1  class NoCopy2
2  {
3  public:
4    ~NoCopy2() = default;
5  private:
6    int &i_ref;
7  };
8
9  NoCopy2 obj1, obj2;
10 obj1 = obj2; // <= Compile error
```

Поскольку поля класса является *ссылкой* на переменную типа **int**, копирующие методы не созданы неявно, несмотря на явно определённый деструктор.

5. Перемещающие методы

Перемещающие конструктор и оператор не создаются неявно, когда:

1. явно определён деструктор класса;
2. определён любой из копирующих методов;
3. определён хотя бы один перемещающий метод из двух;
4. в классе существующют нестатические поля, являющиеся **ссылками** или **ссылками**.

C++: резюме по созданию объектов

Кому интересно, как наглядно увидеть, что происходит?

```
1  class TestCtor
2  {
3  public:
4    TestCtor() { std::cout << "default ctor\n"; }
5    TestCtor(const TestCtor& obj)
6    { std::cout << "copy ctor\n"; }
7    TestCtor(TestCtor&& obj)
8    { std::cout << "move ctor\n"; }
9    ~TestCtor() { std::cout << "de-ctor\n"; };
10   TestCtor& operator=(const TestCtor& obj)
11   { std::cout << "copy assign\n"; }
12   TestCtor& operator=(TestCtor&& obj)
13   { std::cout << "move assign\n"; }
14 };
```

Берём подобный класс, создаём объекты в разных областях видимости (блок, функция (в том числе, её параметры) и т.п.) и смотрим на вывод в терминал

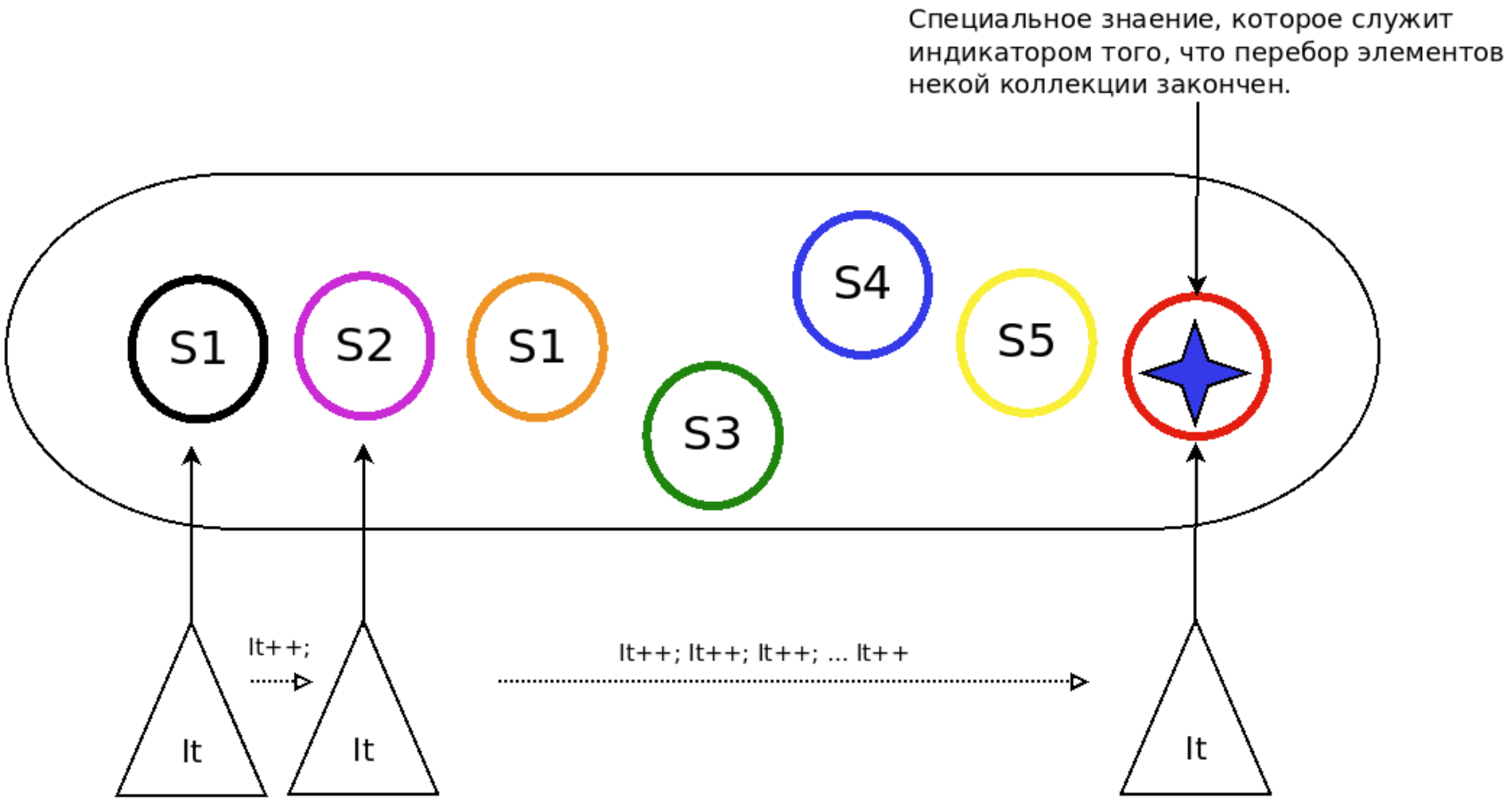
Итераторы в C++

Идея

Предоставить одинаковый (унифицированный) способ доступа к элементам различных коллекций (они же – наборы элементов).

В качестве коллекций: массивы, различные списки, очереди, множества, да и строка (как коллекция символов).
Если способ доступа одинаков: одни и те же алгоритмы можно применить к различным типам.
Большая часть стандартной библиотеки C++ реализует концепцию итераторов.

Идея иллюстрированная



C++: итераторы

Что на картинке:

- есть коллекция, в ней хранится набор однотипных различных объектов;
- у каждого объекта – есть какое-то состояние (набор значений полей в некоторый момент времени);
- итератор – это тоже некий объект, умеющий ссылаться на объект из коллекции и пробегать по всем элементам;
- коллекция предоставляет специальное значение, по которому можно определить, что её обход закончен и были перебраны все её объекты.

Пример итератора, которым уже пользовались

Кому показалось, что итераторы похожи на указатели – так и есть. Указатель на массив в смысле языка C++ (то есть – коллекция элементов какого-либо типа, расположенная непрерывно в памяти) является примером итератора. Он может переходить по элементам массива (причём – в обе стороны), он предоставляет доступ к самому элементу через *разыменование*. Разве что, возникает вопрос об специальном индикаторе окончания перебора. В качестве такого для статических/динамических массивов используют первый блок памяти, следующий за последним (нужным) эмементом коллекции.

И при разработке C++ решили, что неплохо все операции с указателями сделать общим требованием для итераторов.

C++: итераторы

По аналогии с указателями, тип итератора должен предоставить следующие операции:

- оператор *разыменования* – предоставлять доступ к элементу коллекции;
- операторы *инкремента/декремента* – способность объекта итератора последовательно переходить с одного элемента коллекции на другой;
- оператор *проверки на неравенсво* с другим объектом итератора;
- **опционально** оператор *сложения с целым числом* – аналог прибавления целого числа к указателю, **непоследовательный** переход с одного элемента на другой;
- **опционально** операторы *сравнения* с другим объектом итератора, *вычитания* двух объектов итератора (количество элементов между ними), *индексации* (отступ с одновременным разыменованием).

Свойства итератора, минимальный набор

Произвольный итератор:

- знает, как получить доступ к элементу коллекции, в том числе и для изменения его значения (*разыменование*);
- знает, как переходить от одного элемента к другому. При полном проходе каждый элемент попадается только один раз (*пре-инкремент*);
- поддерживает проверку на неравенство с другим объектом итератора (*оператор «*$!=$*»*).

Пример с неявным использованием итераторов – диапазонный цикл **for**

```
1 string my_str = "All right now.";
2
3 for (char symb : my_str) {
4   if (symb != ' ') {
5     cout << "{" << symb << "}";
6   }
7 }
```

Оборачиваем каждый *непробельный* символ строки в фигурные скобки.

Работа такой формы цикла **for** обеспечивается тем, что класс **string** предоставляет итератор в виде отдельного класса. Технически код с предыдущего слайда разворачивается в что-то подобное:

```
1 string my_str = "All right now.";
2
3 for (string::iterator it = my_str.begin();
4      it != my_str.end(); it++) {
5   char symb = *it;
6   if (symb != ' ') {
7     cout << "{" << symb << "}";
8   }
9 }
```

C++: итераторы

На предыдущем слайде:

- **string::iterator** – тип итератора, определён внутри класса **string**;
- **begin** – метод, возвращаюший объект итератора, который ссылается на первый элемент строки;
- **end** – метод, возвращающий специальное знаение, сигнализирующее о том, что элементы в коллекции все пройдены;
- **it++** – последовательный проход объектом итератора по всем символам строки;
- строка 5 – получение значения текущего символа путём *разыменования* итератора.

C++: итераторы

Всё это обсуждение итераторов начато ещё и для того, чтобы для класса **FP003Array** сделать возможным такую работу с его объектами:

```
1  FP003Array obj = FP003Array(8);
2  init_by_squares(obj);
3
4  for (int elem : obj) {
5    elem += *obj.begin();
6  }
7
8  for (const int elem : obj) {
9    cout << elem << ' ';
10 }
11 cout << endl;
```

Первый цикл прибавляет к каждому елементу массива значение первого элемента. Второй – просто печатает массив в терминале.

C++: итераторы

Чтобы заставить код выше работать:

- продумать тип для представления итератора;
- реализовать методы **begin** и **end** в типе **FP003Array**.

begin и end

Повтора ради, смысл этих двух действий в том, чтобы получить значения итераторов, которые будут указывать на первый элемент коллекции и специальное значение кокончания коллекции, соответственно. C++ предоставляет выбор – можно определять эти действия через *методы* класса-коллекции, а можно – в виде отдельных *функций*.

C++: итераторы

Поскольку тип **FP003Array** хранит массив в виде непрерыного куска динамической памяти, то наиболее простой способ реализовать итератор – использовать указатели C++.

```
1  class FP003Array
2  {
3    ...
4    using Iterator = int*;
5
6    Iterator begin() { return data_; };
7    Iterator end() {return data_ + length_; };
8
9    ...
10 };
```

C++: итераторы

Основные моменты:

- использовали псевдоним для задания типа итератора (строка 4);
- для получения объекта итератора, указывающего на первый элемент, просто возвращаем адрес внутреннего блока памяти (строка 6);
- для специального объекта итератора, означающего конец коллекции, используем первый блок памяти **за пределом** памяти под все *текущие* элементы (строка 7);
- типы C++ могут быть вложены в любой составной тип (класс/структура). Тогда текущий класс выступает как пространство имён и до вложенного типа можно добраться используя синтаксис **FP003Array::Iterator**;
- весь код со слайда 42 начал компилировать и исполняться без проблем.

C++: итераторы

Но есть нюанс: указатель в качестве итератора подходит только в том случае, если элементы коллекции действительно располагаются *непрерывно* в памяти. Скажем, в классическом линейном списке (односвязный или двухсвязный) – не всегда данное условие выполняется, зависит от реализации. Для понимания того, как это всё работает, реализуем итератор для типа **FP003Array** в виде вложенного класса.
Задумка следующая: каждый объект итератора будет хранить ссылку на внутренний массив и индекс элемента, на который он в текущий момент ссылается.

C++: итераторы

Тогда часть кода преобразуется[1]

```
1  class FP003Array
2  { ...
3    class Iterator {
4    public:
5      Iterator(size_t idx, FP003Array& obj);
6      bool operator!=(const Iterator& rhs);
7      Iterator& operator++();
8      int& operator*();
9    private:
10     size_t index;
11     FP003Array& arr_ref; };
12
13   Iterator begin();
14   Iterator end();
15   ...
16 };
```

[1] За скобки в строках 4 и 12 – извините, иначе на слайд не влезает код

C++: итераторы

Основные моменты:

- создали отдельный класс (строка 3) с полями для хранения задуманных значений (строки 10, 11);
- добавили минимальный набор методов для итератора: проверка на неравенство (строка 6), пре-инкремент (строка 7), разыменование (строка 8).

Реализация ▸ вся тут [2]. На слайдах не приводится, ибо слишком много места занимает. Например, чтобы в ***.cpp** определить конструктор вложенного класса, требуется начать со стороки **FP003Array::Iterator::Iterator**.

[2] https://github.com/posgen/OmsuMaterials/tree/master/2course/Programming/2021_2022/lectures/examples/lecture5

C++: итераторы

Основные моменты:

- создали отдельный класс (строка 3) с полями для хранения задуманных значений (строки 10, 11);
- добавили минимальный набор методов для итератора: проверка на неравенство (строка 6), пре-инкремент (строка 7), разыменование (строка 8).

Реализация ▸ вся тут[3]. На слайдах не приводится, ибо слишком много места занимает. Например, чтобы в ***.cpp** определить конструктор вложенного класса, требуется начать со стороки **FP003Array::Iterator::Iterator**.

[3] https://github.com/posgen/OmsuMaterials/tree/master/2course/Programming/2021_2022/lectures/examples/lecture5

C++: итераторы

Интерфейс класса-итератора со *слайда 47* представляет собой простейший итератор, который подходит только для алгоритмов, которые подразумевают только один проход по коллекции. Например, тот же диапазонный цикл **for**.
Но чтобы класс, представляющий коллекцию, мог работать с более сложными алгоритамами стандартной библиотеки (<**algorithm**>), итератор должен реализовать интерфейс так называемого *итератора произвольного доступа*.

C++: итераторы

Для его получения всего-то надо класс расширить до:

```
...
    bool operator!=(const Iterator& rhs);
    bool operator==(const Iterator& rhs);
    bool operator<(const Iterator& rhs);
    bool operator<=(const Iterator& rhs);
    bool operator>(const Iterator& rhs);
    bool operator>=(const Iterator& rhs);
    Iterator& operator++();
    Iterator operator++(int);
    int& operator*();
    int& operator[](int n);
    int operator-(const Iterator& rhs);
    Iterator& operator+=(int n);
    Iterator& operator-=(int n);
    Iterator operator+(int n);
    Iterator operator-(int n);
...
```

C++: итераторы

Но об этом –

Но об этом – в следующий раз.

Извините за задержку с текстовой версией. Что-то нюансов многовато оказалось. . .