III

Прежде, чем идти дальше

Как скопировать статический массив одним оператором?

```
1  struct ArrTenthWrapper
2  {
3    int arr[10];
4  };
5
6  ArrTenthWrapper v1 = {{1, 2, 3, 4, 5}},
7                  v2 = {{9, 8, 7, 6, 0}};
8
9  v2 = v1;
10 std::cout << v2[2] << std::endl;
```

Прежде, чем идти дальше

Как скопировать статический массив одним оператором?

```
1  struct ArrTenthWrapper
2  {
3    int arr[10];
4  };
5
6  ArrTenthWrapper v1 = {{1, 2, 3, 4, 5}},
7                   v2 = {{9, 8, 7, 6, 0}};
8
9  v2 = v1;
10 std::cout << v2[2] << std::endl;
```

Как было в C?

Значения полей объектов составных типов при применении присваивания копируются *побайтово*. При использовании *фундаментальных* или *специальных* (статических массивов или указателей) типов C++ ведёт себя точно также.

В дополнении к управлению *конструированием* объектов, C++ даёт возможность:

1. контролировать операцию удаления объекта: объект в C++ удаляется в трёх случаях
   1. выход из локальной области видимости;
2. контролировать операцию копирования объекта.

Рассмотрим обе возможности на примере собственного типа для представления динамического массива.

В дополнении к управлению *конструированием* объектов, C++ даёт возможность:

1. контролировать операцию удаления объекта: объект в C++ удаляется в трёх случаях
   1. выход из локальной области видимости;
   2. перед окончанием работы программы (глобальные объекты);
2. контролировать операцию копирования объекта.

Рассмотрим обе возможности на примере собственного типа для представления динамического массива.

В дополнении к управлению *конструированием* объектов, C++ даёт возможность:

1. контролировать операцию удаления объекта: объект в C++ удаляется в трёх случаях
   1. выход из локальной области видимости;
   2. перед окончанием работы программы (глобальные объекты);
   3. при использовании оператора **delete** (динамические объекты).
2. контролировать операцию копирования объекта.

Рассмотрим обе возможности на примере собственного типа для представления динамического массива.

Какие операции в задуманном типе хотелось бы видеть:

- самое очевидное – *индексация*, доступ к элементам привычным относительно языка программирования способом;

Какие операции в задуманном типе хотелось бы видеть:

- самое очевидное – *индексация*, доступ к элементам привычным относительно языка программирования способом;
- операцию получания длину массива в конкретный момент;

Какие операции в задуманном типе хотелось бы видеть:

- самое очевидное – *индексация*, доступ к элементам привычным относительно языка программирования способом;
- операцию получания длину массива в конкретный момент;
- операцию добавления нового элемента в массив;

Как хотелось бы работать с объектами:

```
1  int main()
2  {
3    using namespace std;
4
5    FP003Array examp1{10};
6
7    examp[0] = 4;
8    examp[3] = -5;
9    cout << "Length: " << examp.length() << endl;
10   cout << "Fourth elem: "<< examp[3] << endl;
11
12   examp.push(45);
13   cout << "Length: " << examp.length() << endl;
14 }
```

Какие поля нужны для начала:

```
1 class FP003Array
2 {
3 public:
4
5 private:
6   int *data_ = nullptr;
7   size_t length_ = 0;
8   size_t capacity_ = 0;
9 };
```

Всё поля получили значения по умолчанию.
Нижнее подчёркивание в названии поля служит стилистическим указателем на то, что это поля. А не какие-нибудь локальные переменные или параметры внутри методов.

Что за **capacity_**?

- поскольку массив динамический, автоматически расширяющийся с добавлением элементов, придётся иметь дело с перевыделением динамической памяти;
- при добавлении каждого нового элемента, увеличивать каждый раз динамический блок на условную единицу – не эффективно с точки зрения частоты обращения за памятью в среде исполнения (ОС);
- будем различать *логический* массив и *динамический* массив, служащий внутренним хранилищем элементов;
- поле **length_** будет отвечать за длину логического массива, поле **capacity_** (ёмкость) – за фактическую длину динамического буфера памяти.

Добавим конструкторы:

```
1 class FP003Array
2 {
3 public:
4     FP003Array() = default;
5     FP003Array(size_t arr_sz);
6
7 private:
8   //поля для данных
9 };
```

Всё-таки дадим возможность создавать объекты, используя конструктор без параметров. Поскольку все значения полей мы задали ранее, а дополнительных действий не требуется для этого случая, вместо явного указания пустого тела конструктора используется синтаксис из строки **4**.

2 варианта определения конструктора с параметром:

1. пишем тело метода:

```
1 FP003Array::FP003Array(size_t arr_sz)
2 {
3   length_ = capacity_ = arr_sz;
4   data_ = new int[arr_sz];
5 }
```

2. сокращённый синтаксис или **список инициализаций**:

```
1 FP003Array::FP003Array(size_t arr_sz) :
2   data_{new int[arr_sz]}, length_{arr_sz},
3   capacity_{arr_sz}
4 {}
```

Двоеточие после списка параметров и присвоение значений полям создаваемого объекта через запятую. Присвоение следует делать в порядке объявления полей.

Вновь о конструкторах

- конструктор отвечает за инициализацию каждого объекта класса (в широком смысле – составного типа данных);

Вновь о конструкторах

- конструктор отвечает за инициализацию каждого объекта класса (в широком смысле – составного типа данных);
- конструкторы могут быть перегружены;

Вновь о конструкторах

- конструктор отвечает за инициализацию каждого объекта класса (в широком смысле – составного типа данных);
- конструкторы могут быть перегружены;
- конструктор вызывается после выделения хранилища в памяти для конкретного объекта.
  - **хранилище в C++** – непрерывный кусок памяти, достаточный для хранения всех полей составного объекта.

После определения конструкторов, становится работоспособным следующий код:

```
1  int main()
2  {
3      FP003Array examp1{10}, examp2;
4
5      cout << "Address (1): " << &examp1 << '\n';
6
7      {
8        FP003Array examp3{120};
9        cout << "Address (3): " << &examp3 << '\n';
10     }
11
12     cout << "Address (2): " << &examp2 << '\n';
13 }
```

Пока методов не добавлено, что-то кроме получения адреса объектов с ними проделать затруднительно.

В примере предыдущего слайда появляется проблема:

- при создании (конструировании) переменных **examp1** и **examp3** выделяется динамическая память при работе конструктора;
- но нет ни одной команды на удаление этой выделенной памяти;
- при окончании области видимости – строка **10** для **examp3**, строка **13** для **examp1** – объекты удаляются в соответсвии с обычными правилами C++. По умолчанию это означает, что в ОС возвращается вся память, выделенная под поля каждого объекта типа **FP001Array**.

Возникает типичная ошибка — *утечка памяти*.

Для её исправления C++ позволяет влиять на опрерацию удаления объекта с помощью ещё однго специального метода – **деструктора**.

Термин

Деструктор – специальный метод класса, позволяющий выполнить дополнительные действия для объектов типа перед тем, как память под объект вернётся операционной системе.

Для её исправления C++ позволяет влиять на опрерацию удаления объекта с помощью ещё однго специального метода – **деструктора**.

Термин

Деструктор – специальный метод класса, позволяющий выполнить дополнительные действия для объектов типа перед тем, как память под объект вернётся операционной системе.

- идентификатор деструктора фиксированный: символ тильды ($\sim$) и имя типа;
- деструктор не может иметь списка параметров;
- деструктор не может быть перегружен.

Добавляем деструктор к рассматриваемому классу:

```
1  class FP003Array
2  {
3  public:
4      FP003Array() = default;
5      FP003Array(size_t arr_sz);
6
7      ~FP003Array();  // <- desctructor
8
9  private:
10   //поля для данных
11 };
```

И его реализацию для удаления в нужный момент динамической памяти:

```
1 FP003Array::~FP003Array()
2 {
3   delete[] data_;
4   data_ = nullptr; // необязательно, но пусть будет
5 }
```

И пример со слайда **(11)** становится безопасным с точки зрения работы с динамической памятью для конкретных объектов.

Пример динамического объекта:

```
1 int main()
2 {
3   FP003Array *arr_ptr = new FP003Array{35};
4
5   cout << "Address : " << arr_ptr << '\n';
6   // другие действия будут вскоре доступны
7
8   delete arr_ptr;
9 }
```

- в строке **3** оператор **new** вызывает конструктор с параметром для создания объекта типа **FP003Array** и создаёт динамический объект;
- в строке **8** оператор **delete** вызывает деструктор и удаляет динамически созданный объект;

Обобщая, полный жизненый цикл любого объекта в C++ (неважно какого типа) состоит из следующих шагов:

1. выделение памяти под объект;
2. вызов конструктора;
   - если объект – составной и используется только конструктор по умолчанию, то происходит последовательный вызов всех конструторов кажого поля в порядке их объявления в классе;
   - если конструктор типа явно определён и используется *список инициализаций*, то сначала вызываются конструкторы всех перечисленных в этом списке полей. Затем – конструкторы для всех неуказанных в нём полей, опять же – в порядке их объявления. И в завершении, исполняются инструкции в **теле** вызванного конструктора.
3. вызов деструктора самого объекта;
4. вызов деструкторов всех полей **в обратном порядке** их объявления;
5. возвращение памяти объекта среде выполнения (ОС).

Конечно же, между пунктами **2** и **3** идёт какая-нибудь работа с объектом.

Время жизни объекта (lifetime)

В соответствии со стандартом C++, время жизни объекта начинается с момента, как конструктор закончил свою работу (технически – выполнились все инструкции в теле конструктора) и заканчивается как только был вызван деструктор объекта (начала исполняться первая инструкция в теле деструктора).

Важно то, что логически (и с точки зрения стандарта языка) понятия конструктора и деструктора применяются и к фундаментальным типам, и к специальным типам — например, указателям.

```
1 int main()
2 {
3   int i1 = 78;
4   {
5     int *i_ptr = &i1;
6     i_ptr *= 2;
7   }
8   cout << "i1 = " << i1 << endl;
9 }
```

- строка **3** – конструируется объект типа **int**;
- строка **5** – конструируется объект типа **указатель на int**;
- строка **7** – происходит деконструирование (удаление) созданного укзателя;

Пример предыдущего слайда – одно из семантических различий между языками C и C++.

В чём суть?

В языке C объект языка считается корректным тогда, когда под него успешно выделено хранилище в памяти. Не имеет значения, обычная ли это переменная, или блок памяти через функции **malloc/calloc/realloc** — если во время работы память получена, то с объектом можно безопасно работать.

C++ добавляет к выделению памяти ещё и стадию конструирования, на которой обязательно должна выполниться одна из форм *инициализации* (см. первую текстовую лекцию). И объект считается корректным тогда и только тогда, когда успешно выполнились обе стадии (выделение памяти + конструирование).

После раскрытия всей подноготной касательно жизни объектов в C++, вернёмся к создаваемому классу и добавим задуманные методы:

- *индексация* – реализуем через перегрузку оператора квадратных скобок ([]) для того, чтобы работать с динамическим массивом привычным встроенным типам массивов образом. И в дополнении сделаем «безопасную» индексацию – какой бы индекс не был указан, он будет приведён к нужному диапазону;
- *получение длины массива* – обычный метод, который будет возвращать значения поля **length_** объекта;
- *добавление нового элемента*.

Определение класса теперь выглядит так:

```
1  class FP003Array
2  {
3  public:
4      FP003Array() = default;
5      FP003Array(size_t arr_sz);
6      ~FP003Array();
7
8      int& operator[](size_t index);
9      size_t length() const;
10     void push(int new_elem);
11
12 private:
13   //поля для данных
14 };
```

Реализация добавления элемента:

```
1 void FP003Array::push(int new_elem)
2 {
3   if (length_ >= capacity_) {
4     re_allocate();
5   }
6
7   data_[length_] = new_elem;
8   length_++;
9 }
```

Если хорошо понимаете операцию *пост-инкремента*, седьмую и восьмую строку можно сократить до

```
7   data_[length_++] = new_elem;
```

Метод **re_allocate** отправим в *закрытую часть* класса (приватный метод):

```
1  class FP003Array
2  {
3  public:
4      FP003Array() = default;
5      FP003Array(size_t arr_sz);
6      ~FP003Array();
7
8      int& operator[](size_t index);
9      size_t length() const;
10     void push(int new_elem);
11
12 private:
13   //поля для данных
14   void re_allocate();
15 };
```

И реализация **re_allocate**:

```
1  void FP003Array::re_allocate()
2  {
3    capacity_ = (capacity_ == 0) ?
4                  2 : capacity_ * 2;
5
6    int *new_data = new int[capacity_];
7    for (size_t i = 0; i < length_; i++) {
8      new_data[i] = data_[i];
9    }
10
11   delete[] data_;
12   data_ = new_data;
13 }
```

Реализация индексации:

```
1 int& FP003Array::operator[](size_t index)
2 {
3   const size_t safe_index = (length_ == 0) ?
4     0 : index % length_;
5   return data_[safe_index];
6 }
```

- перегруженный оператор возвращает ссылку на элемент массива. За счёт ссылки появляется возможность не только получать значение элемента массива по индексу, но и устанавливать его, используя привычную запись:

```
1 FP003Array items{3};
2 items[0] = 101;
```

- строка **4**, первое выражение – пока никак не обрабатываем ошибку разыменования нулевого указателя. Второе выражение – приведение переданного индекса к безопасному.

Метод возвращающий длину:

```
1 size_t FP003Array::length() const
2 {
3   return length_;
4 }
```

После добавления методов появляется уже больше возможностей использовать созданный тип:

```
1  int main()
2  {
3      FP003Array examp1{10}, examp2;
4      examp2.push(20);
5      examp2.push(-5);
6
7      examp[1] = examp2[0] + examp[1];
8      examp[9] = examp2[0] * examp[1];
9      cout << "examp last elem: "
10         << examp[examp.length() - 1] << endl;
11 }
```

И новые проблемы – простое копирование приводит к ошибкам работы с памятью:

```
1  int main()
2  {
3      FP003Array examp1{10}, examp2;
4      examp1[0] = 4;
5      examp1[3] = -3;
6
7      examp2 = examp1;
8      examp2[3] = 9;
9
10     cout << "examp1[3] = " << examp1[3] << endl;
11 }
```

Мало того, что строка **8** теперь влияет на элемент другого объекта, так и в вместо успешного завершения подобная программа выдаст ошибку двойного удаления одного блока динамической памяти. Причина – на втором слайде.

Для управления копированием в составном типе данных в C++ нужно определять два дополнительных метода:

1. **Конструктор копий** (copy constructor, копирующий конструктор – термины на любой вкус). Вызывается:

```
1 void print_array(FP003Array arr);
2 FP003Array examp1{10};
3
4 FP003Array examp2 = examp1; // <- here
5 FP003Array examp3{examp1};  // <- here
6 print_array(examp2);        // <- here
```

2. **Оператор присваивания** для корректоного копирования объектов собственного типа. Вызывается:

```
1 FP003Array examp1{10}, examp2;
2
3 examp2 = examp1;            // <- here
4 examp1 = FP003Array(45);    // <- here
```

По умолчанию (**1**) и (**2**) просто копируют поля побайтово.

Для приведения поведения объектов в порядок, прокачиваем класс дальше:

```
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   int& operator[](size_t index);
11   size_t length() const;
12   void push(int new_elem);
13
14 private:
15   //закрытая часть
16 };
```

Реализация конструктора копий:

```
1 FP003Array::FP003Array(const FP003Array& other) :
2     data_{new int[other.capacity_]},
3     length_{other.length_},
4     capacity_{other.capacity_}
5 {
6   for (size_t i = 0; i < other.length_; i++) {
7     data_[i] = other.data_[i];
8   }
9 }
```

Что-то новенькое

Любой метод класса может обращаться к **закрытым** полям и методам объекта своего же класса, переданного в качестве параметра метода. В данном случае **other** всегда будет каким-то уже созданным объектом, закрытые поля которого мы используем напрямую.

Реализация оператора присваивания:

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

Что за **this**?

this – специальный указатель, доступный в любом (нестатическом) методе класса. Он типизирован и его полный тип определяется как *указатель на тот класс, в котором он используется*. В рамках конкретного объекта **this** ссылается, как бы, на самого себя.

Зачем возвращать ссылку на самого себя?

В C++ нет никаких ограничений на возвращаемое значение (да и на тип параметра – нету) из перегруженного оператора присваивания. Однако, по умолчанию для всех фундаментальных результатом оператора присваивания является **значение его левого операнда**. Чтобы не нарушать общую логику работы языка – делаем аналогично и для собственного типа. А чтобы избежать лишних копирований – возвращаем ссылку.

В принципе, **this** может использоваться для явного выделения полей объекта. Например, тот же оператор присваивания:

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

Стоит или не стоит так делать – вопрос стилистического оформления, никак не требований самого C++.

C++: ООП – резюме на сегодня

- C++ позволяет контролировать полный цикл жизни объекта любого собственного типа;
- возможности конструирования и деконструирования, предоставляемые по умолчанию, не всегда работают для типов, котрые оперируют некоторыми динамическими ресурсами;
- на **this** стоит смотреть как на аргумент, который **всегда** передаётся в любой нестатический метод типа;
- возвращать ссылки на внутренние поля объекта – вполне безопасно (на констрасте с возвратом ссылок на локальные переменные функций);
- неосторожное обращение с копированием – источник неприятных проблем;

C++: ООП – резюме на сегодня

- *конструктор копий* и *оператор присваивания собственных объектов* должны быть или
  1. использованы реализации по умолчанию;
  2. переопределены самостоятельно в типе данных **вместе**;
  3. запрещены для типа.

```
1 FP003Array(const FP003Array& other) = delete;
2 FP003Array& operator=(const FP003Array& rhs) = ↩
      delete;
```

Пример синтаксиса, который явно запрещает работу конструктора копий и оператора присваивания собственных объектов.

C++: ООП – код с лекции

В виде готового кода разобранный пример можно ▸ найти тут[1]

[1] https://github.com/posgen/OmsuMaterials/tree/master/2course/Programming/2021_2022/lectures/examples/lecture3