VI

Заканчивая разборки с итераторами

C++: итераторы – категории

В C++ выделяют несколько видов (*категорий*) итераторов, которые различаются требованиями к операциям в реализующем итераторе типе. Так, класс из предыдущей лекции в окончательном варианте реализовывал самый нетребовательный – *итератор ввода*.
Далее кратко приводятся характеристики основных категорий.

C++: итераторы – категории

1. **Итератор ввода** (он же – LegacyInputIterator)
   Операции: тип, соответствующий данной категории, должен предоставлять следующий минимальный набор операций:
   - **it1 != it2** – проверка двух объектов итератора на неравенство;
   - **++it** – инкремент объекта итератора, т.е. переход к следующему значению в коллекции. Как минимум, нужно реализовать *пре-инкремент*;
   - ***it** – получение доступа к элементу коллекции через оператор *разыменования*.

   Примечание: только алгоритмы, осуществляющие один проход по коллекции и обращающиеся на каждом шаге алгоритма только к единственному элементу коллекции.

Итератора данной категории достаточно для работы *диапазонного цикла* **for**.

2. **Итератор прямого доступа** (LegacyForwardIterator)
   Операции: тип, соответствеющий данной категории, должен предоставлять следующий минимальный набор операций:
   - все операции предыдущей категории;
   - **it++** – *пост-инкремент* объекта итератора.

   Примечание: алгоритмы последовательного доступа к элементам коллекции (как и предыдущая категория), но допускающие несколько проходов по ней (в отличии от предыдущей категории).

C++: итераторы – категории

3. **Двухнаправленный итератор** (LegacyBidirectionalIterator)
   Операции: тип, соответствеющий данной категории, должен предоставлять следующий минимальный набор операций:
   - все операции предыдущей категории;
   - **it–, –it** – *пре- и пост-декременты* объекта итератора, переход к предыдущему элементу коллекции .

C++: итераторы – категории

4. **Итератор произвольного доступа**
(LegacyRandomIterator)
Операции: тип, соответствеющий данной категории, должен предоставлять следующий минимальный набор операций:
   - все операции предыдущей категории;
   - полный набор операций сравнения: больше, меньше, больше либо равно и так далее;
   - операция вычитания двух объектов итератора: получения «расстояния» между тем элементами, на которые указывают итераторы;
   - опрации сложения/вычитания объекта итератора с *целым* числом: сдвиг объекта итератора вправо/влево по коллекции;
   - операция индекации для итератора: получение нового объекта итератора, сдвинутого относительно текущего на переданное в индексе число элементов.

Примечание: категория, подходящия для большинства алгоритмов из стандартной библиотеки (**<algorithm>**).

C++: итераторы – категории

Пример *итератора произвольного доступа* для класса **FP003Array** ▸ доступен онлайн[1].
Далее – примеры работы собственного класса с некоторыми стандартными алгоритмами.

[1] https://github.com/posgen/OmsuMaterials/tree/master/2course/Programming/2021_2022/lectures/examples/lecture6

C++: итераторы – примеры

Сортировка объектов **FP003Array** стандартной функцией **sort**

```
1  FP003Array examp1{10};
2  // тут где-то инициализация каждого элемента скрыта
3
4  std::cout << "Before sorting: ";
5  pretty_print(examp1);
6
7  std::sort(examp1.begin(), exapm1.end());
8
9  std::cout << "After sorting: ";
10 pretty_print(examp1);
```

C++: итераторы – примеры

Присвоение конкретного значения все элементам заданного диапазона коллекции с помощью **fill**

```
1 FP003Array examp2{14};
2
3 std::fill(examp2.begin(), exapm2.end(), 7);
4
5 pretty_print(examp2);
```

C++: итераторы – примеры

Подсчёт количества элементов, равных конкретному значению

```
1 int count_of_four = std::count(
2     examp1.begin(), examp1.end(), 4);
3
4 std::cout << count_of_four
5           << " numbers equals to 4" << std::endl;
```

C++: итераторы – примеры

Поиск конкретного значения в массиве. Если найден элемент, равный искомому значению – возвращается объект итератора, указывающий на него. Иначе – возвращается итератор, равный индикатору окончания коллекции.

```
1  int number_to_find = 16;
2  FP003Array::Iterator it_found
3      = std::find(examp1.begin(), examp1.end(),
4                  number_to_find);
5
6  if ( it_found != examp1.end() ) {
7    cout << "number " << number_to_find
8         << " found at "
9         << (it_found - examp1.begin() + 1)
10        << " place" << endl;
11 } else {
12   cout << "number " << number_to_find
13        << " not found" << endl;
14 }
```

C++: вывод типа переменной компилятором

Предыдущий пример – хороший кандидат на использование ключевого слова **auto**.

Напоминание про auto

В отличие от языка C, в C++ ключевое слово **auto** означает не квалификатор хранилища в памяти для переменной, а указывает компилятору вывести тип переменной **самостоятельно**.

Функция **std::find** из **<algorithm>** всегда возвращает объект итератора коллекции. В этом случае можно исключить написание в коде длинной конструкции **CollectionType::Iterator**, использовав **auto**.

C++: вывод типа переменной компилятором

Вот так пример меняется:

```
1  int number_to_find = 16;
2  auto it_found = std::find(
3      examp1.begin(), examp1.end(), number_to_find
4  );
5
6  if ( it_found != examp1.end() ) {
7    cout << "number " << number_to_find
8         << " found at "
9         << (it_found - examp1.begin() + 1)
10        << " place" << endl;
11 } else {
12   cout << "number " << number_to_find
13        << " not found" << endl;
14 }
```

C++: вывод типа переменной компилятором

И ещё пример кода, подходящий для использования **auto** – работа с динамическими объектами (штуки, временем жизни которых мы управляем вручную). Так, например, создаётся динамический объект типа **FP003Array**

```
1 FP003Array *dyn_arr = new FP003Array{25};
```

И сразу в глаза бросается дублирование названия класса, которое не несёт никакой практической пользы. Поэтому, преобразуем в

```
1 auto *dyn_arr = new FP003Array{25};
```

Никакой неоднозначности, что за тип – видно в операторе **new**. Таким образом – тоже подходящий случай для перекладывания ответственности за вывод типа переменной на компилятор. В примере компилятор выведет тип как *указатель на* ***FP003Array***.

C++: итераторы – примеры

И ещё пример – функция **accumulate**. Попробуйте самостоятельно разобраться, что она делает из примеров. Первый пример:

```
1 int arr_sum = std::accumulate(
2     examp1.begin(), examp1.end(), 0);
3 std::cout << "Sum of all elements in examp1 is "
4           << arr_sum << std::endl;
```

Особенно – что за третий параметр функции? Какой его смысл?

C++: итераторы – примеры

Второй пример:

```
1 int op_product(int accum, int elem)
2 { return accum + elem; }
3
4
5 int arr_prod = std::accumulate(examp1.begin(),
6     examp1.end(), 1, op_product);
7 std::cout << "Product of all elements "
8           << "in examp1 is "
9           << arr_prod << std::endl;
```

▸ Ещё больше алгоритмов[2]

[2] https://en.cppreference.com/w/cpp/algorithm

C++: итераторы – где может быть опасность

Совет: не стоит менять коллекцию в процессе её обхода

Внутри любого цикла, использующего итераторы, не **надо** изменять тот объект, по которому осуществляется обход.

```
1 for (int& elem : examp1) {
2   examp1.push(25);
3   elem *= 2;
4 }
```

Пример с нарушением логики работы с коллекцией. Данный цикл внутри себя использует итераторы для перебора элементов объекта **examp1**. При этом во *второй строке* происходит изменение коллекции. Которое, в случае класса **FP003Array**, может приводить к изменению внутреннего динамического массива. В общем случае, это может привести к тому, что итераторы станут невалидными и обход никак не будет завершён правильно.

C++: итераторы – где может быть опасность

Совет: не разыменовываем итератор **obj.end()**

Не стоит разыменовывать итератор, который указывает на окончание коллекции.

```
1 auto it_end = some_obj.end();
2 std::cout << *it_end << std::endl;
```

В нашей реализации итератора для типа **FP003Array** ничего критичного не произойдёт, но это не верно в отношении коллекций из стандартной библиотеки языка C++. Так что, лучше просто принять к сведениями, что такое действие может приводить к падению программы.

Функциональные объекты

Суть явления

Технически в C++ под **функциональным объектом** или **функтором** понимают объект составного типа, для которого перегружен *оператор круглые скобки* (), он же – **оператор вызова функции**.

Логически, функциональные объекты нужны тогда, когда основное использование их сводится к следующему псевдокоду:

```
1 FunctorClass obj;
2
3 result = obj(arg1, arg2, arg3, ...);
```

В перегруженном *операторе вызова функции* сосредаточена какая-то логика вычислений и он возвращает ожидаемый результат.

C++: функциональные объекты

Один из подходящих случаев для использования *функциональных объектов* в вычислительных программах на C++ являются различные параметрические зависимости. Рассмотрим для начала, обычную линейную функцию:

$$u(x) = ax + b$$

C++: функциональные объекты

Реализовать её можно либо через обычную свободную функцию

```
1 double u(double x, double a, double b)
2 { return a * x + b; }
```

Реализовать её можно либо через обычную свободную функцию

```
1 double u(double x, double a, double b)
2 { return a * x + b; }
```

либо с использованием класса:

```
1 class LinearFn
2 {
3 public:
4   double a, b;
5
6   double operator()(double x)
7   { return a * x + b; }
8 };
```

Вот пример создания типа, объекты которого достойны называться **функторами**.

C++: функциональные объекты

И объекты класса, и функция вычисляют одно и тоже. Посмотрим на то, как выглядит использующий их код:

```
1 LinearFn u1{5, 10};
2 cout << "5*x + 10 (at x = 2): " << u1(2) << endl;
3 cout << "5*x + 10 (at x = 2): " << u(2, 5, 10)
4      << endl;
```

Одно из синтаксических преимуществ функционального объекта – возможность отдельно задать параметры функции (*строка 1*). Альтернативно – параметры таскаются за функцией при каждом её вызове (*строка 3*).

И пример вычисления нелинейной комбинации различных линейных функций

```
1 LinearFn u2 = {-3, 2}, u3{3.5, -0.5};
2
3 double res1 = u1(3.5) + u2(-1.4) * u3(0.33);
4 double res2 = u(3.5, 5, 10) +
5     u(-1.4, -3, 2) * u(0.33, 3.5, -0.5);
6
7 cout << res << ", " << res2 << endl;
```

Здесь можно задуматься о том, какой из подходов более наглядно выражает те действия, которые мы хотим совершить.

Более того, представим, что предыдущая нелинейная комбинация нужна нам в некотором вычислительном алгоритме в различных местах. Тогда эти повторяющиеся действия логично вынести в отдельную функцию, которая будет возвращать результат вычисления при конкретном значении x. Для функциональных объектов, подобная функция может выглядеть как следующая:

```
1 double my_cool_fun(double x, LinearFn u,
2                     LinearFn v, LinearFn w)
3 { return u(x) + v(x) * w(x); }
```

Её код достаточно универсален, чтобы вообще не зависит от того, какие линейные функции используются для вычисления результата. И функция сосредаточена на одной, основной своей задаче – вычислить нужную нелинейную комбинацию в заданной точке.

C++: функциональные объекты

Вопрос: как будет выглядеть подобная функция, если вместо функциональных объектов использовать свободные функции?

Вопрос: как будет выглядеть подобная функция, если вместо функциональных объектов использовать свободные функции? Решение в лоб *эстетически непривлекательно*:

```
1 using lin_fn = double(*)(double, double, double);
2
3 double my_cool_fun2(double x, lin_fn u, lin_fn v,
4     lin_fn w, double ua, double ub, double v1,
5     double vb, double wa, double wb);
```

Тут только сигнатура приведена. Для каждой функции (**u**, **v**, **w**) приходится таскать нужные им параметры. Подобная функция с таким числом параметров никак не может считаться приемлемой с точки зрения понятности кода и минимизации возможных ошибок при её вызове.

Касательно вычислительных задач, можно ещё один случай выделить, который подходит использования функциональных объектов. Это *кэширование* части вычислений на **этапе создания** функционального объекта. Рассмотрим следующую функцию:

$$W(x) = e^{ax+sin(b)}$$

Чуть усложним пример с живой лекции. Данную функцию можно преобразовать к виду:

$$W(x) = (e^a)^x e^{sin(b)}$$

когда становится очевидным, что экспоненты при заданных параметрах a и b являются постоянными для любых значений x.

Приведённую функцию можно описать следующим типом:

```
1  class ExpFn
2  {
3  public:
4      ExpFn(double a, double b) :
5        exp_a{std::exp(a)}, exp_b{std::exp(sin(b))}
6      {}
7
8      double operator()(double x)
9      {
10         return std::pow(exp_a, x) * exp_b;
11     }
12
13 private:
14     double exp_a, exp_b;
15 };
```

Использование очевидно:

```
1 ExpFn u_e1 = {3.5, -1.7};
2 cout << "u_e1(4.5) = " << u_e1(4.5) << endl;
```

В *строке 1* создаём функцию с нужными параметрами. Экспоненты вычислены во время работы конструктора. В *строке 2* уже вычисляется самый минимум операций. Опять же, на простом примере не совсем очевидна польза запоминания, но если, для примера, в некотором вычислительном алгоритме требуется вычислять подобную функцию порядка нескольких миллионов раз, то вычисление синуса каждый раз (не самая быстрая операция в сравнении с тем же умножением) **уже упразднено**.

И пример из стандартной библиотеки – генераторы псевдослучайных чисел из **<random>**

```
1 #include <random>
2
3 std::mt19937_64 gnr{static_cast<size_t>(time(↩
    nullptr))};
4 cout << gnr() << ", " << gnr() << endl;
```

Пример использует ГПСЧ основанный на алгоритме «вихрь Мерсенна», ▸ тут подробности алгоритма, кому интересно.