Ответы на вопросы экзамена по курсу «Языки программирования» 08.01.2020

В ответах курсивом выделены необязательные пояснения, которые можно опустить (особенно на экзамене)

Вариант 1

Задача 1-1

Дайте определение способов передачи параметров по значению/результату и по ссылке(адресу). Приведите пример, который показывает различия этих способов (на любом языке, в котором хотя бы теоретически могут допускаться такие способы).

Ответ

Замечание: пример задачи, формулировку которой большинство студентов не поняли. При этом «непонятки» были двух видов. Первый вид(к сожалению – большинство работ относилось именно к нему) состоял в том, что передачу параметров по значению/результату просто перепутали с передачей по значению, что является безусловной ошибкой. За такую ошибку я снижал оценку за задачу вполовину – до 3 баллов. Второй вид недоразумения состоял в том, что в условии требовалось привести пример, который показывает РАЗЛИЧИЯ этих способов, а не просто пример, иллюстрирующий эти способы. Очевидно, что это два разных типа примеров. Если подпрограмма завершается НОРМАЛЬНО, то обнаружить разницу в том, как именно передавались параметры весьма затруднительно – они получают новые значения корректно и в одном, и в другом случаях. Но если программа завершилась некорректно (то есть по исключению), что вообще совершенно допустимо в современных ЯП, то вот тут-то и возникают нюансы. Поэтому нужно было либо выбрать ЯП, допускающий передачу параметров как по ссылке, так и по значению/результату, а также допускающий обработку исключений (на лекциях я приводил пример из Ады 83). Но в принципе совершенно нормально было привести пример на ГИПОТЕТИЧЕСКОМ языке, то есть представить себе, например, язык C++, в котором появился еще 3 способ передачи параметров – по значению/результату. Обозначим его, например, так: void SampleValueResult (int /x/,/int /y/) – обратите внимание на косые черточки. И далее привести пример того, как эта функция может давать результат, отличный от void SampleValueResult (int &x, int& y). Мы уже использовали такой прием на лекциях, когда обсуждали передачу параметров по имени. Вместо того, чтобы рассматривать синтаксис АЛГОЛА-60 мы рассмотрели гипотетический Паскалеподобный язык, в котором есть спецификатор byname для прердачи параметров по имени, и остался специфкатор var для обозначения передачи параметров по ссылке. Кстати, ряд студентов во втором варианте вполне нормально использовали этот прием (видимо, взяв из конспектов ☺). Но вот когда речь идет не о буквальном переписывании, то тут как-то вся креативность куда-то девается…
Так что я не стал придираться к студентам, которые просто иллюстрировали передачу по ссылке и по значению/результату. Хотя бы правильно понимают, как у нас называются различные методы передачи параметров, и на том спасибо!
Ответ, который хотел увидеть лектор (но увы!), выглядит следующим образом.
Передача параметра по значению/результату является комбинацией способов «по значению» и «по результату», то есть перед входом в подпрограмму значение фактического параметра копируется в соответствующий формальный параметр, который располагается в стековом фрейме (другое название – запись активации) подпрограммы, а после возврата значение формального параметра копируется обратно в фактический параметр. Передача

параметра по ссылке означат передачу по значению адреса фактического параметра. *В зависимости от языка программирования либо программист должен сам вызывать адресную операцию при вызове подпрограммы, и далее разыменовывать адрес при каждом обращении к формальному параметру (это C и Go), либо компилятор сам вставляет нужные команды (Паскаль, C++).*

Ранние компиляторы языка Ада (Ада 83) сами выбирали, какой способ передачи (по ссылке или по значению/результату) лучше подходит для реализации inout-семантики параметров. Например, для массива, конечно, лучше выбирать по ссылке, а для простых переменных базисных скалярных типов может быть эффективнее использовать копирование. Но следующий пример на Аде показывает, что разные способы передачи приводят иногда к тому, что подпрограммы дают разные результаты для разных способов.

```
procedure Sample(X,Y : inout INTEGER) is
begin
     X :=Y+1;
     if Y < 0 then raise MY_EXCEPTION; end if;
     Y := 1;
end Sample;
A: INTEGER := -1;
Sample(A,A);
// в этой точке при передаче по адресу:  A=1 и исключения не было
// а при передаче по значению/результату: A= -1и исключение возникло (некоторые
// компиляторы гипотетически могут отслеживать изменения формальных параметров,
// тогда A=0, но исключение все равно вылетит).
```

Можно переписать этот фрагмент для гипотетического C++ (см.выше) – смысл его будет тот же самый – разные способы дают разный результат.

Задача 1-2

Напишите объявления сущностей F, G, X и Method на языке C# так, чтобы следующий фрагмент компилировался без ошибок.

```
F = X.Method; F(1,2); G = new X().Method; G(1);
```

Ответ

```
class X
{
    public int Method(int i) { return i + 1; }
    public static int Method(int i, int j) { return i + j; }
}
class Program
{

    static void Main(string[] args)
    {
        F = X.Method; F(1, 2); G = new X().Method; G(1);
    }
}
```

Замечание: описывать класс `Program` не обязательно, здесь он приводится только для того, чтобы можно было проверить, что все компилируется без ошибок. Вполне можно было бы сократить, написав так, не заботясь о контексте:

```
class X
{
    public int Method(int i) { return i + 1; }
    public static int Method(int i, int j) { return i + j; }
}
Func<int, int> G;
Func<int, int, int> F;
```

Задача 1-3

Объясните, почему в приведенной программе на языке C++ не вызывается деструктор для объектов John и Mary, но вызывается для Ivan и Petro. Что нужно изменить в объявлении класса Person, чтобы деструктор объектов класса Person вызывался всегда?

```
class Person {
      std::string _name;
      std::shared_ptr<Person> _friend;
public:
      Person(const std::string& name) : _name(name) {}
      ~Person() { std::cout << _name << " retired\n"; }
      void MakeFriend(std::shared_ptr<Person> pPerson) { _friend = pPerson; }
};

int main()
{
      auto Ivan = std::make_shared<Person>("Ivan");
      auto Petro = std::make_shared<Person>("Petro");
      auto John = std::make_shared<Person>("John");
      auto Mary = std::make_shared<Person>("Mary");
      Ivan->MakeFriend(Petro);
      John->MakeFriend(Mary);
      Mary->MakeFriend(John);
      return 0;
}
```

Ответ

Объекты Mary и John связаны кольцевыми ссылками через поле _friend, а std::shared_ptr не умеет обрабатывать кольцевые зависимости, так как использует счетчик ссылок. В случае кольцевых зависимостей счетчик не обнуляется. Поэтому для гарантированного вызова деструктора нужно заменить `std::shared_ptr<Person> _friend` на `std::weak_ptr<Person> _friend`. Вот пример программы:

```
#include <iostream>
#include <memory>
class Person {
      std::string _name;
      std::weak_ptr<Person> _friend;
public:
      Person(const std::string& name) : _name(name) {}
      ~Person() { std::cout << _name << " retired\n"; }
      void MakeFriend(std::shared_ptr<Person> pPerson) { _friend = pPerson; }
```

```
};

int main()
{
      auto Ivan = std::make_shared<Person>("Ivan");
      auto Petro = std::make_shared<Person>("Petro");
      auto John = std::make_shared<Person>("John");
      auto Mary = std::make_shared<Person>("Mary");
      Ivan->MakeFriend(Petro); John->MakeFriend(Mary);
       Mary->MakeFriend(John);
      return 0;
}
```

Замечание: я не стал усложнять пример, добавляя функцию доступа к _friend (в реальном примере она конечно нужна, но здесь усложнит задачу, поскольку надо будет использовать lock() и т.д. – вам оно сильно нужно?), а также написав более сложный вариант функции main() в котором демонстрируется полезность именно std::shared_ptr. Легко дописать пример так, чтобы в нем без использования std::shared_ptr возникли бы «висячие» ссылки. Собственно именно для этого и нужен shared_ptr. Но решить проблему мусора std::shared_ptr иногда не в силах – вот про это и пример.
Я не засчитывал за абсолютно верный вариант предложения сделать `MakeFriend` *пустой (нэт указатэл, нэт проблэм) – это формально и не по существу (ну давайте вообще все выкинем, оставим один деструктор – очень полезный горшочек получится….). Также я не засчитывал предложения заменить _friend на Person& _friend или Person * _friend. Это по сути одно и то же, и обессмысливает пример, поскольку приводит к потенциальным «висячим» ссылкам (ну и зачем тогда огород городить вокруг std::shared_ptr). Если будете когда-либо использовать "усовершенствованные" (слово «разумные» здесь не подходит абсолютно, еще и AI приплетем…) указатели из <memory>, то будьте крайне осторожны с переходом от них к «сырым» указателям и ссылкам. «Мухи отдельно, котлеты отдельно, и не надо их смешивать»*

Задача 1-4

Дайте определение «ленивых» вычислений в языках программирования. В чем заключается связь «ленивых» вычислений и понятия потоков языка Java 8? Что делает следующий фрагмент программы на языке Java 8? Преобразуйте этот фрагмент в эквивалентный код на Java (любой версии) без использования потоков, но сохраняя «ленивость» вычислений. Предполагается, что переменная `S` описана как `Stream<String> S`.

```
int c = S.map(String::toUpperCase).filter(
      x->{ return x.startsWith("J")}).count();
```

Ответ

Здесь 4 подвопроса.

1. Дайте определение «ленивых» вычислений в языках программирования.

В общем случае «ленивые» вычисления – это вычисления, которые выполняются только тогда, когда возникла необходимость в данных, которые являются результатом этих вычислений. *Частным случаем ленивых вычислений является вычисление логических выражений в большинстве ЯП (практически все языки, которые мы разбираем), когда логическое подвыражение вычисляется ТОЛЬКО тогда, когда его результат нужен для*

вычисления значения всего выражения. Другим примером ленивых вычислений служит запрос к СУБД, когда вместо считывания всей последовательности результатов запроса в коллекцию в оперативной памяти мы читаем из курсора СУБД только тогда, когда нам понадобился очередной элемент из результатов запроса.

2. В чем заключается связь «ленивых» вычислений и понятия потоков языка Java 8?

Потоки в языке Java 8 это последовательные коллекции, которые обладают специальным набором методов, которые определяют условие или действие, выполняемое над каждым элементом этой последовательности. Каждый метод, за исключением так называемых терминальных методов(таких, например, как count()), возвращает новый поток, полученный после применения действия или условия к каждому элементу изначального потока. *Например, метод count() возвращает число элементов в последовательности (действие для каждого элемента – это увеличение счетчика элементов, который и будет результирующим значением метода после прохода по всему потоку). Многие методы потоков имеют (хотя бы один) формальный параметр-функцию, которая определяет, что нужно сделать с каждым элементом последовательности, например, метод map применяет функцию-аргумент к каждому элементу последовательности, возвращаемой значение этой функции заменяет текущий элемент. Метод filter применяет предикат-аргумент к каждому элементу, выбрасывая из результирующей последовательности элементы, для которых предикат ложен.*

Потоки очень тесно связаны с ленивыми вычислениями, поскольку они как раз и сделаны для того, чтобы их методы выполнялись «ленивым» образом. То есть нетерминальные методы потока не ждут, пока будет сформирован весь исходный, а пытаются применить свое действие к тем элементами, которые уже доступны, а это и есть принцип «ленивости».

3. Что делает следующий фрагмент программы на языке Java 8?

Считает в строковом потоке S число элементов потока, начинающееся с буквы “j” независимо от регистра.

4. Преобразуйте этот фрагмент в эквивалентный код на Java (любой версии) без использования потоков, но сохраняя «ленивость» вычислений.

Поскольку нам нельзя использовать потоки, то будем считать, что у нас есть список или массив L, в котором находятся все элементы этого потока. *Будем для простоты считать, что у нас есть ArrayList<String> L. Далее можно либо преобразовать его в поток (это один из способов получения первоначальных потоков):*

Stream<String>S = L.stream()

либо, как того требует условие задачи, работаем без потоков, но соблюдая «ленивость», то есть не делаем несколько последовательных циклов, каждый из которых делает новую последовательность, а делаем один цикл, работая каждый раз только с одним (текущим) элементои исходного массива или списка, а в теле будет несколько вложенных друг в друга структур управления. Используем цикл for-each из Java 5 (2005 год), чтобы вообще не зависеть от того, что есть L – массив или список:

```
int cnt = 0;
for (String cur: L) {
      String x = cur.toUpperCase(); // это map
      if (x.startsWith(“J”)) // это filter
            cnt++; // это count()
}
int c = cnt;
```

Замечание: а теперь представьте, что в исходном примере не 3 операции на «конвейере» вычислений (map, filter,count), а 10-15 (как бывает в реальных проектах). Насколько старый код становиться «нечитабельней» новой версии!