Ответы на вопросы экзамена по курсу «Языки программирования» 08.01.2020

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

Вариант 2

Задача 2-1

Дайте определение способов передачи параметров по имени 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: большинство проблем касалось того, что неправильно понимают (то есть совсем не понимают), что такое передача параметров по имени. За эту ошибку снижал вдвое – до 3 баллов. Вторая проблема – какой язык использовать. В современных языках способ не используется, но понимать его нужно. 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 имени. Обозначим его, например, так: void SampleName (int @x,int @y) – обратите внимание на символ @. И далее привести пример того, как эта функция может давать результат, отличный от void SampleName (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 вполне нормально использовали этот прием (видимо, взяв из конспектов ☺).

Ответ. Передача параметров по имени ближе всего по семантике к макроподстановке, но реализуется не подстановкой текста фактического параметра вместо формального, а с помощью так называемых «санков» (thunks), то есть вспомогательных процедур, которые вычисляют действительный адрес фактического параметра КАЖДЫЙ РАЗ при доступе к формальному параметру (неважно, на чтение или запись). *Например, для простой переменной санк возвращает один и тот же адрес (поэтому для простой переменной этот способ фактически эквивалентен передаче по ссылке), а вот для выражения a[i] – ссылке на элемент массива – этот адрес меняется, если во время работы процедуры изменилось значение i.*

Передача параметра по ссылке означает 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++).*

Пример, показывающий различия – это процедура, обменивающая местами значения своих аргументов. Для ссылок (C++):

```
void swap(int& x, int& y)
{
      int tmp = x; x = y; y = tmp;
}
int a[] = {1,2,3,4}, i = 2;
```

```
swap(i, a[i]); // a=={1,2,2,4}, i==3 –не зависит от порядка парамет-
ров!!!!!
```
Для передачи по имени (псевдо-C++):
```
void swap(int@ x, int@ y)
{
      int tmp = x; x = y; y = tmp;
}
int a[] = {1,2,3,4}, i = 2;
swap(i, a[i]); // здесь тело фактически выглядит так:
int tmp = i; i = a[i]; // i == 3!!!!
a[i] = i; // то есть a[3] = 3 => a=={1,2,3,3}, i==3
```

Задача 2-2

Напишите объявления сущностей `F, G и X` на языке Java так, чтобы следующий фрагмент компилировался без ошибок.

```
F = X::Method; int i = F.apply(0.0); G = new X()::Method;
G().apply();
```

Ответ

```
class X
{
    public void Method(){}
    public static int Method(double d) { return (int)d; }
}
@FunctionalInterface
interface Converter {
   int apply(double);
}
@FunctionalInterface
interface Action {
   void apply();
}
Converter F;
Action G;
```

Замечание: можно опустить `@FunctionalInterface` *или static для второй перегрузки Metod.*
Чтобы эти описания компилировались, нужно поместить их в корректный контекст (а корректный контекст – это класс), а фрагмент из условия поместить внутрь какого-нибудь метода контекстного класса. В первом варианте это сделано в задаче на C#, здесь можно сделать аналогично.

Задача 2-3

Объясните, какие ошибки выдаются при трансляции функции `Foo()` в следующем фрагменте программы на C++. Внесите добавления в определение структуры `Owner` так, чтобы фрагмент компилировался без ошибок (менять объявление указателя `_ptr` запрещается).

```
struct X {
};
struct Owner
{
	std::unique_ptr<X> _ptr;
};
void Foo()
{
	Owner o;
	o._ptr = std::make_unique<X>(); // !!! ВНИМАНИЕ – в
//оригинальном варианте задания в параметрах ошибочно стояло
//выражение new X()
	Owner o2(o), o3;  // ошибка – нет конструктора копирования
для структуры Owner
	o3 = o2; // ошибка – нет копирующего присваивания для
структуры Owner
}
```

Ответ

Замечание об опечатке. В оригинальном варианте вместо

`o._ptr = std::make_unique<X>();`

стояло

`o._ptr = std::make_unique<X>(new X());` - *это опечатка. Параметры функции make_unique() передаются в конструктор класса X, а конструктора X(X*) у нас нет (X – вообще пустая структура). Надо либо убрать new X(), либо добавить в структуру X соответствующий конструктор. Справедливости ради надо отметить, что никакого влияния на суть примера эта опечатка не оказывает.*

Ошибки, которые имелись ввиду, проставлены в комментариях к коду. Если их замечали, то я ставил 3 балла. Для получения более высокой оценки требовалось упомянуть про перемещение и корректно описать нужные операции.

Ключевой вопрос здесь – а почему же нет конструктора копирования и операции копирующего присваивания – ведь мы со второго курса знаем, что они должны быть сгенерированы для структуры Owner (равно как и конструктор умолчания, который, может всплыть)! Вот тут-то и зарыта собака! Их нельзя сгенерировать, потому что член этой структуры std::unique_ptr<X> _ptr не имеет этих операций (с C++11 они были объявлены как приватные, а с C++14 появилось понятие удаленной программистом функции, и вот эти то функции из std::unique_ptr были удалены). А зачем удалены? Да потому что std::unique_ptr НЕЛЬЗЯ КОПИРОВАТЬ – на то он и уникальный. Проблема std::auto_ptr, сдуру включенного в старый STL как раз состояла в том, что он мог копироваться! Правда при этом значение указателя, которого копировали «портилось» (так как копия то может быть только одна!!!), что было неприятным сюрпризом для программиста, который некритично использовал auto_ptr.

Таким образом, если структура содержит внутри себя std::unique_ptr, то это означает, что и сама структура НЕ МОЖЕТ КОПИРОВАТЬСЯ!

Ряд студентов вполне справедливо это заметил. Но зачем-то они решили, что это лечится добавлением в класс конструктора копирования и операции копирующего присваивания!!! Их нет, потому что их быть не может – они опасны! И вас об этом по-хорошему предупреждают. Но мы не боимся трудностей и стреляем…себе в ногу. Важно понимать, что, конечно, проблема «как бы лечится» добавлением копирующих операций, но вы их

корректно не напишете (то есть у нас выйдет из структуры очередной auto_ptr, про который создатели STL говорят, что его употреблять ВООБЩЕ нельзя).
Если нельзя копировать, то что можно сделать?
КЛЮЧЕВОЙ МОМЕНТ – можно перемещать (move) единственную копию! А что значит, перемещать, а не копировать? Это значит, что нужно переопределить не копирующие, а перемещающие операции конструирования и присваивания. Я ставил 6 баллов, если а). объясняли про перемещение и б). хотя бы правильно описывали прототипы этих операций (без const, с нужным количеством амперсандов!!!)
Кроме того, нужно убрать ссылки на копирование и заменить их на ссылки на перемещение (то есть обозначить копируемые объекты как перемещаемые, то есть правосторонние ссылки(&&) – с помощью std::move(), которая представляет из себя операцию приведения ссылки X& к правосторонней X&&). Но поскольку в задании спрашивалось только о добавлении в структуру, но не изменениях в Foo(), то я, конечно, никого не штрафовал за опускание этого, но это нужно сделать, чтобы все компилировалось.
Следует отметить, что задачу можно было бы вообще решить по-другому – не добавляя ничего в структуру Owner, а просто вставив std::move внутрь Foo(). Если бы это кто-то сделал, то я задачу, конечно бы зачел. Но этого никто не сделал (правда и спрашивалось про другое)
В структуру Owner надо добавить перемещающий конструктор и перемещающую операцию присваивания. Поскольку появился явный конструктор, то потребуется еще и явный конструктор умолчания. Пример программы (Foo () заменена на main()):

```
#include <iostream>
#include <memory>
struct X
{
};
struct Owner
{
	std::unique_ptr<X> _ptr;
	Owner() { }
	Owner(Owner&& rhs) { _ptr = std::move(rhs._ptr); }
	Owner& operator=(Owner&& rhs) { _ptr = std::move(rhs._ptr); re-
turn *this; }
	// другой вариант:
	//Owner(Owner&& rhs) { _ptr.swap(rhs._ptr); }
	//Owner& operator=(Owner&& rhs) { _ptr.swap(rhs._ptr); return
*this;}
};

int main()
{
	Owner o;
	o._ptr = std::make_unique<X>();
	Owner o2(std::move(o)), o3;
	o3 = std::move(o2);
}
```

Задача 2-4

Дайте определение «ленивых» вычислений в языках программирования. В чем заключается связь «ленивых» вычислений и понятия генераторов языка Python? Что делает следующий

фрагмент программы на языке Python? Преобразуйте этот фрагмент в эквивалентный код на Python, используя генераторы и анонимные функции. Предполагается, что переменная L является списком языка Python.

```
L1 = [i for i in L if i > 0]
J = -1
for i in L1:
     if i%2 == 0:
          J = i; break

print(J)
```

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 генераторов языка Python?

Генераторы непосредственно реализуют идиому ленивых вычислений, так как они предназначены как раз для того, чтобы по запросу (next(gen)) выдавать очередную порцию данных как раз в тот момент, когда они нужны.

3. Что делает следующий фрагмент программы на языке Python?

Находит в списке L первое четное положительное число.

4. Преобразуйте этот фрагмент в эквивалентный код на Python, используя генераторы и анонимные функции.

Один из множества возможных вариантов (правда без -1, но я не обращал внимания на это….):

```
def gen(seq, cond):
     for k in seq:
          if cond(k):
               yield k

g1 = gen(L, lambda x: x > 0)
g2 = gen(g1, lambda x: x%2 == 0)
print(next(g2))
```

Замечание: как и в первом варианте код становится компактней, а в этом случае он и «обленился» и не требует лишнего списка L1.