Ответы на вопросы экзамена по курсу «Языки программирования» 08.01.2019

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

Вариант 1

Задача 1-1

Объясните, что означают сокращения для модели данных C/C++: LP32, ILP32, LP64, ILP64, LLP64.
От каких из перечисленных ниже факторов зависит выбор модели данных:
А. Архитектура компьютера Б. Архитектура компилятора В. Обе архитектуры Г. Другие факторы (поясните, какие именно).

Ответ

Ответ на первую часть вопроса: сокращения поясняют, как соотносятся целочисленные типы данных и указатели в реализации языка C/C++ с точки зрения размера в битах. Во всех указанных моделях предполагается, что sizeof(char) = 1 (так по стандарту), sizeof(short) = 2, а вот размеры остальных (int, long, long long) варьируются в зависимости от архитектуры целевого компьютера (размер типа int стараются выбрать так, чтобы он был «естественным» для архитектуры — совпадал с размерностью регистров) и от архитектуры компилятора — в частности в компиляторах Microsoft в ОС Windows размер типа long для архитектур IA-32 и x86-64 выбран равным размеру int, а компиляторы в ОС Linux для этих же архитектур поддерживают соглашение, принятое еще в 1998 году, о том, что размер типа long должен совпадать с размером указателя.
Сокращения LPXY означают, что размер типа long совпадает с размером указателя и равен XY (в битах), а размер типа int – в 2 раза меньше (16 в LP32, 32 в LP64).
Сокращения ILPXY означают, что размеры типов int, long и указателей совпадают и равны XY (в битах).
Наконец, сокращение LLP64 означает, что размер указателя совпадает с размером типа long long и равен 64 (в битах), а размеры типов int и long совпадают и в 2 раза меньше (32 бита).
Ответ на последний вопрос — В — влияют и архитектура компьютера, и архитектура компилятора
Замечание: некоторые студенты отмечали (абсолютно верно!), что выбор модели зависит как от архитектуры компьютера, так и от архитектуры ОС. Я считал, что архитектура компилятора , в частности, определяется архитектурой ОС, и именно это и подразумевал в ответе Б. Так что это вопрос терминологии. Ответ студентам засчитывался, если они выбирали кроме А еще и Г, и поясняли, что речь идет об ОС.

Задача 1-2

В приведенном ниже фрагменте программы на C# заменить знаки вопроса ????? на одно выражение так, чтобы отобрать все четные элементы списка и упорядочить их по возрастанию. В примере должно быть напечатано (построчно) 4,4,6. Единственное разрешенное изменение — замена вопросов на выражение (указание: использовать методы `Where` и `OrderBy`, применимые к последовательностям и преобразующие их соответствующим образом).

```
List<int> l = new List<int>() { 4, 3, 5, 1, 6, 9, 55, 4 };
var l2 = ????? ;
```

```
foreach (int i in l2) System.Console.WriteLine(i);
```

Ответ

Здесь нужно в правильном порядке вызвать методы `Where` и `OrderBy`, которые возвращают последовательности и принимают аргументы — делегаты, поэтому по условию задачи надо использовать лямбда-выражения.

При этом в методе OrderBy (по аналогии с SQL) лямбда-выражение определяет не функцию сравнения от двух аргументов — а функцию с одним аргументом, которая возвращает для элемента значение, которое и будет сравниваться (в данном случае — это просто само значение элемента) Я снижал на 1 балл, если не так.

```
var l2 = l.Where((x) => x % 2 == 0).OrderBy(x => x);
```

Задача 1-3

Написать на языке Swift объявление перечислимого типа данных, описывающего тип данных Barcode («Штрих-код»). Тип состоит из двух вариантов — upc (одномерный шрих-код), которому соответствуют четыре целых числа, и qrCode (двумерный код), которому соответствует строка. Написать также фрагмент программы, распечатывающий значение переменной prodBarCode, которая описана так:

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)  или
var productBarcode = Barcode.qrCode("ABCDEFGHIJKLMNOP")
```

Ответ

Решение приводилось на лекциях. Сам пример взят из руководства по Swift от компании Apple.

Объявление типа:

```
enum Barcode {
  case upc(Int, Int, Int, Int)
  case qrCode(String)
}
```

Фрагмент:

```
switch productBarcode {
case .upc(let numberSystem, let manufacturer, let product, let check):
    print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
case .qrCode(let productCode):
    print("QR code: \(productCode).")
}
```

Задача 1-4

С помощью каких средств можно смоделировать тип данных из задачи 3 на языке C? Сравните решения в двух языках с точки зрения надежности.

Ответ

В классических императивных языках программирования такие типы описываются с помощью понятия «размеченное объединение типов». В языке Си есть понятие объединения, но объединение Си — неразмеченное, поэтому размеченные объединения нужно моделировать путем задания поля-дискриминанта объединения — как первого поля каждого из вариантов.

Замечание: в принципе код на Си писать необязательно — в условии этого не требовалось. Главное — сформулировать общую идею размеченных объединений и перечислить их недостатки по сравнению с решением Swift.

```
enum EBARCODE {
  EBC_UPC,
  EBC_QRCODE
};
#define QR_STRING_MAX_LEN 2953



union tag_BARCODE {
  // Дискриминант объединения
  enum EBARCODE bcType;
  // первый вариант
  struct {
    enum EBARCODE upcType;  // Дискриминант объединения
    int numberSystem, manufacturer, product, check;
  } upc;
  // второй вариант
  struct {
    enum EBARCODE qrCodeType;  // Дискриминант объединения
    char productCode[QR_STRING_MAX_LEN + 1];
  } qrCode;
}
```

Корректная работа с размеченными объединениями возможна, как правило, только через переключатели. Вот фрагмент на Си:

```
switch(barCode.bcType) {
case EBC_UPC:
  printf("UPC: %d, %d, %d, %d.\n",  barCode.upc.numberSystem,
  barCode.upc.manufacturer,  barCode.upc.product,  barCode.upc.check);
  break;
case EBC_QRCODE:
  printf("QR Code: %s.\n",  barCode.qrCode.productCode);
  break;
default:
  fprintf(stderr, "Unexpected error!!!\n");
  break;
}
```

Решение на Си (как и решение на любом классическом императивном языке) сильно проигрывает как с точки зрения наглядности, так с точки зрения надежности. Во-первых, нужно самим моделировать дискриминант и не забывать это делать при добавлении нового варианта (причем нужно не забыть добавить новую константу в перечисление).

Во-вторых, нужно не забывать корректно инициализировать дискриминант и соответствующие поля (в Swift это делается автоматически при инициализации).

В-третьих, смоделированный дискриминант не защищен от неправильного (случайного, например) изменения (в Swift это невозможно даже синтаксически).

В-четвертых, компилятор не в состоянии проконтролировать корректно ли работа с объединением в переключателе (в коде выше можно переставить printf() в альтернативах — компилироваться и работать будет, но результат — неопределен). В Swift такие ошибки невозможны.

Задача 1-5

В языках C# и Java есть параметризованные библиотечные классы `System.Collections.Generic.List<T> и java.util.ArrayList<T>`, реализующие динамические векторы по аналогии, например, с классом `std::vector<T>` из STL. Они ведут себя очень похоже, содержат набор похожих методов, ведущих себя одинаково (например, `add, clear, contains, indexOf` и др.). Однако метод генерации массива из вектора работает по-разному (предполагается, что `list` объявлен как `List<String>` в C# и как `ArrayList<String>` в Java):

```
(C#)String [] arr = list.ToArray(); //OK – все работает
(Java) String [] arr = list.toArray(); // ошибка компиляции
(Java) Object[] arr = list.toArray(); // OK
```

В Java метод `list.toArray()` без параметров возвращает массив типа `Object[]`. Объясните такое «странное» поведение, а также объясните, почему следующий код скомпилируется, но при выполнении сгенерирует исключение `ClassCastException`

```
(Java) String [] arr = (String[])list.toArray();
```

Ответ

Короткий ответ: так происходит из-за того, что в Java происходит «стирание информации о типе» (type erasure) на этапе генерации байт-кода. Именно поэтому `ArrayList<String>.toArray()` (не только для `String`, но и для любого другого класса) может возвращать только массив из Object – в частности потому, что обобщенный класс не может создавать массивы из объектов типа, являющегося аргументом обобщения. *Такое ограничение наложено потому, что в Java во время выполнения информация об объявленном типе элементов массива должна быть доступна — хотя бы для того, чтобы проверять корректность присваивания элементу массива (см. про ковариантность массивов в Java и связанные с этим проблемы), а поскольку эта информация «стерта», то обобщенный класс может создавать и возвращать только массивы конкретных типов, в данном случае подходит только Object.*

То есть возвращаемый массив состоит из объектов типа String, но доступны они только через ссылку на Object. При этом элементы массива `Object[] arr` вполне можно преобразовать к строке:

```
String s = (String)arr[0]; // будет и компилироваться и работать
```

А вот весь массив приводить к `String[]` нельзя, поскольку JVM “видит”, что типом элемента массива может быть только Object, а информация о том, что там на самом деле String – стерта.

Такое поведение позволяет обеспечить полную совместимость со старым кодом. В частности, метод toArray() «старых» коллекций ArrayList во времена отсутствия обобщений в Java вел себя именно таким образом. Именно решение проблем с совместимостью нового и старого байт-кода и привело к появлению подобного рода «странностей». В C# таких «странностей» нет, однако это достигнуто путем отказа от совместимости новых и старых версий .NET.

Заметим в заключение, что в Java метод ArrayList<T>.toArray() без параметров остался только для совместимости. Программисты реально пользуются перегрузкой метода, которая требует в качестве аргумента ненулевую ссылку на массив, объявленный тип которого — аргумент обобщенного типа ArrayList. Если размер массива не совпадает с необходимым, то он реаллокируется (информация об объявленном типе остается, так что ситуация отличается от варианта создания массива) до нужного размера. Ссылка на этот массив возвращается в качестве результата. Поэтому следующий код и компилируется, и работает:

```
String [] arr = list.toArray(new String[0]);
// а можно и так:
String [] arr = new String[0];
list.toArray(arr); // arr = arr – необязательно
// иногда рекомендуют так:
String [] arr = new String[list.count()];
list.toArray(arr); // заранее аллокировать массив строк
// какой вариант лучше с точки зрения производительности, я не знаю, в
// сети есть диаметрально противоположные мнения на этот счет
```

Замечание: здесь все «держится» на понятии стирания информации о типе. Без этого функциональность обобщений была близка (или почти совпадала) с обобщениями C#. Поэтому без упоминания стирания задача не засчитывалась, несмотря но обилие всяких слов. Как правило, эти слова не по существу...

Задача 1-6

В приведенной ниже программе на языке Go дописать функцию merge так, чтобы она выдавала в стандартный вывод упорядоченные значения без повторений, считанные из каналов — аргументов. Предполагается, что в каналы подаются упорядоченные по возрастанию целые значения без повторений (см. функцию `dump`). В приведенном примере должно быть выдано 1 3 5 6 7 8 23 24 50

```
package main; import ("fmt")
func dump(c chan int, a [] int) {
  for _, i:= range(a) { c <- i }
  close(c)
}
func merge (c1, c2 chan int) {
  buf1,more1 := <-c1
  buf2, more2 := <-c2
  // сюда вставить текст функции
}

func main() {
  c1 := make(chan int)
  c2 := make (chan int)
  a1 := [] int {1,5,7,8,23,24,50}
  // упорядочен и без повторов
  a2 := [] int {1,3,6,8,23}
  // упорядочен и без повторов
  go dump(c1,a1)
  go dump (c2,a2)
  merge(c1,c2)
}
```

Ответ

Один из возможных вариантов:

```
func merge (c1, c2 chan int) {
  buf1,more1 := <-c1
  buf2, more2 := <-c2
```

```
  for more1 && more2 {
    if buf1 < buf2 {
     fmt.Printf("%d ", buf1)
     buf1, more1 = <-c1
    } else if buf1 > buf2 {
      fmt.Printf("%d ", buf2)
      buf2,more2 = <-c2
    } else {
      fmt.Printf("%d ", buf1)
      buf1, more1 = <-c1
      buf2,more2 = <-c2
    }
  }
  for more1 {
    fmt.Printf("%d ", buf1)
    buf1, more1 = <-c1
  }
  for more2 {
    fmt.Printf("%d ", buf2)
    buf2, more2 = <-c2
  }
}
```

Замечание: многие студенты «по привычке» вместо for condition писали while (condition). Я их в глубине души понимаю, особенно когда под рукой нет компилятора, поэтому, конечно, за такие синтаксические мелочи не снижал. Главное было понять, что этот алгоритм прекрасно работает с фиксированной памятью, не требует дополнительных структур и существенно компактнее варианта без каналов.

Сильно штрафовалось отсутствие цикла вообще, а также некорректное использование оператора select – он здесь совсем не по существу….

Некоторые вводили завершающее считывание из каналов внутрь цикла. Так менее красиво и эффективно, но я естественно никак за это не штрафовал.