Ответы на вопросы экзамена по курсу «Языки программирования» 08.01.2019

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

Вариант 2

Задача 2-1

Объясните, почему в списке сокращений для модели данных C/C++ - LP32, ILP32, LP64, ILP64, LLP64 - отсутствует сокращение LLP32.

Где используется модель LLP64?

Объясните, что она означает, ее достоинства и недостатки.

Ответ

Сокращения поясняют, как соотносятся целочисленные типы данных и указатели в реализации языка C/C++ с точки зрения размера в битах. *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).

Поэтому сокращение LLP32 означало бы, что тип long long имеет размер 32, а остальные типы (int, long) — в 2 или более раза короче (иначе это называлось бы LP32 или ILP32), то есть типы int и long были бы в 16 бит — это теоретически возможно только для 16-битных архитектур, но неприемлемо для практических применений (слишком короткий тип long).

Модель LLP64 используется для архитектуры x86-64 в компиляторах Microsoft в ОС Windows.

Ее основное достоинство в том, что с точки зрения размера интегральных типов она НИЧЕМ не отличается от модели ILP32, *которая «естественна» для архитектуры IA-32 (“плоская” виртуальная память до 4ГБ, 32-битные регистры общего назначения), которая стала доминирующей в 90-е годы прошлого века, и для нее была написано очень много программ под управлением ОС Windows. Поэтому облегчение переноса программ в новую архитектуру (x86-64) было выбрано приоритетом для компиляторов Microsoft.*

Но для ОС Linux в конце 90-ч годов проблемы совместимости были менее важны(просто в силу меньшего количества ПО для этой системы в тот момент).Поэтому было принято соглашение о поддержке модели данных LP64, которая лучше подходит для 64-битной архитектуры, нежели LLP64.

Так что недостатком архитектуры LLP64 можно назвать то, что она меньше подходит для доминирующей сейчас архитектуры x86-64 по сравнению с моделью LP64.

Задача 2-2

В приведенном ниже фрагменте программы на Java заменить знаки вопроса ????? на выражение так, чтобы отобрать все строки из потока, начинающиеся с буквы J,

упорядочить их по возрастанию и напечатать. В примере должно быть выдано (построчно) `Java Jim Johnny.` Единственное разрешенное изменение — замена вопросов на выражение (указание: использовать методы потоков `sorted, filter, forEach` . печать строки s осуществляется путем вызова `System.out.println(s)`, проверка на то, что строка s начинается с подстроки s1 – `s.startsWith(s1)`).

```
Stream<String> l = Stream.of("Jim", "Java", "Python", "Brainfuck", "Johnny");
l.?????;
```

Ответ

Здесь нужно в правильном порядке вызвать методы `sorted, filter, forEach`, которые возвращают последовательности и принимают аргументы — функциональные интерфейсы, поэтому по условию задачи надо использовать лямбда-выражения, а для печати — ссылку на метод, либо тоже лямбда-функцию.

```
l.filter(s -> { return s.startsWith("J"); })
 .sorted()
 .forEach(System.out::println);
```

Задача 2-3

Написать на языке Swift объявление перечислимого типа данных, описывающего тип данных SocketAddress. Тип состоит из двух вариантов — afInet, которому соответствуют два целых числа (UInt32 и UInt16), и afUnix, которому соответствует строка. Написать также фрагмент программы, распечатывающий значение переменной sAddr, которая описана так:

```
var sAddr = SocketAddress.afInet(0x7F000001,80)
```
или
```
var sAddr = SocketAddress.afUnix("/home/gig/sa")
```

Ответ

Решение является простой переделкой решения задачи из первого варианта, а это решение приводилось на лекциях.

Объявление типа:

```
enum SocketAddress {
  case afInet(UInt32, UInt16)
  case afUnix(String)
}
```

Фрагмент (конечно, можно ip-адреса распечатывать получше - хотя бы побайтно, но здесь это неважно):

```
switch sAddr {
case .afInet(let ipAddr, let port):
    print("AF_INET address: \(ipAddr), \(port).")
case .afUnix(let path):
    print("AF_UNIX address: \(path).")
}
```

Задача 2-4

С помощью каких средств можно смоделировать тип данных из задачи 3 на языке Паскаль? 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 Паскаль ему соответствует конструкция «запись с вариантами».

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

```
const SUN_PATH_MAX = 128;
type SocketDomain = (AF_INET,AF_UNIX);
SocketAddress = record
  // Дискриминант объединения
 case domain: SocketDomain of
  // первый вариант
  AF_INET: (ipaddr: record ip, port:integer end);
  // второй вариант
  AF_UNIX: (path: packed array [1.. SUN_PATH_MAX] of char);
}
```

Корректная работа с размеченными объединениями возможна, как правило, только через переключатели (в Паскале — операторы выбора). Вот фрагмент на Паскале:

```
case sockAddr.domain of
AF_INET:
  begin
    write("AF_INET address: "); write(sockAddr.ipaddr.ip);
    writeln(sockAddr.ipaddr.port:6)
  end;
AF_UNIX:
  begin
    write("AF_UNIX address: ");  writeln(sockAddr.path)
  end
else { в стандарте Паскаля даже этого else нет!!!!}
  writeln("Unexpected error!!!");
end
```

Решение на Паскале (как и решение на любом классическом императивном языке) сильно проигрывает как с точки зрения наглядности, так с точки зрения надежности.

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

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

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

Даже в таком языке как Ада записи с дискриминтами хоть и превосходят решения Паскаля (и тем более Си), но все равно уступают решению языка Swift. Последний недостаток так и остается...

Задача 2-5

Написать на языке C++ шаблон класса `template <typename T> class OnlyFloats { … };` так, чтобы допустимым фактическим параметром шаблона могли быть ТОЛЬКО типы `float` и `double`. Для остальных фактических параметров (неважно — встроенных типов или классов) должна выдаваться ошибка компиляции (неважно, какая именно). То

есть конкретизации `OnlyFloats<float>` и `OnlyFloats<double>` должны компилироваться нормально, а конкретизации вида `OnlyFloats<int>` или `OnlyFloats<std::string>` должны приводить к ошибкам компиляции. Конкретное содержимое и функциональность класса `OnlyFloats` не имеют значения (за исключением сформулированной особенности). Считаем, что компилятор проводит только поверхностный синтаксический анализ определения шаблона.

Ответ

Здесь можно использовать явные (другое название - полные) корректные частичные специализации шаблона для типов `double` и `float` и ошибочную основную версию шаблона. *Неочевидный вопрос — как вызвать ошибку ТОЛЬКО при конкретизации шаблона, но не при первичном синтаксическом анализе определения шаблона. По условию мы предполагаем самый слабый вариант анализа (так поступает, например, компилятор Visual C++ от Microsoft), и ему вполне хватит строчки типа:*

```
enum { val = T};
```

Для более строгого компилятора (типа GNU C++) можно использовать, например, вариант:

```
static T v = 0; // инициализация неконстантных статических членов внутри класса запрещена, но это обнаружится только при конкретизации
```

В любом случае здесь это неважно — засчитывается любой правдоподобный вариант ошибки.

```
template <typename T> class OnlyFloats
{
	enum { val = T}; // семантически неверно, но компилятор
//проанализирует это только при конкретизации, которая случится только
//при типе отличной от double и float
};
// Явные (или полные) специализации для нужных типов
template <> class OnlyFloats<double> {
};

template <> class OnlyFloats<float> {
};

int main()
{
	OnlyFloats<double> foo; // компилируется
	OnlyFloats<int> bar; // ошибка компиляции
	return 0;
}
```

Замечание: особо продвинутые люди обращались к STL последних стандартов и использовали шаблоны типа static_assert, enable_if, is_float, is_same и так далее. Мы не залезали в курсе так глубоко в STL, но за эрудицию, конечно, задача таким засчитывалась, при условии, конечно, отсутствия явных ошибок при обращении к таким «продвинутым» шаблонам. Отметим, кстати, что реализация этих шаблонов в STL, конечно, также использует упомянутый механизм «основная версия-явная специализация».

Задача 2-6

В приведенной ниже программе на языке Go дописать функцию merge так, чтобы она выдавала в стандартный вывод упорядоченные значения, считанные из каналов — аргументов, за исключением тех, что повторяются. Предполагается, что в каналы подаются упорядоченные по возрастанию целые значения без повторений (см. функцию dump). В приведенном примере должно быть выдано:
3 5 6 7 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 X
      buf1, more1 = <-c1
      buf2,more2 = <-c2
    }
  }
  for more1 {
```

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

Отличие от задачи первого варианта — отсутствует печать элемента в случае, когда элементы из каналов совпадают. В остальном решения — идентичны.

См. также замечания к задаче из первого варианта