Quiz: C++ language, basic subset (non-OOP)

Студент _____ *группы* _____

Я подтверждаю что ответы ниже являются моими собственными _____

(число, подпись)

Для ответов на вопросы, помеченные как (WA) и (PA), возьмите отдельный лист.

1. Стандарт и поведение программ

Студент Разборчивый анализирует большой старый проект и натыкается на следующий код:

```
int a = 0xecfb39f5;
unsigned short *d = (unsigned short *) &a;

int
main (void)
{
  if ((unsigned long) (65536 * d[1] + d[0]) < (1UL << 28))
    abort ();
  return 0;
}
```

Q1: Что-то в этом коде ощутимо режет ему глаза. Подскажите ему какое поведение демонстрирует эта программа?

1) Conforming: никаких проблем нет, сравниваются два беззнаковых числа, причем известных на этапе компиляции
2) Unspecified: не специфицирован порядок вычисления аргументов сравнения, результаты могут быть различны
3) Undefined: возможно знаковое целочисленное переполнение в одном из аргументов
4) Syntax violation: приведение целого числа к массиву является синтаксической ошибкой

2. Объявления и определения

Продолжая читать старый код, студент Разборчивый натыкается на следующие конструкции

```
char *(*(**foo[][8])())[] = { 0 };
void myfunction(char *(*(**foo[][8])())[]);
```

Q2: Что здесь написано?

1) Объявление массива foo и объявление функции myfunction
2) Объявление массива foo и определение функции myfunction
3) Определение массива foo и объявление функции myfunction
4) Определение массива foo и определение функции myfunction

Примечание: понятно, что каждое определение является также объявлением, поэтому слово "объявление" выше следует читать как "объявление, но ***не*** *определение"*

Q3 (WA): Прочитайте тип аргумента функции myfunction, запишите результат.

Q4 (PA) : Напишите серию typedef или using (в синтаксисе нового стандарта), которая позволила бы переписать приведенные строчки без боли для глаз читающего код.

3. Перечисления и константы вместо макросов

Студенты Разборчивый и Внезапный нашли в старом коде макроопределения:

```
#define lexer_TERMINAL 0
#define lexer_NONTERMINAL 10
#define lexer_STARTSYM 20
```

и поспорили о том как их переписать. Студент Разборчивый предлагает вариант (1):

```
namespace lexer
{
  enum lexelem
  {
    TERMINAL = 0,
    NONTERMINAL = 10,
    STARTSYM = 25
  };
}
```

Студент Внезапный предлагает вариант (2):

```
namespace lexer
{
  int const TERMINAL = 0;
  int const NONTERMINAL = 10;
  int const STARTSYM = 25;
}
```

Q5: Каждый студент приводит аргументы за свой вариант. Некоторые из этих аргументов верны, некоторые нет. Отметьте верные утверждения среди перечисленных ниже:

За подход с enum:

1) Явный тип `lexer::lexelem` лучше заметен в сигнатурах функций и обеспечивает более строгий контроль типов
2) Вариант 2 содержит явные определения констант, так что его включение в две или более единицы трансляции приведет к нарушению ODR
3) Перечислимый тип определяет строгое rvalue: его элементы не занимают места в памяти. Взять адрес `&lexer::`TERMINAL можно в случае константы и нельзя в случае перечисления

За подход с константами:

4) Возможно дополнительно вводить константы в `namespace lexer` в других модулях – расширение списка констант не требует модификации исходного заголовочника. В случае enum все его элементы должны быть собраны в одном месте
5) Нельзя взять не только адрес элемента перечисления, но и ссылку на него, а это бывает нужно при передаче аргументов
6) Константы типизированы и их можно сделать как целочисленными, так и с плавающей точкой, в то время как перечислимые типы (если не использовать enum class из нового стандарта) ограничены целочисленными константами.

(WA), (Bonus) За этот вопрос можно получить бонусные баллы если привести разумные аргументы за любую из сторон (не связанные с наследованием).

4. От препроцессора к шаблонам и обратно

Студент Шустрый предлагает следующую альтернативу шаблонам функций:

```
--- template.h
#ifndef TEMPLATE_GUARD_
#define TEMPLATE_GUARD_
#define CAT(X,Y) X##_##Y
#define TEMPLATE(X,Y) CAT(X,Y)
#endif

--- max.h
#include "template.h"
T TEMPLATE(max,T)(T a, T b)
{
  return (a > b) ? a : b;
}

--- allmax.h
#define T int
#include "max.h"
#undef T
#define T float
#include "max.h"
#undef T
/* … etc … */

--- use.cpp
#include "template.h"
#include "allmax.h"
#include <stdio.h>
int main () {
  printf ("MAX(5, 6) = %d\n", TEMPLATE(max,int)(5, 6));
  printf ("MAX(5.0f, 6.0f) = %f\n", TEMPLATE(max,float)(5.0f, 6.0f));
  return 0;
}
```

Q6: Отметьте из перечисленных ниже все типы, для которых этот вариант будет работать:

```
1) uint64_t
2) float*
3) const float&
4) long long
5) unsigned long long * const
6) int_least32_t
7) void (*) (int, float)
```

Примечание: в принципе его можно заставить работать для всех этих типов серией промежуточных typedef. В вопросе имеется в виду: напрямую, при подстановке типа T в allmax.h

Q7: Там где этот вариант не будет работать, будут:

1) Ошибки компиляции
2) Ошибки исполнения
3) И ошибки компиляции и ошибки исполнения
4) Он везде будет работать

Q8: Будет ли шаблонный вариант функции max работать для функций?

```
template <typename T> T
max (T x, T y) { return (x > y) ? x : y; }

int one (void) { return 1; }
int two (void) { return 2; }

int foo () { printf ("%d\n", max(one, two)()); }
```

1) Будет и вернёт 1
2) Будет и вернёт 2
3) Будет, но вернёт unspecified значение (или 1 или 2)
4) Даже не скомпилируется

5. Вывод типов и новый стандарт

Студент Шустрый хочет написать гетерогенный обмен значениями следующим образом:

```
template <typename T, typename U> void
swap (T& t, U& u)
{
  auto tmp = t;
  t = (T) u;
  u = (U) tmp;
}
```

Q9: В следующем коде:

```
const int a = 0;
const float b = 0.0;

const int *x = &a;
const float *y = &b;
swap (x, y);
```

Будет ли ошибка и как будет выведен тип для tmp?

1) Будет ошибка компиляции
2) Ошибки не будет, typeof(tmp) == const int *
3) Ошибки не будет, typeof(tmp) == int *
4) Ошибки не будет, typeof(tmp) == const int
5) Ошибки не будет, typeof(tmp) == int

Q10: Что если сделать так:

```
decltype("Apple") a = "Apple";
decltype("Pear") p = "Pear";
swap (a, p);
```

1) Будет ошибка компиляции
2) Ошибки не будет, typeof(tmp) == const char *
3) Ошибки не будет, typeof(tmp) == char *
4) Ошибки не будет, typeof(tmp) == const char
5) Ошибки не будет, typeof(tmp) == char