```
#include <string>
#include <iostream>
#include <cstdint>
#include <algorithm>
#include <iterator>
#include <memory>

using namespace std;

int main()
{

	//Задание 1. Сырые строковые литералы (Raw String Literals)
	//Выведите строку, например: my name is "Marina"
	//а) без использования Raw String Literals
	//б) посредством Raw String Literals
	//в) добавьте разделители (Delimeter)

	{

		__asm nop
	}

	////////////////////////////////////////////////////////////////////////////
	//Задание 2. Реализуйте вычисление факториала с помощью constexpr-функции.
	//
	//Подсказки/напоминания:
	//		- constexpr – функция должна состоять из единственной инструкции return <выражение>; (пока!)
	//		- но это выражение может включать вызов другой constexpr – функции, в частности рекурсивный
	//		  вызов
	//		- если параметр рекурсивной constexpr- функции - это константа, компилятор вычислит результат
	//		  на этапе компиляции

	//Проверьте тот факт, что компилятор вычисляет значение на этапе компиляции (если в качестве
	//			параметра используется константа, известная компилятору на этапе компиляции).
	//			Для проверки достаточно создать встроенный массив с размерностью, вычисляемой
	//			посредством constexpr-функции:

	{	//Например:
		//int ar[factorial(3)];

		//или
		//constexpr int n = factorial(5);
		//int ar1[n];

		//попробуйте:
		//int m = 7;
		//constexpr int n1 = factorial(m);
		//int ar1[n1];

		//а так?
		//int n2 = factorial(m);
```

```
            __asm nop
      }

      ////////////////////////////////////////////////////////////////////
////////////////////////
      //Задание 3a. Перевод с помощью пользовательского литерала из
двоичного представления строкового
      //в значение, например: строку "100000000" -> в значение 256
      //Проверить результат посредством префикса 0b
      //Чтобы вызов пользовательского литерала выглядел просто и читаемо,
например: 100000000_b
      //логично использовать пользовательский литерал с единственным
параметром - const char*

      //Так как речь идет о литералах, логично вычислять значения на
этапе компиляции
      // => реализуйте пользовательский литерал посредством constexpr -
функций
      //Подсказка/напоминание:
      //          - constexpr – функция должна состоять из единственной
инструкции return <выражение>;
      //          - но это выражение может включать вызов другой constexpr
– функции,
      //          - которая может быть рекурсивной (если параметр такой
функции - это константа,
      //            компилятор вычислит результат вызова рекурсивной
функции на этапе компиляции)

      {

            __asm nop

      }

      //Задание 3b. Перевод в строковое двоичное представление, например:
256 -> "0b100000000"
      //Так как строка может быть любой длины, логичнее и проще
возвращать объект std::string
      //=> возвращаемое значение не может быть constexpr!
      //Подсказка: манипулятора std::bin пока нет => преобразование в
двоичную строку
      //придется делать вручную
      //Подсказка: количество разрядов в байте определяет константа
CHAR_BIT - <cstdint>

      {
            //std::string sBin= 256_toBinStr;
            __asm nop
      }

      ////////////////////////////////////////////////////////////////////
////////////////////////
      //Задание 4a. constexpr - объекты
      //Создать класс (шаблон класса?) для хранения и манипулирования
диапазоном значений.
      //В классе должны быть:
      //    переменные для хранения минимального и максимального значений,
      //    методы для получения каждого из значений
      //    метод для проверки - попадает ли указанное значение в диапазон
      //    метод, который получает любое значение данного типа и
формирует результирующее значение:
```

```
      //                                          если принадлежит диапазону,
то его и возвращаем
      //                                          если меньше минимального
значения, возвращаем минимальное
      //                                          если больше максимального
значения, возвращаем максимальное

      //Проверьте тот факт, что компилятор вычисляет значение на этапе
компиляции.
      //                    Для проверки достаточно создать встроенный
массив с размерностью, вычисляемой
      //                    посредством constexpr-метода:
      {

      __asm nop
      }

      ////////////////////////////////////////////////////////////////////
////////////////////////
      //Задание 5. unique_ptr
      {

            //5.a - обеспечьте корректное выполнение фрагмента
            {
                  std::vector<std::string*> v = { new std::string("aa"),
new std::string("bb"), new std::string("cc") };
                  //Распечатайте все строки

                  __asm nop
                  //???
            } //???

            //5.b - модифицируйте задание 5.a:
             //обеспечьте посредством std::unique_ptr:
             //эффективное заполнение (вспомните про разные способы
формирования std::unique_ptr),
             //безопасное хранение указателей на динамически создаваемые
объекты std::string,
             //манипулирование,
             //и освобождение ресурсов
             //

            {
             //Распечатайте все строки

                  __asm nop
             //??? Уничтожение динамически созданных объектов?
            } //???

            {//5.c - дополните задание 5.b добавьте возможность изменять
хранящиеся строки
             //следующим образом (например, добавить указанный суффикс:
"AAA" -> "AAA_1")



                  __asm nop
            }

            {//5.d - динамический массив объектов
```

```
            //Создайте unique_ptr, который является оберткой для
динамического массива
            //с элементами std::string
            //С помощью unique_ptr::operator[] заполните обернутый массив
значениями
            //Когда происходит освобождения памяти?

                __asm nop
            }

            {//5.e - массивы динамических объектов и пользовательская
delete-функция (функтор)
            //Задан стековый массив указателей на динамически созданные
объекты
            //Создайте unique_ptr для такого массива
            //Реализуйте пользовательскую delete-функцию (функтор) для
корректного
            //освобождения памяти

                std::string* arStrPtr[] = { new std::string("aa"), new
std::string("bb"), new std::string("cc") };

                __asm nop
            }

            {//5.f Создайте и заполните вектор, содержащий unique_ptr для
указателей на std::string
                //Посредством алгоритмя copy() скопируйте элементы
вектора в пустой список с элементами
                //того же типа
                //Подсказка: перемещающие итераторы и шаблон
std::make_move_iterator

                __asm nop

            }
            __asm nop

      }

      ////////////////////////////////////////////////////////////////////
//////////////
      //Задание 6.shared_ptr + пользовательская delete-функция

      //Реализовать возможность записи в файл данных (строчек) из разных
источников
      //(для упрощения пусть источниками являются два массива)
      //Так как все "писатели" будут по очереди записывать свои данные в
один и тот же файл,
      //логично предоставить им возможность пользоваться одним и тем же
указателем FILE* =>
      //безопасной оберткой для такого указателя является shared_ptr
      //а. Первый владелец должен открыть/создать файл для записи
      //б. Все остальные писатели должны присоединиться к использованию
      //в. Последний владелец указателя должен закрыть файл

      //Подсказка: имитировать порядок записи можно с помощью функции
rand()
      /*
      {
```

```
	//"писатели":
	//Создать writer1, writer2


	//например, источники данных:
	char ar1[] = "Writer1";
	char ar2[] = "Writer2";

	//заданное число итераций случайным образом позволяем одному из
"писателей" записать в файл
	//свою строчку
	//Подсказка: строчки удобно записывать в файл посредством функции
fputs()


	__asm nop
	}//закрытие файла???

	*/


}
```