1151044 - Programación Orientada a Objetos
Tarea 4 - 2026-I

Se desea definir la siguiente interfaz:

```
struct flujo_entrada {
   virtual ~flujo_entrada( ) = default;
   virtual int examina( ) const = 0;
   virtual void consume( ) = 0;
};
```

El comportamiento esperado de los tipos derivados que implementen las funciones virtuales puras es el siguiente:

- Función miembro `int examina( ) const;`
  Devuelve un entero correspondiente al valor del byte actual de la entrada, sin consumirlo. Si ésta ya se terminó, debe devolver el valor entero `EOF`.
- Función miembro `void consume( );`
  Consume el siguiente byte de la entrada. Si ésta ya se terminó, no tiene efecto.

Con base en esto, implementa dos tipos de dato `entrada_estandar` y `entrada_cadena` que hereden de la interfaz, que sean instanciables (es decir, que se puedan declarar objetos de dichos tipos) y que se comporten como sigue.

```
class entrada_estandar : public flujo_entrada {
   //...
public:
   //...
};
```

- Constructor `entrada_estandar( );`
  Sin efecto.
- Función miembro `int examina( ) const;`
  Devuelve un entero correspondiente al valor del byte actual de la entrada estándar, sin consumirlo. Si ésta ya se terminó, debe devolver el valor entero `EOF`.
- Función miembro `void consume( );`
  Consume el siguiente byte de la entrada estándar. Si ésta ya se terminó, no tiene efecto.

```
class entrada_cadena : public flujo_entrada {
   //...
public:
   //...
};
```

- Constructor `explicit entrada_cadena(const std::string&);`
  Almacena una copia de la cadena dada. El objeto comienza apuntando al primer byte de la misma.
- Función miembro `int examina( ) const;`
  Devuelve un entero correspondiente al valor del byte actual de la cadena. Si ésta ya se terminó, debe devolver el valor entero `EOF`.
- Función miembro `void consume( );`
  Avanza un byte sobre la cadena. Si ésta ya se terminó, no tiene efecto.

Además, implementa la siguiente función que sea capaz de leer un entero a partir de un `flujo_entrada`:

```
flujo_entrada& operator>>(flujo_entrada& ent, int& v) {
   //...
}
```

Si el entero no se pudo leer, la función `operator>>` debe elevar una excepción de un tipo derivado de `std::exception`, como por ejemplo `std::runtime_error` que se define en `<stdexcept>`. Tu función solamente necesita poder leer enteros no negativos y puedes suponer que no habrá problemas de sobreflujo (*overflow*).

Los tipos de dato no debe exponer públicamente ninguna variable miembro interna y deben implementarse de modo de que sea imposible que un objeto de este tipo adquiera un estado inválido. Los tipos de dato no deben permitir fugas de memoria. Puedes usar cualquier archivo de la biblioteca de C++ excepto por `<sstream>` y `<spanstream>`.

Tu código no puede usar `dynamic_cast` ni `typeid`, y no debe declarar variables estáticas ni globales. Un programa que lea a lo mucho cien enteros debe terminar en menos de un segundo. Puedes consultar una página de prueba en https://racc.mx/uam/trimestre-actual/2026-i/poo/tarea4.html. El código que envíes no debe contener `main`, no debe contener la definición de `flujo_entrada` y tampoco debe leer ni imprimir nada directamente.

Envía tu código fuente desde tu cuenta institucional al formulario en https://forms.gle/hKTWWHzwj873dGN38. Tu código será evaluado con varios casos de prueba y se espera que cumpla la semántica descrita.

Ejemplo de uso	Ejemplo de salida

```
int main( ) {
   entrada_estandar ent1;
   int a, b;
   ent1 >> a >> b;
   // suponer que el usuario ingresa "5 7"
   // desde la entrada estándar

   entrada_cadena ent2("  123     456");
   int x, y;
   ent2 >> x >> y;

   std::cout << a << " " << b << "\n";
   std::cout << x << " " << y << "\n";
}
```

```
5 7
123 456
```