Trabajo práctico 1:
Calculadora programable

Normativa

Límite de entrega: domingo 15 de abril, 23:59hs. Enviar el zip al mail: `algo2.dc+tp1@gmail.com`
Normas de entrega: Ver "Información sobre la cursada" en el sitio Web de la materia.
(`http://campus.exactas.uba.ar`)

1. Introducción

Este trabajo consiste en implementar en C++ una calculadora programable. La calculadora se inicializa cargándole un *programa*. Un programa consta de varias **`rutinas`**. Cada rutina está identificada por un nombre y consta de una secuencia de instrucciones. El siguiente es un ejemplo de un posible programa para la calculadora:

Rutina **`A`**	Rutina **`B`**
`read(x)`	`push(2)`
`push(2)`	`read(x)`
`add`	`mul`
`write(y)`	`write(x)`
	`jump(A)`

Este programa tiene dos rutinas, identificadas por sus nombres **`A`** y **`B`**. La rutina **`A`** consta de una secuencia de cuatro instrucciones, y la rutina **`B`** consta de una secuencia de cinco instrucciones.

La calculadora opera con una *pila* que almacena valores numéricos, y con una *memoria* en la que se pueden guardar **`variables`** identificadas por nombre. Los programas pueden utilizar ocho distintos tipos de operaciones: cuatro operaciones aritméticas (`push, add, sub, mul`), dos operaciones de manejo de variables (`read, write`), y dos operaciones de salto (`jump, jumpz`). A continuación se describe el mecanismo de todas las operaciones.

1.1. Operaciones aritméticas (`push, add, sub, mul`)

Las operaciones aritméticas permiten hacer cómputos con números.

- `push(número)`. Mete una **constante** numérica en la pila.

Estado inicial	Código ejecutado	Resultado
20 10 pila	`push(42)`	42 20 10 pila

- `add`. Saca dos valores del tope de la pila, los suma y apila el resultado. En caso de que la pila se encuentre vacía, se toma `0` como valor por defecto.

Estado inicial	Código ejecutado	Resultado
32 10 5 pila	`add`	42 5 pila

Estado inicial	Código ejecutado	Resultado
12 pila	`add`	12 pila

Estado inicial	Código ejecutado	Resultado
pila	`add`	0 pila

Observar que no es lo mismo la pila vacía que la pila que tiene un único valor `0`.

- `sub`. Saca dos valores del tope de la pila, los resta y apila el resultado. El valor en el tope de la pila es el sustraendo. Igual que para la suma, si la pila está vacía, el valor por defecto es `0`.

Estado inicial	Código ejecutado	Resultado
57 99 11 pila	sub	42 11 pila

- `mul`. Saca dos valores del tope de la pila, los multiplica y apila el resultado. Al igual que antes, si la pila está vacía, el valor por defecto es `0`.

Estado inicial	Código ejecutado	Resultado
7 6 pila	mul	42 pila

Ejemplo de código que utiliza operaciones aritméticas.

Estado inicial	Código ejecutado	Resultado
10 pila	push(4) mul push(2) add	42 pila

1.2. Operaciones de manejo de variables (**`read`**, **`write`**)

Las operaciones de manejo de variables permiten almacenar un valor en la memoria de la calculadora, dándole un nombre como **`x`** o **`y`**, para posteriormente recuperar el valor almacenado.

- `write(variable)`. Saca el valor del tope de la pila y lo guarda en la variable indicada. Si la pila está vacía, el valor por defecto es `0`.

Estado inicial	Código ejecutado	Resultado
42 50 pila **x** = 10 **y** = 20 variables	write(**x**)	50 pila **x** = 42 **y** = 20 variables

- `read(variable)`. Mete en la pila el valor actual de la variable indicada. Si la variable no fue inicializada anteriormente, se considera que su valor por defecto es `0`.

Estado inicial	Código ejecutado	Resultado
pila **x** = 7 **y** = 42 variables	read(**y**)	42 pila **x** = 7 **y** = 42 variables

Estado inicial	Código ejecutado	Resultado
pila **x** = 7 variables	read(**y**)	0 pila **x** = 7 variables

Ejemplo de código que utiliza variables.

Estado inicial	Código ejecutado	Resultado
21 pila variables	write(**x**) read(**x**) read(**x**) add	42 pila **x** = 21 variables

1.3. Operaciones de salto (`jump`, `jumpz`)

Las operaciones de salto permiten definir programas que incluyan condiciones y ciclos. La calculadora cuenta en todo momento con una *rutina actual* (por ejemplo, **A** o **B**) y con un *índice de instrucción actual* que indica cuál es la instrucción actual dentro de dicha rutina (0, 1, 2, ...). La calculadora ejecuta siempre la instrucción indicada de la rutina actual. Además, la ejecución procede hacia adelante, es decir, después de ejecutar una instrucción, la calculadora pasa a la siguiente instrucción de la rutina actual, incrementando el índice de instrucción. La única salvedad está dada por las instrucciones de salto:

- `jump(rutina)`. Salta a la rutina indicada, es decir, pasa a ejecutar la primera instrucción de dicha rutina. Más precisamente, tiene el siguiente efecto:

  ```
  rutina actual ← rutina
  índice de la instrucción actual ← 0
  ```

- `jumpz(rutina)`. Saca el valor del tope de la pila. Si el valor es `0`, salta a la rutina indicada. De lo contrario, sigue con la ejecución normalmente, pasando a la siguiente instrucción. Si la pila está vacía, se toma `0` como valor por defecto.

Condiciones de terminación. La calculadora termina la ejecución cuando se da alguna de las condiciones siguientes:

1. La rutina actual no existe en el programa. Por ejemplo, si se ejecuta `jump(XYZZY)` y la rutina **XYZZY** no está definida, la calculadora finaliza la ejecución del programa.
2. La rutina actual existe en el programa, pero el índice de la instrucción actual está fuera del rango de las instrucciones de dicha rutina. Por ejemplo, si se ejecuta la instrucción `add` y esa es la última instrucción de la rutina actual, la calculadora finaliza la ejecución del programa.

Ejemplo de código que utiliza operaciones de salto.

El siguiente programa le resta 1 a la variable **n**. Si **n** llega a ser `0`, el programa termina, saltando a la rutina **FIN**. En caso contrario, vuelve a comenzar la ejecución de la rutina **A**:

Rutina **A**

```
read(n)
push(1)
sub
write(n)
read(n)
jumpz(FIN)
jump(A)
```

2. Implementación

En el archivo `Utiles.h` provisto por la cátedra se cuenta con la definición del tipo enumerativo **Operacion**. Las operaciones posibles son **PUSH**, **ADD**, **SUB**, **MUL**, **WRITE**, **READ**, **JUMP** y **JUMPZ**. Se cuenta también con la definición del tipo **Id**, que es un renombre de **std::string** y sirve para representar nombres de rutinas y nombres de variables.

Deberán implementar tres clases en C++: **Instruccion**, **Programa** y **Calculadora**, de acuerdo con la especificación que se detalla a continuación:

2.1. Clase `Instruccion`

Las instancias de esta clase representan instrucciones acompañadas de parámetros concretos, por ejemplo `push(42)` o `jumpz(FIN)`. Se deben implementar los siguientes métodos públicos:

1. **Instruccion::Instruccion(Operacion operacion, int valor)**: construye una instrucción acompañada de un parámetro entero. *Precondición:* la operación debe ser **PUSH**.
2. **Instruccion::Instruccion(Operacion operacion)**: construye una instrucción sin parámetros. *Precondición:* la operación debe ser **ADD**, **SUB** o **MUL**.
3. **Instruccion::Instruccion(Operacion operacion, Id nombre)**: construye una instrucción acompañada de un parámetro de tipo **Id**, que puede representar el nombre de una rutina o de una variable. *Precondición:* la operación debe ser **READ**, **WRITE**, **JUMP** o **JUMPZ**.

4. `Operacion Instruccion::operacion() const`: devuelve la operación asociada a esta instrucción. Por ejemplo, la operación asociada a la instrucción `write(resultado)` es `WRITE`.
5. `int Instruccion::valor() const`: devuelve el parámetro numérico que acompaña a esta instrucción. Por ejemplo, el valor asociado a la instrucción `push(17)` es el entero `17`. *Precondición:* la operación de esta instrucción debe ser `PUSH`.
6. `Id Instruccion::nombre() const`: devuelve el parámetro de tipo `Id` que acompaña a esta instrucción. Por ejemplo, el nombre asociado a `read(x)` es el string `"x"`. *Precondición:* la operación de esta instrucción debe ser `READ`, `WRITE`, `JUMP` o `JUMPZ`.

2.2. Clase `Programa`

Las instancias de esta clase representan programas que pueden tener varias rutinas, donde cada rutina tiene un nombre y una lista de instrucciones. Se deben implementar los siguientes métodos:

1. `Programa::Programa()`: construye un programa vacío, en el que no hay ninguna rutina definida.
2. `void Programa::agregarInstruccion(Id idRutina, Instruccion instruccion)`: agrega una instrucción a la rutina indicada. Si la rutina indicada no existe, crea una nueva rutina con ese nombre.
3. `bool Programa::esRutinaExistente(Id idRutina) const`: devuelve `true` si y sólo si la rutina indicada está definida en el programa.
4. `int Programa::longitud(Id idRutina) const`: devuelve la cantidad de instrucciones de la rutina indicada. *Precondición:* `idRutina` debe identificar una rutina existente.
5. `Instruccion Programa::instruccion(Id idRutina, int i) const`: devuelve la `i`-ésima instrucción de la rutina indicada. *Precondición:* `idRutina` debe identificar una rutina existente y además debe cumplirse que `0 <= i && i < longitud(idRutina)`.

Ejemplo.

```
Programa p;
p.agregarInstruccion("A", Instruccion(READ, "x"));
p.agregarInstruccion("A", Instruccion(PUSH, 2));
p.agregarInstruccion("A", Instruccion(ADD));
p.agregarInstruccion("A", Instruccion(WRITE, "y"));

p.agregarInstruccion("B", Instruccion(PUSH, 2));
p.agregarInstruccion("B", Instruccion(READ, "x"));
p.agregarInstruccion("B", Instruccion(MUL));
p.agregarInstruccion("B", Instruccion(WRITE, "x"));
p.agregarInstruccion("B", Instruccion(JUMP, "A"));

// p.esRutinaExistente("A") ===> true
// p.esRutinaExistente("C") ===> false
// p.longitud("B")          ===> 5
// p.instruccion("A", 0)    ===> read(x)
// p.instruccion("B", 3)    ===> write(x)
```

2.3. Clase `Calculadora`

En esta clase se debe implementar el mecanismo de ejecución de la calculadora. Se deben implementar los siguientes métodos:

1. `Calculadora::Calculadora(Programa programa)`: construye una calculadora que tiene cargado el programa indicado.
2. `void Calculadora::asignarVariable(Id idVariable, int valor)`: establece el valor de la variable indicada en la memoria de la calculadora.
3. `void Calculadora::ejecutar(Id idRutina)`: ejecuta el programa hasta alcanzar alguna de las condiciones de finalización. La ejecución comienza en la primera instrucción de la rutina especificada por el parámetro `idRutina`. Observar que si `idRutina` no es una rutina existente en el programa, la ejecución finaliza de manera inmediata.
4. `int Calculadora::valorVariable(Id idVariable) const`: devuelve que tiene la variable indicada en la memoria de la calculadora. Si nunca se le dio valor a dicha variable, se asume que su valor por defecto es 0.

Ejemplo.

```
Programa p;
p.agregarInstruccion("MAIN", Instruccion(READ, "x"));
p.agregarInstruccion("MAIN", Instruccion(PUSH, 2));
p.agregarInstruccion("MAIN", Instruccion(ADD));
p.agregarInstruccion("MAIN", Instruccion(WRITE, "x"));

Calculadora c(p);
c.asignarVariable("x", 38);

c.ejecutar("MAIN");
// c.valorVariable("x") ===> 40

c.ejecutar("MAIN");
// c.valorVariable("x") ===> 42
```

Observaciones

Para el TP deberán:

- Elegir la representación interna de las tres clases (atributos **`private`**).
- Implementar **todos** los métodos públicos solicitados en la interfaz especificada en la sección anterior, para las tres clases pedidas.
- Asegurarse de que los métodos y clases cumplan con la funcionalidad esperada a través de casos de test.

Además, pueden agregar todos los atributos y métodos privados que deseen a las clases. No deberían modificar la interfaz pública.

Se proveen los archivos `.h` incompletos para las clases **`Instruccion`** y **`Programa`**. Los archivos `.cpp` y la clase **`Calculadora`** deben ser implementados desde cero. Asimismo, se proveen tests para las clases **`Instruccion`** y **`Programa`** usando el *framework* `gtest`. Se espera que escriban sus propios casos de test para la clase **`Calculadora`**.

Entrega

El trabajo práctico se entregará por mail a la dirección `algo2.dc+tp1@gmail.com`. El código desarrollado se entregará como un archivo comprimido en formato zip hasta el día domingo 15 de abril a las 23:59hs. El mail deberá tener como **Asunto** los números de libreta separados por punto y coma (;). Por ejemplo:

To: algo2.dc+tp1@gmail.com
From: alumno-algo2@dc.uba.ar
Subject: 102/09; 98/10
Adjunto: tp1.zip

Dentro del zip se deberá encontrar todo el código implementado, cualquier documento que consideren necesario para aclarar detalles del desarrollo del práctico e instrucciones para poder compilar el código desarrollado así como para ejecutar los tests desarrollados. Son válidas instrucciones que requieran el uso de CLion.