Programación 4 – soluciones práctico 1

Ejercicio 2:

Se pide implementar en C++ una clase que represente un punto en un plano, utilizando coordenadas cartesianas.

En principio, una clase de C++ es similar a un estructurado de C que además permite incluir operaciones, pero esa no es la única diferencia.

```
class Punto {
   private:
      float x, y;
};
```

Un objeto de clase `Punto` (terminología de Orientación a Objetos análoga a decir "una variable de tipo `Punto`") tiene dos miembros (análogo a "campo" o "atributo") de tipo flotante para representar sus coordenadas. La etiqueta `private:` es una palabra reservada e indica que no podré acceder desde una operación que no pertenece a la clase `Punto`, a todos los miembros que se declaren a continuación (hasta el final de la declaración de la clase o hasta encontrar una etiqueta distinta).

En particular, si declaro un objeto `Punto` de la siguiente forma:

```
Punto p;
```

no podré hacer lo siguiente (se obtiene un error de compilación):

```
p.x = 3.2;
cout << p.x;
```

Para poder hacer lo anterior, los miembros `x` e `y` deberían estar bajo una etiqueta `public:` pero justamente la idea es utilizar el mecanismo de *ocultamiento de información* que proporciona el lenguaje. Por lo tanto, para acceder a los miembros privados de la clase `Punto`, necesitamos declarar e implementar operaciones públicas, de la siguiente forma:

```
// Declaración
class Punto {
   private:
      float x, y;
   public:
      void setX(float); // Análogo setY
      float getX(); // Análogo getY
};

// Implementación
void Punto::setX(float x) {
    this->x = x;
}

float Punto::getX() {
    return this->x:
}
```

Observaciones sobre las implementaciones de las dos operaciones anteriores:

- La clase `Punto` *encapsula* datos y operaciones en una misma declaración.
- El prefijo `Punto::` que antecede al nombre la operación cuando esta se implementa, es para indicar que la misma pertenece (es miembro) de la clase.
- La palabra reservada `this` representa un puntero al *objeto implícito*, es decir, al objeto a través del cual se invoca a la operación. En el caso de la operación `setX` es necesario utilizarlo para evitar ambigüedad, pues en la asignación `x = x` el compilador no tiene elementos para distinguir al miembro privado de la clase, del parámetro de entrada. Una alternativa sería utilizar otro nombre para el parámetro, por ejemplo `unX`, y luego hacer `x = unX`; en este caso no sería necesario acceder a través de `this`, pues se asume que el identificador `x` refiere a un miembro de la clase. Algunos estilos de codificación en C++ recomiendan siempre utilizar `this` para evitar ambigüedades; como contraparte, el código resulta más cargado de caracteres.

Notar que las operaciones no reciben como parámetro al punto, dado que se trabaja con el objeto implícito. Una forma más clara de ver esto, es a través del siguiente ejemplo:

```
Punto p;

p.setX(3.2);
cout << p.getX();
```

En ambos casos, las operaciones se invocan a través del objeto. En el contexto de las invocaciones a `setX` y `getX`, el objeto implícito será `p`.

Por otra parte, si quisiéramos inicializar puntos de la siguiente forma:

```
Punto p1, p2(3.0, 4.5); // Quiero que p1 se inicialice con las coordenadas
                        // (0.0, 0.0) por defecto y que p2 se inicialice
                        // con los valores pasados como parámetros en x e y
                        // respectivamente
```

Hay que implementar explícitamente el constructor por defecto y el constructor por parámetros:

```
class Punto {
   private:
      float x, y;
   public:
      Punto();
      Punto(float, float);
      void setX(float); // Análogo setY
      float getX(); // Análogo getY
};

Punto::Punto() {
    this->x = 0.0;
    this->y = 0.0;
}

Punto::Punto(float x, float y) {
    this->x = x;
    this->y = y;
}
```

O alternativamente, las dos operaciones en una misma:

```
class Punto {
   private:
      float x, y;
   public:
      Punto(float =0.0, float =0.0);
      void setX(float); // Análogo setY
      float getX(); // Análogo getY
};
```

De esta forma no se implementa el constructor por defecto y el código del constructor por parámetros es exactamente el mismo al ya implementado.

Observaciones:

- Notar que los constructores tienen una sintaxis particular. No retornan nada pero no se declara un resultado `void`.
- Estos constructores se invocan automáticamente cuando se crea un objeto de la clase correspondiente. La memoria ya está reservada y los constructores solamente realizan acciones de inicialización.

Por otro lado, notar que sin implementar nada más es posible hacer lo siguiente, con el comportamiento habitual:

```
{  // Comienzo del bloque, se asigna automáticamente memoria estática
   // parap1 y p2

  Punto p1, p2(3.0, 4.5);
  p1 = p2; // Asigna el contenido de p2 en p1

}  // Fin del bloque, se libera automáticamente la memoria estática ocupada
   // por p1 y p2
```

Cuando una clase maneja explícitamente memoria dinámica (a través de `new` y `delete`), es necesario implementar explícitamente otros constructores, el operador de asignación y el destructor. Se verá en la clase 2.

De la misma forma que se pueden declarar arreglos de variables, se pueden declarar arreglos de objetos, por ejemplo:

- Arreglo estático:

```
Punto pts[10];
// Crea un arreglo de 10 objetos de clase Punto e inicializa a cada
// uno invocando al constructor por defecto.
// Al salir del bloque {} donde se declaró pts, se libera la memoria
// estática ocupada por los 10 objetos.
```

- Arreglo dinámico:

```
Punto *pts;
pts = new Punto[10];
```

```
      // Obtiene memoria dinámica para 10 objetos de clase Punto e
      // inicializa a cada uno invocando al constructor por defecto.

      delete [] pts;
      // Libera la memoria dinámica ocupada por los 10 objetos.
```

En este ejercicio, se pide además implementar una clase que represente un segmento en un plano, que incluya una operación que calcule el largo del segmento.

```
class Segmento {
   private:
      Punto p1, p2;
   public:
      Segmento();
      Segmento(Punto, Punto);
      Segmento(Punto);
      Segmento(float, float, float, float);
      float largo();
};
```

Observaciones:

- Una clase (en este caso, `Segmento`) puede tener como miembros a objetos de otras clases (`Punto`), de la misma forma que los tipos elementales (como `Punto` tiene miembros de tipo `float`).
- Pueden existir diferentes constructores por parámetros según se necesite. En este caso hay tres: (1) para crear un segmento a partir de sus dos puntos extremos, (2) para crear un segmento a partir de un punto y el origen (0.0, 0.0), y (3) para crear un segmento a partir de las coordenadas de sus dos puntos extremos.

Implementación de la clase:

```
Segmento::Segmento() {
}
// Se inicializan los miembros p1 y p2 con el constructor por defecto de
// Punto

Segmento::Segmento(Punto p1, Punto p2) {
    this->p1 = p1;
    this->p2 = p2;
}
```

Alternativamente (y más eficiente), el constructor anterior se puede implementar de la siguiente forma:

```
Segmento::Segmento(Punto unP1, Punto unP2) : p1(unP1), p2 (unP2){
}
// El código después de los : indica la inicialización de los miembros
```

Las demás operaciones:

```
Segmento::Segmento(Punto p) : p1(p) {
}
// Se inicializa p1 con el parámetro p y p2 con el valor por defecto
// (0.0, 0.0)
```

```
Segmento::Segmento(float x1, float y1, float x2, float y2) :
                   p1(x1, y1), p2(x2, y2) {
}
// Inicializa los miembros p1 y p2 del segmento, invocando al contructor
// por parámetros de Punto
```

Notar que lo siguiente no sería válido (no compila):

```
Segmento::Segmento(float x1, float y1, float x2, float y2) {
    this->p1.x1 = x1;
    this->p1.y1 = y1;
    this->p2.x2 = x2;
    this->p2.y2 = y2;
}
```

dado que las coordenadas son miembros privados de `Punto` y por lo tanto no se puede acceder a ellas desde `Segmento`. Alternativamente se podría implementar de la siguiente forma, pero no es práctico:

```
Segmento::Segmento(float x1, float y1, float x2, float y2) {
    this->p1.setX(x1);
    this->p1.setY(y1);
    this->p2.setX(x2);
    this->p2.setY(y2);
}
```

Las demás operaciones:

```
float Segmento::largo() {
    return sqrt(pow(p2.getX() - p1.getX(), 2) +
                pow(p2.getY() - p2.getY(), 2));
}
```

Para utilizar las funciones `pow` y `sqrt` se debe incluir la biblioteca `cmath`.

Organización de los archivos fuente:

- Usualmente se separa la declaración de una clase (archivo `.h`), de su implementación (archivo `.cpp`).
- Lo anterior permite ocultar los detalles de implementación de las operaciones, si bien no se oculta la estructura interna de la clase. Notar que el cliente de la clase (otra clase que necesite utilizarla), basta con que conozca el archivo `.h` (el cual tiene que incluir, mediante la directiva al precompilador `#include`) y el archivo `.cpp` compilado (código objeto).
- Entonces por cada clase habrá un archivo `.h` y otro `.cpp`. No se recomienda declarar varias clases dentro de un mismo archivo `.h`, dado que quedarían acopladas. Esta recomendación se flexibiliza cuando se trata de varias clases altamente acopladas, por ejemplo, que forman parte de una misma biblioteca.
- Dado que una misma clase puede necesitarse desde diferentes clientes (`#include` de un mismo archivo desde varios archivos diferentes), debe evitarse que el compilador reciba varias copias del mismo código, lo que causaría un error de compilación debido a múltiples

definiciones de un mismo identificador. Para eso se utilizan las directivas al precompilador `#ifndef, #define, #endif`. A continuación, se muestra un ejemplo completo:

Archivo Segmento.h

```
#ifndef SEGMENTO
#define SEGMENTO

#include "Punto.h"

class Segmento {
   private:
      Punto p1, p2;
   public:
      Segmento();
      Segmento(Punto, Punto);
      Segmento(Punto);
      Segmento(float, float,
               float, float);
      float largo();
};

#endif
```

Archivo Segmento.cpp

```
#include "Segmento.h"
#include <cmath>

Segmento::Segmento() {
}

Segmento::Segmento(Punto unP1,
                   Punto unP2) :
          p1(unP1), p2 (unP2){
}

Segmento::Segmento(Punto p) : p1(p) {
}

Segmento::Segmento(float x1, float y1,
                   float x2, float y2) :
          p1(x1, y1), p2(x2, y2) {
}

float Segmento::largo() {
    return sqrt(
           pow(p2.getX()-p1.getX(), 2) +
           pow(p2.getY()-p2.getY(), 2));
}
```

Observaciones sobre el código anterior:

- Por convención se nombra a los archivos `.h` y `.cpp` con el mismo identificador de la clase.
- La directiva `#ifndef` causa que cuando el precompilador procese el archivo `Segmento.h`, pregunte si en su tabla de símbolos está definido el identificador `SEGMENTO` (como convención se usa el mismo nombre de la clase, pero todo en mayúscula). Si no está definido, entonces lo ingresa en dicha tabla (mediante `#define`) y procesa todo el código. Si ya está definido (porque ya se incluyó desde otra clase cliente), entonces no se procesa. Notar que la primera línea debe contener la directiva `#ifndef`, mientras que la última debe indicar `#endif`.
- El archivo `Segmento.h` debe incluir al archivo `Punto.h`, dado que la clase `Segmento` tiene miembros de clase `Punto`. El texto entre comillas del `#include` denota el camino hasta el archivo, puede incluir directorios en caso de que el archivo `Punto.h` esté en otro directorio diferente al actual.
- En `Segmento.cpp` se debe incluir `Segmento.h`, dado que se necesita la declaración para compilar la implementación. Además, se incluye `cmath` (en este caso entre llaves `<>`, por ser una biblioteca estándar), dado que se necesita para la implementación de la operación `largo`.

Ejercicio 3:

Se pide implementar una clase que permita representar fechas, con día, mes y año.

La primera decisión que se debe tomar refiere a la representación interna, para la cual en principio se pueden manejar dos opciones:

1. Tres miembros de tipo entero para el día, mes y año, respectivamente.
2. Un entero largo que represente la cantidad de días desde una fecha inicial.

Ambas opciones tienen ventajas y desventajas. La opción 1 permite mostrar directamente la fecha sin realizar conversiones, mientras que algunas operaciones pueden tener una lógica compleja. La opción 2 facilita algunas operaciones, pero requiere realizar una transformación para su visualización, además de restringir el rango de fechas que se puede manejar (depende de la fecha inicial).

Para esta resolución se tomará la opción 1, sin embargo, es importante tener claro que una representación interna intuitiva no necesariamente es la más adecuada en determinadas situaciones. Por simplicidad, asumiremos que todos los meses tienen 30 días.

```
class Fecha {
  private:
    int dia, mes, anio;
  public:
    Fecha(int, int, int);
    Fecha avanzar(int);
    Fecha retroceder(int);
    int diferencia(Fecha);
    bool igual(Fecha);
};

Fecha::Fecha(int dia, int mes, int anio) {
    this->dia = dia;
    this->mes = mes;
    this->anio = anio;
}

Fecha Fecha::avanzar(int inc) {
     int nDia = (this->dia + inc) % 30;
     int nMes = (this->mes + (this->dia + inc) / 30)) % 12;
     int nAnio = (this->anio) + (this->mes + (this->dia + inc) / 30) / 12;

     return Fecha(nDia, nMes, nAnio); // Devuelve una nueva fecha
}

// La implementación de la operación retroceder as análoga a la de avanzar

int Fecha::diferencia(Fecha f) {
     long int f1 = f.dia + f.mes*30 + f.anio*12*30;
     lont int f2 = this->dia + this->mes*30 + this->anio*12*30;

     return f2 - f1;
}

bool Fecha::igual(Fecha f) {
     return this->dia==f.dia && this->mes==f.mes && this->anio==f.anio;
}
```

Ejemplo de declaración de objetos e invocación a operaciones:

```
Fecha f1(12, 1, 2020), f2(14, 1, 2020), f3;
int dif;

f3 = f2.avanzar(4);
dif = f2.diferencia(f1);
if (f1.igual(f2)) {
      // Hacer una cosa
} else {
      // Hacer otra cosa
}
```

Notar que sería más cómodo poder hacer lo siguiente:

```
f3 = f2 + 4;
dif = f2 - f1;
if (f1 == f2) {
      // Hacer una cosa
} else {
      // Hacer otra cosa
}
```

Para lograr lo anterior, hay que hacer los siguientes cambios, tanto en la declaración como en la implementación de la clase Fecha:

- Sustituir `avanzar` por `operator+`
- Sustituir `diferencia` por `operator-`
- Sustituir `igual` por `operator==`

Con lo anterior, estamos haciendo uso del mecanismo de sobrecarga de operadores de C++. El código anterior es una forma abreviada de lo siguiente (que también compila, pero claramente no es cómodo de usar):

```
f3 = f2.operator+(4);
dif = f2.operator-(f1);
if (f1.operator==(f2)) {
      // Hacer una cosa
} else {
      // Hacer otra cosa
}
```

Al momento de sobrecargar operadores, tener en cuenta lo siguiente:

- Hay un conjunto predefinido de operadores que se pueden sobrecargar. Por ejemplo, para `Fecha` podría ser útil sobrecargar el `operator++`:

```
void Fecha::operator++() {
    *this = *this + 1;
}
// Modifica el objeto implícito, invocando a la operación que
// incrementa una fecha, pasando como parámetro el valor de un día.
```

- La cantidad y tipo de los parámetros no están predefinidos, tampoco están relacionados con el operador que se está sobrecargando. Esto significa que hay relativa libertad al momento de sobrecargar los operadores.
- Debido al punto anterior, se debe cuidar de no sobrecargar un operador con un comportamiento que no es el esperado. Por ejemplo, en el siguiente código:

  ```
  String s1, s2, s3;
  s3 = s1 + s2;
  ```

  es razonable esperar que en `s3` se asigne la concatenación de `s1` y `s2`. Pero nada impide que yo sobrecargue el `operator+` en `Fecha` para que decremente su valor, cosa que no es deseable.

- Un mismo operador se puede sobrecargar para distintas clases (por ejemplo, `operator+` está sobrecargado para `String` y también para `Fecha`), pero también se puede sobrecargar en una misma clase, siempre que tenga diferentes parámetros, por ejemplo, en `Fecha` podemos declarar las siguientes operaciones:

  ```
  Fecha operator-(int);
  // Decrementa a la fecha implícita en una cantidad de días indicada
  // por el parámetro

  int operator-(Fecha);
  // Calcula la diferencia en días entre dos fechas
  ```

  Tener en cuenta que sumar dos fechas en principio no tiene sentido, por más que sintácticamente el lenguaje lo permita.

Finalmente, notar que no es necesario implementar el operador de asignación ni el destructor en la clase `Fecha`, dado que el comportamiento por defecto es el deseado.

La implementación del operador de inserción de flujo (`<<`) se verá en la siguiente clase de práctico.

Ejercicio 4:

Se pide implementar en C++ el datatype String.

Observar que:

- Al ser un datatype, las variables se declaran de la misma forma que cualquier tipo elemental, por ejemplo:
  ```
  int i;
  String s;
  ```
- Nunca será necesario declarar algo del tipo puntero a String, es decir, `String *s`. Notar que esto requeriría hacer `s = new String` y luego de que no se necesite más se deberá hacer `delete s`, lo que no es práctico.
- En lenguaje C existe una biblioteca de operaciones para la manipulación de cadenas de caracteres representadas mediante `char *`. La principal desventaja de esta biblioteca es que el manejo de la memoria debe realizarse explícitamente.

- En lenguaje C++ existe el namespace std que incluye a la clase `string`, con funcionalidades y forma de uso muy similares a lo que se pide en este ejercicio. Es la forma más práctica de manejar cadenas de caracteres en un programa C++. En este ejercicio, se pide implementar una clase similar con el objetivo de discutir el manejo explícito de memoria dinámica, copias y sobrecarga de operadores.

Representación interna:

Para la cadena de caracteres se podría utilizar un arreglo estático, pero esto limita el tamaño de las cadenas que pueden manejarse. Para representar cadenas de largo variable se utiliza char *.

Para marcar el fin de la cadena se puede utilizar o bien un centinela (por ejemplo, el caracter ´\0´ que se utiliza en la biblioteca de C) o una variable que indique el tope.

Utilizaremos la siguiente representación, donde el string vacío se representará con una cadena de un solo carácter, que tiene el ´\0´:

```
class String {
   private:
      char *cadena;
      int largo;
}
```

Las variables de tipo String quisiéramos declararas e inicializarlas de la siguiente forma:

```
      String s1, s2("hola");
```

Para eso necesitaremos los siguientes constructores:

```
String::String() { // constructor por defecto
      cadena = char [1];
      cadena[0] = ´\0´;
      largo = 0;
}


String::String(char *c)  { // constructor por parámetros
      int i = 0;
      while (c[i] != ´\0´) // la cadena en c viene inicializada con
                           // un ´\0´ al final (es una constante)
            i++;
      largo = i;
      cadena = new char [largo + 1];
      for (i=0; i<= largo; i++) // copio hasta el ´\0´ inclusive
            cadena[i] = c[i];
}
```

Notar que cuando declaro el string s2, en el stack se reserva memoria (de forma estática) para un puntero a char (la cadena) y un entero (el largo). Cuando se inicializa el contenido, en este caso a partir de la cadena "hola", se reserva memoria dinámica en el heap (a través de new) para 5 caracteres, los de "hola" más el ´\0´. Esa memoria dinámica deberá liberarse cuando corresponda.

Si el string se declara dentro de un bloque {} de la siguiente forma:

```
{
```

```
      String s2("hola");
}
```

Al salir del bloque se libera automáticamente la memoria estática de s2 (el puntero y el entero) y se invoca automáticamente al destructor, que deberá hacer lo siguiente:

```
String::~String() {
      delete [] cadena;
}
```

Si no se implementa el destructor de esta manera, se dejará memoria utilizada e inaccesible. Notar que no es necesario verificar que la cadena sea distinta de NULL, dado que incluso la cadena vacía tiene al menos un elemento.

Notar que las clases Punto y Fecha (Ejercicios 4 y 5 respectivamente) NO necesitan un destructor como este, dado que no manejan explícitamente memoria dinámica, de la misma forma que no se necesita un destructor para los tipos elementales como int o float.

Ahora si queremos hacer lo siguiente:

```
String s1, s2("hola");
s1 = s2; // copiar el contenido de s2 sobre s1, sobreescribiendo lo
          //que hay en s1, si hay algo
```

Necesitamos implementar el operador de asignación de la siguiente forma:

```
String &String::operator=(const String &s) {
      if (this != &s) { // verifico si es autoasignación
                        // (comparación de punteros)
            delete cadena; // elimino lo que haya en el objeto implícito
            largo = s.largo;
            cadena = new char [largo + 1];
            for (i=0; i<= largo; i++) // copio la cadena del parámetro en
                                      //la del objeto implícito
                  cadena[i] = s.cadena[i];
      }
      return *this;
}
```

Dos comentarios sobre el cabezal de la operación:

- El tipo de retorno es para poder hacer asignaciones en cascada, por ejemplo, s1=s2=s3.
- El parámetro s no se modifica, estrictamente debería ser un pasaje por valor, por eficiencia se pasa una referencia, pero es constante para proteger al objeto de eventuales modificaciones.

Ahora si queremos hacer lo siguiente:

```
String s1("hola");
String s2(s1); // inicializamos s2 con el contenido de s1
```

Necesitamos implementar un constructor de copia de la siguiente forma:

```
String::String(const String &s) {
      // NO elimino lo que haya en el objeto implícito, porque no
      // está inicializado, dado que es la primera vez que existe
      largo = s.largo;
      cadena = new char [largo + 1];
      for (i=0; i<= largo; i++) // copio la cadena del parámetro en
                                //la del objeto implícito
```

```
        cadena[i] = s.cadena[i];
}
```

Siempre que implementamos un datatype que maneja explícitamente memoria dinámica, se deben implementar el destructor, el operador de asignación y el constructor de copia, tal como se hizo con la clase String. En particular, el constructor de copia se invoca también automáticamente cuando se pasa un parámetro por valor y cuando se retorna un objeto como resultado de una función.

Otras operaciones relevantes:

```
bool String::operator==(const String &);
```

Operador de comparación, devuelve true sí y solo sí el objeto pasado como parámetro contiene una cadena de caracteres idéntica a la del objeto implícito, false en caso contrario. También se deben sobrecargar los operadores != (distinto), <, >, <=, >= (comparaciones lexicográficas).

```
String String::operator+(const String &s) { // concatenación
      String result;
      result.largo = largo + s.largo;
      result.cadena = new char[result.largo + 1];
      // Copiar en result.cadena, primero el contenido de
      // cadena y a continuación el contenido de s.cadena. Al
      // final poner un ´\0´. Hacerlo.
      return result;
}
```

Notar que se devuelve un nuevo objeto, no se modifica el objeto implícito. Si hago s1=s2 + s3, no quiero que se modifiquen ni s2 ni s3. También notar que el resultado se devuelve por valor, en este caso se invocará automáticamente al constructor de copia para copiar el contenido de la variable result, la cual será destruida al finalizar la operación.

Acceso a una posición de la cadena, tanto para lectura como para escritura, por ejemplo, si queremos hacer lo siguiente:

```
String s("hola");
char c;
c = s[0]; // en c se asigna el caracter ´h´
s[3]= ´i´; // asigna el caracter ´i´ en la última posición del
           // string, resultando "holi"
```

La operación se define como:

```
char &String::operator[](int pos) {
      return cadena[pos];
}
```

Notar que al devolverse una referencia, permite usar la operación tanto del lado izquierdo como del lado derecho del operador de asignación.

Operador de inserción de flujo, para poder hacer lo siguiente:

```
String s("hola");
cout << s; // imprime el contenido de s en la pantalla
```

Para eso es necesario implementar la siguiente operación, que NO es parte de la clase String (si bien se declara en el mismo archivo .h a continuación de la declaración de la clase):

```
ostream &operator<<(ostream &o, const String &s) {
      o << s.getCadena(); // notar que al no ser una operación
                          // miembro de la clase, necesito una
                          // operación pública que devuelva la
                          //cadena char * del String
      return o; // para poder hacer invocaciones en cascada
}
```

El operador de extracción de flujo es análogo al anterior, pero para cargar datos en un string. Se usa de la siguiente forma:

```
String s; // se declara un string vacío
cin << s; // se lee una cadena de caracteres desde el teclado y se
          // carga en s
```

Para lo cual es necesario implementar la siguiente opreración:

```
istream &operator>>(istream &i, String &s)
{
      // s entra por referencia porque se va a modificar
      char sLeer[MAX_LARGO];
      // arreglo de caracteres de largo especificado por una
      // constante
      i.getline(sLeer, MAX_LARGO);
      // función de istream que carga la entrada en una char *
      String sAux(sLeer);
      // se crea un objeto de clase String a partir de un char *
      s = sAux;
      // se asigna a s
      return is;
      // se libera automáticamente la memoria de sLeer y de sAux
}
```

Implementación y uso de excepciones:

Como primera aproximación, se pide:

- Al intentar acceder a una posición del String inválida se lance la excepción "std::out_of_range".
- Al recibir un parámetro inválido en una operación se lance la excepción "std::invalid_argument".

Estas excepciones son genéricas y en una implementación concreta de una nueva clase, debe buscarse los lugares apropiados donde correspondería lanzarlas. En el caso de la clase String, una operación donde es claro que correspondería lanzar una excepción es la siguiente:

```
char &String::operator[](int pos) {
      if (pos < 0 || pos >= largo)
            throw std::out_of_range;
      return cadena[pos];
}
```

Para atrapar la excepción, un posible código es el siguiente:

```
int main () {
      bool salir = false;
      while (!salir) {
            try {
                  // Todo lo que va en el menu prinicpal:
                  // imprimir, pedir opciones, swith, etc
                  // Invoca operaciones que pueden lanzar excepciones
                  . . .
                  MiClase obj;
                  obj.f(); // f puede lanzar una excepción
                  . . .
                 } catch(...) {
                        cout << "Error inesperado" << endl;
                 }
      }
      return 0;
}
```

Notar que en este código, si todos los datos se ingresan correctamente y se hacen todos los chequeos, nunca se imprimirá el mensaje “Error inesperado”. Si ocurriera algún error que causara el lanzamiento de una excepción, el programa principal la atrapará y continuará su ejecución de forma habitual sin abortar.

Las excepciones en C++ se pueden utilizar de diferentes maneras, para diferentes propósitos y en distintas etapas del desarrollo (depuración, testeo, producción). En este curso, como criterio general se recomienda NO usar excepciones para controlar escenarios lógicos esperados (ingreso de un dato inválido, el cual se debería pedir de nuevo hasta que sea correcto), sino para manejar problemas o situaciones que si bien se sabe que podrían ocurrir, se entiendan como excepcionales (por ejemplo: división por cero, memoria insuficiente). Típicamente son situaciones frente a las cuales si se reacciona de forma apropiada, se puede evitar que el programa aborte su ejecución. Finalmente, notar que es posible implementar un mecanismo de escalamiento de lanzamiento de excepciones, desde funciones anidadas hasta eventualmente el programa principal.

Observación final: En este curso generalmente no hacemos manejo de memoria a tan bajo nivel, sin embargo, entender el funcionamiento de la clase String es una muy buena forma de entender el manejo de memoria en general de C++.

Ejercicio 10:

a) Se identifica una clase base para representar empleados, que almacena el nombre. Luego se identifican los empleados comunes, que además del nombre tienen información sobre el sueldo, y los jornaleros, que tienen información sobre la cantidad de horas trabajadas y el valor de la hora. Las clases común y jornalero son clases derivadas de empleado.

b) En este caso, la clase que representa a los empleados es abstracta, dado que en el sistema no existen empleados que no sean comunes ni jornaleros.

c) La operación getTotal() recorre la colección de todos los empleados y para cada uno calcula la liquidación: si es común corresponde al sueldo mensual, si es jornalero corresponde al producto de la cantidad de horas trabajadas por el valor de la hora. En una versión preliminar, para cada empleado se debería averiguar de qué tipo es (común o jornalero), para saber cómo se calcula su respectivo sueldo. Si en un futuro surge un nuevo tipo de empleado, se debe modificar la operación (un nuevo case dentro del switch que discrimina según el tipo de empleado).

d) El polimorfismo es la capacidad de asociar diferentes métodos a una misma operación. Para implementar una alternativa a la versión preliminar de la operación getTotal() también usaremos la propiedad de intercambiabilidad (o subsumption), que implica que un objeto de clase base puede ser sustituido por un objeto de clase derivada. Ver clase de tórico *03 - Conceptos Básicos de Orientación a Objetos 1era Parte*.

e)

```
class Empleado {
      private:
            string nombre;
      public:
            Empleado(string);
            string getNombre();
            virtual float darLiquidacion() = 0;
            // La operación es virtual para habilitar despacho
            // dinámico y es pura (=0) porque no tiene método
            // (no hay forma de calcular el sueldo si el
            // empleado no es común o jornalero)
};

Empleado::Empleado(string unNombre) {
      nombre = unNombre;
}

string Empleado::getNombre() {
      return nombre;
}

class Comun : public Empleado // clase derivada de Empleado {
      private:
            float sueldoMensual; // además herada el nombre de Empleado

      public:
            Comun(string, float);
            float getSueldoMensual(); // además hereda getNombre de
                                      // Empleado
```

```
            virtual float darLiquidacion(); // Tiene método
};

Comun::Comun(string unNombre, float unSueldoMensual) : Empleado(unNombre) {
      sueldoMensual = unSueldoMensual;
}

float Comun::getSueldoMensual() {
      return sueldoMensual;
}

float Comun::darLiquidacion() {
      return sueldoMensual;
}
```

Notar que las dos operaciones anteriores hacen lo mismo, sin embargo, la primera es un get habitual (en este caso, de la clase Comun), mientras que la segunda es una operación polimórfica definida en la clase base Empleado y luego implementada en cada una de sus clases derivadas.

```
class Jornalero : public Empleado // clase derivada de Empleado {
      private:
            int cantHoras;
            float valorHora; // además herada el nombre de Empleado
      public:
            Jornalero(string, int, float);
            int getCantHoras();
            float getValorHora(); // además hereda getNombre de Empleado
            virtual float darLiquidacion(); // Tiene método
};

Jornalero::Jornalero(string unNombre, int unCantHoras, float unValorHora) :
                     Empleado(unNombre) {
      cantHoras = unCantHoras;
      valorHora = unValorHora;
}

int Jornalero::getCantHoras() {
      return cantHoras;
}

float Jornalero::getValorHora() {
      reutrn valorHora;
}

float Jornalero::darLiquidacion() {
      return cantHoras*valorHora;
}
```

f)

Primero implementaremos la colección de empleados como un array con tope.

```
class ColEmpleados {
      private;
            Empleado *emps[MAX_EMPLEADOS];
            // Notar que si la colección es de un tipo T elemental o
```

```
            // datatype (que no requiera polimorfismo), el tipo base del
            // arreglo es T y no T *
            int tope;
      public:
            ColEmpleados();
            void agregarEmpleado(Empleado *);
            Empleado *iesimo(int);
            int getCant();
            void liberar(); // no conviene que el destructor de una
                            // colección libera la memoria de sus objetos
};

ColEmpleados::ColEmpleados() {
      tope = 0;
}

void ColEmpleados::agregarEmpleado(Empleado *e) {
      emps[tope] = e;
      tope++;
}

Empleado *ColEmpleados::iesimo(int i) {
      return emps[i];
}

int ColEmpleados::getCant() {
      return tope;
}

void ColEmpleados::liberar() {
      for (int i=0; i<tope; i++)
            delete emps[i];
}
```

Luego definimos el programa principal:

```
int main() {
      ColEmpleados empleados;
      int opcion = 0;
      do {
            cout << "Ingresar opción (0-Salir, 1-Ingresar empleado,
                                        2-Calcular liquidación)" << endl;
            cin >> opcion;
            switch (opcion) {
                  case 0:
                        break;
                  case 1:
                        string nom;
                        int tipoEmp;
                        Empleado *emp;
                        cout << "Ingresar nombre" << endl;
                        cin >> nom;
                        cout << "Ingrear tipo de empleado
                                 (1-Comun, 2-Jornalero)" << endl;
                        cin >> tipoEmp;
                        if (tipoEmp == 1) {
```

```
                        float sueldo;
                        cout << "Ingresar sueldo" << endl;
                        cin >> sueldo;
                        emp = new Comun(nom, sueldo);
                  } else {
                        int horas;
                        float vHora;
                        cout << "Ingresar horas" << endl;
                        cin >> horas;
                        cout << "Ingresar valor hora" << endl;
                        cin >> vHora;
                        emp = new Jornalero(nom, horas, vHora);
                  }
                  empleados.agregarEmpleado(emp);
                  break;
            case 2:
                  float liqTotal = 0;
                  for (int i=0; i<empleados.getCant(); i++)
                        liqTotal = liqTotal +
                         empleados.iesimo(i)- >darLiquidacion();
                  cout << "Total liquidación: " << liqTotal << endl;
      } // end switch
  } while (opcion != 0);
  return 0;
}
```

Observar:

- Cuando se ingresa un empleado se declara un puntero a Empleado, que luego se instancia en Comun o Jornalero, cada uno con sus datos específicos (intercambiabilidad). Luego se agrega el empleado a la colección, no importando de qué clase sea.
- Cuando se liquida el sueldo, se recorre toda la colección y para cada empleado se invoca a darLiquidacion. En tiempo de ejecución (despacho dinámico) se decidirá cual versión de la operación invocar (polimorfmismo), dependiendo de si el puntero apunta a un Comun o a un Jornalero. Notar que la colección debe ser de puntero a Empleado para que este mecanismo funcione. Por el contrario, si definimos un array de Empleado hay un problema de consistencia, pues una instancia de Comun no ocupa la misma cantidad de memoria que una de Jornalero.

Nota: En todos los códigos C++ presentados se asume que:

- Cada clase tiene su archivo .h con la declaración y su correspondiente .cpp con la implementación, el cual incluye al .h.
- Cuando se utiliza stream de entrada y salida (cin y cout) se debe incluir la biblioteca iostream.
- Cuando se utiliza la clase string se debe anunciar el uso del namespace std.
- Se trabaja con el flujo habitual de eventos, por sencillez de la exposición. Es decir, no se controlan los casos de borde (ingresar un empleado en una colección sin espacio suficiente) y se asume que los datos se ingresan correctamente por parte del usuario.