Práctico de Programación 4 – clase 2

Práctico 1, Ejercicio 6:

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++.

Práctico 1, Ejercicio 12:

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.