Ayudantía 0

Introducción a IIC3413-DB
y C++ Moderno

Introducción a IIC3413-DB

Instalación

- Seguir instrucciones de README.md:
  - https://github.com/PUC-IIC3141/IIC3413-DB

IIC3413-DB

- Base de datos pensada para property graphs.
- Aún en desarrollo
  - es muy probable que tenga bugs.
  - si detectan errores o no entienden algo en particular de su funcionamiento, abran un nuevo issue: https://github.com/PUC-IIC3141/IIC3413-DB/issues e intentaré arreglarlo a la brevedad.
  - por ahora solo soporta algunas funcionalidades.

Consultas soportadas

consultas tipo SELECT/MATCH/WHERE(opcional)

Ej:

```
SELECT ?n.Name, ?n.Age // Si se encuentra un n en match y no tiene Name no aparecerá
MATCH (?n:Person { BornCountry: "Chile" })
WHERE ?n.Age > 18 AND ?n.Age < 65


SELECT ?p.Name
MATCH (?p)-[:PresidentOf]->(:Country {Name: "Chile"})


SELECT * // Entrega ids internas de los nodos y edges
MATCH (?n)->(m), (?m)->(?n)
```

Aún no soportado

- Funciones de agregación
  - SUM
  - AVERAGE, MAX, MIN
  - COUNT

- Operaciones de caminos
  - Por ejemplo nos gustaría expresar la búsqueda de 2 nodos conectados por un camino de largo arbitrario, donde unos de esos edges pertenecientes al camino tiene cierto label/property

Arquitectura del sistema

- 3 módulos por ahora:
  - Base
  - Storage
  - RelationalModel

Introducción a C++ (c++17)

Principales diferencias con Python

- Compilado
  - Debería ser más rápido si se programa adecuadamente
- Fuertemente Tipado
- Sin Garbage Collector
- Manejo de memoria a bajo nivel:
  - Se puede acceder, reservar y liberar memoria a libre disposición.
  - Que se pueda hacer no significa que funcione: acceder o liberar a memoria no reservada finalizará el programa.
  - S.O. y memoria virtual protegen a otros procesos de ser afectados.
- Riesgoso: Undefined Behaviour, Unspecified Behaviour
  -

Hello World

```
/* hello_world.cpp */
#include <iostream> // to use std::cout

int main() {
    std::cout << "Hello World\n";
    return 0;
}
```

```
Para compilar:
    g++ hello_world.cpp -o hello_world

Para ejecutar:
    ./hello_world
```

Tipos de datos primarios

- Numéricos
  - int,
  - long,
  - float,
  - double,
  - long long,
  - long double

  cada uno puede tener opcionalmente el prefijo signed o unsigned (si no tiene se asume signed)
- bool
- char
- void

Tipos de datos derivados

- Arreglos
- Punteros
- Referencias
- Funciones
  - al ser un tipo de dato pueden ser pasadas como argumento

Punteros

- Todo valor usado en la ejecución de un programa debe estar almacenado en alguna parte de la memoria RAM.
- Los punteros permiten trabajar directamente con direcciones de memoria.
- Útil por ejemplo para pasar una estructura muy grande a una función, en lugar de copiar la estructura completa en el stack solo es necesario copiar la posición de memoria donde se encuentra esa estructura.

Punteros

```
// declaración de un puntero:
int* a;        // iniciado por defecto a un valor especial: nullptr
int i = 5;
int* b = &i;   // iniciado a la posición de memoria donde está i;

std::cout << b << "\n";  // imprime la posición de memoria donde apunta b
std::cout << *b << "\n"; // imprime el valor al que apunta b
                         // hacer *b se suele denominar como ‘dereferenciar’

b++;    // b ahora apunta a la siguiente posición de memoria
        // (saltándose sizeof(int) = 4 Bytes)

b+= 10  // Ahora aumentaría la posición a la que apunta en 4*10 Bytes.
        // Tratar de dereferenciar el puntero puede dar error o un valor
        // pseudo-random
```

Referencias

- Está pensada para ser un alias de una variable existente
- Al momento de ejecutar código no tienen diferencia alguna con los punteros.
- Al escribir código sí tienen diferencias:
  - Una referencia no ser inicializada como nullptr
  - Diferente sintaxis para obtener el valor al que apuntan
  - Referencia no permite aritmética de punteros

Referencias

```
// declaración de una referencia:
int a = 10;
int& r = a;

std::cout << r << "\n";  // lo mismo que imprimir a
```

Paso por valor y referencia

- Al declarar métodos que reciben parámetros tenemos la opción de recibirlos por parámetros o por referencia

```
void al_cuadrado(int n) { // paso por valor
    n = n*n;
}

int main() {
    int i = 10;
    al_cuadrado(i);
    std::cout << i << "\n"; // i no cambio su valor
}
```

Paso por valor y referencia

- Al declarar métodos que reciben parámetros tenemos la opción de recibirlos por parámetros o por referencia

```
void al_cuadrado(int& n) { // paso por referencia
    n = n*n;
}

int main() {
    int i = 10;
    al_cuadrado(i);
    std::cout << i << "\n"; // i cambio su valor
}
```

Paso por valor y referencia

- Al declarar métodos que reciben parámetros tenemos la opción de recibirlos por parámetros o por referencia

```
void al_cuadrado(int* n) { // paso por referencia (usando punteros)
    *n = (*n) * (*n);      // en este caso queda bastante peor la
                           // sintaxis
}

int main() {
    int i = 10;
    al_cuadrado(&i);
    std::cout << i << "\n"; // i cambio su valor
}
```

Arreglos

- int a[10]; // zero initialized
- int b[5] = { 1, 2, 3, 4, 5 };
- Arreglos multidimensionales solo son una ilusión:
  - int matriz3D [3][5][7]; // es equivalente a
  - int arreglo [105]; // (3 * 5 * 7 = 105)
- De hecho los arreglos son en realidad un puntero
  - int b[5] = { 1, 2, 3, 4, 5 };
  - std::cout << b[6] << "\n"; // Al igual que al usar punteros, si tienen suerte arrojará error, sino tendran un valor "pseudo-random"
  - lo que pasa es que b apunta al comienzo del arreglo, acceder a un índice es cómo dereferenciar:
    - *b == b[0]
    - *(b+6) = b[6]

Tipos de datos definidos por el usuario

- Class: base de los lenguajes orientados a objetos, permite definir atributos y métodos para crear estructuras más complejas.
- Struct: básicamente lo mismo que una clase, pero sus elementos públicos por defecto.
- Union: representa un objeto que puede tener más de un tipo, pero solo uno válido a la vez. Ocupa tanta memoria como el tipo más grande.
- Enumeration: para representar listas de constantes.
- Typedef: proporciona una forma de crear un alias que se puede utilizar en cualquier lugar, en lugar de un nombre de tipo (posiblemente largo de escribir y/o difícil de leer)

Clases

- Clases tienen atributos y métodos
- Cada atributo/método es public, protected o private
- Tienen un constructor y un destructor
  (aunque se omita su declaración siempre existen)
- Permite herencia y polimorfismo
- Generalmente declarada en 2 archivos distintos:
  - Header (.h, .hpp): declara atributos y métodos
  - Implementation (.cpp, .cc, .cxx): define los métodos que no fueron definidos en el header.

Ejemplo Clase

```
/* rectangle.h */
class Rectangle {
private:
     int alto;
     int ancho;

public:
     Rectangle(int ancho, int largo);
     ~Rectangle();

     int area();
     int get_alto(){ return alto; }
     int get_ancho(){ return ancho; }
}
```

```
/* rectangle.cc */
#include "rectangle.h"

Rectangle::Rectangle(int ancho, int largo)
  : ancho(ancho), largo(largo) { }

Rectangle::~Rectangle() = default;

int Rectangle::area() {
     return ancho*alto;
}
```

Usando clases

```
Rectangle rect1 = Rectangle(2,3); // Inicializa una instancia de la clase Rectangle
                                  // construyendo de este modo se usa el stack.
                                  // al salir del scope donde fue declarado se llamará
                                  // automáticamente al destructor

auto rect2 = Rectangle(2,3);      // hace lo mismo que la línea anterior.
                                  // Es una buena práctica usar auto cuando sea posible

auto rect3 = new Rectangle(4,5);  // Crea un objeto en el heap y devuelve un puntero
                                  // a ese objeto.

delete rect3;                     // Usando punteros se debe destruir explícitamente el
                                  // objeto usando delete para recuperar la memoria.

rect2.area();                     // para acceder a los atributos/métodos se usa el '.'
rect3->area();                    // si es un puntero se usa ->
```

Control de flujo

- Similar a otros lenguajes
  - if, if/else, if/else if…
  - while, do/while
  - for, range-based for
  - switch/case
  - goto

Estructuras de datos típicas de la STL: std::string

- En C se solía usar char* para representar cadenas de texto.
- En C++ se aconseja siempre usar std::string a menos que se trabaje con librerías heredadas que reciben un char*.
- Se puede convertir de std::string a char* usando el método c_str()
  - `std::string s = "Hola Mundo"; // inicializando un string`
  - `auto s = "Hola Mundo";        // OJO! en este caso no se puede usar auto`
    `// porque se infiere el tipo const char*.`
- Permite concatenación mediante la suma.
- A diferencia de otros lenguajes (Java, C#) la comparación(==) funciona como se esperaría

  (Se está comparando por valor y no por referencia)

Estructuras de datos típicas de la STL: std::vector

- En c++ los arreglos son de tamaño fijo, para listas tamaño variable se recomienda usar std::vector.
- Al declarar un vector se le especifica el tipo de dato que contendrá:

```
std::vector<int> v1;                 // inicialización de vector vacío
std::vector<int> v2 = {1, 2, 3, 4}; // inicialización con elementos
v2.push_back(5);                     // añade un elemento al final de v2
v2.size();                           // entrega el tamaño de v2
v2[i];                               // accede a la posición i del vector
```

Recorriendo vector

- #include <vector>
- Se puede usar un for típico o range-based for loop:

```
std::vector v1 = { 10, 20, 30, 40};
for (int i = 0; i < v1.size(); i++) {
    std::cout << v1[i] << "\n";
}
```

```
std::vector v1 = { 10, 20, 30, 40};
for (auto n : v1) { // n es copia
    std::cout << n << "\n";
}
```

```
std::vector v1 = { 10, 20, 30, 40};
for (auto &n : v1) { // n es referencia
    std::cout << n << "\n";
}
```

Estructuras de datos típicas de la STL: std::set

- #include <set>
- Útil para cuando tienes una lista varios con elementos y necesitas hacer búsquedas en tiempo logarítmico (buscar en std::vector sería lineal).

```
std::set<int> s0;                 // inicialización de set vacío
std::set<int> s = {1, 2, 3, 4};   // inicialización con elementos
s.insert(5);                      // añade un elemento a s
s.size();                         // entrega el tamaño de s

auto search = s.find(10);         // busca el número 10 en s
if (search != s.end()) {
    std::cout << *search << "\n"; // search no es un puntero, pero se sobrecarga el
}                                 // operador ‘*’ para entregar valor encontrado
else {
    std::cout << "Not found\n";
}
```

Estructuras de datos típicas de la STL: std::map

- #include <map>
- Útil para guardar key/values. Permite hacer búsquedas por key en tiempo logarítmico.

```
std::map<std::string, int> m;  // declara un map con llave string y valor int vacío
m.insert({"uno", 1});          // inserta un par key/value
m.size();                      // devuelve cuantos elementos hay en m

auto search = m.find("cuatro");
if (search != m.end()) {
    std::cout << "key:" << search->first << "\n";     // search no es puntero
    std::cout << "value:" << search->second << "\n";  // sobrecarga de operador ->
}
else {
    std::cout << "Not found\n";
}
```

Recorriendo map

```
// Solo funciona si se compila usando c++17 (agregar parámetro g++: -std=c++1z)
for (auto&&[key, value] : m) {
    std::cout << key << ", " << value << "\n";
}

// Sin c++17
for (auto it = m.begin(); it != m.end(); ++it) {
    std::cout << it->first << ", " << it->second << "\n";
}
```

Smart Pointers

- Confiar en que se usará correctamente new/delete puede ser peligroso, muchas veces no es así, sobretodo con flujos complejos.
- En c++ moderno se recomienda evitar el uso de new y delete y reemplazar su uso por smart pointers.
- Sobrecargan los operadores ‘*’ y ‘->’ por lo que se pueden usar como si fueran punteros.
- En el momento en que se dejen de ocupar, se destruirán automáticamente.

Smart pointers

Existen 3 tipos:

- std::unique_ptr<Type>
  - Indica pertenencia, sólo puede ser apuntado de un lugar a la vez.
  - Destrucción automática al salir de scope.
  - Si voy a pasar el puntero debo “transferir esa pertenencia” ocupando std::move().
  - Al transferir la pertenencia a otro, no puedo volver a ocupar el puntero.

- std::shared_ptr<Type>
  - No hay pertenencia, puede ser apuntado de varios lugares distintos a la vez.
  - Posee contador de referencias, cuando no hay más referencias apuntando se destruye.

- std::weak_ptr<Type>
  - Similar a std::shared_ptr, pero no son considerados en la cuenta de referencias.
  - No existe certeza de que el objeto apuntado existe.
  - Necesario para solucionar dependencias cíclicas (ej: lista doblemente ligada)

Ejemplo smart pointers

```
#include "rectangle.h"

#include <iostream>
#include <memory> // to use smart pointers

using namespace std;

void print(unique_ptr<Rectangle> rect) {
    cout << rect->alto << ", " << rect->ancho << "\n";
}

int main() {
    auto ptr = make_unique<Rectangle>(10, 20);   // como parámetro lo mismo que se pasaría al constructor
    if (ptr == nullptr) {                        // comprobamos que no es nulo
        cout << "ptr == nullptr\n";
    }
    // print(ptr);                               // daría error de compilación
    print(move(ptr));                            // transferimos el ownership de ptr que tenemos
    if (ptr == nullptr) {                        // comprobamos que es nulo luego de haber usado el move()
        cout << "ptr == nullptr\n";
    }
}
```

Recursos Adicionales

- http://www.cplusplus.com/doc/tutorial/ (tutorial)
- https://www.studytonight.com/cpp/ (tutorial más completo)
- https://en.cppreference.com/book/intro/smart_pointers (smart pointers)
- http://www.cplusplus.com/reference/stl/ (documentación de la Standard Library)
- https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md (recomendado para cuando crean que crean que manejan bien c++)
- Effective Modern C++ (Scott Meyers). También recomendado para cuando ya se manejen en c++. No lo descarguen de http://gen.lib.rus.ec/, es ilegal.