UNIVERSIDAD NACIONAL DE SAN AGUSTÍN

ASIGNATURA: PROGRAMACIÓN DE SISTEMAS

PRÁCTICA LABORATORIO N°04

OBJETIVO

- La presente práctica de laboratorio tiene como objetivo el uso de punteros en C++.

TEMA

- Introducción.
- Variables de Apuntador.
- Direcciones Números.
- Operadores * y &.
- Operador New.

MARCO TEÓRICO

INTRODUCCIÓN:

Un apuntador es la dirección en memoria de una variable. Recuerde que la memoria de una computadora se divide en posiciones de memoria numeradas (llamadas bytes), y que las variables se implementan como una sucesión de posiciones de memoria adyacentes. Recuerde también que en ocasiones el sistema C++ utiliza estas direcciones de memoria como nombres de las variables. Si una variable se implementa como tres posiciones de memoria, la dirección de la primera de esas posiciones a veces se usa como nombre para esa variable. Por ejemplo, cuando la variable se usa como argumento de llamada por referencia, es esta dirección, no el nombre del identificador de la variable, lo que se pasa a la función invocadora.

Una dirección que se utiliza para nombrar una variable de este modo (dando la dirección de memoria donde la variable inicia) se llama apuntador porque podemos pensar que la dirección "apunta" a la variable. La dirección "apunta" a la variable porque la identifica diciendo dónde está, en lugar de decir qué nombre tiene. Digamos que una variable está en la posición número 1007; podemos referirnos a ella diciendo "es la variable que está allá, en la posición 1007".

VARIABLES DE APUNTADOR:

Un apuntador se puede guardar en una variable. Sin embargo, aunque un apuntador es una dirección de memoria y una dirección de memoria es un número, no podemos guardar un apuntador en una variable de tipo int o double. Una variable que va a

contener un apuntador se debe declarar como de tipo apuntador. Por ejemplo, lo que sigue declara p como una variable de apuntador que puede contener un apuntador que apunta a una variable de tipo double:

double *p;

La variable p puede contener apuntadores a variables de tipo double, pero normalmente no puede contener un apuntador a una variable de algún otro tipo, como int o char. Cada tipo de variable requiere un tipo de apuntador distinto.

En general, si queremos declarar una variable que pueda contener apuntadores a otras variables de un tipo específico, declaramos las variables de apuntador igual que declaramos una variable ordinaria de ese tipo, pero colocamos un asterisco antes del nombre de la variable. Por ejemplo, lo siguiente declara las variables p1 y p2 de modo que puedan contener apuntadores a variables de tipo int; también se declaran dos variables ordinarias v1 y v2 de tipo int:

int *p1, *p2, v1, v2;

Es necesario que haya un asterisco antes de cada una de las variables de apuntador. Si omitimos el segundo asterisco en la declaración anterior, p2 no será una variable de apuntador; será una variable ordinaria de tipo int. El asterisco es el mismo símbolo que hemos estado usando para la multiplicación, pero en este contexto tiene un significado totalmente distinto.

Declaraciones de variables de apuntador

Una variable que puede contener apuntadores a otras variables de tipo

Nombre_de_Tipo se declara del mismo modo que una variable de tipo Nombre_de_Tipo, excepto que se antepone un asterisco al nombre de la variable.

Sintaxis:

Nombre_de_Tipo *Nombre_de_Variable1, *Nombre_de_Variable2, ...;

Ejemplo:

double * apuntador1, * apuntador2;

Cuando tratamos apuntadores y variables de apuntador, normalmente hablamos de apuntar en lugar de hablar de direcciones. Cuando una variable de apuntador, como p1, contiene la dirección de una variable, como v1, decimos que dicha variable apunta a la variable v1 o es un apuntador a la variable v1.

- **DIRECCIONES NÚMEROS:**

Un apuntador es una dirección, una dirección es un entero, pero un apuntador no es un entero. Esto no es absurdo, ¡es abstracción! C++ insiste en que usemos un apuntador como una dirección y no como un número. Un apuntador no es un valor de

tipo int ni de ningún otro tipo numérico. Normalmente no podemos guardar un apuntador en una variable de tipo int. Si lo intentamos, casi cualquier compilador de C++ generará un mensaje de error o de advertencia. Además, no podemos efectuar las operaciones aritméticas normales con apuntadores.

(Podemos realizar una especie de suma y una especie de resta con apuntadores, pero no son la suma y resta normales con enteros.)

Las variables de apuntador, como p1 y p2 en nuestro ejemplo de declaración, pueden contener apuntadores a variables como v1 y v2. Podemos usar el operador & para determinar la dirección de una variable, y luego podemos asignar esa dirección a una variable de apuntador. Por ejemplo, lo siguiente asigna a la variable p1 un apuntador que apunta a la variable v1:

```
p1 = &v1;
```

Ahora tenemos dos formas de referirnos a v1: podemos llamarla v1 o "la variable a la que p1 apunta". En C++ la forma de decir "la variable a la que p1 apunta" es *p1. Éste es el mismo asterisco que usamos al declarar p1, pero ahora tiene otro significado. Cuando el asterisco se usa de esta manera se le conoce como operador de desreferenciación, y decimos que la variable de apuntador está desreferenciada.

Si armamos todas estas piezas podemos obtener algunos resultados sorprendentes. Consideremos el siguiente código:

```
v1 = 0;

p1 = &v1;

*p1 = 42;

cout << v1 << endl;

cout << *p1 << endl;
```

Este código despliega lo siguiente en la pantalla:

```
42

42
```

En tanto p1 contenga un apuntador que apunte a v1, entonces v1 y *p1 se referirán a la misma variable. Así pues, si asignamos 42 a *p1, también estamos asignando 42 a v1.

El símbolo & que usamos para obtener la dirección de una variable es el mismo símbolo que usamos en una declaración de función para especificar un parámetro de llamada por referencia. Esto no es una coincidencia. Recuerde que un argumento de llamada por referencia se implementa dando la dirección del argumento a la función invocadora. Así pues, estos dos usos del símbolo & son básicamente el mismo. Sin

embargo, hay ciertas diferencias pequeñas en la forma de uso, así que los consideraremos dos usos distintos (aunque íntimamente relacionados) del símbolo &.

❖ OPERADORES * y &:

El operador * antepuesto a una variable de apuntador produce la variable a la que apunta. Cuando se usa de esta forma, el operador * se llama operador de desreferenciación.

El operador & antepuesto a una variable ordinaria produce la dirección de esa variable; es decir, produce un apuntador que apunta a la variable. El operador & se llama simplemente operador de dirección de.

Por ejemplo, consideremos las declaraciones

```
double *p, v;
```

Lo siguiente establece a p de modo que apunte a la variable v:

```
p= &v;
```

*p produce la variable a la que apunta p, así que después de la asignación anterior *p y v se refieren a la misma variable. Por ejemplo, lo siguiente establece el valor de v a 9.99, aunque nunca se usa explícitamente el nombre v:

```
*p = 9.99;
```

Podemos asignar el valor de una variable de apuntador a otra variable de apuntador. Esto copia una dirección de una variable de apuntador a otra. Por ejemplo, si p1 todavía está apuntando a v1, lo que siguiente establecerá el valor de p2 de modo que también apunte a v1:

```
p2 = p1;
```

Siempre que no hayamos modificado el valor de v1, lo siguiente también desplegará 42 en la pantalla:

```
cout << *p2;
```

Asegúrese de no confundir

```
p1 = p2;
```

con

```
*p1 = *p2;
```

Cuando añadimos el asterisco, no estamos tratando con los de apuntadores p1 y p2, sino con las variables a las que estos apuntan.

Puesto que podemos usar un de apuntador para referirnos a una variable, el programa puede manipular variables, aunque éstas carezcan de identificadores que las nombren. Podemos usar el operador new para crear variables sin identificadores que sean sus nombres. Hacemos referencia a estas variables sin nombre mediante apuntadores. Por ejemplo, lo siguiente crea una nueva variable de tipo int y asigna a la variable de apuntador p1 la dirección de esta nueva variable (es decir, p1 apunta a esta nueva variable sin nombre):

```
p1 = new int;
```

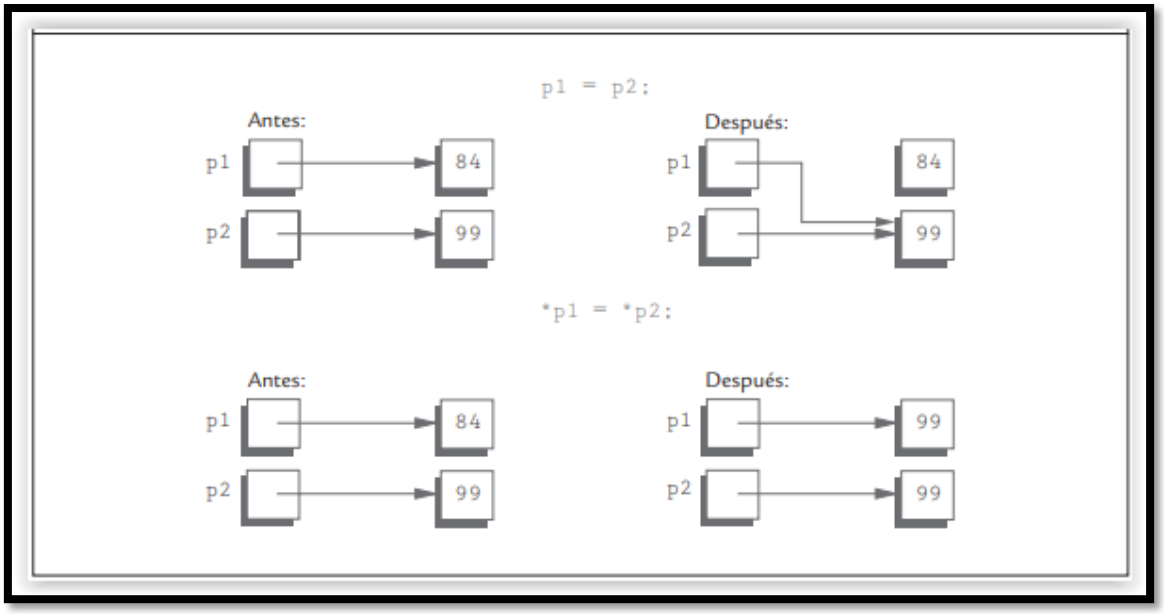
Podemos usar *p1 para referirnos a esta nueva variable (es decir, la variable a la que p1 apunta). Podemos hacer con esta variable sin nombre lo mismo que con cualquier otra variable de tipo int. Por ejemplo, lo que sigue lee un valor de tipo int del teclado y lo coloca en la variable sin nombre, suma 7 al valor y luego despliega el nuevo valor en la pantalla:

```
cin >> *p1;

*p1 = *p1 + 7;

cout << *p1;
```

El operador new produce una nueva variable sin nombre y devuelve un apuntador que apunta a esta nueva variable.



❖ OPERADOR NEW:

El operador new crea una nueva variable dinámica del tipo que se especifica y devuelve un apuntador que apunta a esta nueva variable. Por ejemplo, lo que sigue

crea una variable dinámica nueva del tipo MiTipo y deja a la variable del apuntador p apuntando a esa nueva variable.

```
MiTipo *p;

p = new MiTipo;
```

Si el tipo es una clase que tiene un constructor, se invoca el constructor predeterminado para la variable dinámica recién creada. Se pueden especificar inicializadores que hagan que se invoquen otros constructores:

```
int *n;

n = new int(17); //inicializa n con 17

MiTipo *apuntMt;

apuntMt = new MiTipo(32.0, 17); // invoca MiTipo(double, int);
```

El estándar de C++ estipula que, si no hay suficiente memoria desocupada para crear la nueva variable, la acción predeterminada del operador new es terminar el programa.

- **EJEMPLO BASICO DE APUNTADORES:**

```
//Programa para demostrar apuntadores y variables dinámicas.

#include <iostream>
using namespace std;
int main()
{
int *p1, *p2;
p1 = new int;
*p1 = 42;
p2 = p1;
cout << "*p1 == " << *p1 << endl;
cout << "*p2 == " << *p2 << endl;
*p2 = 53;
cout << "*p1 == " << *p1 << endl;
cout << "*p2 == " << *p2 << endl;
p1 = new int;
*p1 = 88;
cout << "*p1 == " << *p1 << endl;
cout << "*p2 == " << *p2 << endl;
cout << "Este es un ejemplo de uso de apuntadores!\n";
return 0;
}
```

❖ PROYECTO DE PRÁCTICAS DEL SEMESTRE 2020-A

Para el Proyecto de Prácticas del Semestre 2020-A en grupos de cinco personas deberán escoger un líder de grupo y seleccionar un Sistema a Desarrollar en el siguiente Semestre. Se sugiere escoger uno de los siguientes el cual no debe repetirse por grupo de prácticas.

1. Sistema de pedidos y pago de cuentas de restaurante.
2. Sistema de préstamo de libros de una biblioteca.
3. Sistema de ventas en farmacia.
4. Sistema de pedidos de delivery.
5. Sistemas de reserva de citas de consultorio médico.
6. Sistemas de reserva de películas de Cine.
7. Sistema de reserva de pasajes de avión.
8. Sistema de reserva de pasajes terrestre.
9. Sistema de registro de huéspedes de hotel.
10. Sistema de pedidos de taxis.

Una vez seleccionado el Sistema, deberá realizar un modelamiento de las clases que van a intervenir en el mismo, identificando sus atributos (público, privado, protegido), métodos u operaciones (público, privado, protegido) y las relaciones que deben existir entre ellas con su respectiva cardinalidad argumentando en todos ellos la razón de su uso y posterior implementación.

Deberá implementar el Sistema utilizando la POO (Clases, Estructura, Funciones, etc.). Debe comentar el código por bloques, indicando su funcionalidad y separando la lógica en archivos cabecera (,h), los métodos u operaciones (.cpp) y el programa principal (.cpp).

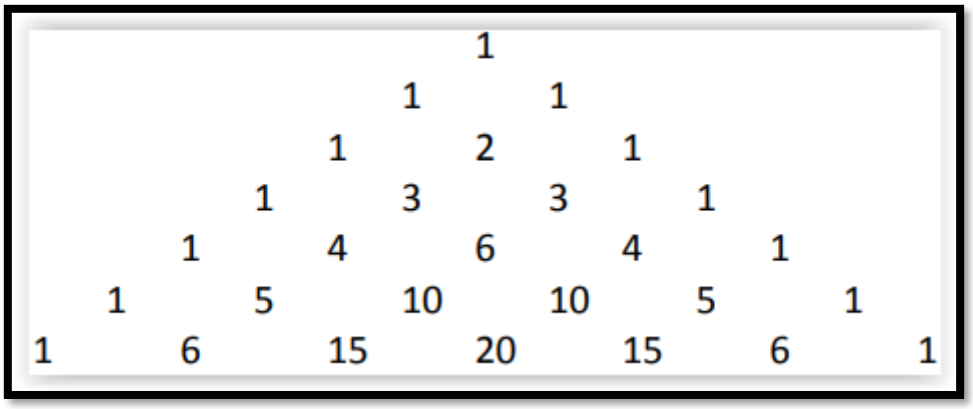
❖ ACTIVIDADES

- Consideraciones: Puede utilizar como IDE: Zinjal, Visual Studio 2019, entre otros. Ello debe ser especificado en su informe.

Nota: Para los siguientes ejercicios, programar de forma ordenada de tal manera que las clases y/o estructuras queden definidas en archivos cabecera (.h), los métodos u operaciones en un archivo .cpp y finalmente el programa principal en otro archive .cpp. Asimismo, debe utilizar la experiencia de la práctica anterior y debe estar visible en el código.

- **EJERCICIO 1.**

Escribir un programa que permita visualizar el triángulo de pascal. En el triángulo de pascal cada número es la suma de los dos números situados encima de él. Este problema se debe resolver utilizando un arreglo de una sola dimensión.



Restricciones: Uso arreglos, funciones, así como las estructuras de control selectivas y repetitivas.

- **EJERCICIO 2.**

  El juego del ahorcado se juega con dos personas (o una persona y una computadora). Un jugador selecciona una palabra y el otro jugador trata de adivinar la palabra adivinando las letras individuales. Diseñar un programa para jugar al ahorcado.

  Restricciones: Uso arreglos, funciones, así como las estructuras de control selectivas y repetitivas.

- **EJERCICIO 3.**

  Se dice que una matriz tiene un punto de silla si alguna posición de la matriz es el menor valor de su fila, y a la vez el mayor de su columna. Escribir un programa que tenga como entrada una matriz de números reales, y calcular la posición de un punto de silla (si es que existe).

  Restricciones: Uso arreglos, funciones, así como las estructuras de control selectivas y repetitivas.

- **EJERCICIO 4.**

  Se desea escribir un programa que permita manejar la información de habitantes de un complejo habitacional. El mismo posee 7 torres; a su vez cada torre posee 20 pisos y cada piso 6 departamentos.

  Se desea saber:
  a- Cantidad total de habitantes del complejo
  b- Cantidad promedio de habitantes por piso de cada torre
  c- Cantidad promedio de habitantes por torre

  Restricciones: Uso arreglos, funciones, así como las estructuras de control selectivas y repetitivas.

- **EJERCICIO 5.**

  Implemente la multiplicación de matrices utilizando punteros. Restricciones: Uso de punteros, funciones, así como las estructuras de control selectivas y repetitivas.

- **EJERCICIO 6.**

  Sea A una matriz de tamaño NxN, implemente un programa que dado un menú de opciones resuelva:

  • La transpuesta de A (At).
  • Si A es simétrica o antisimétrica.
  • Si A es una matriz triangular superior o triangular inferior.

  Restricciones: Uso de punteros, funciones, así como las estructuras de control selectivas y repetitivas.

❖ BIBLIOGRAFIA

- Deitel, Paul J., Deitel, Harvey M., “Cómo Programar en C++”, 6ta Edición, Ed. Pearson Educación, México 2009.
- Stroustrup, Bjarne, “El Lenguaje de Programación C++”, 3ra Edición, Adisson Pearson Educación S.A., Madrid 2002.
- Eckel, Bruce, “Thinking in C++”, 2da Edición, Prentice Hall, 2000.