**Instituto Superior de Formación Técnica Nº 151 logo151-trans.gif  
Carrera: Analista de Sistemas  
1 Año. Algoritmos y Estructuras de Datos I**

|  |  |
| --- | --- |
| **Trabajo Práctico Nº 9** | **Unidad 9** |
| **Modalidad:** Semi-Presencial | **Estratégica Didáctica:** Trabajo Individual |
| **Metodología de Desarrollo:** acordar | **Metodología de Corrección:** acordar docente |
| **Carácter de Trabajo:** Obligatorio – con Nota | **Fecha Entrega:** A confirmar por el Docente. |

**PUNTEROS**

**Marco Teórico:**

1. Explicar con un ejemplo las Direcciones de memoria (tomar un Integer para el Ejemplo)
2. ¿Qué entiende por punteros, que representan?
3. ¿Qué utilidades tienen los Operadores Monarios “\*” y “&”?
4. ¿Qué direcciones válidas (valores) tengo para los Punteros?
5. ¿Cuál es el Problema de no inicializar un Puntero?
6. ¿Por qué decimos que NULL es un valor seguro?
7. ¿Qué entiende por Copia y comparación de punteros?
8. ¿Qué entendemos por Tipos puntero (piense en tipos de datos)
9. Dar ejemplos de Punteros a estructuras
10. Diferencie con un ejemplo Punteros a constantes y punteros constantes
11. ¿Qué relación hay entre Punteros y paso de parámetros? (relacione referencia y valor)
12. Explicar la relación entre Punteros y Arrays
13. Explicar el “Esquema de Memoria y datos del programa”
14. ¿Qué entiende por Memoria dinámica? ¿Qué uso tiene?

**Marco Practico:**

**Tener en Cuenta:**  
  
. Modularizar el Programa.  
. Proteger contra Inclusiones Múltiples.  
. Aplicar Espacios de Nombres.  
. Aplica apropiadamente los conceptos de abstracción, encapsulación y ocultamiento de información.  
. Realiza una apropiada distribución de responsabilidades entre las entidades del espacio de la solución.   
. Desarrolla para reusar.  
. Reusa apropiadamente las entidades desarrolladas en el espacio curricular.  
. Demuestra un uso apropiado de la sintaxis y semántica del lenguaje de programación C++.

**Desarrollar un Programa** que:

**1.** Sin ejecutarlo, ¿qué mostraría el siguiente código?

int x = 5, y = 12, z;

int \*p1, \*p2, \*p3;

p1 = &x;

p2 = &y;

z = \*p1 \* \*p2;

p3 = &z;

(\*p3)++;

p1 = p3;

cout << \*p1 << " " << \*p2 << " " << \*p3;

***Rta.:*** 61 12 61

**2.** ¿Qué problema hay en el siguiente código?

int dato = 5;

int \*p1, p2;

p1 = &dato;

p2 = p1;

cout << \*p2;

***Rta.:*** Que p1 es un puntero a entero, es decir que guarda la dirección de memoria (en hexadecimal) de un entero, mientras que p2 es un entero. Luego, la asignación p2 = p1 es inconsistente. (no puedo asignarle a un entero una dirección de memoria)

**3.** ¿Qué problema hay en el siguiente código?

double d = 5.4, e = 1.2, f = 0.9;

double \*p1, \*p2, \*p3;

p1 = &d;

(\*p1) = (\*p1) + 3;

p2 = &e;

(\*p3) = (\*p1) + (\*p2);

cout << \*p1 << " " << \*p2 << " " << \*p3;

***Rta.:*** El problema está en la línea (\*p3) = (\*p1) + (\*p2); puesto que p3 no fue inicializado nunca, es decir: no apunta a ningún lugar, razón por la cual mal podría guardar la información contenida en p1 y p2.

**4.** ¿Cómo declararías un puntero constante p para apuntar a una constante

entera? (Repasa las diapositivas 877‐878 de la presentación del tema.)

const int const p;

**5.** Dado el siguiente tipo:

typedef struct {

string nombre;

double sueldo;

int edad;

} tRegistro;

Y el siguiente subprograma:

void func(tRegistro &reg, double &irpf, int &edad) {

const double TIPO = 0.18;

reg.edad++;

irpf = reg.sueldo \* TIPO;

edad = reg.edad;

}

Reescribe el subprograma para que implemente el paso de parámetros por variable con punteros, en lugar de las referencias que usa ahora (modifica el prototipo y la implementación convenientemente).

***Rta.:***

void func(tRegistro\* reg, double\* irpf, int\* edad)

{

const double TIPO = 0.18;

reg->edad++;

irpf = (reg->sueldo) \* TIPO;

edad = (reg->edad);

}

Lic. Oemig José Luis.

**PUNTEROS**

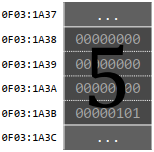
**Marco Teórico:**

1. Explicar con un ejemplo las Direcciones de memoria (tomar un Integer para el Ejemplo)

Durante la ejecución de un programa, todo dato se guarda en memoria. La unidad mínima de almacenamiento es el bit (que sólo puede almacenar un 0 o un 1) aunque la unidad de almacenamiento más común es el byte (8 bits). Cada espacio de 1 byte está identificado con una dirección de memoria y, de acuerdo al tipo de dato a almacenar, se destinan bloques consecutivos de 1 byte.

Por ejemplo, para almacenar un dato *int* (entero) como

int i = 5;



El dato ‘*i*’ se accede a partir de su dirección base (0F03:1A38), dirección de la primera celda (byte) de memoria utilizada por el dato. Como el tipo de dato int utiliza 4 bytes, se almacena:

00000000 00000000 00000000 00000101 → 5

1. ¿Qué entiende por punteros, que representan?   
   Un puntero es una variable que contiene una dirección de memoria. Sirve para acceder a otros datos.
2. ¿Qué utilidades tienen los Operadores Monarios “\*” y “&”?

Ambos operadores tienen dos utilidades:

Respecto del operador “**\***”:

* Cuando se usa en la declaración (Por ejemplo: int\* ptr), crea una variable de tipo puntero (que apunta a un entero en este caso)
* Cuando no se usa en la declaración, actúa como sinónimo de “contenido de”, retornando el valor de la variable localizada en la dirección especificada por su operando. Se lo llama también “operador indirección” u “operador desreferencia”.

Respecto del operador “**&**”:

* Cuando se usa en una declaración (Por ejemplo: string& meal) crea una referencia a la variable que le asignemos a meal.
* Cuando no se usa en una declaración, actúa como un operador “dirección de”, es decir que obtiene la dirección de memoria de su operando. Se lo llama también “operador de referencia”.

1. ¿Qué direcciones válidas (valores) tengo para los Punteros?

Las variables de tipo puntero no se inicializan automáticamente. Por ende, si la declaramos pero no la inicializamos, contiene una dirección no válida (puede apuntar a cualquier cosa).

Hay tres maneras de hacer que un puntero apunte a una dirección válida:

* Asignando la dirección de otro dato (mediante el operador &)
* Asignando otro puntero (mismo tipo base) que ya sea válido
* Asignando el valor NULL (puntero nulo, no apunta a nada)

1. ¿Cuál es el Problema de no inicializar un Puntero?

El problema es que puede apuntar a cualquier lado.

1. ¿Por qué decimos que NULL es un valor seguro?  
   Porque es más seguro apuntar a “la nada” que apuntar a una dirección que pudiera contener información importante (corriendo el riesgo de cambiarla accidentalmente)
2. ¿Qué entiende por Copia y comparación de punteros?

Al copiar un puntero en otro, ambos apuntarán al mismo dato. Ejemplo:

*int x = 5;*

*int \*punt1 = NULL;* // punt1 no apunta a nada

*int \*punt2 = &x;* // punt2 apunta a la variable x

*punt1 = punt2;* // ambos apuntan a la variable x

*\*punt1 = 8;*

Al dato x ahora se puede acceder de tres formas:

*x \*punt1 \*punt2*

1. ¿Qué entendemos por Tipos puntero? (piense en tipos de datos)

Declaramos tipos para los punteros con distintos tipos base:

*typedef int \*tIntPtr;* // Tipo Puntero a Entero: es el tipo de dato al q apunta *tIntPtr*

*typedef char \*tCharPtr;* // Tipo Puntero a char

*typedef double \*tDoublePtr;* // Tipo Puntero a double

1. Dar ejemplos de Punteros a estructuras

*typedef struct {*

*int codigo;*

*string nombre;*

*double sueldo;*

*} tRegistro;*

*tRegistro registro;*

*typedef tRegistro \*tRegistroPtr;* // *tRegistroPtr* es un puntero a tRegistro

1. Diferencie con un ejemplo Punteros a constantes y punteros constantes

C++ permite definir un puntero a una constante, de forma que el valor al que el puntero apunta no pueda ser cambiado, pero el puntero en sí puede cambiar para apuntar a otra variable o constante:

*const int l = 2;*

*const int \*p= &l;*

*p* es un puntero a un entero, declarado como constante. El entero al que apunta *p* no debe ser cambiado. El puntero *p* en sí puede modificarse.

Así, *\*p = 3* no es válido, pero *p++* sí lo es.

--------------------------------------------------------------------------------------------------------

Además, se puede declarar un puntero constante, que no puede ser cambiado:

En la declaración:

*int \*const p;*

*p* es un puntero constante que no puede ser cambiado. El carácter al que apunta sin embargo sí puede modificarse.

Por último :

*const int m = 3;*

*const int \*const p = &m;*

indica que ni el puntero *p* ni el carácter al que apunta pueden ser modificados.

Estas construcciones pueden utilizarse para mejorar la calidad del código. Si un puntero o una variable no serán nunca cambiados deben declararse como constantes. El compilador indicará si inadvertidamente se han modificado estos valores.

1. ¿Qué relación hay entre Punteros y paso de parámetros? (relacione referencia y valor)

En el lenguaje C no hay mecanismo de paso por referencia (&)

Sólo se pueden pasar parámetros por valor

Para simular el paso por referencia, se utilizan punteros:

***void incrementa(int \*punt)*** *{*

*(\*punt)++;*

*}*

Para llamar a la función:

*int entero = 5;*

***incrementa(&entero);***

*cout << entero << endl;*

Mostrará **6** en la consola

En definitiva, punt funciona como un puntero constante: El argumento (el puntero) no cambia pero aquello a lo que apunta (el entero) sí puede cambiar.

1. Explicar la relación entre Punteros y Arrays

Una variable array es un puntero al primer elemento del array. De hecho el nombre del array es un puntero constante: siempre apunta al primer elemento.

Así, si tenemos:

*int dias[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};*

Entonces:

*cout << \*dias << endl;*

Muestra 31 en la consola, el primer elemento del array

1. Explicar el “Esquema de Memoria y datos del programa”

El sistema operativo distingue varias regiones en la memoria:

* La pila o stack, que es donde se almacenan los datos locales.
* El montón o Heap, que es donde se almacenan los datos dinámicos.
* La Memoria Principal, donde se guardan los datos globales, el código del programa y el Sistema Operativo.

1. ¿Qué entiende por Memoria dinámica? ¿Qué uso tiene?

Es un sector de la memoria donde se guardan datos que se crean y se destruyen durante la ejecución del programa, a medida que se necesita. Para gestionarla existe lo que se denomina un Sistema Gestor de memoria dinámica (SGMD). Cuando se necesita memoria para una variable se solicita El SGMD reserva espacio y devuelve la dirección base.

Cuando ya no se necesita más la variable, se destruye.

Se libera la memoria y el SGMD cuenta de nuevo con ella.