UNIVERSIDAD ADOLFO IBÁÑEZ
FACULTAD DE INGENIERÍA Y CIENCIAS

Ayudantía 2 Estructura de datos

Profesores: Sebastián Sáez, Diego Ramos

Ayudantes: Diego Duhalde, Benjamín Wiedmaier, Fernando Zamora

Preparación para Prueba 1

1. Lenguaje C

   1.1. ¿Qué es el operador `sizeof`? ¿Cómo se utiliza?

   1.2. ¿Qué es la aritmética de punteros? Proporcione un ejemplo.

   1.3. Explique la diferencia entre pasar un argumento por valor y pasar un argumento por referencia.

   1.4. Explique la diferencia entre una variable local y una variable global.

   1.5. ¿Cual es la diferencia entre & y * en C?

   1.6. Indique la diferencia central entre la memoria estática (stack) y la memoria dinámica (heap).

2. Punteros

   2.1. ¿Qué es un puntero? ¿Cuál es la diferencia entre un puntero y una variable normal?

   2.2. ¿Qué es un puntero nulo? ¿Para qué se utiliza?

   2.3. ¿Qué es la aritmética de punteros? Proporcione un ejemplo.

   2.4. Suponga que p es un puntero de tipo float que almacena la dirección $0x845b342c0$. ¿A qué dirección corresponde el puntero $p + 3$?

3. Complejidad temporal y espacial

   3.1. Ordene las complejidades en notación Θ de menor a mayor:

i. n^2	v. $\log n$	ix. $n^2 + n^3$
ii. n^3	vi. $n \log n$	x. $n^3 + n^2$
iii. n	vii. $n^2 \log n$	xi. $n^3 + n$
iv. 1	viii. $n^2 + n$	xii. $n^3 + n^2 + n$

   3.2. ¿Cuál es la complejidad temporal de la siguiente función? Justifique su respuesta.

```
int power(int x, int n) {
    if (n == 0)
        return -1;
    int temp = power(x, n/2);
    if (n % 2 == 0)
        return temp * temp;
    else
        return x * temp * temp;
}
```

```
void invertir(int A[], int n) {
    int B[n];
    for (int i = 0; i < n; i++) {
        B[i] = A[i];
    }

    for (int i = 0; i < n; i++) {
        A[i] = B[n-1-i];
    }
}

void bubbleSort(int arr[], int n) {
    int temp;
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-1-i; j++) {
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

3.3. ¿Cuál es la complejidad temporal y espacial de la siguiente función? Justifique su respuesta.

```
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

int LongestIncreasingSubsequence(int arr[], int n) {
    int lis[n];
    for (int i = 0; i < n; i++)
        lis[i] = 1;

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;
        }
    }

    int max = 0;
    for (int i = 0; i < n; i++) {
        if (lis[i] > max)
            max = lis[i];
    }

    return max;
}
```

4. Recursión

4.1. Escribe una función recursiva en C que calcule el n-ésimo término de la secuencia de Fibonacci. La secuencia de Fibonacci se define de la siguiente manera:

$$F(0) = 0, \quad F(1) = 1$$

$$F(n) = F(n-1) + F(n-2), \quad \text{para } n \geq 2$$

El programa debe solicitar al usuario un número entero n y mostrar el n-ésimo término de la serie.

4.2. Ejecute la función anterior con argumento n igual a 6, indicando todas las llamadas recursivas a la función Fibonacci y que retorna cada llamada.

RESPUESTAS

1. Lenguaje C

   1.1. El operador `sizeof` es un operador en tiempo de compilación que se utilza para obtener el tamaño (en bytes) de una variable o tipo de dato. Un ejemplo de uso podría ser el siguiente:

   ```
   int x;
   printf("El tamaño de int es: %zu bytes\n", sizeof(int));
   printf("El tamaño de x es: %zu bytes\n", sizeof(x));
   ```

   En este ejemplo, `sizeof(int)` devuelve el tamaño en bytes del tipo `int` y `sizeof(x)` devuelve el tamaño en bytes de la variable `x`.

   1.2. La aritmética de punteros consiste en realizar operaciones (como suma, resta o diferencias) sobre punteros. Estas operaciones se ajustan automáticamente al tamaño del tipo de dato al que apunta el puntero. Un ejemplo podría ser:

   ```
   int arr[5] = {10, 20, 30, 40, 50};
   int *p = arr;  // p apunta al primer elemento de arr
   printf("Valor en *p: %d\n", *p);   // Imprime 10
   printf("Valor en *(p+1): %d\n", *(p+1));  // Imprime 20
   ```

   Aquí, al sumar 1 a `p`, se mueve la dirección de memoria al siguiente elemento del arreglo, teniendo en cuenta el tamaño de un entero.

   1.3. En el paso por valor, se pasa una copia del valor de la variable al argumento de la función. Esto significa que cualquier cambio realizado en el argumento dentro de la función no afecta la variable original. Por ejemplo:

   ```
   void incrementar(int x) {
       x = x + 1;
   }

   int main() {
       int a = 5;
       incrementar(a);
       printf("El valor de a es: %d\n", a); // Imprime 5
       return 0;
   }
   ```

   En este caso, el valor de `a` no cambia porque se pasa una copia de su valor a la función.

   En el paso por referencia, se pasa la dirección de memoria de la variable al argumento de la función. Esto permite que la función modifique directamente el valor de la variable original. Por ejemplo:

   ```
   void incrementar(int *x) {
       *x = *x + 1;
   }

   int main() {
       int a = 5;
       incrementar(&a);
       printf("El valor de a es: %d\n", a); // Imprime 6
       return 0;
   }
   ```

En este caso, el valor de `a` cambia porque se pasa su dirección de memoria a la función.

1.4. Una variable local es aquella que se declara dentro de una función o bloque y solo es accesible dentro de ese ámbito. Su tiempo de vida está limitado a la ejecución de la función o bloque en el que se declara. Por ejemplo:

```
void funcion() {
    int x = 10; // Variable local
    printf("x: %d\n", x);
}
```

En este caso, `x` solo existe mientras se ejecuta `funcion`.

Una variable global, en cambio, se declara fuera de cualquier función y es accesible desde cualquier parte del programa. Su tiempo de vida es el mismo que el del programa. Por ejemplo:

```
int x = 10; // Variable global

void funcion() {
    printf("x: %d\n", x);
}
```

Aquí, `x` es accesible tanto dentro como fuera de `funcion`.

1.5. La diferencia entre & y * en C radica en su uso y propósito:

- `&` (operador de dirección): Se utiliza para obtener la dirección de memoria de una variable. Por ejemplo:

  ```
  int x = 10;
  int *p = &x; // p almacena la dirección de x
  ```

  Aquí, `&x` devuelve la dirección de memoria de la variable `x`.

- `*` (operador de desreferencia): Se utiliza para acceder al valor almacenado en la dirección de memoria a la que apunta un puntero. Por ejemplo:

  ```
  int x = 10;
  int *p = &x; // p almacena la dirección de x
  printf("Valor de x: %d\n", *p); // Imprime 10
  ```

  Aquí, `*p` accede al valor almacenado en la dirección de memoria a la que apunta `p`.

1.6. La memoria estática (stack) es una región de memoria que se utiliza para almacenar variables locales y llamadas a funciones. Su tamaño es fijo y se gestiona automáticamente. Por otro lado, la memoria dinámica (heap) es una región de memoria que se utiliza para almacenar datos cuyo tamaño puede cambiar en tiempo de ejecución. Su gestión es manual, utilizando funciones como `malloc` y `free`. Por ejemplo:

```
// Memoria estática
void funcion() {
    int x = 10; // Variable en el stack
}

// Memoria dinámica
void funcion() {
    int *p = (int *)malloc(sizeof(int)); // Variable en el heap
    *p = 10;
    free(p); // Liberar memoria
}
```

2. Punteros

2.1. Un puntero es una variable que almacena la dirección de memoria de otra variable. La diferencia principal entre un puntero y una variable normal es que un puntero no almacena directamente un valor, sino la dirección donde se encuentra ese valor. Por ejemplo:

```
int x = 10;
int *p = &x; // p almacena la dirección de x
printf("Valor de x: %d\n", x);   // Imprime 10
printf("Valor de x usando puntero: %d\n", *p); // Imprime 10
```

En este caso, `p` almacena la dirección de `x`, y `*p` accede al valor almacenado en esa dirección.

2.2. Un puntero nulo es un puntero que no apunta a ninguna dirección de memoria válida. Se utiliza para indicar que el puntero no está inicializado o que no tiene un objetivo válido. En C, un puntero nulo se representa con la constante `NULL`. Por ejemplo:

```
int *p = NULL; // p es un puntero nulo
if (p == NULL) {
    printf("El puntero no apunta a ninguna dirección válida.\n");
}
```

En este caso, `p` no apunta a ninguna dirección de memoria válida, lo que puede ser útil para evitar errores al intentar acceder a memoria no inicializada.

2.3. La aritmética de punteros también se puede utilizar para recorrer un string. Por ejemplo:

```
char str[] = "Hola";
char *p = str;  // p apunta al primer carácter de str
while (*p != '\0') {  // Recorre hasta encontrar el carácter nulo
    printf("%c\n", *p);
    p++;
}
```

En este caso, `p` se incrementa para apuntar al siguiente carácter del string hasta llegar al carácter nulo `'\0'`, que indica el final del string.

2.4. En C, la dirección del puntero $p+3$ se calcula sumando $3 \times$sizeof(float) a la dirección base p. Dado que la dirección base es $0x845b342c0$ y el tamaño de un `float` es típicamente 4 bytes, la dirección resultante será:

$$p + 3 = 0x845b342c0 + (3 \times 4) = 0x845b342c0 + 12 = 0x845b342cc$$

Por lo tanto, la dirección del puntero $p + 3$ es $0x845b342cc$.

3. Complejidad temporal y espacial

3.1. 1

4. Recursión