Universidad De Las Américas

ISW111 Algoritmos

Proyecto Integrador

Nombres:

- Ismael Darío Espinoza Granja
- Leandro Xavier Simbaña Imbaquingo

Problema 1:

Cotizaciones

En el gobierno está licitando la construcción de una carretera. Se tienen 5 empresas candidatas. Elabora un programa que pida el monto de las 5 cotizaciones. Enseguida se deberá descartar la más barata y la más cara, luego se deberá obtener el promedio de las que restan. El programa deberá mostrar en pantalla las cotizaciones que se eliminan y el promedio del resto de las cotizaciones.

En el primer ejercicio se debe realizar un análisis funcional para identificar los requisitos clave y las tareas a realizar antes de iniciar el proceso de codificación. A continuación, veremos cuáles son los aspectos claves que se tendrá en cuenta para realizar el código en C++:

1. Entradas:
    - El programa debe solicitar el monto de las cotizaciones para las 5 empresas candidatas.

2. Proceso:
    - El programa debe descartar la cotización más barata y la más cara.
    - Luego, debe calcular el promedio de las cotizaciones restantes.

3. Salidas:
    - El programa debe mostrar en pantalla las cotizaciones que se eliminan (la más barata y la más cara).
    - También, debe mostrar el promedio del resto de las cotizaciones.

4. Estructura del Programa:
    - Se puede utilizar un bucle para solicitar las cotizaciones y almacenarlas en un array.

- Se deben implementar funciones para encontrar la cotización más barata y la más cara, así como para calcular el promedio de las cotizaciones restantes.

Una vez teniendo claro cuáles serán las partes que tendrá nuestro código procedemos a codificar con 2 posibles soluciones y escogeremos cuál es la más óptima para este caso de estudio

- Primera solución planteada con la estructura FOR:

```
#include <iostream>

int main() {
    const int empresas = 5;
    float cotizacion[empresas];

    // Ingreso de las cotizaciones
    std::cout << "Ingrese el monto de la cotizacion:\n";
    for (int i = 0; i < empresas; ++i) {
        std::cout << "Cotizacion empresa " << i + 1 << ": ";
        std::cin >> cotizacion[i];
    }

    // Esta parte inicia la diferenciación de la cotización más alta vs la más baja
    int cotizacionmin = 0;
    int cotizacionmax = 0;

    for (int i = 1; i < empresas; ++i) {
            if (cotizacion[i] > cotizacion[cotizacionmax]) {
            cotizacionmax = i;
        }

        if (cotizacion[i] < cotizacion[cotizacionmin]) {
            cotizacionmin = i;
        }
    }

    // Mostrar cotizaciones eliminadas
    std::cout << "\nCotizaciones eliminadas:\n";
    std::cout << "La cotización mas alta es: " << cotizacion[cotizacionmax] << std::endl;
    std::cout << "La cotización mas baja es: " << cotizacion[cotizacionmin] << std::endl;

    // Para las cotizaciones restantes
    float sumacot = 0;
    int contador = 0;

    for (int i = 0; i < empresas; ++i) {
        if (i != cotizacionmin && i != cotizacionmax) {
            sumacot += cotizacion[i];
            contador++;
        }
    }
```

```
        float promediocot = sumacot / contador;

        std::cout << "\nEl promedio de las cotizaciones restantes es: " << promediocot <<
    std::endl;

        return 0;
}
```

El programa comienza declarando una constante `empresas` con valor 5, representando el número de empresas candidatas para la construcción de la carretera. Se utiliza un array llamado `cotizacion` para almacenar las cotizaciones de cada empresa, y se especifica que estas cotizaciones son de tipo `float`.

El usuario es guiado a través de un bucle `for` para ingresar las cotizaciones de las empresas. Dentro de este bucle, se utiliza `std::cout` para imprimir mensajes solicitando las cotizaciones de cada empresa, y `std::cin` para leer las cotizaciones ingresadas por el usuario. Las cotizaciones se almacenan en el array `cotizacion`.

Después de ingresar todas las cotizaciones, el programa pasa a determinar la cotización más alta y la más baja. Esto se realiza mediante otro bucle `for`, que compara cada cotización con la cotización actual más alta y más baja. Se utilizan variables `cotizacionmin` y `cotizacionmax` para realizar un seguimiento de los índices de las cotizaciones más baja y más alta, respectivamente.

Una vez que se han identificado las cotizaciones más baja y más alta, se imprime un mensaje utilizando `std::cout` para informar al usuario sobre cuáles cotizaciones se han eliminado. Se muestra la cotización más alta y la más baja utilizando los índices obtenidos previamente.

Luego, el programa calcula el promedio de las cotizaciones restantes. Se utiliza un bucle `for` para recorrer todas las cotizaciones y sumar solo aquellas que no fueron eliminadas (las que no coinciden con los índices de la cotización más baja y más alta). Se lleva un contador para determinar cuántas cotizaciones se están sumando.

El promedio se calcula dividiendo la suma de las cotizaciones restantes por el número de cotizaciones restantes. El resultado se almacena en una variable llamada `promediocot`.

Finalmente, se imprime el resultado del promedio utilizando `std::cout`. El programa termina con `return 0;`.

En términos de eficiencia, el código es relativamente simple y directo. Utiliza bucles para iterar sobre las cotizaciones, lo que es una elección eficiente para este tipo de tarea repetitiva. También utiliza variables para realizar un seguimiento de las cotizaciones más baja y más alta, evitando la necesidad de anidar múltiples condicionales. La introducción de un bucle `do-while` para la entrada de cotizaciones también habría sido una opción válida, aunque el bucle `for` utilizado cumple eficientemente con los requisitos.

Es importante destacar que el código no maneja situaciones de entrada incorrecta por parte del usuario, como la introducción de letras en lugar de números. Agregar validación y manejo de errores podría ser una mejora para garantizar la robustez del programa en condiciones de entrada inesperadas.

En resumen, el código en C++ es una implementación efectiva para solicitar, procesar y analizar cotizaciones de empresas en el contexto de una licitación para la construcción de una carretera.

- Segunda solución planteada mediante DO-While

```
#include <iostream>

int main() {
    const int empresas = 5;
    float cotizacion[empresas];

    // Ingreso de las cotizaciones usando do-while
    int i = 0;
    do {
        std::cout << "Cotizacion empresa " << i + 1 << ": ";
        std::cin >> cotizacion[i];
        i++;
    } while (i < empresas);

    // Inicialización de índices para la cotización más alta y más baja
    int cotizacionmin = 0;
    int cotizacionmax = 0;

    // Identificación de la cotización más alta y más baja
    i = 1;  // Reiniciamos i ya que se usó anteriormente
    do {
        if (cotizacion[i] > cotizacion[cotizacionmax]) {
            cotizacionmax = i;
        }

        if (cotizacion[i] < cotizacion[cotizacionmin]) {
            cotizacionmin = i;
        }
        i++;
    } while (i < empresas);

    // Mostrar cotizaciones eliminadas
    std::cout << "\nCotizaciones eliminadas:\n";
    std::cout << "La cotización mas alta es: " << cotizacion[cotizacionmax] << std::endl;
    std::cout << "La cotización mas baja es: " << cotizacion[cotizacionmin] << std::endl;

    // Calcular el promedio de las cotizaciones restantes
    float sumacot = 0;
    int contador = 0;

    i = 0;  // Reiniciamos i para utilizarlo en el nuevo bucle
    do {
        if (i != cotizacionmin && i != cotizacionmax) {
            sumacot += cotizacion[i];
```

```
            contador++;
        }
        i++;
    } while (i < empresas);

    // Calcular y mostrar el promedio de las cotizaciones restantes
    float promediocot = sumacot / contador;
    std::cout << "\nEl promedio de las cotizaciones restantes es: " << promediocot << std::endl;

    return 0;
}
```

En cuanto a la estructura general, ambos códigos siguen una secuencia lógica, comenzando con la entrada de cotizaciones, seguido por la identificación de las cotizaciones más extremas y concluyendo con el cálculo del promedio. Ambos utilizan arrays para almacenar las cotizaciones, variables para realizar un seguimiento de las cotizaciones más baja y más alta, y bucles para iterar sobre las cotizaciones.

La diferencia principal reside en la elección del bucle para la entrada de datos. En el primer código, se utiliza un bucle for para la entrada de cotizaciones, que es una opción común y eficiente en este contexto. Por otro lado, el segundo código opta por un bucle do-while para la entrada de cotizaciones. Este enfoque garantiza que al menos una cotización sea ingresada antes de realizar comparaciones, lo que puede ser considerado un beneficio en términos de robustez del programa.

El bucle do-while utilizado en el segundo código asegura que el programa no avance sin la entrada mínima necesaria, brindando una capa adicional de validación. Aunque ambas implementaciones son válidas, la elección entre bucles puede depender de preferencias de estilo de codificación o requisitos específicos de la aplicación.

Ambos códigos demuestran eficiencia en términos de simplicidad y legibilidad. Utilizan variables para rastrear las cotizaciones más extremas, evitando anidaciones innecesarias de condicionales. Sin embargo, es importante destacar que ninguno de los códigos aborda la validación de entrada del usuario, como la detección de entradas no válidas, lo que podría considerarse una mejora para garantizar la robustez del programa en situaciones de entrada inesperadas.

En resumen, ambas implementaciones logran el mismo propósito, pero difieren en el tipo de bucle utilizado para la entrada de datos. La elección entre un bucle for y un bucle do-while puede depender de las preferencias del programador y de la importancia de garantizar la entrada mínima necesaria antes de continuar con el programa. Ambos códigos demuestran enfoques efectivos y eficientes para el procesamiento de cotizaciones de empresas en el contexto de una licitación para la construcción de una carretera

Ejercicio 12

- Peces

Se ha instalado un sensor submarino para contar el número de peces que pasan bajo él en un lago. El sensor tiene un temporizador que envía el carácter “T” al procesador cada segundo. De detectarse la presencia de un pez, el sensor envía el carácter “P” al procesador. Al finalizarse el período de temporización, a la “T” le sigue inmediatamente una F. Como ejemplo, un flujo de datos normales podría ser: T P T T T P P T P P P P T P T F

De ello se desprende que la primera detección de un pez tuvo lugar en el segundo 2. Luego, en el 6 y 7 pasaron dos peces bajo el sensor, etc. Construya un programa que introduzca los datos procedentes del sensor y que produzca la siguiente salida:

1. Número de segundos que el sondeo estuvo funcionando.
2. Total de peces que pasaron bajo el sensor.
3. Mayor número de peces en segundos consecutivos que pasaron por el sensor.

Los datos que constituirían la salida del ejemplo anterior serían:

1. El sondeo duró 16 segundos.
2. Un total de 8 peces pasaron bajo el sensor.
3. El mayor número de peces que pasaron en segundos consecutivos bajo el sensor fue de 4.

Para la realización del segundo ejercicio se debe realizar un análisis para considerar cuales son las variables para tener en cuenta al momento de pasar este análisis a un código funcional. En este caso estas variables serán:

1. Entrada de Datos:
   - El programa debe recibir datos del sensor submarino, que emite "T" cada segundo y "P" cuando detecta un pez. La secuencia finaliza con "TF".

2. Proceso:
   - El programa debe calcular el número de segundos que el sondeo estuvo en funcionamiento.
   - También, debe determinar el total de peces que pasaron bajo el sensor.
   - Debe identificar el mayor número de peces en segundos consecutivos.

3. Salida:
   - El programa debe producir la siguiente salida:

1. Número de segundos que el sondeo estuvo funcionando.
2. Total de peces que pasaron bajo el sensor.
3. Mayor número de peces en segundos consecutivos que pasaron por el sensor.

4. Estructura del Programa:

- Puede utilizar un bucle para leer los datos del sensor hasta que se detecte la secuencia "TF".
- Llevar un contador de tiempo para calcular la duración del sondeo.
- Mantener un contador de peces y otro para seguir la cuenta de peces consecutivos.
- Actualizar el máximo de peces consecutivos cuando sea necesario.

- Primera solución planteada con la estructura DO-While

```
#include <iostream>

int main(){

printf("--- Sensor Peces ---\n \n");
printf("Opciones del sondeo (T = Sin registro, P = Pez detectado, F = Fin del Sondeo)\n \n");

char ctrl;
int i=0, pez=0, pezcon=0;
int max=0;

do{
   i++;
   printf("Ingrese el dato del sensor [%d]: ", i);
   scanf("%c",&ctrl);
   getchar();

   if(ctrl == 'P'){
      pez++;
      pezcon++;

   }else if (ctrl == 'T'){

      if(pezcon>=max){
         max=pezcon;
         pezcon=0;
      }

   }

}while (!(ctrl == 'F'));

printf("RESULTADOS.\n1. El sondeo duro %d segundos\n2. Un total de %d peces pasaron
bajo el sensor.\n3. El mayor numero de peces consecutivos que pasaron bajo el sensor fue de
%d.", i, pez, max);
   return 0;
}
```

- Estructura del Código:

- Encabezado y Declaraciones: El programa comienza incluyendo la biblioteca `<iostream>`, que proporciona funciones para entrada y salida en C++. Luego, se declaran las variables necesarias, como `ctrl` para almacenar la entrada del usuario, `i` para contar el tiempo de sondeo, `pez` para el número total de peces, `pezcon` para el número consecutivo de peces, y `max` para el mayor número consecutivo de peces.

- Mensaje Inicial: Se imprime un mensaje informativo al usuario utilizando `std::cout` para indicar las opciones de sondeo.

- Bucle de Sondeo: El programa utiliza un bucle `do-while` para realizar el sondeo. Dentro del bucle, se incrementa el tiempo de sondeo (`i`) en cada iteración y se solicita al usuario que ingrese el dato del sensor.

- Lógica de Procesamiento: Se utiliza una estructura condicional para procesar la entrada del usuario. Si se detecta un pez (`P`), se incrementa el contador de peces (`pez`) y el contador consecutivo de peces (`pezcon`). Si no hay registro (`T`), se verifica si el contador consecutivo de peces es mayor o igual al máximo registrado hasta ahora (`max`), y si es así, se actualiza `max` y se reinicia `pezcon` a cero.

- Finalización del Sondeo: El bucle se repite hasta que se ingresa la opción de finalización (`F`). Después, se imprime un mensaje con los resultados del sondeo, incluyendo la duración del sondeo, el total de peces detectados y el mayor número consecutivo de peces.

Análisis:

- Eficiencia: El código es eficiente en términos de ejecución. Utiliza un bucle `do-while` para garantizar que el sondeo se realice al menos una vez, y luego se mantiene ejecutando hasta que el usuario ingresa la opción de finalización (`F`).

- Legibilidad: El código es relativamente fácil de leer y entender. Los nombres de las variables son descriptivos, y los mensajes de salida están formateados de manera clara.

- Portabilidad: El código utiliza características de C++ estándar, por lo que debería ser portátil en la mayoría de los entornos que admitan C++.

En resumen, el código implementa de manera efectiva un programa de sondeo de un sensor de peces en C++, con un diseño claro y lógica de procesamiento eficiente. Puede mejorarse considerando la validación de entrada para aumentar su robustez en situaciones de entrada inesperadas.

- Segunda solución planteada con la estructura While

```
#include <iostream>
```

```
// Función principal del sondeo
void realizarSondeo() {
   std::cout << "--- Sensor Peces ---\n\n";
   std::cout << "Opciones del sondeo (T = Sin registro, P = Pez detectado, F = Fin del
Sondeo)\n\n";

   char ctrl;
   int i = 0, pez = 0, pezcon = 0;
   int max = 0;

   // Bucle de sondeo
   while (true) {
      i++;
      std::cout << "Ingrese el dato del sensor [" << i << "]: ";
      std::cin >> ctrl;
      std::cin.ignore(); // Para consumir el carácter de nueva línea

      if (ctrl == 'F') {
         // Salir del bucle si se ingresa la opción de finalización
         break;
      }

      if (ctrl == 'P') {
         pez++;
         pezcon++;

      } else if (ctrl == 'T') {

         if (pezcon >= max) {
            max = pezcon;
            pezcon = 0;
         }
      }
   }

   // Imprimir resultados del sondeo
   std::cout << "\nRESULTADOS.\n1. El sondeo duro " << i << " segundos\n";
   std::cout << "2. Un total de " << pez << " peces pasaron bajo el sensor.\n";
   std::cout << "3. El mayor numero de peces consecutivos que pasaron bajo el sensor fue de "
<< max << ".";
}

int main() {
```

```
    // Llamar a la función de sondeo
    realizarSondeo();

    return 0;
}
```

En el código de la primera solución, se utilizó un bucle do-while, que garantiza que el cuerpo del bucle se ejecute al menos una vez antes de verificar la condición de salida. Este enfoque es válido y eficiente, pero puede resultar en código ligeramente menos modular y más denso. La estructura del bucle do-while puede ser preferida cuando se necesita garantizar al menos una iteración del bucle antes de evaluar la condición de salida.

En el segundo código, se optó por un bucle while que se ejecuta indefinidamente hasta que se encuentra una declaración break al ingresar la opción de finalización (F). Este cambio mejora el modularidad al encapsular la lógica del sondeo en una función separada llamada realizarSondeo. Esta función encapsula la lógica del sondeo y proporciona una estructura más clara y fácilmente comprensible. El modularidad es una práctica recomendada en programación, ya que facilita el mantenimiento y la reutilización del código.

Ambas versiones del código carecen de validaciones de entrada detalladas, lo que podría resultar en comportamientos inesperados si el usuario ingresa datos no válidos. Sin embargo, la comparación se centra en las diferencias estructurales y de estilo entre las implementaciones.

En términos de eficiencia y legibilidad, ambas versiones son comparables. Ambas implementaciones son portátiles y compatibles con C++ estándar. La elección entre la estructura del bucle do-while y while(true) es más una cuestión de estilo y preferencia del programador. La versión modificada destaca la importancia de la modularidad al encapsular la lógica del sondeo en una función separada, mejorando así la estructura del código principal.

En resumen, la versión modificada del código demuestra una mayor modularidad y claridad estructural, lo que facilita la comprensión y el mantenimiento del programa a medida que crece en complejidad. Ambas implementaciones son efectivas, y la elección entre ellas dependerá de las preferencias del programador y los requisitos específicos del proyecto.