Resumen Primer Capitulo

programming: a general overview

objetivos del capitulo:

El capitulo trata sobre los objetivos de todo el libro y ve un pequeño repaso sobre conceptos básicos de la programación y de las matemáticas discretas.

- Se vera como trabaja un programa para largas entradas y como su funcionamiento importa.
- Se vera todas las matemáticas necesarias para el resto del libro.
- También un breve repaso a recursión.
- también un breve resumen a ciertas características de c++ que serán necesarias para el libro.

1.1: whats this book about

Se vera las problemáticas del libro con dos ejemplos de manejo de búsqueda o datos uno con números y otro con letras:

Primer ejemplo.

Suponemos que tenemos una lista de N números, y quieres determinar el Kth largo.

- La primera solución acomoda los datos de forma decreciente y regresa el numero en la posición k.
- La segunda solución arregla los elementos de forma decreciente luego lee los datos uno por uno y compara en un nuevo arreglo de k espacios, en el que compara los datos, si es mas chico que el k elemento lo omite pero si es mas grande lo acomoda en el arreglo. cuando el algoritmo acaba retorna el elemento en la posición k.
  ambos son algoritmos muy sencillos pero se complican mucho si les agregas una gran cantidad de elementos ya que ambos optan por comparar 1 por 1.

Segundo ejemplo

El objetivo es encontrar las palabras ocultas en dos arreglos de caracteres (char), las palabras pueden estar escondidas de forma horizontal, vertical y diagonal en cualquier dirección.

- la primera solución usa ciclos for para buscar la palabra, selecciona un carácter y busca a través de las filas y columnas en todas las orientaciones.
- la segunda solución es prácticamente la misma pero para que sea mas fácil decir si es la palabra usa también el numero de caracteres.
  De nuevo tenemos el problema de que el programa funciona perfecto mientras menos datos maneje, es decir si le agregas una gran cantidad de elementos va se vuelve impráctico debido a problemas de largos tiempos de carga.

1.2 Mathematics Review

en este apartado se centra en un repaso a partir de formulas y datos matemáticos que pueden ser de nuestro interés.

1.2.1 Exponentes:

Da un repaso por la ley de los exponentes.

1.2.2 Logaritmos:

- en la computación todos los algoritmos son de base dos a menos que se indique de otra forma.
- definición.
- da un repaso a dos teoremas

1.2.3 Series:

- revisa los conceptos básicos de las series y muestra algunas de las formula de sumatorias que mas se usan en la algebra

1.2.4 Aritmética modular:

- La aritmética modular es un sistema de aritmética para números enteros, en el que los números se envuelven al alcanzar un determinado valor, el módulo. En la aritmética modular de la congruencia, los números congruentes al mismo número módulo de un determinado módulo se consideran iguales.

1.2.5 The P word:

las dos formas mas comunes de comprobar las afirmaciones en las estructuras de datos las comunes son:

- prueba por inducción

- prueba por contradicción
  existe una tercera que es prueba por intimidación pero usualmente es usada solo por docentes, la mejor forma de comprobar si tu teorema es verdadero o falso es usando un contra ejemplo.

prueba por inducción:

La prueba por inducción tiene dos partes base, la primera es crear un caso base en el que se compruebe que el teorema es verdadero para algunos casos. luego se hace una hipótesis inductiva. asumiendo que el teorema sirve para todos los casos con índice *k* , usando esta suposición se demuestra si el teorema es verdadero para el siguiente numero *k+1* , resulta que es verdadero, siempre y cuando *k* sea finito.

prueba por contradicción:

La prueba por contradicción parte de la suposición de que el teorema es falso y demuestra que es así, el teorema implica que alguna propiedad es falsa por ende el teorema original tambien.

1.3 introducción a la recursividad.

La recursividad es la capacidad de determinar un elemento en lo que respecto a su propia constitución, es decir, en una recurrencia indeterminada, dicha recurrencia es un recurso básico en la programación asociada a la repetición de funciones.

```
#include <iostream>

//function
void recursive(int n);

int main() {
    int n;
    std::cout << "ingresa un numero grande papu" << std::endl;
    std:: cin >> n;
    recursive(n);
    return 0;
}
void recursive (int n)
{
    if (n >= 10 )
    {
        std::cout << "El numero es: "<< n <<std::endl;
```

```
        n = n/10;
        recursive(n);
    }
    else
    {
        std::cout << "El numero es: " << n << std:: endl;
    }
}
```

1.4 Clases C++

Que es?

- Una clase en C++ es una plantilla que define el estado y el comportamiento de un tipo de objeto. En C++, las clases son una parte fundamental de la programación.
- En C++, las clases se definen utilizando la palabra clave class seguida del nombre de la clase y un bloque de código entre llaves {}. Dentro del cuerpo de la clase, se pueden definir propiedades y métodos que describen el estado y el comportamiento del objeto1.

1.4.1 sintaxis básica de una clase en c++

Una clase tiene lo conocido como miembros, los cuales son datos o variables y funciones, dichas funciones se llamas de otra forma como método.

ejemplo de código básico de una clase en c++

```
class Intcell
{
public:
        IntCell()
                { storedValue = 0;}
        IntCell(int initialValue)
                { storedValue = initialValue; }

        int read()
                { return storedValeu; }

        void  write (int x)
                { stored = x; }

private:
```

```
        int storedValue;
};
```

1.4.2 Extra constructor Syntax And accesors

aunque ya es un código completo y bueno tiene muchas fallas y hay sintaxis adicional que van a mejorar el código.

Initialization list:

El constructor IntCell utiliza una lista de inicialización antes del cuerpo del constructor. La lista de inicialización se utiliza para inicializar los miembros de datos directamente. Usas listas de inicialización en lugar de una asignación en el cuerpo porque te ahorra tiempo, especialmente cuando los miembros de datos son tipos de clase que tienen inicializaciones complejas. En algunos casos, es obligatorio utilizar listas de inicialización. Por ejemplo, si un miembro de datos es constante (lo que significa que no se puede cambiar después de que se ha construido el objeto), entonces el valor del miembro de datos solo se puede inicializar en la lista de inicialización. Además, si un miembro de datos es en sí mismo un tipo de clase que no tiene un constructor de parámetro cero, entonces debe ser inicializado en la lista de inicialización utilizando la sintaxis : valor_almacenado{valor_inicial} {} en lugar de la sintaxis tradicional : valor_almacenado(valor_inicial) { }. El uso de llaves en lugar de paréntesis es nuevo en C++ 11 y es parte de un esfuerzo mayor para proporcionar una sintaxis uniforme para la inicialización en todas partes.

```
class IntCell
{
public:
        IntCell( )
                { storedValue = 0; }
        IntCell( int initialValue )
                { storedValue = initialValue; }
        int read( )
                { return storedValue; }
        void write( int x )
                { storedValue = x; }
private:
        int storedValue;
};
```

constructor Explicito:

Un constructor explícito es un tipo de constructor en programación orientada a objetos que se utiliza para convertir un objeto de una clase a otro tipo de objeto. Este constructor se utiliza para convertir un objeto de una clase a otro tipo de objeto. Por ejemplo, si tenemos una clase A y queremos convertirla en una clase B, podemos utilizar un constructor explícito para hacerlo. El constructor explícito se define con la palabra clave explicit y toma un solo argumento que es el objeto que se va a convertir.

separación de interfaz e implementación:

La interfaz usualmente la colocamos en un archivo .h o de header, el de implementación se tiene un archivo .cpp igual que el main, que requiere incluir la librería.

muestra de como se vería el código de interfaz:

```
#ifndef IntCell_H
#define IntCell_H
class IntCell
{
public:
explicit IntCell( int initialValue = 0 );
int read( ) const;
void write( int x );
private:
int storedValue;
};
#endif
```

Muestra de como se vería el archivo de implementación:

```
#include "IntCell.h"
6 IntCell::IntCell( int initialValue ) : storedValue{ initialValue }
{
}
int IntCell::read( ) const
{
return storedValue;
}
void IntCell::write( int x )
{
```

```
storedValue = x;
}
```

Muestra de como se veria el cpp del main

```
#include <iostream>
#include "IntCell.h"
using namespace std;

int main( )
{
IntCell m;
m.write( 5 );
cout << "Cell contents: " << m.read( ) << endl;
return
```

1.4.4 vector and string:

En El estándar de c++ define dos tipos de clases el **vector** y el **string**
el primero se creo con el interés de remplazar el segundo, ya que su string incorporada causa muchos problemas.
El principal problema de los arreglos es que no se comporta como una clase de primer orden, es decir no se puede copiar con un simple igual y batallas para saber cual es su numero de elementos. Otro problema es que es en si una cadena de chars o caracteres por lo tanto suele tener problemas de los mismos y algunos mas, un claro ejemplo es que no compara correctamente cadenas con el uso del doble igual.
Por el otro lado un vector sabe que tan grande es y no solo eso trabaja las matrices y cadenas como un objeto de primera clase es decir que también puede comparar vectores con el doble igual, el mayor que, menor que, etc.

- su mayor ventaja que yo veo es que un vector es dinámico ósea crece y decrece mientras mas o menos elementos ocupes que guarde, pero también, puedes decirle cuantos espacios quieres que tenga.

código de uso de vectores

```
#include <iostream>
#include <vector>
using namespace std;
```

```
int main( )
{
vector<int> squares( 100 );
for( int i = 0; i < squares.size( ); ++i )
squares[ i ] = i * i;

for( int i = 0; i < squares.size( ); ++i )
cout << i << " " << squares[ i ] << endl;

return 0;
}
```

1.5 C++ Details

C++ igual que todos los lenguajes tiene si características especificas del lenguaje y aquí se discutirán con un breve resumen de los mismos.

1.5.1 Punteros

Lo mas importante de un puntero es que almacena la dirección de memoria de lo que le des no el valor, todo puntero depende de dos valores, es signo de ampersand & y el asterisco, dichos punteros son muy útiles a la hora de hacer estructuras dinamicas, como por ejemplo el hacer un arreglo o vector definido dinámico es decir que el tamaño del mismo sea definido durante el tiempo de ejecución.

```
int main( )
{
IntCell *m;
m = new IntCell{ 0 };
m->write( 5 );
cout << "Cell contents: " << m->read( ) << endl;
delete m;
return 0;
}
```

1.5.2 L, R valor y referencias.

Otro tema mas aparte de los punteros, se define como tipos de referencias, a partir de la versión 11 de c++ se crearon los R valor y L valor.

Un L valor es una esprecion identica a un objeto no temporal y un r valor es una exprecion que identifica a un objeto temporal o un valor, como una constante.

```
vector<string> arr( 3 );
const int x = 2;
int y;
...
int z = x + y;
string str = "foo";
vector<string> *ptr = &arr;
```

para resumir los l valores van a la izquierda y usualmente pueden ser modificables, como otra característica es que los l valores también pueden ser r valores al ponerse del lado derecho, en cambio los R valores van a la derecha y también son valores fijos y no modificables un l valor puede ser un r valor pero un r valor no siempre puede ser un l valor.

Por otra parte las referencias se utilizan principalmente para proporcionar interfaces de funciones mas limpias, las funciones necesitan obtener objetos para acceder a sus valores y así usarlos en dicha función como referencias.

1.5.3 Paso de parámetro

c++ pasa todos los parámetros utilizando llamada por valor, esto quiere decir que el argumento se copia en el parámetro formal sin embargo se pueden pasar objetos tan grandes que el pasar por copia es ineficiente, además a veces uno busca poder alterar el el valor que se pasa gracias a esto c++ tiene 4 formas de diferentes de pasar parámetros, las cuales son:

- doble promedio (doble a, doble b). devuelve un promedio de a y b
- intercambio vacío (doble a, doble b). intercambia a y b, tipos de parametros incorrectos
- cadenas de elementos aleatorios `vector<string> arr`. devuelve un elemento aleatorio en arr, pero se vuelve ineficiente
- referencia-valor, su concepto fice que es dado a almacenar un valor temporal a punto de ser destruido

```
string randomItem( const vector<string> & arr );
string randomItem( vector<string> && arr );
vector<string> v { "hello", "world" };
cout << randomItem( v ) << endl;
cout << randomItem( { "hello", "world" } ) << endl;
```

1.5.4 return passing

En c++ existen varias formas de regresar un valor de una función, las mas comunes son:

```
double average( double a, double b );
LargeType randomItem( const vector<LargeType> & arr );
vector<int> partialSum( const vector<int> & arr );
```

todas estas devuelven el valor pero se vuelven ineficientes entonces vamos a buscar una forma mejor.

```
LargeType randomItem1( const vector<LargeType> & arr )
{
return arr[ randomInt( 0, arr.size( ) - 1 ) ];
}

const LargeType & randomItem2( const vector<LargeType> & arr )
{
return arr[ randomInt( 0, arr.size( ) - 1 ) ];
}

vector<LargeType> vec;
...
LargeType item1 = randomItem1( vec ); // copy
LargeType item2 = randomItem2( vec ); // copy
const LargeType & item3 = randomItem2( vec ); // no copy
```

- Lo primero que consideras sucede en las primeras 4 líneas el ejemplo anterior que se utiliza un retorno por valor
- la segunda forma sucede de las líneas 6 a 9 utilizando un retorno por referencia constante para evitar una copia inmediata.
  el siguiente ejemplo también muestra formas similares en la que la llamada al valor resulta ineficiente debido a la creación y eventual limpieza de una copia.

```
vector<int> partialSum( const vector<int> & arr )
{
vector<int> result( arr.size( ) );
result[ 0 ] = arr[ 0 ];
for( int i = 1; i < arr.size( ); ++i )
```

```
result[ i ] = result[ i-1]+ arr[ i ];
return result;
}
vector<int> vec;
vector<int> sums = partialSum( vec ); // Copy in old C++; move in C++11
```

1.5.5 std::swap y std::move

Otros dos ejemplos de formas en las que se pueden evitar copias de elementos al programar es la implementación de una rutina por intercambio o std::swap. el intercambio de dobles se implementa fácilmente por 3 copias con la ayuda de un auxiliar, aunque se vuelve difícil con valores muy grandes.

```
void swap( double & x, double & y )
{
double tmp = x;
x = y;
y = tmp;
}
void swap( vector<string> & x, vector<string> & y )
{
vector<string> tmp = x;
x = y;
y = tmp;
}
```

Una forma de resolverlo es con un std::move

```
void swap( vector<string> & x, vector<string> & y )
{
vector<string> tmp = static_cast<vector<string> &&>( x );
x = static_cast<vector<string> &&>( y );
y = static_cast<vector<string> &&>( tmp );
}
void swap( vector<string> & x, vector<string> & y )
{
vector<string> tmp = std::move( x );
x = std::move( y );
```

```
y = std::move( tmp );
}
```

la función std::move convierte un valor l a un valor r, pero la función std::move no mueve nada mas bien permite o ayuda a que un valor sea movido.

The big Fives:

Destructor

Se llama al destructor cuando quieres eliminar a un objeto, es decir la unica responsabilidad del destructor es liberar los recursos adquiridos por el objeto durante su uso, esto puede ser desde eliminar variables hasta cerrar archivos que se hayan abierto.

Copy constructor

un constructor especializado usado para construir un nuevo objeto inicializado es el constructor por copia, el cual funciona si el objeto existente es un l valor.

```
IntCell( const IntCell & rhs );
IntCell( IntCell && rhs );
```

el constructor de copia se implementa a miembros de datos primitivos.

Move Constructor

un constructor de movimiento es muy similar a un constructor de copia solo que en este caso es enfocado en los r valores.

Copy and move assignment operator=

El operador de asignación se llama cuando = se aplica a dos objetos que han sido ambos previamente construido. lhs=rhs está destinado a copiar el estado de rhs en lhs. Si rhs es un valor l, esto se hace usando el operador de asignación de copia; si rhs es un valor r (es decir, un valor temporal que está a punto de ser destruido de todos modos), esto se hace usando la asignación de movimiento

```
IntCell & operator= ( const IntCell & rhs );
IntCell & operator= ( IntCell && rhs );
```

todos estos grandes 5 vistos en un código:

```
~IntCell( ); // Destructor
IntCell( const IntCell & rhs ); // Copy constructor
IntCell( IntCell && rhs ); // Move constructor
IntCell & operator= ( const IntCell & rhs ); // Copy assignment
IntCell & operator= ( IntCell && rhs ); // Move assignment
```

1.6 templates

Una plantilla es una construcción que genera un tipo o función normal en tiempo de compilación en función de los argumentos que proporciona el usuario para los parámetros de la plantilla.

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

En el código anterior se describe una plantilla para una función genérica con un único parámetro de tipo *T*, cuyo valor devuelto y parámetros de llamada (lhs y rhs) son todos de este tipo. Se puede asignar un nombre a un parámetro de tipo como se desee, pero por convención se usan letras mayúsculas únicas.

1.6.1 funtion templates.

una plantilla de función no es como tal una función si no es una plantilla de algo que puede llegar a ser una función.

```
template <typename Comparable>
const Comparable & findMax( const vector<Comparable> & a )
{
int maxIndex = 0;
for( int i = 1; i < a.size( ); ++i )
if( a[ maxIndex ] < a[i])
maxIndex = i;
return a[ maxIndex ];
}
```

Debido a que los argumentos de la plantilla pueden asumir cualquier tipo de clase, al decidir sobre el parámetro convenciones de paso y retorno-paso, se debe asumir que los argumentos de plantilla no son tipos primitivos.

1.6.2 class template

Una plantilla de clase funciona igual o casi igual que una de función.

```
template <typename Object>
class MemoryCell
{
public:
explicit MemoryCell( const Object & initialValue = Object{ } )
: storedValue{ initialValue } { }
const Object & read( ) const
{ return storedValue; }
void write( const Object & x )
{ storedValue = x; }
private:
Object storedValue;
};
```

memoryCell es como la clase intCell, pero funciona para cualquier tipo.

1.6.3 objects, comparable

se utiliza repetitivamente objects y comprable como tipos genéricos de la siguiente forma:
Se supone que el objeto tiene un constructor de parámetro cero, un operador = y un constructor de copia. Comparable, como se sugiere en el ejemplo de findMax, tiene una funcionalidad adicional en forma de operador< que
se puede utilizar para proporcionar un pedido total.

```
class Square
{
public:
explicit Square( double s = 0.0 ) : side{ s }
{ }

double getSide( ) const
{ return side; }
double getArea( ) const
```

```
{ return side * side; }
double getPerimeter( ) const
{ return 4 * side; }
void print( ostream & out = cout ) const
{ out << "(square " << getSide( ) << ")"; }
bool operator< ( const Square & rhs ) const
{ return getSide( ) < rhs.getSide( ); }
private:
double side;
};
ostream & operator<< ( ostream & out, const Square & rhs )
{
rhs.print( out );
return out;
}
int main( )
{
vector<Square> v = { Square{ 3.0 }, Square{ 2.0 }, Square{ 2.5 } };
cout << "Largest square: " << findMax( v ) << endl;
return 0;
}
```

1.6.4 funcion objects

Anterior mente vimos las plantillas de funciones, dichas plantillas tienen la limitación que es que solo funciona para objetos que tienen una función operator definida y utilizan ese operador.

```
Generic findMax, with a function object, Version #1.
Precondition: a.size( ) > 0.
template <typename Object, typename Comparator>
const Object & findMax( const vector<Object> & arr, Comparator cmp )
{
int maxIndex = 0;
for( int i = 1; i < arr.size( ); ++i )
if( cmp.isLessThan( arr[ maxIndex ], arr[ i ]))
maxIndex = i;
return arr[ maxIndex ];
}

class CaseInsensitiveCompare
```

```
{
public:
bool isLessThan( const string & lhs, const string & rhs ) const
{ return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
};
int main( )
{
vector<string> arr = { "ZEBRA", "alligator", "crocodile" };
cout << findMax( arr, CaseInsensitiveCompare{ } ) << endl;
return 0;
}
```

1.6.5 Separate Compilation of Class Templates

Al igual que las clases normales las plantillas de clases se pueden implementar completamente a la declaración, de esa forma se puede separar la interfaz de la implementación.

1.7 using matrices

muchos algoritmos usan matrices bidimensionales pero c++ no proporciona ese tipo sin embargo esto se soluciona usando un vector de vectores, pero el hacer esto requiere saber sobre la sobrecarga de la matriz.

Opiniones criticas y conclusiones

opinión critica:

EL libro es confuso y de resumir un capitulo entero de temas tan densos terminas no confundió pero si sobresaturado seria mas ameno si le pudiera de poco a poco o en vez de solo resumir leer un apartado y hacer un código referente al apartado. se haría no solo mas llevadero en mi opinión mas interesante y aprendería mas.

conclusión:

todo el texto toma desde una explicación básica de lo que se va a ver, pasa por una muy breve explicación matemática, un repaso diría total a las clases en c++, etc.
Es un libro muy completo y vale mucho la pena de leer, o tenerlo para consulta, aunque es muy revoltoso y da temas de forma incompleta anexando que en el capitulo x, y o z están se puede entender lo suficiente con el suficiente esfuerzo.

Referencias:

- J. D. M. González. “Punteros en C++ ¿Qué son, cómo hacerlos y cómo usarlos?” Aprende a programar desde cero en múltiples lenguajes. Accedido el 20 de septiembre de 2023. [En línea]. Disponible: https://www.programarya.com/Cursos/C++/Estructuras-de-Datos/Punteros
- “Referencias en C++”. Delft Stack. Accedido el 20 de septiembre de 2023. [En línea]. Disponible: https://www.delftstack.com/es/howto/cpp/cpp-reference-in-cpp/#:~:text=Las%20referencias%20se%20utilizan%20principalmente%20para%20proporcionar%20interfaces,no%20hubiera%20conceptos%20de%20referencia%20en%20el%20idioma.
- “Plantillas (C++)”. Microsoft Learn: Build skills that open doors in your career. Accedido el 21 de septiembre de 2023. [En línea]. Disponible: https://learn.microsoft.com/es-es/cpp/cpp/templates-cpp?view=msvc-170