Resumen capitulo 1

Materia:
Estructura de datos

Profesor :
Leonardo Juárez Zucco

Alumno:
Fabio Valdes Paz

UNIAT:
University of Advanced Technologies

Aunque un algoritmo funcione en estricto sentido de la palabra, es decir, que produzca un resultado, es importante que el algoritmo sea factible. Es decir, no nos servirá si tarda demasiado tiempo en resolver un problema, incluso si al final resuelve el problema. Lo mejor es tener un algoritmo que pueda resolverlo en el menor tiempo posible y utilizando la menor cantidad de recursos para considerarse bueno.

La recursividad es una función que se llama a sí misma en la función y contiene un caso base y un caso general.[4]

```
1 int f( int x )
2 {
3 if( x == 0 )
4 return 0;
5 else
6 return 2 * f( x - 1 ) + x * x;
7 }
```

En el código anterior, tenemos una función llamada f que toma un parámetro x. En el primer caso, que es el caso base, se verifica si x es igual a cero. Si es así, la función devuelve 0. Luego, en el segundo caso, que es el caso general, si x no es igual a cero, la función se llama a sí misma con el argumento (x - 1). Después, multiplica por 2 el resultado de la llamada recursiva y suma x * x al resultado. Finalmente, devuelve este resultado.

Siempre se deben cumplir las dos reglas de recursión, ya que de lo contrario podría ocasionar problemas en el código, como un mal rendimiento o que no funcione correctamente. Las reglas son las siguientes: el caso base siempre tiene que tener un caso que se pueda resolver sin recursión, y la segunda regla es progreso; para los casos que se tienen que resolver recursivamente, la llamada recursiva siempre tiene que ser un caso que lleve al caso base. Por ejemplo, si el caso base es 0 = 0, entonces el caso general no puede ser -1, ya que llevaría a -2 y -3, y nunca llegaría al caso base,Regla de diseño Supongamos que todas las llamadas recursivas funcionan
Regla del interés compuesto. Nunca dupliques el trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

```
#include <iostream>

class Perro{
private:// atributos
    std::string color;
    std::string raza;
public://metodos
    Perro(std::string,std::string);
    void comer();
    void correr();
};
//constructor
Perro::Perro(std::string _color, std::string _raza) {
    color = _color;
```

```
    raza = _raza;
}

void Perro::comer() {
    std::cout<<"perro de la raza: "<<raza<<" esta comiedo"<<std::endl;

}

void Perro::correr() {
    std::cout<<"perro de color: "<<color<<" salio a correr"<<std::endl;
}
int main() {

    Perro p1("cafe","mastn_tibetano");
    p1.correr();
    p1.comer();
    return 0;
}
```

La sintaxis de clases en C++ comienza con la palabra clave "class", seguida de llaves de apertura y cierre que terminan con un punto y coma. La clase necesita atributos y métodos. Los atributos suelen ser privados y se declaran usando la palabra clave "private", mientras que los métodos suelen ser públicos y se declaran con la palabra clave "public". Además, la clase debe tener un constructor.

Los arrays y strings no son una buena idea para usar en C++; aunque el lenguaje los proporciona, tienen muchas limitaciones. Por ejemplo, una vez que especificas cuántos elementos puede contener un array, no se puede cambiar. Además, existen varias otras complicaciones. La mejor opción es usar vectores, como se muestra en el siguiente ejemplo.

```
#include <iostream>
 #include <vector>


int main( )
 {
 std::vector<int> numero={5,5,4,8,5};

for (int i=0;i<numero.size();i++)
{
    std::cout<<numero[i]<<std::endl;
}
std::cout<<std::endl;

//acceder a el 4 numero
int cuarto=numero[3];
```

```
//acceder a el 3 numero
int tercer=numero[2];
//manipulacion de los numeros de los vectores
std::cout<<"suma: "<< cuarto+tercer<<" "<<"multiplicacion"<<" "<<cuarto*tercer<<std::endl;

std::cout<<std::endl;
//agregar un nuevo elemento en el vector
numero.push_back(10);

for (int i=0;i<numero.size();i++)
   {
      std::cout<<numero[i]<<std::endl;
   }
std::cout<<std::endl;
//eliminar elemento espesifico del arrey
   numero.erase(numero.begin() + 3);


   for (int i=0;i<numero.size();i++)
   {
      std::cout<<numero[i]<<std::endl;
   }

 return 0;
 }
```

los punteros son variables que almacenan donde está otro objeto el siguiente codigo hace ejemplo de los punteros

```
#include <iostream>

int main() {
   int numero = 50;
   int *puntero;

   int digito;
   int *punt;
//guardar direccion
   puntero=&numero;


   std::cout<<"numero: "<<*puntero <<std::endl;
   //direccion de numero
   std::cout<<"direccion de numero: "<<puntero<<std::endl;

   //arreglo
```

```
    int arreglo[]={1,2,3,4,5,6};
    int *arr;
    arr=arreglo;

    for(int i=0;i<6;i++)
    {
        std::cout<<"elementos: "<<*arr++<<std::endl;
    }


    return 0;
}
```

(*uando un objeto que es asignado por new ya no está referenciado, la operación delete debe aplicarse al objeto (a través de un puntero). De lo contrario, la memoria que consume se pierde (hasta que el programa termina). Esto se conoce como una fuga de memoria.[1]*)
Lvalues y Rvalues:
En el contexto de la declaración int i = 10;, el término "lvalue" se refiere a lo que está en la parte izquierda, en este caso, i, mientras que "rvalue" se refiere a lo que está en la parte derecha, en este caso, 10.
No es posible intercambiar lvalues y rvalues, por ejemplo, int 10 = i; no es válido sintácticamente.
Es incorrecto afirmar que los valores son constantes que no pueden cambiar. Los valores son expresiones temporales que no tienen una ubicación de memoria permanente y no pueden aparecer en el lado izquierdo de una asignación. Por otro lado, los valores son variables, lo que significa que pueden recibir el valor de otra cosa.[5]

call by value y call by reference
pasar por valor es cuando por ejemplo tu declaras una variable por ejemplo x = 5 y luego creas una funcion que mención la x en alguna parte de la memoria hay dos variables llamadas x que valen 5
pero no es la misma son distintas por lo que si dentro de la función tú cambias el valor de x a 20
por ejemplo un a x la original que usaste para pasar el valor seguirá valiendo 5 en cambio la de la función en este caso cambiará a 20
ejemplo

```
#include <iostream>
void numero(int x);
int main() {

    int x=5;
    numero(x);
    //el valor de x afuera de la funcion despues de que la función terminara std::cout<<"valor de x fuera de la función después de que termine la función : "<<x<<std::endl;
    return 0;
}
```

```
void numero(int x)
{
    std::cout<<"valor de x en este momento en la función:
    "<<x<<std::endl; //cambiamos x
    x=20;
    std::cout<<"valor de x después de cambiarlo en la función:

"<<x<<std::endl; }
```

en cambio pasar por referencia es un poco diferente esto lo que provoca es que la x que declaraste fuera de la función ocupará el mismo espacio de la
x de la fusión por lo que si tu cambias dentro de la función x a 20 por ejemplo la x de fuera de la funcion tambien cambiara
y esto se logra con punteros como ejemplo el siguiente código

```
#include <iostream>
void numero(int& x);
int main() {

    int x=5;
    numero(x);
    //el valor de x afuera de la función después de que la función terminará std::cout<<"valor
de x fuera de la función después de que termine la función : "<<x<<std::endl;
    return 0;
}

void numero(int& x)
{
    std::cout<<"valor de x en este momento en la función:
    "<<x<<std::endl; //cambiamos x
    x=20;
    std::cout<<"valor de x después de cambiarlo en la función:

"<<x<<std::endl; }
```

swap sirve para cambiar los valores por ejemplo si quieres que una variable sea igual a otra variable diferente ejemplo

```
1 void swap( double & x, double & y )
2 {
3 double tmp = x;
4 x = y;
5 y = tmp;
6 }
```

una plantilla es para poder obtener toda la cantidad de clases que hay para un solo código

por ejemplo que puedes usar int double o flotante

```
1 /**
2 * Return the maximum item in array a.
3 * Assumes a.size( ) > 0.
4 * Comparable objects must provide operator< and operator=
5 */
6 template <typename Comparable>
7 const Comparable & findMax( const vector<Comparable> & a )
8 {
9 int maxIndex = 0;
10
11 for( int i = 1; i < a.size( ); ++i )
12 if( a[ maxIndex ] < a[i])
13 maxIndex = i;
14 return a[ maxIndex ];
15 }[1]
```

En algunos algoritmos genéricos utilizados para encontrar el número máximo de un arreglo, estas plantillas siempre tienen limitaciones: solo funcionan si el operador del objeto es <, lo que ocasiona problemas si no hay una comparación clara, como en rectángulos o cadenas de texto que necesitan ser comparadas sin diferencias entre mayúsculas y minúsculas. Esta limitación puede solucionarse si se reescribe la función findMax(*encuenta maximo*) para arrays de objetos y funciones de comparación. Esto se puede lograr mediante el uso de objetos de función (*un objeto de función, o functor, es cualquier tipo que implementa operador(). Este operador se conoce como el operador de llamada o, a veces, el operador de aplicación.*[2]) Se utilizan para implementar comparaciones personalizadas en lugar de depender del operador < predeterminado.

```
1 // Generic findMax, with a function object, Version #1.
2 // Precondition: a.size( ) > 0.
3 template <typename Object, typename Comparator>
4 const Object & findMax( const vector<Object> & arr, Comparator cmp )
5 {
6 int maxIndex = 0;
7
8 for( int i = 1; i < arr.size( ); ++i )
9 if( cmp.isLessThan( arr[ maxIndex ], arr[ i]))
10 maxIndex = i;
11
12 return arr[ maxIndex ];
13 }
```

```
14
15 class CaseInsensitiveCompare
16 {
17 public:
18 bool isLessThan( const string & lhs, const string & rhs )
const 19 { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
20 };
21
22 int main( )
23 {
24 vector<string> arr = { "ZEBRA", "alligator", "crocodile" };
25 cout << findMax( arr, CaseInsensitiveCompare{ } ) << endl;
26
27 return 0;
28 }
[1]
```

la expresión cmp.operator()(x,y) también conocida como cmp(x,y) en el que operator () es el operador de llamada de función. con esta opción puedes cambiar el nombre del parámetro a isLessThan (*es menor que*), se puede usar una opción findMax, usando la plantilla less se puede observar en el siguiente código

```
1 // Generic findMax, with a function object, C++ style.
2 // Precondition: a.size( ) > 0.
3 template <typename Object, typename Comparator>
4 const Object & findMax( const vector<Object> & arr, Comparator isLessThan )
5 {
6 int maxIndex = 0;
7
8 for( int i = 1; i < arr.size( ); ++i )
9 if( isLessThan( arr[ maxIndex ], arr[ i]))
10 maxIndex = i;
11
12 return arr[ maxIndex ];
13 }
14
15 // Generic findMax, using default ordering.
16 #include <functional>
17 template <typename Object>
18 const Object & findMax( const vector<Object> & arr )
19 {
```

```
20 return findMax( arr, less<Object>{ } );
21 }
22
23 class CaseInsensitiveCompare
24 {
25 public:
26 bool operator( )( const string & lhs, const string & rhs ) const
27 { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
28 };
29
30 int main( )
31 {
32 vector<string> arr = { "ZEBRA", "alligator", "crocodile" };
33
34 cout << findMax( arr, CaseInsensitiveCompare{ } ) << endl;
35 cout << findMax( arr ) << endl;
36
37 return 0;
38 }
```
[1]

Las matrices son conocidas como arreglos bidimensionales. En el siguiente código se muestra cómo se crea una clase matriz. Primero, se define una clase llamada Matrix, la cual representará una matriz que utiliza vectores de vectores. Para esto, se agrega la biblioteca <vector>. Esta clase tiene tres constructores. Los cuales toman un número de filas y columnas como argumentos y crean una matriz inicializando el vector de vectores. También toman un vector de vectores como argumento para inicializar la matriz y toman un rvalue reference (*el cual se utiliza para inicializar la matriz moviendo los datos.*[3]), se sobrecarga el operador de indexación para acceder a los elementos de la matriz. Además, se proporcionan las funciones miembro numrows() y numcols() para obtener el número de filas y columnas de la matriz, respectivamente. El arreglo es un vector de vectores que sirve para almacenar los datos de la matriz.

```
1 #ifndef MATRIX_H
2 #define MATRIX_H
3
4 #include <vector>
5 using namespace std;
6
7 template <typename Object>
8 class matrix
```

```
9 {
10 public:
11 matrix( int rows, int cols ) : array( rows )
12 {
13 for( auto & thisRow : array )
14 thisRow.resize( cols );
15 }
16
17 matrix( vector<vector<Object>> v ) : array{ v }
18 { }
19 matrix( vector<vector<Object>> &&v): array{ std::move( v ) }
20 { }
21
22 const vector<Object> & operator[]( int row ) const
23 { return array[ row ]; }
24 vector<Object> & operator[]( int row )
25 { return array[ row ]; }
26
27 int numrows( ) const
28 { return array.size( ); }
29 int numcols( ) const
30 { return numrows( ) ? array[ 0 ].size( ) : 0; }
31 private:
32 vector<vector<Object>> array;
33 };
34 #endif
```

El operador se utiliza para acceder a elementos de una matriz. Si la matriz fuera m y usáramos m[i], devolverá un valor de la fila i. Si deseas acceder a un elemento específico, debes usar m[i][j], donde j es la columna.

Existen dos opciones para devolver un valor del operador: por referencia constante o por referencia simple.

conclusión: podemos decir que todos estos temas son la base en programación en c++ esto nos ayudara y enseñara a tener una mejor estructura y optimizar el codigo de la mejor manera.

Referencias

[1] M. Allen Weiss, *Data Structures & Algorith Analysis Mark Allen Weiss*, 4th ed. Addison-Wesley,

1992. [Online]. Available: https://docs.google.com/document/d/1BNwfcJiVOfmO2smpC9ya-aOHqkoG0ibWBSnIDfNBp8w/edit

[2] TylerMSFT, “Objetos de función en la biblioteca estándar de C++,” Microsoft Learn, Jun. 16, 2023. https://learn.microsoft.com/es-es/cpp/standard-library/function-objects-in-the-stl?view=msvc-170

[3] TylerMSFT, “Rvalue reference declarator: &&,” *Microsoft Learn*, Sep. 28, 2022. https://learn.microsoft.com/en-us/cpp/cpp/rvalue-reference-declarator-amp-amp?view=msvc-170

[4] Programación ATS, «79. Programación en C++ || Funciones || Recursividad - Factorial de un número», *YouTube*. 8 de agosto de 2016. [En línea]. Disponible en: https://www.youtube.com/watch?v=i9roxX8z7tk

[5] The Cherno, «lvalues and rvalues in C++», *YouTube*. 2 de abril de 2020. [En línea]. Disponible en: https://www.youtube.com/watch?v=fbYknr-HPYE

[6] Caleb Curry, «C++ Programming Tutorial 67 - Pass By Reference and Pass By Value», *YouTube*. 19 de marzo de 2019. [En línea]. Disponible en: https://www.youtube.com/watch?v=FXzpFn8LJUI