Mario Alberto Montes Girón

Resumen: Programming: A General Overview I

1.1 What's This Book About?

Ideas Principales:

- Writing a working program is not good enough. If the program is to be run on a large data set, then the running time becomes an issue.
- We will see how to estimate the running time of a program for large inputs.
- We will see how to compare the running times of two programs without coding them.
- We will see techniques for drastically improving the speed of a program and for determining program bottlenecks.

Opinión Crítica y Conclusiones:

Estoy alineado a las expectativas de este libro. Entiendo que muchas veces escribir un código no es suficiente. Si queremos hacer la diferencia, habrá que optimizarlo (tiempo/ recursos) para que sea realmente eficiente.

1.2 Mathematics Review

Ideas Principales

- Exponents
- Logarithms
- Series
- Modular Arithmetic
- The P Word (Proof by induction, counterexample, contradiction)

Opinión Crítica y Conclusiones:

Tengo problemas entendiendo los conceptos y ejemplos usados para las Series y los Módulos Aritméticos. No me siento sólido en este tema. En cuanto a Exponentes, Logaritmos y Pruebas, me siento mas seguro ya que son conceptos que he visto anteriormente.

1.3 A Brief Introduction to Recursion

Ideas Principales:

- A function that is defined in terms of itself is called recursive.

- The idea is that the recursive function f ought to be expressible in only a few lines.
- Base case is the value for which the function is directly known without resorting to recursion.
- Recursive calls will keep on being made until a base case is reached.
  Fundamental rules of recursion:
- Base cases. You must always have some base cases, which can be solved without recursion.
- Making progress. For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case.
- Design rule. Assume that all the recursive calls work.
- Compound interest rule. Never duplicate work by solving the same instance of a problem in separate recursive calls.

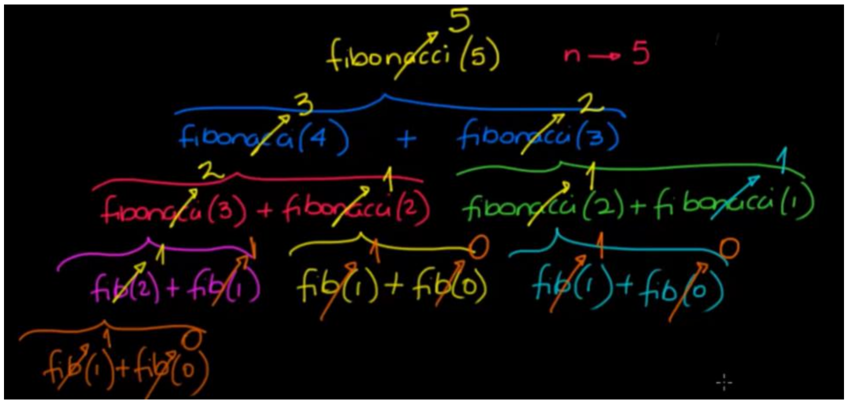
Referencias Adicionales:

"La serie de Fibonacci: 0,1,1,2,3,5,8,13, 21

Empieza con 0 y 1 y tiene la propiedad que cada número subsecuente es la suma de los dos números previos. Puede ser definida en forma recursiva por:

```
long Fibonacci (int n)
{
if (n==0 || n==1)
return n;
 else
return Fibonacci(n-1)+Fibonacci(n-2);
}
```

Usando como ejemplo n = 5: " [1]



Opinión Crítica y Conclusiones:

La recursividad se refiere a una función que se llama a si misma. Esto se hace con el fin de optimizar un código y este sea más eficiente. La recursividad tiene 4 reglas esenciales para poder trabajar con ella.

Base Cases (casos que se pueden resolver sin recursividad), Making Progress (se refiere al cuerpo de la función, donde se hacen los llamados repetitivos), Design Rule (asumir que la llamada de recursividad funciona) y Compound Interest rule (que básicamente es no crear distintas funciones de recursividad dentro del mismo código para resolver el problema).

1.4 C++ Classes

Ideas Principales

- A class in C++ consists of its members. These members can be either data or functions.
- The functions are called member functions.
- Each instance of a class is an object.
- Each object contains the data components specified in the class.
- A member function is used to act on an object. Often member functions are called methods.
- A member that is public may be accessed by any method in any class.
- A member that is private may only be accessed by methods in its class.
- A constructor is a method that describes how an instance of the class is constructed.
- If no constructor is explicitly defined, one that initializes the data members using language defaults is automatically generated.
- Default parameters can be used in any function, but they are most used in constructors.
- The initialization list is used to initialize the data members directly.
- Explicit Constructor: You should make all one-parameter constructors explicit to avoid behind-the-scenes type conversions.
- Example of an incorrect declaration:

```
IntCell obj; // obj is an IntCell
obj = 37; // Should not compile: type mismatch
```

- Example of a correct declaration:

```
IntCell temporary = 37;
obj = temporary;
```

- The use of explicit means that a one-parameter constructor cannot be used to generate an implicit temporary.
- Accessor: A member function that examines but does not change the state of its object.
- Mutator: A member function that changes the state of the object.
- Mutators cannot be applied to constant objects.
- To make a member function an accessor, we must add the keyword const after the closing parenthesis that ends the parameter type list.
- in C++ it is more common to separate the class interface from its implementation.
- The interface lists the class and its members (data and functions). Placed in ".h" file.
- The implementation provides implementations of the functions. Placed in ".cpp" file.
- Each member function (in the .cpp file) must identify the class that it is part of. Otherwise, it would be assumed that the function is in global scope. Example > ClassName::member
- The :: is called the scope resolution operator.
- The signature of an implemented member function must match exactly the signature listed in the class interface.
- Example of declaring an object:

```
IntCell obj1; // Zero parameter constructor
IntCell obj2( 12 ); // One parameter constructor
IntCell obj4{ }; // Zero parameter constructor
```

- The C++ standard defines two classes: the vector and string.
- The built-in string is simply an array of characters.
- A vector knows how large it is. Size is a method that returns the size of the vector.
- Example of declaring a vector:

```
vector daysInMonth = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Opinión Crítica y Conclusiones:

Esta sección habló sobre lo que es un "Class" (que básicamente es algo parecido a un modelo/ prototipo), cómo se encuentran estructurados en los lenguajes de programación (los archivos .h y .cpp) y los elementos que los integran (member functions, objects, methods, etc.) También se revisaron los conceptos de "Array" y "Vector" (que en esencia son estructuras de datos donde se puede almacenar información). Para todos los conceptos se mostraron ejemplos de como deben de ser declarados.

A pesar de que entiendo los conceptos, no me queda muy clara la aplicación de las clases al usarlas con funciones. El semestre anterior solo vimos como usar las Clases como un molde para generar mas objetos que heredaran datos/ atributos.

1.5 C++ Details

Ideas Principales

- A pointer variable is a variable that stores the address where another object resides.
- The ***** indicates that is a pointer variable.
- The use of uninitialized pointers typically crashes programs, because they result in access of memory locations that do not exist.

```
int main( )
{
IntCell *m;

m = new IntCell{0 };
m->write (5);
cout << "Cell contents: " << m->read () << endl;

delete m;
return 0;
 }
```

- In C++ “new” returns a pointer to the newly created object. Example: *m = new IntCell;*
- When an object that is allocated by new is no longer referenced, the “delete” operation must be applied to the object (through a pointer). Otherwise, the memory that it consumes is lost (memory leak)
- Assignment and comparison of pointer variables in C++ is based on the value of the pointer.
- Two pointer variables are equal if they point at the same object.
- If a pointer variable points at a class type, then a (visible) member of the object being pointed at can be accessed via the -> operator.
- Address-of operator “&” > returns the memory location where an object resides.
- An lvalue is an expression that identifies a non-temporary object.
- An rvalue is an expression that identifies a temporary object.
- an lvalue reference is declared by placing an & after some type.

- An rvalue reference is declared by placing an && after some type.
- lvalue references use #1: aliasing complicated names.
- lvalue references use #2: range for loops.
- lvalue references use #3: avoiding a copy.

- Parameter passing mechanisms in C++:
- 1. Call-by-value is appropriate for small objects that should not be altered by the function.
- 2. Call-by-constant-reference is appropriate for large objects that should not be altered by the function and are expensive to copy.
- 3. Call-by-reference is appropriate for all objects that may be altered by the function.

- Return passing mechanisms in C++:
- 1. Return-by-value: the function returns an object of an appropriate type that can be used by the caller; in all cases the result of the function call is an rvalue.
- 2. Return by-constant-reference
- 3. Return-by-reference: is used in a few places to allow the caller of a function to have modifiable access to the internal data representation of a class

- Function std::move exists that converts any lvalue (or rvalue) into an rvalue.
- The swap function std::swap is also part of the Standard Library and will work for any type.

- The Big Five: classes come with five special functions:
- Destructor: is called whenever an object goes out of scope or is subjected to a delete. The default simply applies the destructor on each data member.
- Copy Constructor and Move Constructor: There are two special constructors that are required to construct a new object, initialized to the same state as another object of the same type. These are the copy constructor if the existing object is an lvalue, and the move constructor if the existing object is an rvalue. By default, the copy constructor is implemented by applying copy constructors to each data member in turn.
- Copy Assignment and Move Assignment (operator=): The assignment operator is called when = is applied to two objects that have both been previously constructed.
  lhs=rhs is intended to copy the state of rhs into lhs.
  If rhs is an lvalue, this is done by using the copy assignment operator; if rhs is an rvalue, this is done by using the move assignment operator. By default, the copy assignment operator is implemented by applying the copy assignment operator to each data member in turn.

- When defaults don’t work: Occurs when a data member is a pointer type, and the pointer is allocated by some object member function (such as the constructor).
- To declare an array, arr1, of 10 integers, one writes:

  int arr1[10];

- arr1 is actually a pointer to memory that is large enough to store 10 ints, rather than a first-class array type.
- When arr1 is passed to a function, only the value of the pointer is passed; information about the size of the array is lost.
- If the size is unknown, we must explicitly declare a pointer and allocate memory via new[]
- because memory has been dynamically allocated, at some point it must be freed with delete[]:

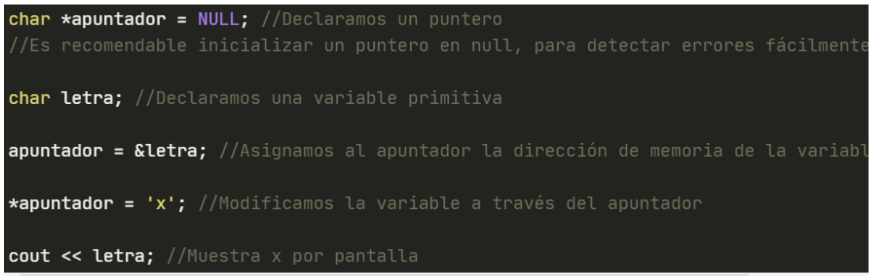
Referencias Adicionales:

“Los punteros y el ampersand

El ampersand es un operador de C++ y es comúnmente utilizado para los punteros. Este operador nos permite obtener la dirección de memoria de una variable cualquiera y es justo esto (la dirección en memoria) lo que utilizan los punteros para referenciar valores.

Los apuntadores y el asterisco

El asterisco es, por decirlo de alguna forma, el operador por excelencia de los punteros. Su utilidad radica en que, si el valor de dicho apuntador corresponde a una dirección de memoria, el asterisco nos permite resolverla y acceder al valor almacenado allí. Viéndolo desde otro enfoque, un apuntador es únicamente una dirección de memoria (un número) y el asterisco es el que hace la magia de obtener el valor referenciado por dicha dirección.” [2]



```
char *apuntador = NULL; //Declaramos un puntero
//Es recomendable inicializar un puntero en null, para detectar errores fácilmente

char letra; //Declaramos una variable primitiva

apuntador = &letra; //Asignamos al apuntador la dirección de memoria de la variabl

*apuntador = 'x'; //Modificamos la variable a través del apuntador

cout << letra; //Muestra x por pantalla
```

Opinión Crítica y Conclusiones:

En esta parte del capítulo se abarcaron temas que van más enfocados a como hacer uso de los espacios de memoria. Entre los conceptos principales se encuentran los punteros, los “rvalues y lvalues”, los parámetros para pasar valores, para retornar valores, los arreglos y los “Big 5”. Estos últimos no me quedan del todo claro ya que no los he usado antes, al menos de manera consciente.

1.6 Templates

Ideas Principales

- Type-independent algorithms (also known as generic algorithms) are written in C++ using the template.
- Templates can be used to write generic algorithms.
- It works only for objects that have an operator< function defined, and it uses that operator< as the basis for all comparison decisions.
- A function template is not an actual function, but instead is a pattern for what could become a function.
- Function templates are expanded automatically as needed.
- An expansion for each new type generates additional code.
- A class template works much like a function template but works for any type of object.
- Object is assumed to have a zero-parameter constructor, an operator=, and a copy constructor.
- Comparable has additional functionality in the form of operator< that can be used to provide a total order.
- Function object: a function is being passed by placing it inside an object. This object is commonly known as a function object.
- Class templates can be implemented either entirely in their declarations, or we can separate the interface from the implementation.
- The entire class template with its implementation is placed in a single header file.
- we implement all class templates entirely in its declaration in a single header file.

Referencias Adicionales:

“We use C++ function templates for creating functions that can be used with different data types. A function template is used to perform the same function on multiple data types. An alternative approach to this is using function overloading. However, using a function template is a better approach to writing less and more maintainable code.

```
template <class type> ret-type func-name(parameter_list) {
   // body of function
}
```

The class templates in C++ make the class using it generic. It operates similar to function templates except that it is used on the entire class instead of a single function. Once the class is created, you can declare the data type while declaring the class instance.

Template<class type>
class class_name
{
.
.
}" [3]

Opinión Crítica y Conclusiones:

Esta sección se enfoca principalmente en los Templates. Es decir, modelos que podemos definir para aplicarlos en la creación de Funciones y Clases (siempre y cuando se puedan adaptar al tipo de datos que se va a usar). Es una manera de agilizar la creación de nuestros algoritmos para no comenzar siempre desde cero. Nunca he usado un Template por lo que las particularidades no me han quedado del todo claras. Por el momento me quedo con el concepto general.

1.7 Using Matrices

Ideas Principales

- Two-dimensional arrays, which are popularly known as matrices.
- The matrix is represented by an array data member that is declared to be a vector of vector.
- The constructor first constructs array as having rows entries each of type vector that is constructed with the zero-parameter constructor.
- The body of the constructor is then entered, and each row is resized to have cols columns.
- The constructor terminates with what appears to be a two-dimensional array.
- The idea of operator[] is that if we have a matrix m, then m[i] should return a vector corresponding to row i of matrix m.

Opinión Crítica y Conclusiones:

El concepto principal que se revisó en esta sección fueron las Matrices, que en esencia son arreglos de dos dimensiones. Se examina su declaración, los usos que se le pueden dar y la importancia del operador [ ] para poder obtener valores específicos dentro de la Matriz. Estoy algo familiarizado con este tema ya que lo he puesto en práctica con anterioridad, pero no me siento seguro de poder implementarlo cuando se pide mas grado de dificultad.

Conclusión General

Muchos de los temas que se cubrieron en este capítulo ya los conocía y otros no. Para los temas que ya conocía, me sirvió para refrescar el conocimiento, pero también para darme cuenta de que hay muchas cosas que todavía desconozco y que no domino. Hay muchas particularidades y grados de dificultad que aún no manejo pero que me gustaría dominar.

Me cuesta mucho trabajo leer códigos que todavía no entiendo. Varios de los ejemplos que dio este capítulo no los pude entender en su totalidad. Esto es en gran parte porque no los he hecho antes. Yo aprendo cuando hago las cosas con alguien guiándome por primera vez y luego refuerzo el conocimiento al hacerlo ahora por mi cuenta.

Hubo otros temas de los cuales nunca había escuchado y mucho menos llevarlos a la práctica. En general, me emociona saber que hay muchas cosas todavía por aprender que me ayudarán a subir mi nivel como programador.

Referencias

[1] Universidad de la República de Uruguay, “Principios de Programación”. [Online]. Available : https://www.fing.edu.uy/tecnoinf/mvd/cursos/prinprog/material/teo/prinprog-teorico10.pdf. [Accessed Sept. 20, 2023].

[2] J. Meza, “Los punteros y elementos dinámicos en C++ con ejemplos y ejercicios resueltos”, 2020. [Online]. Available : https://www.programarya.com/Cursos/C++/Estructuras-de-Datos/Punteros. [Accessed Sept. 21, 2023].

[3] A S. Ravikiran, “Understanding Templates in C++”, 2023. [Online]. Available : https://www.simplilearn.com/tutorials/cpp-tutorial/templates-in-cpp. [Accessed Sept. 21, 2023].

Weiss, M. A. (2014). Data structures and algorithm analysis in C++ (4th ed.).Pearson Education.