Introducción a los algoritmos en c++

OscarGauss :P

PUBLISHED BY PUBLISHER

BOOK-WEBSITE.COM

First printing, March 2013

Contenidos

1 Introducción a la programación en c++ 7

1.1 Introducción **7**

1.1.1 ¿Qué es un Lenguaje de Programación? 7
1.1.2 Historia de C++ 7
1.1.3 ¿Qué es C++? 8
1.1.4 Herramientas Necesarias 8
1.1.5 Consejos iniciales antes de programar 9
1.1.6 Ejemplos 9

1.2 Lo mas basico **10**

1.2.1 Proceso de desarrollo de un programa 10
1.2.2 Sintaxis 12
1.2.3 Comentarios 12
1.2.4 Datos primitivos 13
1.2.5 Variables y constantes 14
1.2.6 Lectura e Impresión 14
1.2.7 Operadores 17
Operadores Aritméticos 17
Operadores de Asignación 17
Operadores de Relación 18
Operadores Lógicos 18
Operadores Relacionados con punteros 18
Operadores de Estructuras y Uniones 18
Operadores Lógicos de Bits 18
Misceláneas 18

1.3 Estructuras de Control **18**

1.3.1 Sentencias de decisión 19
Sentencia if 19
Sentencia switch 21
Operador condicional ternario ?: 22

1.3.2 Sentencias de iteración . . . 23
Sentencias For . . . 23
Sentencia While . . . 24
Sentencia Do - While . . . 25
1.3.3 Sentencias Break y Continue . . . 26
Break . . . 26
Continue . . . 26
Uso de break y continue junto con while . . . 27

2 Estructura de datos 1 . . . 29

3 Programación modular . . . 31

3.1 Introduccion 31
3.1.1 ¿Qué es la Programación Modular? . . . 31

3.2 Modulos 31
3.2.1 Concepto de Modulo . . . 31
3.2.2 Elementos de declaracion del Modulo . . . 31
3.2.3 Ejemplos . . . 31

3.3 Recursividad 32
3.3.1 Que es recursvidad? . . . 32
3.3.2 Ejemplos . . . 32

4 Algoritmos Basicos . . . 35

4.1 Algoritmos de Ordenanmiento 35
4.1.1 Ordenamiento Rapido . . . 35
Descripcion del Algoritomo . . . 35
Codigo en C++ . . . 35
4.1.2 Ordenamiento por mezcla . . . 36
Descripcion del Algoritomo . . . 36
Codigo en C++ . . . 36
4.1.3 Ordenamiento por Monticulos . . . 37
Descripcion del Algoritmo . . . 37
4.1.4 Ordenamiento por Cuentas . . . 38
Descripcion del Algortimo . . . 38

4.2 Busqueda Binaria 38
4.2.1 Descripcion del Algoritmo . . . 38
4.2.2 Ejemplo . . . 38

Bibliography . . . 39

Books 39

Articles 39

Index 41

1. Introducción a la programación en c++

1.1 Introducción

1.1.1 ¿Qué es un Lenguaje de Programación?

Antes de hablar de C++, es necesario explicar que un lenguaje de programación es una herramienta que nos permite comunicarnos e instruir a la computadora para que realice una tarea específica. Cada lenguaje de programación posee una sintaxis y un léxico particular, es decir, forma de escribirse que es diferente en cada uno por la forma que fue creado y por la forma que trabaja su compilador para revisar, acomodar y reservar el mismo programa en memoria.
Existen muchos lenguajes de programación de entre los que se destacan los siguientes:

- C
- C++
- Basic
- Ada
- Java
- Pascal
- Python
- Fortran
- Smalltalk

1.1.2 Historia de C++

C++ es un lenguaje de programación creado por Bjarne Stroustrup en los laboratorios de At&T en 1983. Stroustrup tomó como base el lenguaje de programación más popular en aquella época el cual era C.
El C++ es un derivado del mítico lenguaje C, el cual fue creado en la década de los 70 por la mano del finado Dennis Ritchie para la programación del sistema operativo (un sistema parecido a Unix es GNU/Linux), el cual surgió como un lenguaje orientado a la programación de sistemas (System Programming) y de herramientas (Utilities) recomendado sobre todo para programadores expertos, y que no llevaba implementadas muchas funciones que hacen a un lenguaje más comprensible.
Sin embargo, aunque esto en un inicio se puede convertir en un problema, en la práctica es su mayor virtud, ya que permite al programador un mayor control sobre lo que está haciendo. Años más tarde, un programador llamado Bjarne Stroustrup, creo lo que se conoce como C++.
Necesitaba ciertas facilidades de programación, incluidas en otros lenguajes pero que C no soportaba, al menos directamente, como son las llamadas clases y objetos, principios usados en la programación actual. Para ello rediseñó C, ampliando sus posibilidades pero manteniendo su mayor cualidad, la de

permitir al programador en todo momento tener controlado lo que está haciendo, consiguiendo así una mayor rapidez que no se conseguiría en otros lenguajes.
C++ pretende llevar a C a un nuevo paradigma de clases y objetos con los que se realiza una comprensión más humana basándose en la construcción de objetos, con características propias solo de ellos, agrupados en clases. Es decir, si yo quisiera hacer un programa sobre animales, crearía una clase llamada animales, en la cual cada animal, por ejemplo un pato, sería un objeto, de tal manera que se ve el intento de esta forma de programar por ser un fiel reflejo de cómo los humanos (en teoría) manejamos la realidad.
Se dice que nuestro cerebro trabaja de forma relacional (relacionando hechos), es por ello que cada vez que recuerdas algo, (cuentas un hecho), termina siendo diferente (se agregan u omiten partes).

1.1.3 ¿Qué es C++?

C++ es un lenguaje de programación orientado a objetos que toma la base del lenguaje C y le agrega la capacidad de abstraer tipos como en Smalltalk.
La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitieran la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.
Posteriormente se añadieron facilidades de programación genérica, que se sumó a los otros dos paradigmas que ya estaban admitidos (programación estructurada y la programación orientada a objetos). Por esto se suele decir que el C++ es un lenguaje de programación multiparadigma.

1.1.4 Herramientas Necesarias

Las principales herramientas necesarias para escribir un programa en C++ son las siguientes:

1. Un equipo ejecutando un sistema operativo.
2. Un compilador de C++
   - Windows MingW (GCC para Windows) o MSVC (compilador de microsoft con versión gratuita)
   - Linux (u otros UNIX): g++
   - Mac (con el compilador Xcode)
3. Un editor cualquiera de texto, o mejor un entorno de desarrollo (IDE)
   - Windows:
     - Microsoft Visual C++ (conocido por sus siglas MSVC). Incluye compilador y posee una versión gratuita (versión express)
     - Bloc de notas (no recomendado)
     - Editor Notepad++
     - DevCpp (incluye MingW - en desuso, no recomendado, incluye también un compilador)
     - Code::Blocks
   - Linux (o re-compilación en UNIX):
     - Gedit
     - Kate
     - KDevelop
     - Code::Blocks
     - SciTE
     - GVim

- Mac:
  - Xcode (con el compilador trae una IDE para poder programar)

4. Tiempo para practicar
5. Paciencia

Adicional

- Inglés (Recomendado)
- Estar familiarizado con C u otro lenguaje derivado (PHP, Python, etc).

Es recomendable tener conocimientos de C, debido a que C++ es una mejora de C, tener los conocimientos sobre este te permitira avanzar mas rapido y comprender aun mas. Tambien, hay que recordar que C++, admite C, por lo que se puede programar (reutilizar), funciones de C que se puedan usar en C++.

Aunque No es obligacion aprender C, es recomendable tener nociones sobre la programación orientada a objetos en el caso de no tener conocimientos previos de programación estructurada. Asimismo, muchos programadores recomiendan no saber C para saber C++, por ser el primero de ellos un lenguaje imperativo o procedimental y el segundo un lenguaje de programación orientado a objetos.

1.1.5 Consejos iniciales antes de programar

Con la práctica, se puede observar que se puede confundir a otros programadores con el código que se haga. Antes de siquiera hacer una línea de código, si se trabaja con otros programadores, ha de tenerse en cuenta que todos deben escribir de una forma similar el código, para que de forma global puedan corregir el código en el caso de que hubieran errores o rastrearlos en el caso de haberlos.

También es muy recomendable hacer uso de comentarios (comenta todo lo que puedas, hay veces que lo que parece obvio para ti, no lo es para los demás) y tratar de hacer un código limpio y comprensible, especificando detalles y haciendo tabulaciones, aunque te tome un poco mas de tiempo, es posible que mas adelante lo agradezcas tu mismo.

1.1.6 Ejemplos

Codigo 1.1: Ejemplo de C++

```cpp
#include <iostream>

using namespace std;

int main()
{
    int numero;
    cin>>numero;
    if(numero%2==0){
        cout<<"El numero es par\n";
    }else{
        cout<<"EL numero es impar\n";
    }
    return 0;
}
```

1.2 Lo mas basico

1.2.1 Proceso de desarrollo de un programa

Si se desea escribir un programa en C++ se debe ejecutar como mínimo los siguientes pasos:

1. Escribir con un editor de texto plano un programa sintácticamente válido o usar un entorno de desarrollo (IDE) apropiado para tal fin
2. Compilar el programa y asegurarse de que no han habido errores de compilación
3. Ejecutar el programa y comprobar que no hay errores de ejecución

Este último paso es el más costoso, por que en programas grandes, averiguar si hay o no un fallo prácticamente puede ser una tarea totémica.
Un archivo de C++ tiene la extención *cpp* a continuación se escribe el siguiente codigo en c++ del archivo 'hola.cpp'

Codigo 1.2: Hola Mundo

```
1  // Aquí generalmente se suele indicar qué se quiere con el programa a hacer
2  // Programa que muestra 'Hola mundo' por pantalla y finaliza
3
4  // Aquí se sitúan todas las librerias que se vayan a usar con include,
5  // que se verá posteriormente
6  #include <iostream> // Esta libreria permite mostrar y leer datos por consola
7
8  int main()
9  {
10      // Este tipo de líneas de código que comienzan por '//' son comentarios
11      // El compilador los omite, y sirven para ayudar a otros programadores o
12      // a uno mismo en caso de volver a revisar el código
13      // Es una práctica sana poner comentarios donde se necesiten,
14
15      std::cout << "Hola Mundo" << std::endl;
16
17      // Mostrar por std::cout el mensaje Hola Mundo y comienza una nueva línea
18
19      return 0;
20
21      // se devuelve un 0.
22      //que en este caso quiere decir que la salida se ha efectuado con éxito.
23 }
```

Mediante simple inspección, el código parece enorme, pero el compilador lo único que leerá para la creación del programa es lo siguiente:

Codigo 1.3: Hola Mundo Compilado

```
1  #include <iostream>
2  int main(void){ std::cout << "Hola Mundo" << std::endl; return 0; }
```

Como se puede observar, este código y el original no difieren en mucho salvo en los saltos de línea y que los comentarios, de los que se detallan posteriormente, están omitidos y tan sólo ha quedado "el esqueleto" del código legible para el compilador. Para el compilador, todo lo demás, sobra.

Aquí otro ejemplo

Codigo 1.4: Hello World

```
#include <iostream>

int main()
{
    std::cout << "Hola Mundo" << std::endl;
    std::cout << "Hello World" << std::endl;
    std::cout << "Hallo Welt" << std::endl;
    return 0;
}
```

Para hacer el código mas corto debemos incluir *using namespace std* con lo que le estamos diciendo al compilador que usaremos el espacio de nombres std por lo que no tendremos que incluirlo cuando usemos elementos de este espacio de nombres, como pueden ser los objetos cout y cin, que representan el flujo de salida estándar (típicamente la pantalla o una ventana de texto) y el flujo de entrada estándar (típicamente el teclado).
Se veria de la siguiente manera:

Codigo 1.5: Using Namespace Std

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hola Mundo" << endl;
    cout << "Hello World" << endl;
    cout << "Hallo Welt" << endl;
    return 0;
}
```

Los pasos siguientes son para una compilación en GNU o sistema operativo Unix, para generar el ejecutable del programa se compila con g++ de la siguiente forma:

Codigo 1.6: Compilar

```
| g++ hola.cpp -o hola
```

Para poder ver los resultados del programa en acción, se ejecuta el programa de la siguiente forma:

Codigo 1.7: Ejecutar

```
|./hola
```

Y a continuación se debe mostrar algo como lo siguiente:

Codigo 1.8: Resultado

```
Hola Mundo
```

1.2.2 Sintaxis

Sintaxis es la forma correcta en que se deben escribir las instrucciones para el computador en un lenguaje de programación específico. C++ hereda la sintaxis de C estándar, es decir, la mayoría de programas escritos para el C estándar pueden ser compilados en C++.

El punto y coma

El punto y coma es uno de los simbólos más usados en C, C++; y se usa con el fin de indicar el final de una línea de instrucción. El punto y coma es de uso obligatorio.

Ejemplo

Codigo 1.9: Sintaxis

```
1  clrscr(); //Limpiar pantalla, funciona con la librería conio de Borland C++
2  x = a + b;
3
4  string IP = "127.0.0.1"; // Variable IP tipo string
5  cout << IP << endl; // Devuelve 127.0.0.1
6
7  char Saludo[5] = "Hola"; // Variable Saludo tipo char
8  cout << Saludo[0] << endl; // Igual a H
9  cout << Saludo[1] << endl; // Igual a o
10 cout << Saludo[2] << endl; // Igual a l
11 cout << Saludo[3] << endl; // Igual a a
```

El punto y coma se usa también para separar contadores, condicionales e incrementadores dentro de un sentencia for

Ejemplo

Codigo 1.10: Sintaxis

```
1  for (i=0; i < 10; i++) cout << i;
```

Espacios y tabuladores

Usar caracteres extras de espaciado o tabuladores (caracteres tab) es un mecanismo que nos permite ordenar de manera más clara el código del programa que estemos escribiendo, sin embargo, el uso de estos es opcional ya que el compilador ignora la presencia de los mismos. Por ejemplo, el segundo de los ejemplos anteriores se podría escribir como:

Codigo 1.11: Sintaxis

```
1  for (int i=0; i < 10; i++) { cout << i * x; x++; }
```

y el compilador no pondría ningún reparo.

1.2.3 Comentarios

Existen dos modos basicos para comentar en c++:

//
Comentan solo una linea de código
/*Comentario*/
Comentan estrofas de código
A continuación un ejemplo:

Codigo 1.12: Comentarios

```
#include <iostream>

using namespace std;

int main()
{
    /*
    cout    es para imprimir
    <<      se utiliza para separar elementos para cout
    endl    es un salto de linea
    */

    //En español
    cout << "Hola Mundo" << endl;
    //En ingles
    cout << "Hello World" << endl;
    //En aleman
    cout << "Hallo Welt" << endl;
    return 0;
}
```

1.2.4 Datos primitivos

En un lenguaje de programación es indispensable poder almacenar información, para esto en C++ están disponibles los siguientes tipos que permiten almacenar información numérica de tipo entero o real:

Nombre	Descripción	Tamaño	Rango de valores
bool	Valor booleano	1byte	true o false
char	Carácter o entero pequeño	1byte	De -128 a 127
short int	Entero corto	2bytes	De -32768 a 32767
int	Entero	4bytes	De -2147483648 a 2147483647
long long	Entero largo	8bytes	-9223372036854775808 a 9223372036854775807
float	Número de punto flotante	4bytes	3.4e +/- 38 (7 digitos)
double	Float con doble precisión	8bytes	1.7e +/- 308 (15 digitos)

Los valores dependen de la arquitectura utilizada. Los mostrados son los que generalmente se encuentran en una máquina típica de arquitectura 32 bits.

El Modificador unsigned

El modificador unsigned es utilizado únicamente con los enteros, su utilización permite utilizar en los enteros únicamente la parte positiva,

Codigo 1.13: Unsigned

```
1 int a;                    // Almacena valores entre -32768 y 32767
2 unsigned int a; // Almacena valores entre 0 y 65535
```

1.2.5 Variables y constantes

Una variable, como su nombre lo indica, es un determinado objeto cuyo valor puede cambiar durante el proceso de una tarea específica. Contrario a una variable, una constante es un determinado objeto cuyo valor no puede ser alterado durante el proceso de una tarea específica. En C, C++ para declarar variables no existe una palabra especial, es decir, las variables se declarán escribiendo el tipo seguido de uno o más identificadores o nombres de variables. Por otro lado, para declarar constantes existe la palabra reservada const, así como la directiva *#define*. A continuación se muestran ejemplos de declaración de variables y constantes.

Variables	Constantes	Constantes
int a;	const int a = 100;	#define a 100
float b;	const float b = 100;	#define b 100

Notas:
A diferencia de las constantes declaradas con la palabra const los símbolos definidos con *#define* no ocupan espacio en la memoria del código ejecutable resultante.
El tipo de la variable o constante puede ser cualquiera de los listados en Tipos primitivos, o bien de un tipo definido por el usuario.
Las constantes son usadas a menudo con un doble propósito, el primero es con el fin de hacer más legible el código del programa, es decir, si se tiene (por ejemplo) la constante numerica 3.1416 y esta representa al número pi, entonces podemos hacer declaraciones tales como:

Codigo 1.14: Constantes

```
1 #define pi 3.1416
```

En este caso podremos usar la palabra pi en cualquier parte del programa y el compilador se encargará de cambiar dicho simbolo por 3.1416. o bien,

Codigo 1.15: Constantes

```
1 const pi = 3.1416;
```

En este otro caso podremos usar la palabra pi en cualquier parte del programa y el compilador se encargará de cambiar dicho símbolo por una referencia a la constante pi guardada en la memoria.

1.2.6 Lectura e Impresión

La iostream
La iostream es la biblioteca estándar en C++ para poder tener acceso a los dispositivos estándar de entrada y/o salida. En sus programas, si usted desea hacer uso de los objetos cin, cout, cerr y clog tendrá que incluir (por medio de la directiva *#include*) el uso de la biblioteca iostream. En la iostream se encuentran definidas las clases ios (misma que es la base para las clases que implementen operaciones de entrada y/o salida de datos), istream (para operaciones de entrada) y

ostream (para operaciones de salida). Aparte de las clases mencionadas, en la iostream se encuentra una lista de variables y constantes (atributos) que son accesibles por el usuario a través del operador de ámbito (::) si es que no se incluyera *using namespace std*.
Si usted usa la directiva #include <iostream.h> o #include <iostream> en sus programas, automáticamente la iostream pone a su disposición los objetos cin, cout, clog y cerr en el ámbito estándar (std), de tal manera que usted puede comenzar a enviar o recibir información a través de los mismos sin siquiera preocuparse de su creación. Asi, un sencillo ejemplo del uso de los objetos mencionados se muestra en seguida.

Codigo 1.16: Iostream

```
// De nuevo con el hola mundo...
#include <iostream.h>
int main()
{
     std::cout << "Hola mundo";     // imprimir mensaje (en la pantalla)
     std::cin.get();     // lectura ( entrada del teclado )
     return 0;
}
```

Operadores de direccionamiento
Los operadores de direccionamiento son los encargados de manipular el flujo de datos desde o hacia el dispositivo referenciado por un stream específico. El operador de direccionamiento para salidas es una pareja de símbolos de "menor que" «, y el operador de direccionamiento para entradas es una pareja de símbolos de "mayor que" ». Los operadores de direccionamiento se colocan entre dos operandos, el primero es el Stream y el segundo es una variable o constante que proporciona o recibe los datos de la operación. Por ejemplo, en el siguiente programa y en la instrucción cout « "Entre su nombre: "; la constante "Entre su nombre: " es la fuente o quien proporciona los datos para el objeto cout. Mientras que en la instrucción cin » nombre la variable nombre es el destino o quien recibe los datos provenientes del objeto cin.

Codigo 1.17: Oper Direc

```
// De nuevo con el hola mundo...
#include <iostream.h>
int main()
{
     char nombre[80];
     cout << "Entre su nombre: ";
     cin  >> nombre;
     cout << "Hola," << nombre;
     cin.get();
     return 0;
}
```

Observe que si en una misma línea de comando se desea leer o escribir sobre varios campos a la vez, no es necesario nombrar más de una vez al stream. Ejemplos:

Codigo 1.18: Ejemplo Cin Cout

```
cout << "Hola," << nombre;
cin >> A >> B >> C;
```

Otro ejemplo del manejo de *cin*:

Codigo 1.19: Impresión

```
// Programa que muestra diversos textos por consola

// Las librerías del sistema usadas son las siguientes
#include <iostream>

using namespace std;
// Es la función principal encargada de mostrar por consola diferentes textos
int main(void)
{
    // Ejemplo con una única línea, se muestra el uso de cout y endl
    cout << "Bienvenido. Soy un programa. Estoy en una linea de
       codigo." << endl;

    // Ejemplo con una única línea de código que se puede fraccionar
    // mediante el uso de '«'
    cout << "Ahora "
         << "estoy fraccionado en el codigo, pero en la consola me
            muestro como una unica frase."
         << endl;

    // Uso de un código largo, que cuesta leer para un programador,
    // y que se ejecutará sin problemas.
    // *** No se recomienda hacer líneas de esta manera,
    // esta forma de programar no es apropiada ***
    cout << "Un gran texto puede ocupar muchas lineas."
         << endl
         << "Pero eso no frena al programador a que todo se pueda
            poner en una unica linea de codigo y que"
         << endl
         << "el programa, al ejecutarse, lo situe como el
            programador quiso"
         << endl;

    return 0; // Y se termina con éxito.
}
```

Consola:

Codigo 1.20: Resultado Impresión

```
Bienvenido. Soy un programa. Estoy en una linea de codigo.
Ahora estoy fraccionado en el codigo, pero en la consola me muestro como una unica
    frase.
Un gran texto puede ocupar muchas lineas.
Pero eso no frena al programador a que todo se pueda poner en una unica linea de
    codigo y que
```

```
el programa, al ejecutarse, lo situe como el programador quiso
```

1.2.7 Operadores

El C++ está lleno de operadores. Presentamos aquí una tabla de los mismos indicando el rango de prioridad de cada uno, y cómo se ejecutan. A continuación comentaremos brevemente los operadores.

Operadores (de mayor a menor prioridad)	Sentido
( ) { }	I-D
! ++ – * &	D-I
* / %	I-D
+ -	I-D
« »	I-D
< <= > >=	I-D
== !=	I-D
&	I-D
^	I-D
|	I-D
&& (and)	I-D
|| (or)	I-D
?: (Operador Condicional Ternario)	I-D
= += -= *= /= %=	D-I

La acción de estos operadores es la siguiente:

Operadores Aritméticos

+ Suma los valores situados a su derecha y a su izquierda.
- Resta el valor de su derecha del valor de su izquierda.
- Como operador unario, cambia el signo del valor de su izquierda.
***** Multiplica el valor de su derecha por el valor de su izquierda.
/ Divide el valor situado a su izquierda por el valor situado a su dercha.
% Proporciona el resto de la división del valor de la izquierda por el valor de la derecha (sólo enteros).
++ Suma 1 al valor de la variable situada a su izquierda (modo prefijo) o de la variable situada a su derecha (modo sufijo).
– Igual que ++, pero restando 1.

Operadores de Asignación

= Asigna el valor de su derecha a la variable de su izquierda.
Cada uno de los siguientes operadores actualiza la variable de su izquierda con el valor de su derecha utilizando la operación indicada. Usaremos de *d* e *i* para iquierda.
+= Suma la cantidad *d* a la variable *i*.
-= Resta la cantidad *d* de la variable *i*.
***=** Multiplica la variable *i* por la variable *d*.
/= Divide la variable *i* entre la cantidad *d*.

%= Proporciona el resto de la división de la variable *i* por la cantidad *d*
Ejemplo
conejos *= 1.6; //es lo mismo que conejos = conejos * 1.6;

Operadores de Relación

Cada uno de estos operadores compara el valor de su izquierda con el valor de su dercha. La expresión de relación formada por un operador y sus dos operandos toma el valor 1 si la expresión es cierta, y el valor 0 si es falsa.
< menor que
<= menor o igual que
== igual a
>= mayor o igual que
> mayor que
!= distinto de

Operadores Lógicos

Los operadores lógicos utilizan normalmenteexpresiones de relación como operadores. El operador *!* toma un operando situado a su derecha; el resto toma dos: uno a su derecha y otro a su izquierda.
&& and La expresión combinada es cierta si ambos operandos lo son, y falsa en cualquier otro caso.
|| or La expresión combinada es cierta si uno o ambos operandos lo son, y falsa en cualquier otro caso.
! not La expresión es cierta si el operador es falso, y viceversa.

Operadores Relacionados con punteros

& Operador de Dirección Cuando va seguido por el nombre de una variable, entrega la dirección de dicha variable *&abc* es la dirección de la variable *abc*.
*** Operador de Indirección** Cuando va seguido por un puntero, entrega el valor almacenado en la dirección apuntada por él *abc = 22; def = &abc; val = *def* el efecto es asignar a *val* el valor de 22.

Operadores de Estructuras y Uniones

El operador de pertenencia (punto) se utiliza junto con el nombre de la estructura o unión, para especificar un miembro de las mismas. Si tenemos una estructura cuyo *nombre* es nombre, y *miembro* es un miembro especificado por el patrón de la estructura, **nombre.miembro** identifica dicho miembro de la estructura. El operador de pertenencia puede utilizarse de la misma forma en uniones. Ejemplo

Operadores Lógicos de Bits

Misceláneas

1.3 Estructuras de Control

Las estructuras de control permiten modificar el flujo de ejecución de las instrucciones de un programa.
Con las estructuras de control se puede:

- De acuerdo a una condición, ejecutar un grupo u otro de sentencias (**If**)
- De acuerdo al valor de una variable, ejecutar un grupo u otro de sentencias (**Switch**)
- Ejecutar un grupo de sentencias mientras se cumpla una condición (**While**)

- Ejecutar un grupo de sentencias hasta que se cumpla una condición (**Do-While**)
- Ejecutar un grupo de sentencias un número determinado de veces (**For**)

Todas las estructuras de control tienen un único punto de entrada y un único punto de salida.

1.3.1 Sentencias de decisión

Definición

Las sentencias de decisión o también llamadas de CONTROL DE FLUJO son estructuras de control que realizan una pregunta la cual retorna verdadero o falso (evalúa una condicion) y selecciona la siguiente instrucción a ejecutar dependiendo la respuesta o resultado.

Sentencia if

La instrucción if es, por excelencia, la más utilizada para construir estructuras de control de flujo.

Sintaxis

Primera Forma

Ahora bién, la sintaxis utilizada en la programación de C++ es la siguiente:

Codigo 1.21: Sintaxis If

```
1  if (condicion)
2     {
3          Set de instrucciones
4     }
```

siendo *"condicion"* el lugar donde se pondrá la condición que se tiene que cumplir para que sea verdadera la sentencia y así proceder a realizar el *"set de instrucciones"* o código contenido dentro de la sentencia.

Segunda Forma

Ahora veremos la misma sintaxis pero ahora le añadiremos la parte *"Falsa"* de la sentencia:

Codigo 1.22: Sintaxis If Else

```
1  if (condicion)
2     {
3          Set de instrucciones     //Parte VERDADERA
4     }
5  else
6     {
7          Set de instrucciones 2   //Parte FALSA
8     }
```

La forma mostrada anteriormente muestra la union de la parte *"VERDADERA"* con la nueva secuencia la cual es la parte *"FALSA"* de la sentencia de decision *"IF"* en la cual esta compuesta por el:

Codigo 1.23: Sintaxis Else

```
1  else
2     {
```

```
3       Set de instrucciones 2   //Parte FALSA
4    }
```

la palabra *"else"* o *"De lo contrario"* indica al lenguaje que de lo contrario al no ser verdadera o no se cumpla la parte verdadera entonces realizara el *"set de instrucciones 2"*.

Ejemplos De Sentencias If

Ejemplo 1:

Codigo 1.24: If Ejemplo 1

```
1  if(numero == 0) //La condicion indica que tiene que ser igual a Cero
2      {
3        cout<<"El Numero Ingresado es Igual a Cero";
4      }
```

Ejemplo 2:

Codigo 1.25: If Ejemplo 2

```
1  if(numero > 0) // la condicion indica que tiene que ser mayor a Cero
2      {
3        cout<<"El Numero Ingresado es Mayor a Cero";
4      }
```

Ejemplo 3:

Codigo 1.26: If Ejemplo 3

```
1  if(numero < 0) // la condicion indica que tiene que ser menor a Cero
2      {
3        cout<<"El Numero Ingresado es Menor a Cero";
4      }
```

Ahora uniremos todos estos ejemplos para formar un solo programa mediante la utilización de la sentencia "Else" e introduciremos el hecho de que se puede escribir en este espacio una sentencia if ya que podemos ingresar cualquier tipo de código dentro de la sentencia escrita después de un Else.
Ejemplo 4:

Codigo 1.27: If Ejemplo 4

```
1  if(numero == 0) //La condicion indica que tiene que ser igual a Cero
2     {
3       cout<<"El Numero Ingresado es Igual a Cero";
4     }
5  else
6     {
7        if(numero > 0) // la condicion indica que tiene que ser mayor a Cero
8        {
9            cout<<"El Numero Ingresado es Mayor a Cero";
```

```
10          }
11          else
12          {
13              if(numero < 0) // la condicion indica que tiene que ser menor a Cero
14              {
15                cout<<"El Numero Ingresado es Menor a Cero";
16              }
17          }
18      }
```

Sentencia switch

switch es otra de las instrucciones que permiten la construcción de estructuras de control. A diferencia de if, para controlar el flujo por medio de una sentencia switch se debe de combinar con el uso de las sentencias *case* y *break*.
Notas: cualquier número de casos a evaluar por switch así como la sentencia default son opcionales. La sentencia switch es muy útil en los casos de presentación de menus.

Sintaxis

Ahora bién, la sintaxis utilizada en la programación de C++ es la siguiente:

Codigo 1.28: Sintaxis Switch

```
switch (condicion)
{
    case primer_caso:
         bloque de instrucciones 1
    break;

    case segundo_caso:
         bloque de instrucciones 2
    break;

    case caso_n:
         bloque de instrucciones n
    break;

    default: bloque de instrucciones por defecto
}
```

Ejemplos De Sentencias Switch

Ejemplo 1

Codigo 1.29: Switch Ejemplo 1

```
switch (numero)
{
    case 0: cout << "numero es cero";
}
```

Ejemplo 2

Codigo 1.30: Switch Ejemplo 2

```
switch (opcion)
{
    case 0: cout << "Su opcion es cero"; break;
    case 1: cout << "Su opcion es uno"; break;
    case 2: cout << "Su opcion es dos";
}
```

Ejemplo 3

Codigo 1.31: Switch Ejemplo 3

```
switch (opcion)
{
    case 1: cout << "Su opcion es 1"; break;
    case 2: cout << "Su opcion es 2"; break;
    case 3: cout << "Su opcion es 3"; break;
    default: cout << "Elija una opcion entre 1 y 3";
}
```

Operador condicional ternario ?:

En C/C++, existe el operador condicional (?:) el cual es conocido por su estructura como ternario. El comportamiento de dicho operador es el mismo que una estructura if - then - else del lenguaje BASIC (y de la función IIf de Visual Basic). El operador condicional ?: es útil para evaluar situaciones tales como:
Si se cumple tal condición entonces haz esto, de lo contrario haz esto otro.

Sintaxis

Codigo 1.32: Sintaxis Switch

```
( (condicion) ? proceso1 : proceso2 )
```

En donde, condicion es la expresión que se evalua, proceso1 es la tarea a realizar en el caso de que la evaluación resulte verdadera, y proceso2 es la tarea a realizar en el caso de que la evaluación resulte falsa.

Ejemplos De Sentencias Operador condicional ternario ?:

Ejemplo 1

Codigo 1.33: Condicional Ejemplo 1

```
int edad;
cout << "Cual es tu edad: ";
cin >> edad;
```

```
cout << ( (edad < 18) ? "Eres joven aun" : "Ya tienes la mayoria de
    edad" );
El ejemplo anterior podria escribirse de la siguiente manera:

int edad;
cout << "Cual es tu edad: ";
cin >> edad;
if (edad < 18) cout << "Eres joven aun";
    else cout << "Ya tienes la mayoria de edad";
```

Ejemplo 2
Vamos a suponer que deseamos escribir una función que opere sobre dos valores numéricos y que la misma ha de regresar 1 (true) en caso de que el primer valor pasado sea igual al segundo valor; en caso contrario la función debe retornar 0 (false).

Codigo 1.34: Condicional Ejemplo 2

```
int es_igual( int a, int b)
{
    return ( (a == b) ? 1 : 0 )
}
```

1.3.2 Sentencias de iteración

Definición

Las Sentencias de Iteración o Ciclos son estructuras de control que repiten la ejecución de un grupo de instrucciones. Básicamente, una sentencia de iteración es una estructura de control condicional, ya que dentro de la misma se repite la ejecución de una o más instrucciones mientras que una a condición especifica se cumpla. Muchas veces tenemos que repetir un número definido o indefinido de veces un grupo de instrucciones por lo que en estos casos utilizamos este tipo de sentencias. en C++ los ciclos o bucles se construyen por medio de las sentencias for, while y do - while. La sentencia for es útil para los casos en donde se conoce de antemano el número de veces que una o más sentencias han de repetirse. Por otro lado, la sentencia while es útil en aquellos casos en donde no se conoce de antemano el número de veces que una o más sentencias se tienen que repetir.

Sentencias For

Ahora bién, la sintaxis utilizada en la programación de C++ es la siguiente:

Sintaxis

Codigo 1.35: Sintaxis For

```
for(contador; final; incremento)
{
    Codigo a Repetir;
}
```

donde:

contador es una variable numérica
final es la condición que se evalua para finalizar el ciclo (puede ser independiente del contador)
incremento es el valor que se suma o resta al contador
Hay que tener en cuenta que el *"for"* evalua la condición de finalización igual que el while, es decir, mientras esta se cumpla continuaran las repeticiones.

Ejemplos De Sentencias For

Ejemplo 1

Codigo 1.36: For Ejemplo 1

```
1 for(int i=1; i<=10; i++)
2 {
3     cout<<"Hola Mundo";
4 }
```

Esto indica que el contador *"i"* inicia desde 1 y continuará iterando mientras i sea menor o igual a 10 (en este caso llegará hasta 10) e *"i++"* realiza la sumatoria por unidad lo que hace que el for y el contador se sumen. repitiendo 10 veces *"HOLA MUNDO"* en pantalla.
Ejemplo 2

Codigo 1.37: For Ejemplo 2

```
1 for(int i=10; i>=0; i--)
2 {
3     cout<<"Hola Mundo";
4 }
```

Este ejemplo hace lo mismo que el primero, salvo que el contador se inicializa a 10 en lugar de 1; y por ello cambia la condición que se evalua así como que el contador se decrementa en lugar de ser incrementado.
La condición también puede ser independiente del contador:
Ejemplo 3

Codigo 1.38: For Ejemplo 3

```
1 int j = 20;
2 for(int i=0; j>0; i++){
3     cout<<"Hola"<<i<<" - "<<j<<endl;
4     j--;
5 }
```

En este ejemplo las iteraciones continuaran mientras j sea mayor que 0, sin tener en cuenta el valor que pueda tener i.

Sentencia While

Ahora bién, la sintaxis utilizada en la programación de C++ es la siguiente:

Sintaxis

Codigo 1.39: Sintaxis While

```
while(condicion)
{
    codigo a Repetir
}
```

donde:
condicion es la expresión a evaluar

Ejemplos De Sentencias While

Ejemplo 1

Codigo 1.40: While Ejemplo 1

```
int contador = 0;

while(contador<=10)
{
    contador=contador+1;
    cout<<"Hola Mundo";
}
```

El contador Indica que hasta que este llegue a el total de 10 entonces se detendrá y ya no se realizará el código contenido dentro de la sentencia while, de lo contrario mientras el *"contador"* sea menor a 10 entonces el código contenido se ejecutará desplegando hasta 10 veces "Hola Mundo" en pantalla.

Sentencia Do - While

Sintaxis

La sentencia do es usada generalmente en cooperación con while para garantizar que una o más instrucciones se ejecuten al menos una vez. Por ejemplo, en la siguiente construcción no se ejecuta nada dentro del ciclo while, el hecho es que el contador inicialmente vale cero y la condición para que se ejecute lo que está dentro del while es *"mientras el contador sea mayor que diez"*. Es evidente que a la primera evaluación hecha por while la condición deja de cumplirse.

Codigo 1.41: Sintaxis Do While

```
int contador = 0;

while(contador > 10)
{
    contador ++;
    cout<<"Hola Mundo";
}
```

Al modificar el segmento de código anterior usando do tenemos:

Codigo 1.42: Sintaxis Do While

```
int contador = 0;

do
{
    contador ++;
    cout<<"Hola Mundo";
}
while(contador > 10);
```

Observe cómo en el caso de do la condición es evaluada al final en lugar de al principio del bloque de instrucciones y, por lo tanto, el código que le sigue al *do* se ejecuta al menos la primera vez.

1.3.3 Sentencias Break y Continue

En la sección (Sentencia switch) vimos que la sentencia break es utilizada con el propósito de forzar un salto dentro del bloque switch hacia el final del mismo. En esta sección volveremos a ver el uso de break, salvo que esta ocasión la usaremos junto con las sentecias for y la sentencia while. Además, veremos el uso de la sentencia continue.

Break

La sentencia break se usa para forzar un salto hacia el final de un ciclo controlado por for o por while.

Ejemplo

En el siguiente fragmento de código la sentencia break cierra el ciclo for cuando la variable (i) es igual a 5.

Codigo 1.43: Break

```
for (int i=0; i<10; i++) {
    if (i == 5) break;
    cout << i << " ";
}
```

La salida para el mismo será:

Codigo 1.44: Resultado Break

```
0 1 2 3 4
```

Continue

La sentencia continue se usa para ignorar una iteración dentro de un ciclo controlado por for o por while.

Ejemplo

En el siguiente fragmento de código la sentencia continue ignora la iteración cuando la variable (i) es igual a 5.

Codigo 1.45: Continue

```
for (int i=0; i<10; i++) {
    if (i == 5) continue;
    cout << i << " ";
}
```

La salida para el mismo será:

Codigo 1.46: Resultado Continue

```
0 1 2 3 4 6 7 8 9
```

Uso de break y continue junto con while

Los dos ejemplos anteriores se presentan en seguida, salvo que en lugar de for se hace uso de while. **Nota:** no deje de observar que la construcción del ciclo while para el caso de la sentencia continue es diferente, esto para garantizar que el ciclo no vaya a caer en una iteración infinita.

Break

Codigo 1.47: Break While

```
int i = 0;
while (i<10) {
    if (i == 5) break;
    cout << i << " ";
    i++;
}
```

Continue

Codigo 1.48: Continue While

```
int i = -1;
while (i<10) {
    i++;
    if (i == 5) continue;
    cout << i << " ";
}
```

2. Estructura de datos 1

3. Programación modular

3.1 Introduccion

3.1.1 ¿Qué es la Programación Modular?

La programacion modular es un paradigma (modelo) de programacion, que consiste en dividir un problema en subproblemas con el fin de simplificarlo.
Aplicando la Programacion Modular podemos llevar problemas grandes y tediosos, a pequeños subproblemas, y estos a su ves en otros, hasta poder resolverlos facilmente con un lenguaje de programacion, a esta tecnica la llamaremos divide y venceras.
Un modulo es cada una de las partes del programa que resuelve un subproblema en las que se dividio el problema original.

3.2 Modulos

3.2.1 Concepto de Modulo

Un Modulo es una segmento de codigo separado del bloque principal que puede ser invocado en cualquier momento desde este o desde otro modulo.

3.2.2 Elementos de declaracion del Modulo

Un Moduo o subrutina se declara, generalmente, por:

- Un nombre unico con el que se lo identifica y distingue de otros.
- Un tipo de dato de retorno.
- Una lista de parametros, puede ser cero, uno o varios.
- Conjunto de ordenes que debe ejecutar la subrutina.

3.2.3 Ejemplos

El siguiente ejemplo muestra una subrutina que tiene un parametro de entrada (x) del cual calcula su factorial y lo devuelve como parametro de salida.
El factorial de un numero n se define como el producto de todos los numeros desde 1 hasta n, y se simboliza n!.

$$5! = 1x2x3x4x5 = 120$$

Codigo 3.1: Factorial

```
#include <iostream>
using namespace std;
//"int" Tipo de dato de retorno.
//"fact" Nombre de la subrutina.
//"(int x)" Paranetros que recibe.
int fact(int x){
        //Ordenes que ejecuta
        int f=1
        for(int i=1; i<=x;i++)
                f=f*i;
        return f;
}
int main(){
        int n;
        cin>>n;
        cout<<fact(n)<<endl;
        return 0;
}
```

3.3 Recursividad

3.3.1 Que es recursvidad?

La recursividad es una tecnica de programacion en la que un Modulo hace una llamada a si mismo con el fin de resolver el problema. La llamada a si mismo se conoce como llamada recursiva.
Dicho formalmente, un algoritmo se dice recursivo si calcula instancias de un problema en función de otras instancias del mismo problema hasta llegar a un caso base, que suele ser una instancia pequeña del problema, cuya respuesta generalmente está dada en el algoritmo y no es necesario calcularla.
Aun asi la definicion puede ser confusa, por eso ahora presentaremos 2 ejemplos que ilustran la recursividad:

3.3.2 Ejemplos

- Al igual que en el primer ejemplo tendremos una subrutina o funcion que calcule el factorial de un numero, en este caso lo hara de manera recursiva.

Codigo 3.2: Factorial Recursivo

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

int fact(int n){
        /*
        Caso base: donde el algoritmo se detendra por que tendra
        un valor conocido y no necesitara calcularlo pues 1!=1
        */
        if(n==1) return 1;
        /*
        Si el valor es diferente de 1, entonces es un valor que
```

```
12          necesitamos calcular, por la definicion de factorial
13          podemos apreciar que, por ejemplo: 5!=5x4! y que a su
14          vez 4!=4*3! y asi sucesivamente, esto es recursividad
15          ya que esta funcion se llama asi misma para calcular el
16          siguiente factorial.
17          */
18          return n*fact(n-1);
19  }
20  int main(){
21          int n;
22          cin>>n;
23          cout<<fact(n)<<endl;
24          return 0;
25  }
```

- Ahora calcularemos el n-simo termino de la sucesion Fibonacci por medio de un algoritmo recursivo.
  La sucecion Fibonacci, comienza con los valores 1 y 1 y a partir de estos cada termino es la suma de los 2 anteriores:

  $$1, 1, 2, 3, 5, 8, 13, 21, 34, ...$$

Codigo 3.3: Sucesion Fibonacci

```
1   #include <iostream>
2   using namespace std;
3
4   int Fibo(int n){
5           //Caso base:  El primer y segundo termino de la sucesion
6           //son 1.
7           if(n==1 or n==2) return 1;
8           //Por la definicion de la sucecion un termino es igual a
9           //la suma de los 2 terminos anteriores.
10          return Fibo(n-1)+Fibo(n-2);
11  }
12  int main(){
13          int n;
14          cin>>n;
15          cout<<Fibo(n)<<endl;
16          return 0;
17  }
```

4. Algoritmos Basicos

4.1 Algoritmos de Ordenanmiento

4.1.1 Ordenamiento Rapido

El ordenamiento rapido (Quicksort en ingles) es un algortimo basado en la tecnica divide y venceras, que permite ordenar n elementos en un tiempo proporcional a $n\log(n)$

Descripcion del Algoritomo

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento del vector a ordenar, al que llamaremos pivote.
- Mover todos los elementos menores que el pivote a un lado y los mayores al otro lado.
- El vector queda separado en 2 subvectores una con los elementos a la izquierda del pivote y otra con los de la derecha,
- Repetir este proceso recursivamente en las sublistas hasta que estas tengan un solo elemento.

Codigo en C++

Codigo 4.1: QuickSort

```
1  #include <iostream>
2  using namespace std;
3  int pivot(int a[], int first, int last) {
4      int  p = first;
5      int pivotElement = a[first];
6
7      for(int i = first+1 ; i <= last ; i++) {
8          if(a[i] <= pivotElement){
9              p++;
10             swap(a[i], a[p]);
11         }
12     }
13
14     swap(a[p], a[first]);
15
16     return p;
17 }
18 void quickSort( int a[], int first, int last ) {
19     int pivotElement;
20
```

```
21      if(first < last){
22          pivotElement = pivot(a, first, last);
23          quickSort(a, first, pivotElement-1);
24          quickSort(a, pivotElement+1, last);
25      }
26  }
27  void swap(int& a, int& b){
28      int temp = a;
29      a = b;
30      b = temp;
31  }
```

4.1.2 Ordenamiento por mezcla

El algoritmo de ordenamiento por mezcla (Merge Sort) es un algoritmo de ordenacion externo estable basado en la tecnica divide y venceras. Su complejidad es $O(n \log n)$

Descripcion del Algoritomo

Conceptualmente el algoritmo funciona de la siguiente manera:

- Si la longitud del vector es 1 o 0, entonces ya esta ordenado, en otro caso:
- Dividir el vector desordenado en dos subvectores de aproximadamente la mitad de tamaño.
- Ordenar cada subvector recursivamente aplicando el ordenamiento por mezcla.
- Mezclar los dos subvectores en un solo subvector ordenado.

Codigo en C++

Codigo 4.2: Merge Sort

```
1   #include <bits/stdc++.h>
2   using namespace std;
3
4   void mergeSort(int list[], int lowerBound, int upperBound){
5       int mid;
6
7       if (upperBound > lowerBound){
8           mid = ( lowerBound + upperBound) / 2;
9           mergeSort(list, lowerBound, mid);
10          mergeSort(list, mid + 1, upperBound);
11          merge(list, lowerBound, upperBound, mid);
12      }
13  }
14  void merge(int list[], int lowerBound, int upperBound, int mid){
15      int* leftArray = NULL;
16      int* rightArray = NULL;
17      int i, j, k;
18      int n1 = mid - lowerBound + 1;
19      int n2 = upperBound - mid;
20      leftArray = new int[n1];
21      rightArray = new int[n2];
22      for (i = 0; i < n1; i++)
23          leftArray[i] = list[lowerBound + i];
```

```
24      for (j = 0; j < n2; j++)
25          rightArray[j] = list[mid + 1 + j];
26
27      i = 0;
28      j = 0;
29      k = lowerBound;
30
31      while (i < n1 && j < n2){
32          if (leftArray[i] <= rightArray[j]){
33              list[k] = leftArray[i];
34              i++;
35          }
36          else{
37              list[k] = rightArray[j];
38              j++;
39          }
40
41          k++;
42      }
43      while (i < n1){
44          list[k] = leftArray[i];
45          i++;
46          k++;
47      }
48      while (j < n2){
49          list[k] = rightArray[j];
50          j++;
51          k++;
52      }
53      delete [] leftArray;
54      delete [] rightArray;
55  }
```

4.1.3 Ordenamiento por Monticulos

Descripcion del Algoritmo

Es un algoritmo de ordenacion no recursivo, con complejidad $O(n \log n)$

Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él. El algoritmo, después de cada extracción, recoloca en el nodo raíz o cima, la última hoja por la derecha del último nivel. Lo cual destruye la propiedad heap del árbol. Pero, a continuación realiza un proceso de "descenso" del número insertado de forma que se elige a cada movimiento el mayor de sus dos hijos, con el que se intercambia. Este intercambio, realizado sucesivamente "hunde" el nodo en el árbol restaurando la propiedad montículo del arbol y dejándo paso a la siguiente extracción del nodo raíz.

El algoritmo, en su implementación habitual, tiene dos fases. Primero una fase de construcción de un montículo a partir del conjunto de elmentos de entrada, y después, una fase de extracción

sucesiva de la cima del montículo. La implementación del almacén de datos en el heap, pese a ser conceptualmente un árbol, puede realizarse en un vector de forma fácil. Cada nodo tiene dos hijos y por tanto, un nodo situado en la posición i del vector, tendrá a sus hijos en las posiciones 2 x i, y 2 x i +1 suponiendo que el primer elemento del vector tiene un índice = 1. Es decir, la cima ocupa la posición inicial del vector y sus dos hijos la posición segunda y tercera, y así, sucesivamente. Por tanto, en la fase de ordenación, el intercambio ocurre entre el primer elemento del vector (la raíz o cima del árbol, que es el mayor elemento del mismo) y el último elemento del vector que es la hoja más a la derecha en el último nivel. El árbol pierde una hoja y por tanto reduce su tamaño en un elemento. El vector definitivo y ordenado, empieza a construirse por el final y termina por el principio.

4.1.4 Ordenamiento por Cuentas

Descripcion del Algortimo

El ordenamiento por cuentas es un algortimo en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Sólo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo).

El primer paso consiste en averiguar cuál es el intervalo dentro del que están los datos a ordenar (valores mínimo y máximo). Después se crea un vector de números enteros con tantos elementos como valores haya en el intervalo [mínimo, máximo], y a cada elemento se le da el valor 0 (0 apariciones). Tras esto se recorren todos los elementos a ordenar y se cuenta el número de apariciones de cada elemento (usando el vector que hemos creado). Por último, basta con recorrer este vector para tener todos los elementos ordenados.

4.2 Busqueda Binaria

4.2.1 Descripcion del Algoritmo

El algortimo de busqueda binaria esta basado en la tecnica divide y venceras. Se usa sobre un conjunto de elemntos ordenados y tiene complejidad de $O(n \log n)$

En una forma general el algoritmo de busqueda binaria se puede usar en cualquier funcion "binaria", es decir que sea falsa para un primer intervalo del conjunto y verdadera para el resto del conjunto, o viceversa.

4.2.2 Ejemplo

Bibliography

Books

[Smi12] John Smith. *Book title*. 1st edition. Volume 3. 2. City: Publisher, Jan. 2012, pages 123–200.

Articles

[Smi13] James Smith. “Article title”. In: 14.6 (Mar. 2013), pages 1–8.

Index

C

Citation 6
Corollaries 8

D

Definitions 7

E

Examples 8
    Equation and Text 8
    Paragraph of Text 9
Exercises 9

F

Figure 11

L

Lists 6
    Bullet Points 6
    Descriptions and Definitions 6
    Numbered List 6

N

Notations 8

P

Paragraphs of Text 5
Problems 9
Propositions 8
    Several Equations 8
    Single Line 8

R

Remarks 8

T

Table 11
Theorems 7
    Several Equations 7
    Single Line 7

V

Vocabulary 9

Lista de Codigos

1.1 Ejemplo de C++ . . . 9
1.2 Hola Mundo . . . 10
1.3 Hola Mundo Compilado . . . 10
1.4 Hello World . . . 11
1.5 Using Namespace Std . . . 11
1.6 Compilar . . . 11
1.7 Ejecutar . . . 11
1.8 Resultado . . . 11
1.9 Sintaxis . . . 12
1.10 Sintaxis . . . 12
1.11 Sintaxis . . . 12
1.12 Comentarios . . . 13
1.13 Unsigned . . . 14
1.14 Constantes . . . 14
1.15 Constantes . . . 14
1.16 Iostream . . . 15
1.17 Oper Direc . . . 15
1.18 Ejemplo Cin Cout . . . 15
1.19 Impresión . . . 16
1.20 Resultado Impresión . . . 16
1.21 Sintaxis If . . . 19
1.22 Sintaxis If Else . . . 19
1.23 Sintaxis Else . . . 19
1.24 If Ejemplo 1 . . . 20
1.25 If Ejemplo 2 . . . 20
1.26 If Ejemplo 3 . . . 20
1.27 If Ejemplo 4 . . . 20
1.28 Sintaxis Switch . . . 21
1.29 Switch Ejemplo 1 . . . 21
1.30 Switch Ejemplo 2 . . . 22
1.31 Switch Ejemplo 3 . . . 22

1.32 Sintaxis Switch . . . 22
1.33 Condicional Ejemplo 1 . . . 22
1.34 Condicional Ejemplo 2 . . . 23
1.35 Sintaxis For . . . 23
1.36 For Ejemplo 1 . . . 24
1.37 For Ejemplo 2 . . . 24
1.38 For Ejemplo 3 . . . 24
1.39 Sintaxis While . . . 25
1.40 While Ejemplo 1 . . . 25
1.41 Sintaxis Do While . . . 25
1.42 Sintaxis Do While . . . 25
1.43 Break . . . 26
1.44 Resultado Break . . . 26
1.45 Continue . . . 27
1.46 Resultado Continue . . . 27
1.47 Break While . . . 27
1.48 Continue While . . . 27
3.1 Factorial . . . 31
3.2 Factorial Recursivo . . . 32
3.3 Sucesion Fibonacci . . . 33
4.1 QuickSort . . . 35
4.2 Merge Sort . . . 36