Introducción a la Programación en C++

Cátedra: Algoritmos y Estructuras de Datos

Acuerdo de Licencia

Buena parte del contenido de este apunte se copió o modificó, tomándolo del apunte "Introducción en C++ a la Programación Competitiva", cuya autoría pertenece a varios autores del "Capítulo estudiantil ACM-ICPC UMSA", disponible en
https://www.academia.edu/37590839/Introduccion_en_C_a_la_Programacion_Competitiva

¿Por qué este apunte?

Los profesores de Algoritmos y Estructuras de Datos (AEDD) de la UTN Santa Fe decidimos armar este apunte, el cual está formado por material didáctico e introductorio sobre la *programación en C++* en particular y *fundamentos de algoritmia* en general. Hemos seleccionado material de diferentes fuentes que de manera generosa comparten sus creaciones mediante licencias del tipo "Atribución-No Comercial-Compartir Igual". Este apunte también incluye algunas secciones con contenido propio, y se comparte con el mismo tipo de licencia (creemos que es lo que corresponde a trabajos de este tipo realizados en una universidad pública).

El objetivo de esta recopilación-elaboración es el de brindar a los alumnos de 1er. año de la carrera de Ingeniería en Sistemas de Información de la Universidad Tecnológica Nacional, un material de apoyo que los estimule en el aprendizaje de la programación, complementando las clases de teoría y práctica del cursado anual de esta asignatura (AEDD).

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, una forma de escribirse que es diferente en cada uno por la manera en que fue creado y por la forma en que trabaja su compilador para revisar, acomodar y alojar el mismo programa en memoria.

Existen muchos lenguajes de programación entre los que se destacan los siguientes:

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

1.1.2 ¿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 la de 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, sumándose así a los otros dos paradigmas que ya estaban admitidos (programación estructurada y programación orientada a objetos). Por esto se suele decir que C++ es un lenguaje de programación multiparadigma.

1.1.3 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++

3. Un editor cualquiera de texto, o mejor un entorno de desarrollo (IDE)
   - Windows
     - Bloc de notas (no recomendado)
     - Editor Notepad++
     - DevCpp (incluye MingW - en desuso, no recomendado, incluye también un compilador)
     - Zinjal (es el que usamos en AEDD)
   - Linux (o re-compilación en UNIX):
     - Gedit
     - Kate
     - KDevelop
     - Code::Blocks
     - SciTE
     - GVim

4. Tiempo para practicar

5. Paciencia

Adicional

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

1.1.4 Consejos iniciales antes de programar

Aunque en esta etapa el objetivo es aprender a programar (de forma individual, cada uno de nosotros), vale la pena ser conscientes que los programadores habitualmente trabajan en equipo y es muy valioso escribir código que sea de fácil comprensión para todos sus miembros. Antes de escribir código, si se trabaja con otros programadores, hay que tener en cuenta que todos deben hacerlo de una forma similar, para así poder corregirlo en el caso de que hubieran errores, o rastrearlos en el caso de sospechar que pueda existir alguno.

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 más de tiempo, es posible que más adelante lo agradezcas tú mismo.

1.1.5 Ejemplo

Código 1.1: Ejemplo de C++

```
#include <iostream>

using namespace std;  //Comentario

int main () {
	int numero ;

	cout << "Ingrese un número: " << endl;
	cin >> numero;

	if (numero%2==0) {
		cout << "El número es par." << endl;
	} else {
		cout << "El número es impar." << endl;
	}

	return 0;
}
```

1.2 Lo más básico

1.2.1 Proceso de desarrollo de un programa

Si se desea escribir un programa en C++ se deben 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.
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, porque en programas grandes, averiguar si hay o no un fallo prácticamente puede ser una tarea totémica.
Un archivo de C++ tiene la extensión ".cpp". A continuación se visualiza el código en C++ del archivo 'hola.cpp'

Código 1.2: Hola Mundo

```
// Aquí generalmente se suele indicar qué se quiere con el programa a hacer
// Programa que muestra 'Hola mundo' por pantalla y finaliza

// Aquí se sitúan todas las librerías que se vayan a usar con include,
// que se verá posteriormente

#include <iostream> // Esta librería permite mostrar y leer datos por
                    // consola

int main () {
      // Este tipo de líneas de código que comienzan por '//' son
      // comentarios
      // El compilador los omite, y sirven para ayudar a otros
      // programadores o a uno mismo en caso de volver a revisar
      // el código.
      // Es una práctica sana poner comentarios donde se necesiten.

      std::cout << "Hola Mundo" << std::endl;

      // Mostrar por std::cout el mensaje Hola Mundo y comienza una nueva
      // línea

      return 0;

      // Se devuelve un 0, que en este caso quiere decir que la salida
      // se ha efectuado con éxito.
}
```

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

Código 1.3: Hola Mundo - versión 2

```
# include <iostream>
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, que se detallarán posteriormente, están omitidos. Tan sólo ha quedado "el esqueleto" del código legible para el compilador. Para el compilador, todo lo demás, sobra. Aquí otro ejemplo:

Código 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 más corto podemos incluir *using namespace std*. Asi podremos obviar la indicación std:: para 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) respectivamente..

Se vería de la siguiente manera:

Código 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:

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

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

```
|./hola
```

Y a continuación debería mostrarse algo como lo siguiente:

```
Hola Mundo
```

1.2.2 Sintaxis

Sintaxis se refiere a 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 C estándar pueden ser compilados en C++.

1.2.2.1 El punto y coma

El punto y coma es uno de los símbolos más usados en C, C++; y se usa para indicar el final de una línea de instrucción. Como se puede observar en los ejemplos anteriores, el punto y coma es de uso obligatorio.
También se usa para separar contadores, condicionales e incrementadores dentro de una sentencia *for*

Ejemplo:

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

1.2.2.2 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 se podría escribir así:

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

y el compilador no pondría ningún reparo.

1.2.2.3 Comentarios

Existen dos modos básicos para comentar en C++:

//	Comentan solo una línea de código
/*Comentario*/	Comentan estrofas o bloques de código

A continuación un ejemplo:

Código 1.6: Comentarios

```
#include <iostream>

using namespace std;

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

	// En español
	cout << "Hola Mundo" << endl;
	// En inglés
	cout << "Hello World" << endl;
	// En alemán
	cout << "Hallo Welt" << endl;

	return 0;
}
```

1.2.3 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	1 byte	true o false
char	Carácter o entero pequeño	1 byte	De -128 a 127
short int	Entero corto	2 bytes	De -32768 a 32767
int	Entero	4 bytes	De -2147483648 a 2147483647
long long	Entero largo	8 bytes	De -9223372036854775808 a 9223372036854775807
float	Número de punto flotante	4 bytes	3.4e +/- 38 (7 dígitos)
double	Float con doble precisión	8 bytes	1.7e +/- 308 (15 dígitos)

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, y permite especificar que de los enteros se utilizarán únicamente valores positivos.

```
int a;       // Almacena valores entre -2,147,483,648 a 2,147,483,647
unsigned int a;    // Almacena valores entre 0 a 4,294,967,295
```

1.2.4 Variables y constantes

Una variable, como su nombre lo indica, es un elemento cuyo valor puede cambiar durante el proceso de una tarea específica. Por el contrario, una constante es un elemento 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 declaran 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

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 “Datos 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. Por ejemplo, si se tiene la constante numérica 3.1416, esta representa al número pi, entonces podemos hacer declaraciones tales como:

```
#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 símbolo por 3.1416. O bien,

```
const float 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.

1.2.5 Lectura e Impresión

La lectura e impresión de datos se puede realizar con los métodos cin y cout primitivos de C++.

1.2.5.1 cin y cout

En sus programas, si usted desea hacer uso de los objetos cin y cout tendrá que incluir la biblioteca iostream (por medio de la directiva #include). iostream es la biblioteca estándar en C++ que permite tener acceso a los dispositivos estándar de entrada y/o salida.

Si en los programas se utilizan las directivas #include <iostream.h> o #include <iostream>, automáticamente la *iostream* pone a su disposición los objetos *cin* y *cout* en el ámbito estándar (std), de tal manera que se puede comenzar a enviar o recibir información a través de los mismos sin siquiera preocuparse de su creación. Así, se muestra a continuación un sencillo ejemplo del uso de los objetos mencionados.

Código 1.7: Iostream

```
// De nuevo con el hola mundo...
#include <iostream.h>

using namespace std;

int main (){
	cout << "Hola mundo"; // imprimir mensaje (en la pantalla)
	cin.get(); // lectura (entrada del teclado)

	return 0;
}
```

Para el manejo de *cin* y *cout* se necesita también el uso de los operadores de inserción de flujo << y >>. Los operadores de inserción de flujo son los encargados de manipular el flujo de datos desde o hacia el dispositivo referenciado por un stream específico. El operador de inserción de flujo para salidas es una pareja de símbolos "menor que" <<, y el operador de inserción de flujo para entradas es una pareja de símbolos "mayor que" >>. Los operadores de inserción de flujo 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 se tiene la instrucción cout << "Ingrese su nombre: ";. La constante "Ingrese 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.

Código 1.8: Operador inserción de flujo

```
// Una vez más...
#include <iostream>
```

```
using namespace std;

int main () {
	char nombre[80];
	cout << "Ingrese su nombre: " << endl;
	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:

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

Otro ejemplo del manejo de cout:

Código 1.9: 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 única linea de codigo y que "
            << endl
            << "el programa , al ejecutarse , lo situe como el "
            << "programador quiso "
            << endl ;

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

Salida por consola:

```
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 única linea de codigo y que
el programa , al ejecutarse , lo situe como el programador quiso
```

1.2.6 Operadores

C++ tiene varios 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	Sentido
( ) { } -> .	I-D
! ~++ – - (tipo) * & sizeof(todos unarios)	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
,	I-D

La acción de estos operadores es la siguiente :

1.2.6.1 Operadores Aritméticos

C++ tiene varios operadores. Presentamos aquí una tabla de los mismos indicando el rango.

+ : 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 derecha.

* : 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 derecha.

% : 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 preincremento) o de la variable situada a su derecha (modo postincremento).

-- : Similar a ++, pero restando 1.

1.2.6.2 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 a su derecha utilizando la operación indicada. Usaremos *d* para referir la variable de la derecha e *i* para la de la izquierda.

+= 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

Ejemplo:

```
conejos *= 1.6; //es lo mismo que conejos = conejos * 1.6;
```

1.2.6.3 Operadores de Relación

Cada uno de estos operadores compara el valor de su izquierda con el valor de su derecha. 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

1.2.6.4 Operadores Lógicos

Los operadores lógicos utilizan normalmente expresiones 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.

1.2.6.5 Operadores Relacionados con punteros

& : Operador de Dirección. Cuando va seguido por el nombre de una variable, entrega la dirección de dicha variable. &aux es la dirección de la variable aux.

***** : Operador de Indirección. Cuando va seguido por un puntero, entrega el valor almacenado en la dirección apuntada por él.

```
abc  =  22;
def  =  & abc ;  // puntero a abc
val  =  * def
```

El efecto neto es asignar a val el valor 22

1.2.6.6 Operadores de structs (registros)

El operador de pertenencia (punto) se utiliza junto con el nombre asociado al struct, para especificar un miembro del mismo. 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. Ejemplo:

```
struct {
      int codigo;
      float precio;
} articulo;

articulo.codigo = 1265;
```

Con esto se asigna un valor al miembro código de la estructura artículo.

-> : Operador de pertenencia indirecto: se usa con un puntero al struct para identificar un miembro de las mismas. Supongo que ptrstr es un puntero a una estructura que contiene un miembro especificado en el patrón de estructura con el nombre miembro. En este caso:

```
ptrstr -> miembro;
```

identifica al miembro correspondiente de la estructura apuntada.

```
struct {
      int codigo;
      float precio;
} articulo, *ptrstr;

ptrstr = &articulo;

ptrstr->codigo = 1265;
```

De este modo se asigna un valor al miembro código de la estructura articulo. Las tres expresiones siguientes son equivalentes:

```
ptrstr->código          articulo.código          (*ptrstr).codigo
```

1.2.6.7 Operadores Lógicos y de Desplazamiento de Bits

Operadores Lógicos

Estos son cuatro operadores que funcionan con datos de clase entera, incluyendo char. Se dice que son operadores que trabajan "bit a bit" porque pueden operar cada bit independientemente del bit situado a su izquierda o derecha:

~ El **complemento a uno o negación** en bits es un operador unario, cambia todos los 1 a 0 y los 0 a 1. Así:

~(10011010) == 01100101

& El **and** de bits es un operador que hace la comparación bit por bit entre dos operandos. Para cada posición de bit, el bit resultante es 1 únicamente cuando ambos bits de los operandos sean 1. En terminología lógica diríamos que el resultado es cierto si, y sólo si, los dos bit que actúan como operandos lo son también. Por tanto:

(10010011) & (00111101) == (00010001)

ya que únicamente en los bits 4 y 0 existe un 1 en ambos operandos.

| El **or** para bits es un operador binario que realiza una comparación bit por bit entre dos operandos. En cada posición de bit, el bit resultante es 1 si alguno de los operandos o ambos contienen un 1. En terminología lógica, el resultado es cierto si uno de los bits de los operandos es cierto, o ambos lo son. Así:

(10010011) | (00111101) == (10111111)

porque todas las posiciones de bits con excepción del bit número 6 tenían un valor 1 en, por lo menos, uno de los operandos.

^ El **or exclusivo** de bits es un operador binario que realiza una comparación bit por bit entre dos operandos. Para cada posición de bit, el resultante es 1 si alguno de los operandos contiene un 1; pero no cuando lo contienen ambos a la vez. En terminología lógica, el resultado es cierto si lo es el bit u otro operando, pero no si lo son ambos. Por ejemplo:

(10010011) ^(00111101) == (10101110)

Observe que el bit de posición 0 tenía valor 1 en ambos operandos; por tanto, el bit resultante ha sido 0.

Operadores de Desplazamiento de Bits

Estos operadores desplazan bits a la izquierda o a la derecha. Seguiremos utilizando terminología binaria explícita para mostrar el funcionamiento.:

<< : El desplazamiento a la izquierda es un operador que desplaza los bits del operando izquierdo a la izquierda el número de sitios indicando por el operando de su derecha. Las posiciones vacantes se rellenan con ceros, y los bits que superan el límite del byte se pierden. Así:

(10001010)«2 == (1000101000)

cada uno de los bits se ha movido dos lugares hacia la izquierda

>> : El desplazamiento a la derecha es un operador que desplaza los bits del operando situado a su izquierda hacia la derecha el número de sitios marcado por el operando situado a su derecha. Los bits que superan el extremo derecho del byte se pierden. En tipos unsigned, los lugares vacantes a la izquierda se rellenan con ceros. En tipos con signo el resultado depende del ordenador utilizado; los lugares vacantes se pueden rellenar con ceros o bien con copias del signo (bit extremo izquierdo). En un valor sin signo tendremos:

(10001010)»2 == (00100010)

en el que cada bit se ha movido dos lugares hacia la derecha.

1.2.6.8 Misceláneas

sizeof Devuelve el tamaño en bytes, del operando situado a su derecha. El operando puede ser un especificador de tipo, en cuyo caso se emplean paréntesis; por ejemplo, sizeof(float). Puede ser también el nombre de una variable concreta o de un array, en cuyo caso no se emplean paréntesis: sizeof foto

(tipo) Operador de moldeado, convierte el valor que vaya a continuación en el tipo especificado por la palabra clave encerrada entre los paréntesis. Por ejemplo, (float)9 convierte el entero 9 en el número de punto flotante 9.0.

, El operador coma une dos expresiones en una, garantizando que se evalúa en primer lugar la expresión situada a la izquierda. Una aplicación típica es la inclusión de más información en la expresión de control de un bucle for:

```
for (jugadores=2, ronda=0; ronda<1000; jugadores*=2){
     ronda += jugadores;
}
```

? El operador condicional se explicará más adelante

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 condición) y selecciona la siguiente instrucción a ejecutar dependiendo la respuesta o resultado:

1.3.1.1 Sentencia if

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

Sintaxis

Primera Forma

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

```
if (condicion){
      Set de instrucciones
}
```

siendo condición 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 le añadiremos la parte Falsa de la sentencia:

```
if (condicion){
      Set de instrucciones     //Parte VERDADERA
} else {
      Set de instrucciones 2   //Parte FALSA
}
```

La forma mostrada anteriormente muestra la unión de la parte verdadera con la nueva secuencia, que es la parte falsa de la sentencia de decisión if.

La palabra *else* indica al lenguaje que si la condición no se cumple o es falsa, entonces realizará el set de instrucciones 2.

Ejemplos de Sentencias If

Ejemplo 1:

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

Ejemplo 2:

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

Ejemplo 3:

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

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:

```
if (numero == 0) { //La condicion indica que tiene que ser igual a 0
	cout <<"El Numero Ingresado es Igual a Cero";
} else {
	if (numero > 0) { //La condicion indica que tiene que ser > a 0
		cout <<"El Numero Ingresado es Mayor a Cero";
	} else { // por descarte...
		cout <<"El Numero Ingresado es Menor a Cero";
		}
	}
}
```

1.3.1.2 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 combinar con el uso de las sentencias *case* y *break*. 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 menús.

Sintaxis

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

```
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:

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

Ejemplo 2:

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

Ejemplo 3:

```
switch (numero) {
      case 1:
            cout << "Número es uno";
            break;
      case 2:
            cout << "Número es dos";
            break;
      case 3:
            cout << "Número es tres";
            break;
      default:
            cout << "Elija una opción entre 1 y 3";
}
```

1.3.1.3 Operador condicional ternario ?:

En C/C++, existe el operador condicional (? :) el cual es conocido por su estructura como ternario.

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

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

En donde condición es la expresión que se evalúa, 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 Operador condicional ternario:

Ejemplo 1:

```
int edad ;

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

//Usando el operador condicional ternario
cout << ((edad < 18) ? " Eres joven" : "Ya tienes la mayoria de 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).

```
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 también una estructura de control condicional, ya que dentro de la misma se repite la ejecución de una o más

instrucciones mientras que una condición específica 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.

1.3.2.1 Sentencias For

Sintaxis

La sintaxis utilizada en la programación de C++ es la siguiente:

```
for (inicialización ; final; incremento){
      Código a Repetir ;
}
```

donde inicialización suele ser una variable numérica que recibe un valor inicial, final es la condición que se evalúa para finalizar el ciclo (puede ser independiente del contador) e incremento es el valor que se suma o resta a la variable. Hay que tener en cuenta que el for evalúa la condición de finalización igual que el while, es decir, mientras esta se cumpla continuarán las repeticiones.

Ejemplos de Sentencias For

Ejemplo 1

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

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 suma por unidad lo que hace que el contador se incremente repitiendo 10 veces “Hola Mundo” en pantalla. (el valor de i llega a ser 11, y ese es el caso en el que no se cumple la condición del for, y se termina la ejecución de la sentencia).

Ejemplo 2

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

Este ejemplo hace algo similar al primero, “al revés”: el contador se inicializa en 10 en lugar de 1; y por ello cambia la condición que se evalúa así como que el contador se decrementa en lugar de ser incrementado. (en este caso el cuerpo del for se ejecuta 11 veces: para los valores 10,9,8,..2,1,0 de i; y cuando i toma el valor -1, se termina la ejecución del for).

La condición también puede ser independiente del contador:

Ejemplo 3

```
int j = 20;

for (int i=0; j>0; i++){
	cout << "Hola " << i << " - " << j << endl ;
	j--;
}
```

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

1.3.2.2 Sentencia While

Sintaxis

La sintaxis utilizada para el *while* es la siguiente:

```
while (condicion) {
	// código a repetir
}
```

Donde condición es la expresión a evaluar.

Ejemplos de Sentencias While

Ejemplo 1

```
int contador = 0;

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

El contador indica que se realizará el código contenido dentro de la sentencia *while*, mientras que este contador sea menor o igual que 10, entonces se detendrán las iteraciones y se continuará en la sentencia siguiente. Por lo tanto se mostrará 11 veces “Hola Mundo” en

pantalla. (y el valor de la variable contador inmediatamente luego de la ejecución del while será 11).

1.3.2.3 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.

Pensemos la siguiente situación: el usuario tiene que ingresar una letra por teclado, y el programa debe asegurarse que dicha letra sea una mayúscula antes de avanzar en su ejecución.

Podríamos escribir el siguiente código:

```
char letra;

cin >> letra;
while (not (letra>='A' and letra<='Z'){
      cin >> letra;
};
```

El código anterior funciona, pero si observamos que el ingreso de una letra por parte del usuario se debe ejecutar sí ó sí una vez, nos damos cuenta que la estructura adecuada para este caso es un *do..while*, que nos permite un código más conciso y elegante:

```
char letra;

do {
      cin >> letra;
} while (not (letra>='A' and letra<='Z');
```

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 sentencias *for* y la sentencia *while*. Además, veremos el uso de la sentencia continue.

1.3.3.1 Break

La sentencia *break* se usa para forzar un salto hacia el final de un ciclo controlado por *for* o por *while*. (aunque por cuestiones de estilo el for está pensado para estructuras repetitivas exactas, y por lo tanto no deberíamos usar break dentro de un for)

Ejemplo:

En el siguiente fragmento de código la sentencia break cierra el ciclo *while* cuando la variable i es igual a 5. (el ejemplo ilustra una forma de codificar un ciclo exacto con un *while*):

```
int i=1;
while (true) {
	if (i==5) break ;
	cout << i << " ";
	i++;
}
```

La salida para el mismo será:

```
0 1 2 3 4
```

El uso de la sentencia *break*, permite implementar ciclos con la última iteración "incompleta", dado que en la última ejecución del cuerpo del ciclo, las sentencias que se encuentran despúes del break, no se ejecutan

1.3.3.2 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.

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

La salida para el mismo será:

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