UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño
y programación de Videojuegos

UNLVIRTUAL

Introducción a la programación

Unidad Temática Número 3

Estructuras de Control

Objetivo: Implementar Estructuras de Control en C++

Temas: Programación Estructurada, Secuencia, Expresiones, Operadores, Estructuras de Control Repetitivas y Condicionales

INTRODUCCIÓN

Hasta el momento sabemos realizar operaciones manejando variables, introduciendo valores y mostrándolos en pantalla. Pero esto recién empieza. Si pudiéramos unir con líneas el flujo de información de cada una de las instrucciones que realizamos en nuestros códigos, tendríamos una línea recta desde la primera instrucción hasta la última, ya que en todo momento estamos yendo a la siguiente instrucción para ejecutarla y luego a la siguiente y así. Todavía no sabemos cómo hacer si queremos tener alguna decisión sobre los valores que ingresamos, o tener un manejo de operaciones automático que no requiera escribir tantas veces lo que queremos hacer. Bien, aquí es donde introducimos las Estructuras de Control. Que como su nombre lo indica, nos permite controlar el flujo de datos para que varíe según nuestras necesidades. No se preocupen si esta introducción se presta un poco confusa, ya nos encargaremos de ver los ejemplos suficientes como para que se comprenda el concepto.

ESTRUCTURA DE CONTROL: SI CONDICIONAL

Imaginemos que alguien nos encarga ir a la verdulería a comprar un kilo de manzanas y en caso de no haber, comprar un kilo de peras (Una de las dos frutas va a haber sí o sí). En nuestra mente podríamos imaginar esta orden como:

Comprar manzanas si hay, sino comprar peras.

Ese "si" nos implica una condición, que en este caso es comprar manzanas, en caso de no cumplirse por no haber, directamente realizaríamos la contraparte adversativa, es decir lo que nos indica el "sino", que en este caso es comprar peras. Esa misma pregunta es la que podemos hacer en nuestro código y nos permite realizar un quiebre en nuestro flujo de datos y poder así bifurcar el camino.

La estructura del código del SI es la siguiente:

```
SI (PASA ESTO)
    HACEMOS ESTO
SINO
    HACEMOS ESTO
```

Cuando escribimos PASA ESTO nos estamos refiriendo una expresión lógica. ¿Qué es una expresión? Una **expresión** es un concepto que ya vimos pero no lo habíamos nombrado de esta manera. En informática le decimos expresión al conjunto de operandos relacionados por operadores. Es decir *x = 10* es una expresión, es este caso es una asignación, *x+5+y* es una expresión aritmética, *x<10* es una expresión lógica, etc.
Son ejemplos de expresiones:

```
manzanas = 5;
manzanas + peras < 4;
peras++;
```

Ahora sí estamos en condiciones de exponer, finalmente, la sintaxis del SI condicional en C++:

```
if (expresion){
    ...
}else{
    ...
}
```

La palabra **if** (SI en inglés) nos indica que si **expresion** es una expresión cierta (true), se realizará lo que se escriba entre las primeras llaves **{.}**, en caso de la que expresión sea falsa, se ejecutará directamente lo que se escriba dentro de las llaves contiguas a la palabra **else**. Tanto **if**, como **else** son palabras reservadas en C++.

Veamos un ejemplo. Debemos indicar si el número que ingresamos por teclado es mayor o menor a 5.

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

int main (int argc, char *argv[]) {

    int numero;

    cout<<"Ingrese un numero: ";

    cin>>numero;

    if (numero > 5){

        cout<<"El numero ingresado es mayor a 5"<<endl;

    }else{

        cout<<"El numero ingresado es menor a 5"<<endl;
    }

    return 0;
}
```

Como ya mencionamos, la expresión del SI debe ser sí o sí una expresión lógica, es decir, que su resultado sea **true** o **false**. Es muy común utilizar con las estructuras condicionales como el SI, las variables banderas. ¿Qué son las banderas? Las banderas son variables booleanas que almacenan el resultado de una expresión lógica, y sirven para controlar estados.

Supongamos que necesitamos definir la lógica de nivel de nuestro juego, y una de las reglas dice que no podemos pasar de nivel si no juntamos 100 monedas. Podríamos entonces definir una bandera llamada "monedas" que sea **true** si ya tenemos 100 o más monedas y **false** si todavía no las juntamos. Vamos a dar por sentado que en nuestro código cuando llegamos a las 100 monedas prendemos la bandera (es decir, la ponemos en *true*), entonces al momento de pasar de nivel hacemos el siguiente código:

```
    if (monedas){
        //pasamos de nivel
    }else{
        //debes juntar mas monedas
    }
```

¿Cuál es la utilidad de usar banderas? La utilización de banderas permite agilizar mucho la lectura del código, sobre todo cuando tenemos varios estados. Es mucho más fácil leer el contenido de una variable que andar acumulando expresiones lógicas. Sobre todo cuando las banderas acumular varias expresiones. Ya veremos eso más adelante.

Para finalizar con el SI debemos mencionar que la condición del SINO puede ser opcional, no así la del IF. Es decir, siempre que coloquemos un **if** debemos tener una condición que lo evalúe y un código que se ejecute cuando esa expresión es verdadera, pero podemos optar por no tener la instrucción **else.**

Por ejemplo, si ahora quisiéramos hacer esa parte del juego que coloca en true la bandera cuando junta las 100 monedas, podríamos tener el siguiente código que se ejecuta cada vez que se junta una moneda:

```
    if (coins > 99){
        monedas = true;
    }
```

Donde *coins* es la variable que va contando las monedas que vamos juntando. De esta manera cuando *coins* llegue a 100, la expresión del **if** será cierta (true) y entrará a ejecutarse el código *monedas = true*. Podría incluso ejecutarse cada vez que se junta una moneda, total una vez que pase las 100 entrará siempre y por ende *monedas* ya valdrá *true*. Así, cuando tengamos que pasar de nivel y preguntemos por *monedas* está

será true o false y me indicará si ya junté las 100 monedas. En ese código no hace falta tener una condición que salga por el SINO (**else**) así que no la utilizamos.

Por el momento esto es todo lo que refiere a la sintaxis del IF en C++. Aunque aún falta explayarnos sobre un par de conceptos que veremos más adelante.

ESTRUCTURA DE CONTROL: CASE (SWITCH CASE)

Volvamos al ejemplo del pedido a la verdulería:

Comprar manzanas si hay, sino comprar peras.

Pero esta vez vamos a hacerlo un poco más real e imaginar que puede no haber peras tampoco en la verdulería. En ese caso el pedido debería reformularse de la siguiente manera:

Comprar manzanas si hay, sino comprar peras si hay.

En este caso no podríamos simplemente comprar peras en la instrucción falsa, sino que deberíamos volver a preguntar si hay peras. De esta manera:

```
SI (MANZANAS)
    COMPRAR
SINO SI (PERAS)
    COMPRAR
```

Es decir, estamos colocando un SI dentro de otro SI, colocar una estructura dentro de otra se denomina **anidar**.

La sintaxis de este código sería de la siguiente manera:

```
    if (manzanas){
        // comprar manzanas
    } else {
        if (peras){
        // comprar peras
        }
    }
```

Notar que aquí la pregunta si hay peras no se realiza sino hasta que se verifique que no hay manzanas. Y hay una clara diferencia con lo siguiente:

```
    if (manzanas){
        // comprar manzanas
    }
    if (peras){
        // comprar peras
    }
```

Si bien este segundo código podría parecer similar, su funcionamiento tiene algunas diferencias. En caso que haya manzanas, efectivamente compraremos manzanas, pero luego si hay peras, también compraremos peras. A diferencia del primer ejemplo, donde si hay manzanas compramos manzanas y no continuamos para ver si hay peras.

Entonces, si tenemos la necesidad de verificar dos estados consecutivos, podemos utilizar dos SI anidados. ¿Pero qué ocurre si debemos consultar varios estados, quizás 5, 10 o más? Sería todo un ovillo de SI anidados.

Imaginemos por ejemplo la siguiente situación:
En una escuela primera, los niños salieron al recreo y ahora debemos volver a ubicarlos a cada uno en su respectivo curso. Vamos a suponer que por edad a cada uno le corresponde un curso y no hay niños de la misma edad en distintos grados. Tenemos la siguiente relación.

Edad	Curso
6	1ro
7	2do
8	3ro
9	4to
10	5to
11	6to
12	7mo

Si utilizamos si consecutivos, corremos el riesgo de continuar haciéndole preguntas a un niño que ya ubicamos en un aula. Debemos utilizar SI anidados, aunque como mencionamos, puede volverse un poco confuso tener tantas estructuras una dentro de otra.

Una alternativa a utilizar varios SI anidados es utilizar una estructura llamada CASE o SWITCH CASE. Podría traducirse como "según" en español. Su forma es la siguiente:

SEGUN "ESTADO"
CASO 1 HACER:
CASO 2 HACER:
CASO 3 HACER:
….
PREDETERMINADO HACER:

Donde "ESTADO" es una variable que puede tomar distintos valores que contemplamos en cada uno de los CASO, si la variable toma un valor no contemplado, podemos poner una acción para realizar en todos los valores no contemplados, esta acción se ejecutará en la opción PREDETERMINADO. Estamos entonces en condiciones de exponer la sintaxis del CASE en C++:

```
switch (variable){
    case 1:
        //en caso de ser 1
        break;
    case 2:
        //en caso de ser 2
        break;
    case 3:
        //en caso de ser 3
        break;
    case 4:
        //en caso de ser 4
        break;
    default:
        //cualquier otro caso
}
```

La estructura CASE nos permite comparar estados de diversos tipos de variables, tanto enteros, como strings. Hasta ahora sólo hemos comparado números, pero también podemos comparar cadenas. Básicamente es como tener muchos IF anidados comparando por una variable por un estado: *if (variable == estado)*. Las palabras **switch**, **case**, **break** y **default**, son palabras reservadas. La instrucción **default** es opcional (al igual que el **else** en la instrucción **if**). La instrucción **break** realiza el corte del algoritmo del **case** correspondiente. Ojo, En caso de no colocar el **break** no será un error de sintaxis, sino que se ejecutarán todas las acciones hasta llegar a un break o bien a la llave de cierre de la estructura. A continuación analizaremos una caso completo de un case con cada una de las situaciones que planteamos anteriormente.

Imaginemos la misma escuela del ejemplo anterior pero esta vez tenemos en el mismo patio a los niños de guardería que tienen 1, 2 y 3 años (todos van al mismo curso), también los niños de 4 y 5 años que van al mismo jardín y como si fuera poco también están en el patio los niños de la escuela secundaria de al lado que se cruzaron al patio a jugar. No sabemos a qué curso va cada uno, sólo que si tienen más de 12 deberemos mandarlos a la escuela secundaria.

El código fuente de este ejercicio quedaría (por cada niño) de la siguiente manera:

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

int main (int argc, char *argv[]) {

    int edad;
    cout<<"Cual es tu edad? ";
    cin>>edad;
    cout<<endl;

    switch (edad){
    case 1: case 2: case 3:
        cout<<"Va a guarderia"<<endl;
        break;
    case 4: case 5:
        cout<<"Va a jardin"<<endl;
        break;
    case 6:
        cout<<"Va a 1ro"<<endl;
        break;
    case 7:
        cout<<"Va a 2do"<<endl;
        break;
    case 8:
        cout<<"Va a 3ro"<<endl;
        break;
    case 9:
        cout<<"Va a 4to"<<endl;
        break;
    case 10:
        cout<<"Va a 5to"<<endl;
        break;
    case 11:
        cout<<"Va a 6to"<<endl;
        break;
    case 12:
        cout<<"Va a 7mo"<<endl;
        break;
    default:
        cout<<"Va a secundaria"<<endl;
    }

    return 0;
}
```

OPERADORES LOGICOS (2DA PARTE)

Habíamos visto que <, >, ==, != eran operadores lógicos y que usados en una expresión arrojan valores booleanos (true o false). Sin embargo nos quedaron algunos más sin ver, pero ahora vamos a ponernos al día.

Como siempre, vamos a partir de una situación. Imaginemos que tenemos que restringir el ingreso de un número para que se encuentre entre 1 y 10.

Con lo que sabemos podríamos intentar limitar primero los números mayores a 1 y luego los menores a 10.

```
    if(edad >= 1){
        if(edad<=10){
        cout<<"entre 1 y 10"<<endl;
        }
    }
```

¿Por qué dos if anidados? Bien, el primero permite quitarnos de encima todos aquellos números menores a 1, con el resto de los números que pasan la verificación, debemos quitarnos todos aquellos mayores de 10. Esto funciona para un único intervalo de condiciones, pero si tenemos varias condiciones en nuestro intervalo, la sumatoria de estructuras puede ponerse muy compleja. Imaginemos que tenemos que filtrar todos aquellos números pares, entre 1 - 10 y 20 - 30. Podríamos intentar con IF aunque no sería fácil. La estructura CASE no nos sirve ya que aquí tenemos rangos de valores. Lo

que sí podemos utilizar son los operadores OR y AND para operar con distintas expresiones.

El operador OR (O en español) se lo conoce como suma lógica o la "Unión" en teoría de conjuntos. De la misma manera el operador AND (Y en español) se lo conoce como producto lógico o "Intercepción" en teoría de conjuntos. Ambos se utilizan para operar conjuntos de expresiones lógicas simples.

El AND exige que ambas condiciones se cumplan para que la operación sea cierta.
El OR exige que al menos una de las condiciones sea correcta para que la operación sea cierta.

Volviendo al ejemplo de los números entre 1 y 10. En este caso podemos utilizar el operador AND entre ambas expresiones. Un número entre 1 y 10 quedaría comprendido si cumple ambas condiciones conjuntamente.
Un número está entre 1 y 10 si:

El número es **>=** 1 **Y** el número es **<=**10

Veamos tres valores de ejemplo: 4, -1, 15.

Número	>= 1	<=10	>=1 **Y** <=10
4	true	true	true
-1	false	true	false
15	true	false	false

Aquí se puede ver como sólo el 4 cumple ambas condiciones simultáneamente y por ende cumple la condición de ser un número entre 1 y 10.

En C++ el operador AND se escribe con doble caracter ampersand **&&**, mientras el operador OR se escribe con doble caracter pipe **||**. Si quisiéramos utilizarlos en el ejemplo de los números del 1 al 10, sería así:

```
if( (edad >= 1) && (edad <= 10) ){
   // tiene entre 1 y 10 anios
}
```

¿Recuerdan la película Titanic? Cuando estaban rescatando a los tripulantes el capitán dejaba subir pasajeros a los botes con la instrucción "mujeres y niños primero", aludiendo a niños de ambos sexos y mujeres de cualquier edad. Sim embargo, si usamos la lógica esa expresión no es correcta con el operador Y. Parecería ser que para subir al bote deberíamos ser niños y mujeres. Un niño varón cumple la condición de niño, pero no la de ser mujer, una mujer cumple la condición de mujer, pero no la de niño, parecería ser que las únicas habilitadas para subir serían niñas mujeres. El operador correcto en ese caso es O.

```
if((menores)||(mujeres)){
    //subir al bote
}
```

Basta que cumpla cualquiera de ambas condiciones para que la operación sea correcta.

La tabla comparativa con 4 casos distintos quedaría:

Número	menor	mujer	menor **O** mujer
Menor varón	true	false	true
Menor mujer	true	true	true
Mayor varón	false	false	false
Mayor mujer	false	true	true

Estas tablas se denominan tabla de verdad. De manera general podemos escribirlas de la siguiente manera:

Suma lógica (OR, O)		
Primera expresión	Segunda expresión	Resultado
0	0	0
0	1	1
1	0	1
1	1	1

Producto lógico (AND, Y)		
Primera expresión	Segunda expresión	Resultado
0	0	0
0	1	0
1	0	0
1	1	1

En las tablas, un 0 es un false y el 1 es un true. Es decir, en la suma lógica, si la primer expresión es falsa y la segunda cierta, el resultado de la operación es cierta.

La gran utilidad de los operadores OR y AND radica en la posibilidad de comparar varias expresiones, como mencionamos antes. Vamos a suponer un juego, donde para pasar de nivel debemos haber cumplido con las siguientes condiciones:

+ haber juntado al menos 100 monedas o al menos 10 de energía
+ Tener la llave roja o verde
+ **NO** tener la llave dorada

Podríamos escribir esa operación de la siguiente manera.

```
if( ((monedas >= 100)||(energia >= 10)) && ((llaveVerde)||(llaveRoja)) && (!llaveDorada) ){
    //pasa siguiente nivel
}
```

Aprovecho el ejemplo para presentarles al operador **negado** (**!**). El operador negado se escribe como un símbolo de interrogación de cierre, su función es la de negar (cambiar el valor) de la expresión que precede. Si esperamos valuar un true, el negado lo convierte en false, si esperamos valuar un false, el negado lo convierte en true. Hay ocasiones donde es más fácil escribir una expresión para evaluarla por true, aunque lo que necesitamos es que esa condición no se cumpla (y viceversa), en ese caso usamos el negado. En ese ejemplo necesitamos que *llaveDorada* sea false, es decir que cuando esa bandera de false, el negado dará true
Volviendo al ejemplo, la forma de trabajar una operación compuesta es desde adentro hacia afuera de los paréntesis (al igual que la operación aritmética:

Vamos a suponer el conjunto de valores:
monedas = 120
energia = 15
llaveVerde = false
llaveRoja = true
llaveDorada = false

(monedas >= 100)||(energia >= 10)	&&	(llaveVerde)||(llaveRoja)	&&	! llaveDorada
true || true	&&	false || true	&&	! false
true	&&	true	&&	true

Luego, una vez que está todo descompuesto en su mínima expresión, resolvemos la primer con la segunda expresión, el resultado de esas con la tercera y así.

(monedas >= 100)||(energia >= 10)	&&	(llaveVerde)||(llaveRoja)	&&	! llaveDorada
true || true	&&	false || true	&&	! false
true	&&	true	&&	true
true			&&	true
true				

AMBITO

El ámbito es el contexto (scope) en el cual existe una variable o función. Fuera de ese contexto esa variable o función no existen. En C++ el ámbito se delimita por las llaves (**{ }**). A la hora de utilizar estructuras, es muy importante tener bien el claro el concepto de ámbito. Gráficamente es muy sencillo de verlo, una variable existe y tiene valor sólo dentro de las llaves donde ésta fue declarada (y dentro de las llaves internas, pero no hacia afuera). Por ejemplo:

```
1  #include<iostream>
2  using namespace std;
3
4  int main (int argc, char *argv[]) {
5
6
7  int x1 = 10;
8  if(x1>3){
9     int x2 = 20;
10    cout<<"x1 "<<x1<<endl
11    cout<<"x2 "<<x2<<endl;
12 }
13 if(x1>4){
14    int x3 = 30;
15    cout<<"x1 "<<x1<<endl;
16    cout<<"x2 "<<x2<<endl; //error
17    cout<<"x3 "<<x3<<endl;
18 }
19
20 cout<<"x1 "<<x1<<endl;
21 cout<<"x3 "<<x3<<endl; //error
22 return 0;
23 }
```

Si intentamos ejecutar el código anterior, daría dos errores. El 1ro es que no existe *x2* en la línea 16 y el 2do que no existe *x3* en la línea 21.

En el ejemplo, x1 existe dentro de las llaves que lo contienen y en todos aquellos ámbitos creados dentro de esas llaves. X1 fue creada dentro del main y existe para todo el main y dentro de las estructuras if. En cambio x2 fue creada en el 1er if y sólo existe dentro de éste. En el 2do if se creo x3, x2 no existe dentro de este ámbito, y por ende, dentro de esta estructura, por lo que intentar mostrarlo en pantalla dará un error en la línea 16. Por último, intentar mostrar x3 dentro del main tampoco será posible, ya que fue creada dentro del ámbito del 2do if y sólo existe dentro de éste. Incluso, podríamos declarar otra variable llamada x2 dentro del 2do if, que no chocaría con la declarada en el 1er if.

Si hacemos una tabla de los ámbitos existentes en el este ejemplo con sus variables, sería así:

AMBITO			VARIABLES
main	1er if	2do if	
si	si	si	X1
no	si	no	X2
no	no	si	X3

Para terminar de afianzar el concepto vamos a ver otro ejemplo, un poco más complejo:

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

int main (int argc, char *argv[]) {

    int x1 = 10;
    if(x1>3){ //1er IF
        int x2 = 20;

        if (x2 > 20){ //2do IF
            int x3 = 100;
        }
        if (x2 < 20){ //3er IF
            int x4 = 200;
        }
    }
    if(x1>4){ //4to IF
        int x5 = 30;
    }
    return 0;
}
```

Si hacemos una tabla de AMBITO – VARIABLES para este código. Se vería así:

AMBITO					VARIABLES
main	1er if	2do if	3er if	4to fi	
si	si	si	si	si	X1
no	si	si	si	no	X2
no	no	si	no	no	X3
no	no	no	si	no	X4
no	no	no	no	si	X5