UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño
y programación de Videojuegos

UNL VIRTUAL

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

En las unidades anteriores hemos aprendido a formular algoritmos, y nos hemos introducido a la programación en el lenguaje C/C++, vimos conceptos como variable y constante y también como introducir algunos datos por el teclado, hacer algunos cálculos y mostrar los resultados por pantalla y además aprovechamos todas esas herramientas para hacer un sencillo juego.

Ahora vamos a dar un paso más que son las estructuras de control. Seguramente nos hubiera gustado que nuestro programa informara y nos felicitara cuando diéramos en el blanco, pero con las pocas herramientas que teníamos no podíamos, por ejemplo, tomar decisiones.

Con las estructuras de control tienen el núcleo básico de programación, y el dominar su uso es la clave para programar, ya que con ellas podemos tomar decisiones, reaccionando a las distintas alternativas posibles que nos plantean las distintas situaciones en la resolución de un problema complejo.

Programación Estructurada

A finales de los años 1960 surgió una nueva forma de programar que no solamente daba lugar a programas fiables y eficientes, sino que además estaban escritos de manera que facilitaba su comprensión posterior.

El teorema del programa estructurado, demostrado por Böhm-Jacopini, demuestra que todo programa puede escribirse utilizando únicamente las tres instrucciones de control siguientes:

- Secuencia
- Instrucción condicional.
- Iteración (bucle de instrucciones) con condición al principio.

Solamente con estas tres estructuras se pueden escribir todos los programas y aplicaciones posibles. Si bien los lenguajes de programación tienen un mayor repertorio de estructuras de control, éstas pueden ser construidas mediante estas tres estructuras.

Secuencia

En los ejemplos de las unidades anteriores solo hemos visto esta estructura, que como habrán adivinado se refiere a la secuencia de instrucciones, primero se ejecuta la primera, luego la segunda y así sucesivamente hasta completar nuestro programa.

Antes de ver las estructuras de control vamos a estudiar como se construyen las expresiones en C/C++.

Expresiones

Toda expresión consiste en *un conjunto de operandos ligados por operadores.* Utilizaremos expresiones en C/C++ habitualmente para efectuar cálculos, relaciones, asignaciones, etc.

Para plantear expresiones en C/C++ debemos conocer los numerosos operadores que posee este lenguaje.

Operadores

Operadores Aritméticos

Como su nombre lo indica, los operadores aritméticos nos permiten efectuar cálculos aritméticos. La *jerarquía o precedencia* de estos operadores es idéntica a la que empleamos en el álgebra de números. Junto con los operadores aritméticos se pueden emplear también los paréntesis.

Prioridad de los Operadores Aritméticos

Todas las expresiones entre paréntesis se evalúan primero. Las expresiones con paréntesis anidados se evalúan de dentro a fuera, el paréntesis más interno se evalúa

primero.
Dentro de una misma expresión los operadores se evalúan en el siguiente orden:

1. *, / Multiplicación, división.
2. +, - Suma y resta.

Los operadores en una misma expresión con igual nivel de prioridad se evalúan de izquierda a derecha.

Ejemplos:

4 + 2 * 5 = 14	23 * 2 / 5 = 9.2
3 + 5 * (10 - (2 + 4)) = 23	2.1 * (1.5 + 12.3) = 2.1 * 13.8 = 28.98

Operador	En tipos Enteros	En tipos Reales
+	símbolo + unario	símbolo + unario
-	símbolo – unario	símbolo – unario
+	suma	suma
-	resta	resta
*	producto	producto
/	división entera	división en punto flotante
%	resto de la división entera	NA (no aplicable)
++	incremento	NA
--	decremento	NA

Nota: cuando nos referimos a unario se refiere que la operación se realiza sobre un solo elemento, por ejemplo -a que quiere decir que le estamos anteponiendo el símbolo - al valor que tenga a, el resultado de esta operación sería si a es positivo pasará a ser negativo, si a es negativo pasara a ser positivo) en el caso del símbolo + o sea +a, no revestiría ningún cambio. Cuando nos referimos a operaciones binarias serían del tipo a-b o a+b en las cuales el símbolo opera no sobre un valor sino sobre dos valores.

Se debe tener en cuenta que las operaciones algebraicas se resuelven de izquierda a derecha pero respetando la precedencia de los operadores aritméticos.

Ejemplos:

10 + 7 – 4 arroja como resultado **13**. Se resuelve: **(10 + 7) – 4** porque ante igual prioridad de operadores se asocia de izquierda a derecha.

6 + 2 * 9 arroja como resultado **24** . Se resuelve: **6 + (2 * 9)** por que el operador producto tiene prioridad (precede) al operador de la suma.

21 / 4 arroja **5** (entero)

20.0 / 4.0 arroja **5.0** (punto flotante)

21 % 4 arroja **1** (entero)

++x incrementa en **1** el valor de la variable **x**

x++ toma el valor de la variable **x** y luego lo incrementa a en **1**

--x decrementa en **1** el valor de la variable **x**

x-- pos decrementa en **1** el valor de la variable **x**

Nota: obsérvese en los 2 ejemplos siguientes la diferencia entre los operadores de incremento y decremento cuando preceden a una variable y cuando la suceden.

```
int n=2;
cout<< n++ ; /* Se visualiza un 2. C++ envía el contenido de la variable n a la salida a través de cout y luego incrementa en 1 a n */
```

```
int n=2;
cout << ++ n ; /* Se visualiza un 3. C++ incrementa en 1 a la variable n y luego muestra el nuevo valor de n */
```

Operadores de asignación

El operador = permite asignar el resultado de la expresión de la derecha a la variable de la izquierda.

```
x=130;
```

Debemos observar que este operador es asociativo por la derecha, por lo cual podemos hacer asignaciones múltiples o simultáneas.

```
a=b=c=30 ;
```

Esto permite asignar simultáneamente a las tres variables el valor 30. El compilador realiza la asociación del modo siguiente: **a = (b = (c = 30))** . Debe observarse que para C++ la proposición **c = 30** tiene doble sentido: 1) se trata de una asignación y 2) se trata de una expresión que arroja el resultado **30**.

```
cout << (n=5) ; /* asigna 5 a la variable n y visualiza 5 (resultado de la expresión)*/
```

Operadores relativos de asignación

Una de las estructuras más comunes de programación son los contadores, que deben ser incrementados o decrementados a lo largo del programa, normalmente tienen la forma, siendo x el contador, x=x+b donde b es una expresión del valor que se le agregara x. Como son muy frecuentes, C/C++ tiene una forma de escribirlos más reducido, en el caso de la operación suma sería x+=b, además de esta manera reconoce más fácilmente una estructura de tipo contador y la optimiza.

Operador	Asignación abreviada	Asignación no abreviada
+ =	x + = y	x = x + y
- =	x - = y	x = x - y
* =	x * = y	x = x * y
/ =	x / = y	x = x / y
%=	x % = y	x = x % y

Operadores Relacionales

Los operadores relacionales son aquellos que comparan dos elementos a través de una relación válida entre ellos, las más comunes serían menor que (<) mayor que (>) igual (=), etc. El resultado de una comparación solo puede tomar dos valores, **verdadero** o **falso**. por ejemplo 6<8 sería verdadera, 4=5 sería falsa.

C/C++ dispone de operadores relacionales. Su simbología también es similar (< > etc), a excepción de los operadores *igual que* y *distinto que* como veremos a continuación.

Estos operadores nos permitirán plantear expresiones relacionales, las cuales al ser evaluadas arrojarán un valor de verdad: **verdadero** o **falso**. Sin embargo la representación en C/C++ no tiene un tipo original para verdadero o falso sino que usa el valor **int** cero (0) para representar al valor **falso** y de un valor **int** distinto de cero para **verdadero**.

Valor de verdad	Representación en C/C++
Falso	Cero
Verdadero	Distinto de Cero

Los siguientes son los operadores relacionales que permiten comparar operandos de tipos compatibles (no se puede comparar una palabra con un número, pero si una palabra con otra palabra y un número con otro número)

Operador	Significado	Ejemplo
==	Igual que	a == b (Es a igual que b ?)
!=	Distinto que	a != b (Es a distinto que b ?)
<	Menor que	a < b (Es a menor que b?)
>	Mayor que	a > b (Es a mayor que b?)
<=	Menor o igual que	a <= b (Es a menor o igual que b?)
>=	Mayor o igual que	a >= b (Es a mayor o igual que b?)

Utilizaremos expresiones relacionales y lógicas para varias estructuras de control de C++ , que nos permitirán plantear decisiones en el programa ya que podremos elegir un camino u otro dependiendo si el resultado es falso o verdadero.

Los operadores relacionales se asocian de izquierda a derecha y tienen menor prioridad que los operadores aritméticos por lo tanto una expresión del tipo:

a+b<10*c equivale a **(a+b)<(10*c)**

Es posible asignar el resultado de una expresión relacional a una variable ya que en el fondo es un entero:

int m=(12+3<=10); *// asigna cero **(falso)** a la variable entera **m**, porque la expresión 12+3<=10 es falsa*

Ya vimos que en C/C++ el verdadero y falso son tratados en última instancia como números enteros, el falso es el valor 0 (cero), y el verdadero es cualquier otro valor que no sea cero (como no está definido hay que tener cuidado cuando se quiere usar como número), para evitar estos problemas esta definido un tipo de variable llamada **bool**, y los valores **true** (verdadero) y **false** (falso). Luego lo siguiente se puede escribir y funciona:

```
bool a;
a=5<7; // en este caso a contendrá true porque 5 es realmente menor a 7
a=false; // ahora tendrá false
```

Operadores Lógicos

Los operadores lógicos son:

- **&&** (AND) el resultado es verdadero si ambas expresiones son verdaderas
- || (OR) el resultado es verdadero si alguna expresión es verdadera
- ! (NOT) el resultado invierte la condición de la expresión

AND y OR trabajan con dos operandos y retornan un valor lógico basadas en las denominadas tablas de verdad. El operador NOT actúa sobre un operando. Estas tablas de verdad son conocidas y usadas en el contexto de la vida diaria, por ejemplo: "si hace sol Y(AND) tengo tiempo, iré a la playa", "si NO(NOT) hace sol, me quedaré en casa", "si llueve O(OR) hace viento, iré al cine". Las tablas de verdad de los operadores AND, OR y NOT se muestran en las tablas siguientes:

El operador lógico **&& (AND)**

x	**y**	**resultado**
true	true	true
true	false	false
false	true	false
false	false	false

El operador lógico **|| (OR)**

x	**y**	**resultado**
true	true	true
true	false	true
false	true	true
false	false	false

El operador lógico **! (NOT)**

x	**resultado**
true	false
false	true

Los operadores AND y OR combinan expresiones relacionales cuyo resultado viene dado por la última columna de sus tablas de verdad. Por ejemplo:

(a<b) && (b<c)

es verdadero (**true**), si ambas son verdaderas. Si alguna o ambas son falsas el resultado es falso (**false**).

En cambio, la expresión

(a<b) ||(b<c)

es verdadera si una de las dos comparaciones lo es. Si ambas, son falsas, el resultado es falso.

La expresión " NO *a* es menor que *b*"

!(a<b)

es falsa si *(a<b)* es verdadero, y es verdadera si la comparación es falsa. Por tanto, el operador NOT actuando sobre *(a<b)* es equivalente a

(a>=b)

La expresión "NO *a* es igual a *b*"

!(a==b)

es verdadera si *a* es distinto de *b*, y es falsa si *a* es igual a *b*. Esta expresión es equivalente a

(a!=b)

Nota aclaratoria: Las operadores lógicos se conocen con los nombres de AND OR y NOT (y existe un cuarto llamado XOR que no está implementado en C/C++) pero en C/C++ los símbolos que se usan son && para AND, || para OR y ! para NOT, las palabras AND OR y NOT no funcionan en el lenguaje C/C++.

Estás operaciones cobrarán sentido cuando tengamos que tomar decisiones dentro del programa, lo cual vamos a ver un poco más adelante en el punto 2 Estructuras de control.

Evaluación en cortocircuito

C/C++ evalúa una expresión lógica de izquierda a derecha. Si el operando de la izquierda es suficiente para determinar el resultado de la proposición, no se evalúa el operando de la derecha. Por ejemplo:

En **6 < 3 && z == 4** el operando **z == 4** no llegará a evaluarse pues **6 < 3** ya decidió el resultado **false** (falso) de toda la proposición.

Otros Operadores

C++ dispone de otros operadores que describiremos más adelante conforme avancemos en el desarrollo de nuevos temas. Por ejemplo: operador de desplazamiento, el de direcciones, operador condicional, operador de ámbito de resolución, operador coma, operador **( )** , operador **[ ]**.

Precedencia de Operadores en C++

La precedencia o prioridad de un operador determina el orden de aplicación de los operadores de una expresión. En la tabla siguiente se indican los grupos de operadores según orden de prioridad para que puedan consultarlos si lo necesitan.

Prioridad y categoría	Operador	Función o significado	Asociati-vidad
1. (Prioridad más alta)	**( )**	Llamada a función	I-D
	[ ]	Subíndice de arreglos	
	->	Selector indirecto de miembro	
	::	Selector de ámbito de resolución	
	.	Selector directo de miembro	
2. Unarios	**!**	Negación lógica	D-I
	~	Complemento a uno (bits)	
	+	Más (unario)	
	-	Menos (unario)	
	++	Incremento	
	--	Decremento	
	&	Dirección	
	*****	Indirección	
	sizeof	Tamaño del operando en bytes	
	new	Alocación dinámica en memoria	
	delete	Eliminación dinámica	
3. Acceso a miembros	*****	Lee o modifica el valor apuntado	I-D
	->	Accede a un miembro de un objeto apuntado	
4. Multiplicativos	*****	Multiplicación	I-D
	/	División entera o flotante	
	%	Resto de la división entera (módulo)	
5. Aditivos	**+**	Más binario	I-D
	-	Menos binario	
6.	**>>**	Desplazamiento a la derecha	I-D

Desplazamiento	<<	Desplazamiento a la Izquierda	
7. Relacional	<	Menor que	I-D
	<=	Menor o igual que	
	>	Mayor que	
	>=	Mayor o igual que	
8. Igualdad	==	Igual que	I-D
	!=	Distinto que	
9.	&	And (manipulación de bits)	I-D
10.	^	Xor (manipulación de bits)	I-D
11.	|	Or (manipulación de bits)	I-D
12.	&&	Conjunción lógica and	I-D
13.	||	Disyunción lógica or	I-D
14. Condicional	?:	a ? b : c (significa: if a then b, else c)	D-I
15. Asignación	=		D-I
	*=		
	/=		
	%=		
	+=		
	-=		
	&=		
	^=		
	| =		
	<<=		
	>>=		
16. Coma (prioridad más baja)	,	Evaluación múltiple	

Si en una expresión aparecen operadores consecutivos de igual prioridad debe considerarse la forma de asociarlos para resolver la expresión. Por ejemplo en aritmética: ante la presencia de operadores de suma (**+**) y resta (**-**), los cuales tienen igual prioridad, C++ asocia de izquierda a derecha como corresponde al álgebra de números. La expresión **a+b-c** se resuelve: **(a+b)-c**

Deben tenerse en cuenta las reglas siguientes para el planteo de expresiones:

- · Todos los operadores de un mismo grupo tienen igual prioridad y asociatividad.
- · Si dos operadores se aplican a un operando, se aplica antes el de mayor prioridad
- · Asociatividad I-D significa que primero se aplica el operador de la izquierda y luego el siguiente hacia la derecha. Asociatividad D-I significa hacer lo contrario.
- · La precedencia o prioridad de operadores puede alterarse con los paréntesis, quienes tienen máxima prioridad.

Estructuras de Control

C++ dispone de varias estructuras para controlar la lógica de ejecución de los programas. Estas estructuras pueden clasificarse en: *condicionales o selectivas, repetitivas y de interrupción o salto no condicional*

Tipo de estructura	Sentencia C++
Repetitiva	while do-while for
Condicional o selectiva	if-else switch
Salto no condicional o interrupción	break continue

Repetitivas

while

Las expresiones dentro de esta estructura se ejecutarán repetidamente mientras la condición sea verdadera.

Sintaxis	Ejemplo

while (expresión lógica)
{
 	
acciones
 	
}
	int a=0;
	while (a<100)
	{
	cout << a<< "\n";
	a++;
	}

En donde dice expresión lógica se debe colocar un operador relacional que vimos en los puntos anteriores, ese operador dará un resultado de verdadero o falso, si es verdadero las acciones que estén entre las llaves se ejecutarán, cuando la última acción se ejecute, volverá a evaluar la expresión lógica, si vuelve a ser verdadera volverá a ejecutar las acciones entre las llaves, pero si es falsa continuará con las acciones después de las llaves.

En el ejemplo ilustrativo comenzamos con a =0, al entrar en el **while** evaluamos a<100, como a es 0 y 0 <100 es verdadero ejecuta el **cout** e imprime 0 en la pantalla (ya que a vale 0) , luego incrementa a (a++) por lo que a pasa a valer 1, entonces como es la última acción dentro de la llave vuelve a evaluar a<100 como a vale 1 y 1<100 entonces vuelve a ejecutar las instrucciones dentro de las llaves. Y así

seguirá hasta que en a++ pase de 99 a 100, cuando evalúe a<100 dará falso porque a=100 y 100 no es menor a 100 sino igual. Entonces seguirá con el programa después de las llaves.

do-while
Las acciones abarcadas por esta estructura se ejecutan repetidamente mientras la expresión lógica sea verdadera

Sintaxis	Ejemplo

do
{
 	
acciones
 	
} while (expresión lógica);
	int b=0;
	do
	{
	b++ ;
	cout << b<< "\n" ;
	} while (b<100)

Es muy parecida a la anterior, excepto que la evaluación la hace al final de la llave y no al principio con lo cual las acciones se ejecutan al menos una vez, el ejemplo hace exactamente lo mismo que el anterior, la diferencia es que con **while** si la primera vez que evalúa da falso nunca se va a ejecutar lo que esta entre llaves, en cambio en el **do while** si se va a ejecutar al menos una vez.

for
Anteriormente habíamos visto que los operadores relativos de asignación surgen para escribir en forma más simplificada una operación que es muy frecuente, en el caso del **for** es una manera de escribir la estructura **while** pero en forma simplificada en el caso que se use un contador (ver documento de problemas y soluciones).

La estructura for consta de tres expresiones a las que llamaremos exp1, exp2 y exp3 y se escribiría de la siguiente forma:

```
for(exp1; exp2; exp3)
{
        acciones
}
```

La **exp1** es la inicialización del contador, en nuestro ejemplo la variable a con el valor 0
La **exp2** es la condición del while en nuestro ejemplo a<100
La **exp3** es el incremento del contador, en nuestro ejemplo a++

Forma while	Forma for

int a=0;
while (a<100)
{
cout << a<< "\n";
a++;
}
	for (int a=0 ; a<100 ; a++)
	cout << a<< "\n" ;

Nota: si la acción es solo una instrucción no es necesario usar las llaves, el compilador interpretará que lo que está desde el paréntesis cerrado hasta el punto y coma es la sentencia a repetir, como está en el ejemplo del **for**. Esto vale tanto para **while**, **for** e **if**.

Aclaraciones:

En **exp1** no es necesario declarar el tipo, puede estar declarado antes fuera del **for**, si el tipo se declara dentro del **for** en **exp1**, esa variable será válida dentro del **for**, al salir del mismo se destruirá. Eso significa que si piensan seguir usando esa variable fuera del **for** deben declararla afuera del mismo.

En **exp3** se puede colocar cualquier operación, no necesariamente ++, pueden sumar, multiplicar, dividir, llamar a otras funciones, esta instrucción será la última en ejecutarse antes de evaluar nuevamente la condición de la **exp2**, como si estuviera puesta justo antes de la llave en el **while**.

El operador coma y la sentencia for

C++ permite a través del operador coma (**,**) realizar más de una instrucción donde generalmente se admite una.
Por ejemplo, en el ciclo **for**, la primer expresión es usada comúnmente para inicializar una variable y la tercer expresión para modificar la variable que controla la estructura. Empleando el operador coma, podemos efectuar más de una inicialización.

Ejemplo

```
 int i, j;
for (i=0, j=10; i < 10 ; i++, j--)
cout << i << " " << j << endl;
```

Condicional o Selectiva

If-else

Esta estructura sirve para tomar decisiones en el programa, sobre que hacer ante determinadas situaciones, la situación debe plantearse en los términos de verdadero o falso a través de una expresión lógica (como las del **while**), se estructura de la siguiente manera:

```
if (condición lógica)
{
        acciones a tomarse si la condición es verdadera
}
else
{
        acciones a tomarse si la condición es falsa
}
```

Si no se quiere tomar acciones si la condición es falsa, entonces se puede escribir más abreviado:

```
if (condición lógica)
{
        acciones a tomarse si la condición es verdadera
}
```

Ejemplo:

```
if(colision()) //si colisión devuelve verdadero
{
        vida--;
        reiniciarNivel();
}
else //sino
{
        moverEnemigos();
        moverHeroe();
}
```

Si las acciones son una sola sentencia se puede omitir las llaves, el punto y coma señala el final del **if**, por ejemplo:

```
if (x<1) x=1;
```

sería si x es menor a 1 hacer que x sea igual a 1. En el caso que x sea mayor o igual a 1 no hacer nada (no hay instrucción **else**). Con **else** sería:

```
if (x<1) x=1 else x-=2;
```

sería si x es menor a 1 hacer que x sea igual a 1. En el caso que x sea mayor o igual a 1 x será el valor de x menos 2 (x=x-2). Fijarse donde está el punto y coma.

Switch

Es una especie de **if** múltiple, es para el caso especial que tengamos una variable de la cual según los valores que tome tengamos varías alternativas, por ejemplo:

Queremos dar un premio según un sorteo, digamos: una vida extra, más puntos, más dinero, mejorar armamento, más energía, o nada.

Calculamos un número al azar entre 0 y 19 y decidimos que si sale 0 tendrá una vida extra, si sale 1 más puntos, si sale 2 más dinero, sale 3 mejora el armamento, si sale 4 más energía y si sale cualquier otro número no pasa nada.

```
int a=rand()%20;
switch(a)
{
   case 0:
     ++vida;
     break;
   case 1:
     puntos+=100;
     break;
   case 2:
     dinero+=20;
     break;
   case 3:
    ++nivelArmamento;
    break;
   case 4:
     energia+=10;
     break;
   default:
     cout<<"Siga participando";
}
```

Aclaraciones:

Aunque en el ejemplo hay solo una instrucción después de cada **case** en realidad puede contener todas las instrucciones que se quiera, ya que el **break** marca el final, si el **break** no se colocara seguiría con las siguientes instrucciones en el siguiente **case** hasta encontrar un **break** para terminar.

default se ejecuta cuando no entra en ninguno de los case anteriores.

default también es opcional, si no se quiere hacer nada no es necesario colocarlo.

Salto no condicional o interrupción

break y continue

Estas sentencias funcionan dentro de las estructuras repetitivas (**while**, **do while** y **for**).

Ambas sentencias interrumpen la ejecución del grupo de acciones abarcadas entre las llaves.

La interrupción, **break**, sale del bucle y continua con la ejecución de la sentencia siguiente a las llaves. **continue** en cambio, vuelve a comenzar una nueva iteración.

Ejemplo de break	Ejemplo de continue

int a=0;
while (a<5)
{
a++;
if a == 4 break;
cout << a;
}
	int a=0;
	while (a<5)
	{
	a++;
	if a == 4 continue;
	cout << a;
	}

Salida: 1 2 3	**Salida: 1 2 3 5**

exit( )

Ésta función del lenguaje C++ permite interrumpir un programa, devolviendo un valor al entorno o plataforma empleado (WINDOWS, LINUX, DOS, UNIX).
Se halla definida en **stdlib.h**, por lo cual se debe incluir a este archivo en la cabecera del programa y devuelve el valor de su argumento: **void exit(int)**
El valor entero que se indica como argumento se retorna al proceso padre que invocó al programa, corresponde a cero si se ha interrumpido el programa con éxito. Un código distinto de cero indica que la interrupción del programa se ha debido a un error. El valor que devuelve el **exit** es el mismo que devuelve el **return** de la función **main()**.

Ejemplo

```
char resp;

cout<<“Desea continuar operando con el programa (S/N)?”;
cin >> resp ;
resp = toupper( resp ); // pasa a mayúsculas
if (resp==´S´) exit(0);
```

Funciones de biblioteca de C++

Las instrucciones propias del lenguaje C/C++ son pocas, alrededor de 32, el resto son funciones que se agrupan por bibliotecas y están disponibles al ser llamadas (las bibliotecas) con la instrucción **#include**.

Como C/C++ depende en su mayor parte de las funciones y se decidió normalizarlas o estandarizarlas para garantizar que todas las funciones básicas tengan el mismo nombre no importa el compilador ni el sistema operativo en el que se compile. Entonces se creo el ANSI/ISO C++ que es el estándar de las funciones de C/C++, todo compilador de C/C++ que se precie de serlo las traerá incorporadas, listas para usar. Existe mucha bibliografía sobre ellas, y en Internet hay miles de páginas describiendo su uso.

Esto garantiza que cualquier programa escrito en C/C++ que use estas funciones podrá ser compilado sin cambios en otro sistema o compilador.

Para usarlas, el programador C/C++ solo debe considerar el tipo de argumento requerido, el tipo de resultado que devuelve y el nombre del archivo de inclusión donde se halla el prototipo de la función para indicarlo en la sentencia #**include** correspondiente. Más adelante aprenderemos a crear nuestras propias funciones.

Ejemplo

```
#include <iostream.h>
#include <math.h> // archivo con el prototipo de sin(x)y M_PI
int main ( ) {
 int ang;
 cout << "Ingrese un ángulo en grados:";
 cin >> ang;
 float  angr = ang*M_PI/180; // pasa a radianes el ángulo
 cout << "El seno del ángulo es:"<< sin(angr);
 return 0; }
```

Funciones no ANSI/ISO C++

Aún así existen funciones que no están contempladas en el ANSI/ISO C++ con lo cual nos obliga a nosotros escribirlas (lo cual veremos como hacerlo más adelante) o usar las escritas por otras personas, estas se llaman bibliotecas o librerías de terceros.

La ventaja de estás librerías es que tiene funciones que nos ahorran mucho trabajo, la desventaja es que si le pasamos a otra persona nuestro fuente para que lo compile, también tenemos que pasarle la librería.

Existen tantas librerías como temas se les ocurra, existe además, dos modalidades, las comerciales que hay que pagar por usarlas y las gratuitas. También se las puede conseguir con fuentes o sin fuentes (solo el código compilado, el enlazador se encarga de juntar las partes de nuestro programa con las funciones).

Las librerías de terceros no están obligadas a funcionar en todos los compiladores y todos los sistemas.

Conio2

Como ejemplo vamos a instalar para el ZinjaI una librería llamada conio2 que la pueden encontrar en sourceforge (junto con miles de software libre).

En primer lugar, esta librería esta hecha para Win32, así que en otros sistemas operativos no va a funcionar.
En segundo lugar esta hecha para MiniGW/Dev C++. MiniGW es el compilador del ZinjaI, pero Dev C++ es otro IDE, si bajan el archivo está preparado para intalarse en el Dev C++ no en el ZinjaI, para instalarlo hay que hacer unos pasos extras.

Esta biblioteca emula la biblioteca propietaria conio de Borland, pero además la extiende para usarla sobre cout con los <<. Existen otros emuladores de conio, incluso para Linux (si buscan en SourceForge van a encontrar al menos uno), pero este es el que vamos a usar.

Conio es una biblioteca de manipulación de la entrada y salida de consola, permite imprimir en cualquier lugar de la pantalla con colores, así como un sistema de detección de pulsación de teclas rudimentario. Con ella podremos mejorar sustancialmente la presentación gráfica de nuestros programas de consola.

Primero que nada, no bajen el instalador que esta en SourceForge porque no van a poder usarlo, bajen el zip que tenemos preparado.

Dentro del mismo van a encontrar cinco archivos:

conio.chm este archivo es la ayuda con las funciones de biblioteca, en ingles
conio2.h este archivo es el prototipo para C
constream este archivo es el prototipo para C++ (el que modifica cout)
libconio.a este archivo es la biblioteca compilada, la va a usar el enlazador
ZinjaI.txt este es un archivo txt con estas mismas instrucciones de instalación.

conio2.h y **constream** deben ser copiados a la carpeta *include* del MinGW **libconio.a** debe ser copiado a la carpeta *lib* del MinGW

El ZinjaI coloca el MinGW dentro de su carpeta de instalación, por defecto ZinjaI se instala en *c:\Archivos de Programas* en la carpeta ZinjaI.

En resumen, por defecto sería: *c:\Archivos de Programas\ZinjaI\MinGW*

conio.chm colóquenlo donde quieran, téngalo a mano por si necesitan algo de ayuda. Si les molesta el ingles hay mucha ayuda para la versión de conio de Borland en castellano, pueden usarla porque es compatible, pero recuerden que conio2 tiene funciones mejoradas esas no están en las de Borland.

Y para finalizar, hay que indicarle al enlazador que cuando enlace tenga en cuenta a **libconio.a**, para eso, en el ZinjaI, debemos ir al menú *Ejecucion* donde dice O*pciones...*, y agregar en donde dice *Parámetros extras para el compilador*: **-lconio** (la primera después de - es una ele después el nombre del archivo pegado a la ele) separado con un espacio de la anterior.

Con esto ya van a tener un abanico de grandes posibilidades para mejorar sus programas de consola.

Bibliografía:

Ing. Horacio Loyarte. Estructuras de Control [UNL-FICH] [2008]