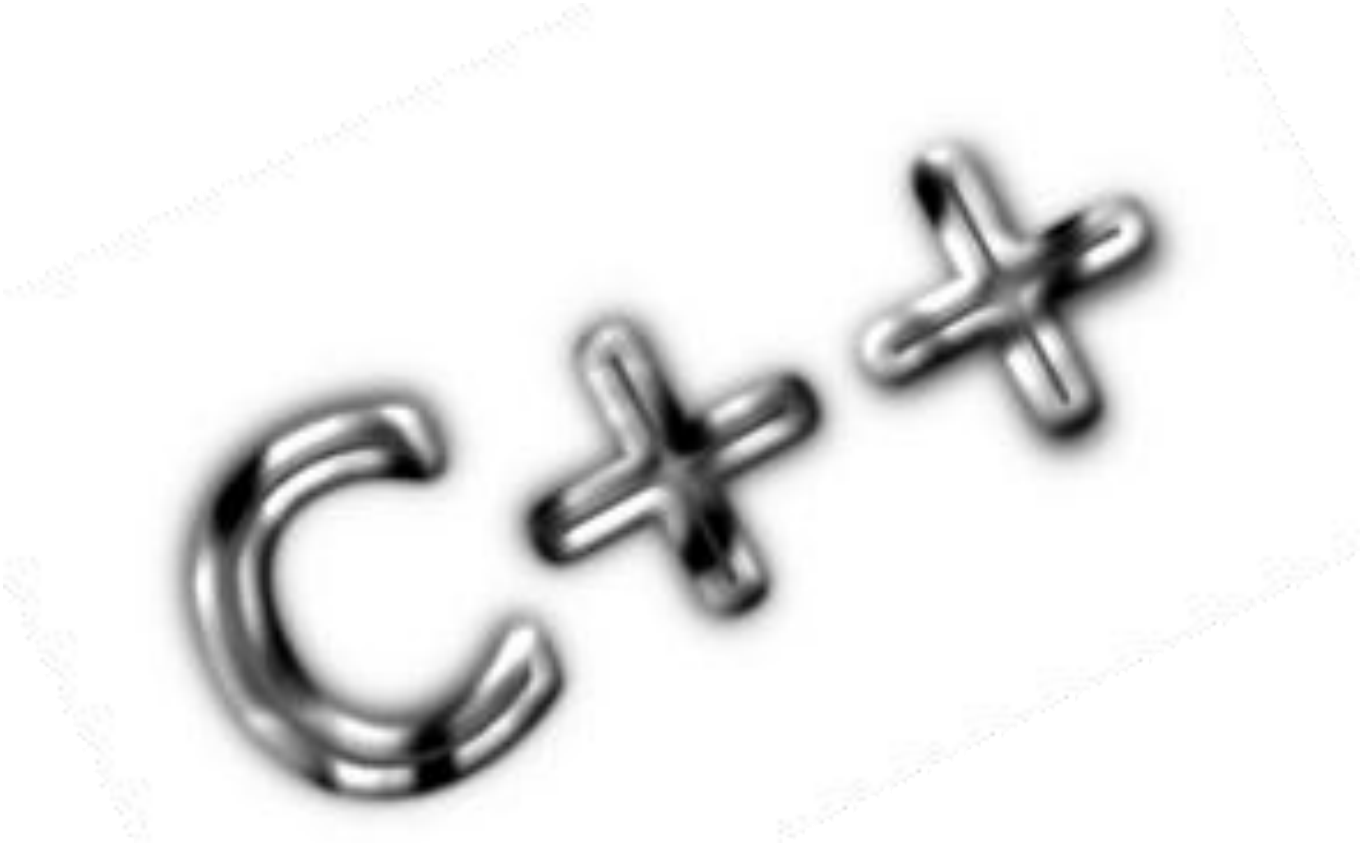

INTRODUCCIÓN A C++

TRANSVERSAL DE PROGRAMACIÓN BÁSICA

INGENIERÍA DE SISTEMAS

En el presente documente se hace una breve introducción al lenguaje de C++. Así mismo se presenta información correspondiente a la estructura general del programa y a las diferentes reglas sintácticas del lenguaje

CRISTIAN GUILLERMO GARCÍA
MONITORIA 2012-3 | Universidad Distrital francisco José de Caldas

INTRODUCCIÓN A C++

Tabla de contenido

INTRODUCCIÓN A C++

1. INTRODUCCIÓN

El presente documento pretende servir de guía para la enseñanza en la programación bajo el lenguaje C++. Pese a que se ha desarrollado siguiendo los lineamientos establecidos en el syllabus de la asignatura transversal de programación básica de la universidad distrital Francisco José de Caldas, es posible utilizarla como un manual en cualquier campo o ámbito siempre y cuando se relacione con el aprendizaje del lenguaje mencionado.

Es necesario resaltar que este documento es completamente tomado de [1] de quien no se precisa autor. Sin embargo, se han hecho algunos modificaciones en cuanto al formato de presentación.

2. ELEMENTOS DEL LENGUAJE

Comenzaremos viendo los elementos más simples que integran un programa escrito en C++, es decir palabras, símbolos y las reglas para su formación.

2.1. Identificadores

Son nombres elegidos por el programador para representar entidades (variables, tipos, constantes, etc.) en el programa. El usuario puede elegir cualquier identificador excepto un pequeño grupo que se reserva para usos específicos.

C++ distingue entre letras **mayúsculas y minúsculas**, por lo que a efectos del programa serán identificadores distintos *hola*, *Hola* y *hoLa*.

La gramática que define la sintaxis de los identificadores es la siguiente:

```
<ident> ::= <carácter_ident> {<carácter_ident>|<dígito>}
<carácter_ident> ::= a | b |···| z | A | B |··· | Z | _
<dígito> ::= 0 | 1 | 2 | ··· | 9
```

No se impone en principio ningún límite en cuanto a la longitud de los identificadores. Dichos límites dependerán del compilador que se esté empleando. Más aún, en esta asignatura no emplearemos identificadores tan largos como para darnos problemas en este sentido.

2.2. Palabras Reservadas

Tienen un significado predeterminado para el compilador y sólo pueden ser usadas con dicho sentido. En C++ se escriben siempre con minúsculas.

and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	cdecl	char
class	compl	const	const_cast	continue	default
delete	do	double	dynamic cast	else	enum
explicit	export	extern	false	float	for
friend	goto	if	inline	int	long
mutable	namespace	new	not	not_eq	operator
or	or_eq	private	protected	public	register
reinterpret_cast		return	short	signed	sizeof
static	static cast	struct	switch	template	this
throw	true	try	typedef	typeid	typename
union	unsigned	using	virtual	void	volatile
wchar_t	xor	xor_eq	while		

2.3. Identificadores Estándares

Tienen un significado predefinido dentro del lenguaje, pero a diferencia de las palabras reservadas, el programador puede redefinirlos dentro de su programa (aunque esto normalmente no es aconsejable). Dos ejemplos de dichos identificadores estándares serán presentados en este tema: los flujos de entrada y salida *cin* y *cout*.

2.4. Literales constantes

Se distingue entre números reales, números enteros y cadenas de caracteres.

```
<literal >      ::= <num_entero> | <num_real> | <car> | <cadena_car>
<num_entero>    ::= [+|-] <dígito> {<dígito>}
<num_real>      ::= <num_entero>.{<dígito>} [<factor_esc>]
<factor_esc>    ::= (e|E) <num_entero>
<car>           ::= ' <carácter> '
<cadena_car>    ::= " {<carácter>} "
```

Cuando se usa **e** en un literal constante real se dice que el número está expresado en *punto flotante* (notación científica). En cuanto a las cadenas de caracteres (o *strings*), en C++ se

expresan mediante una secuencia de caracteres cualesquiera[1] encerrados entre comillas simples.

2.5. Delimitadores

Son símbolos que indican el comienzo o fin de una entidad.

```
!   %   ^   &   (   )   -   +   =
{   }   ~   [   ]   \   ;   '   :
<   >   ?   ,   .   /   "
```

2.6. Comentarios

Un comentario es una secuencia de caracteres que es ignorada por el compilador. Se usan para documentar el programa, de manera que aunque no contribuyan a resolver el problema, sí ayudan a mejorar la comprensión del programa. Se pueden insertar en el programa de dos maneras: escribiendo un texto entre los símbolos /* y */ (este texto se puede extender a lo largo de varias líneas), o usando el símbolo // (se interpretará como comentario todo lo que siga hasta llegar al final de la línea).

Debe tenerse cuidado para no encerrar trozos de código útil dentro de un comentario, ya que un comentario puede extenderse a lo largo de múltiples líneas. Encerrar trozos de código como comentarios es útil cuando se está desarrollando un programa para comprobar si ciertas partes funcionan o no, pero es un error muy común encerrar por descuido otras zonas de código que son necesarias o bien olvidarse de sacar órdenes de dentro de un comentario.

El anidamiento de comentarios (poner comentarios dentro de otros comentarios) puede dar problemas en algunos compiladores. **Nunca emplearemos este tipo de anidamiento**.

2.7. Separadores

Son espacios en blanco, tabuladores, retornos de carro, fin de fichero y fin de línea. Habrá ocasiones en que se pongan para dar legibilidad al programa, y otras por necesidad. Los comentarios son considerados también separadores por el compilador.

[1] El símbolo \ tiene un significado especial en C/C++ como introductor de caracteres especiales (e.g., \n para un salto de línea, \t para una tabulación, \" para las comillas dobles, etc.), lo que deberá tenerse en cuenta a la hora de su introducción como literal. Si se desea usar literalmente, utilícese '\\'.

INTRODUCCIÓN A C++

3. ESTRUCTURA GENERAL DE UN PROGRAMA

Comenzaremos viendo la estructura global de un programa escrito en C++ para a continuación ir desglosando y analizando cada uno de sus componentes. El programa que se muestra es el famoso "Hola Mundo", un programa que únicamente muestra el mensaje "Hola Mundo" en pantalla antes de finalizar.

```
/*
 * Nombre:       HolaMundo.cpp
 * Autor:        Johann Sebastian Mastropiero
 * Fecha:        01/01/2001
 * Descripción: Este es el programa "Hola Mundo".
 *
 */


#include <iostream>   // Para usar cout
#include <cstdlib>    // No hace falta en este caso

using namespace std;  // Para usar cout

int main ()
{
   cout << "Hola Mundo" << endl;
}
```

Como puede apreciarse, el programa comienza con un comentario a lo largo de siete líneas, y en el cual se da información sobre el nombre, autor, fecha y propósito del programa. Esto no es sintácticamente necesario como es bien sabido, pero siempre lo incluiremos como norma de estilo. A continuación nos encontramos con dos operaciones de inclusión. Sin entrar en detalles sobre lo que internamente significan dichas inclusiones[2], lo que hacemos mediante las mismas es indicarle al compilador que pretendemos hacer uso de las funcionalidades que nos proporcionan unas ciertas bibliotecas (en este caso predefinidas; en temas posteriores aprenderemos a definirlas por nuestra cuenta). En este caso concreto queremos emplear la biblioteca de entrada/salida `iostream` para lo cual tenemos que incluirla. La otra inclusión que se realiza corresponde a una biblioteca (`cstdlib`) que no es necesaria en este caso dada la simplicidad del programa, pero que sí lo será en la mayoría de los programas que realicemos más adelante. La línea **using namespace** `std`; la ignoraremos por el momento.

[2] Todas las órdenes que comienzan por la almohadilla (#) van dirigidas al preprocesador, un programa que repasa el código y realiza diversas modificaciones sobre el mismo antes de proceder a la compilación del mismo.
En este caso se le está indicando que incluya en el texto del programa el contenido completo de los ficheros `iostream.h` y `cstdlib.h`.

El algoritmo comienza en la línea **int** `main()`. Aquí aparecen varios elementos cuyo significado veremos en temas posteriores. Lo que sí debemos saber es que todo programa que realicemos tendrá un algoritmo principal (el que empezará a realizarse cuando ejecutemos el programa), y que el comienzo de dicho algoritmo principal se indica mediante una línea de estructura como la mostrada[3].

A continuación encontramos el cuerpo del algoritmo principal, que abarca desde la llave de apertura ({) hasta la llave de cierre (}). Siempre aparecerán estas llaves delimitando el cuerpo de un algoritmo. En este caso, el algoritmo consta de una sola orden, mostrar en pantalla el mensaje "Hola Mundo" y saltar a la línea siguiente. Como puede verse, esto se expresa haciendo uso del objeto predefinido (en `iostream` precisamente) `cout`. Todo lo que redirijamos (mediante el empleo del operador <<) hacia dicho objeto será mostrado en pantalla. En el ejemplo se redirige en primer lugar la cadena "Hola Mundo", y a continuación la indicación de salto de linea (`endl` – abreviatura de end of line). La instrucción termina mediante un punto y coma (;), que la separa de instrucciones posteriores (es preciso poner dicho delimitador incluso si como es el caso aquí, no hay una instrucción posterior).

4. CONSTANTES. VARIABLES Y TIPOS DE DATOS

4.1. Constantes y Variables

En un programa intervienen objetos sobre los que actúan las instrucciones que lo componen. Algunos de estos objetos tomarán valores a lo largo del programa. Dependiendo de si pueden cambiar de valor o no, podemos distinguir dos tipos de objetos:

Constante: Objeto –referenciado mediante un identificador- al que se le asignará un valor que no se podrá modificar a lo largo del programa.

Variable: Objeto –referenciado por un identificador– que puede tomar distintos valores a lo largo del programa.

Será misión del programador asignar el identificador que desee a cada constante y variable. El identificador o nombre de cada objeto sirve para identificar sin ningún tipo de ambigüedad a cada objeto, diferenciándolo de los demás objetos que intervienen en el programa. En C++ hay que indicar (de la manera que más adelante se indicará) el nombre de las constantes y variables que vamos a usar, para que el compilador pueda asociar internamente a dichos nombres las posiciones

[3] Las variantes que pueda tener esta línea de comienzo se verán más adelante cuando se tengan los conceptos precisos para su compresión. De momento asumiremos que siempre es esa la forma de comenzar el algoritmo principal.

de memoria correspondientes. Para esto último es preciso también que junto con dichos nombres, se indique explícitamente para cada constante o variable el tipo de los datos que pueden contener.

4.2. Tipos de Datos

En C++ existen una serie de tipos básicos predefinidos, así como herramientas para la construcción de tipos más complejos. Los principales tipos de datos predefinidos en C++ son los siguientes:

- *Enteros:* el tipo básico es int. Las variables definidas de este tipo podrán tomar típicamente valores entre -2147483648 y 2147483647. En determinadas circunstancias podremos querer modificar este rango, para lo que disponemos de dos tipos de modificadores *short/long y signed/unsigned.*

    - *short int:* el rango de representación es menor (–32768 a 32767 típicamente), y por lo tanto también lo es la memoria ocupada (la mitad).
    - *long int*: mayor rango de representación[4] (e.g., -9223372036854775808 a 9223372036854775807), y más memoria necesaria (el doble).
    - *unsigned int*: ocupa la misma memoria que un int, pero sólo toma valores positivos (e.g., 0 a 4294967295).
    - *signed int*: es equivalente a un int.

  Los modificadores mencionados se pueden combinar, e.g., *long unsigned int* (ocupa la misma memoria que un *long int*, pero sólo almacena números positivos, con lo que el rango de representación es mayor). Adicionalmente, puede suprimirse la indicación *int* en todos los casos anteriores, asumiéndose ésta por defecto (e.g., *unsigned* es lo mismo que *unsigned int*).

- *Carácter:* se emplea el tipo char.

- *Lógico:* se utiliza el tipo bool. Los objetos de este tipo sólo pueden tomar dos valores:
- true (cierto) y false (falso).

- *Reales:* los números reales (en realidad números en punto flotante, ya que los números reales en sentido estricto no son representables en ningún computador digital) se representan mediante los tipos *float* y *double*. El segundo tipo permite representar números con mayor magnitud y precisión, empleando para ello el doble de memoria que *float*. Si se precisa mayor rango de representación o de precisión puede emplearse el

[4] El rango de representación en cada caso depende tanto del compilador como del sistema subyacente. Particularizando para la versión de Dev-C++ sobre sistemas Windows que estamos empleando (v4.9.8.1), se da la circunstancia de que internamente un *long int* es equivalente a un *int*. Por ese motivo el rango de representación es exactamente el mismo.

modificador *long* sobre *double*, i.e., *long double*. No se pueden emplear los modificadores signed/unsigned sobre double.

Todos estos tipos tienen dos propiedades en común: cada uno está formado por elementos *indivisibles* o atómicos que además están *ordenados*. Los tipos de datos con estas características se denominan *tipos de datos escalares*. Cuando decimos que un valor es atómico, nos referimos a que no contiene partes componentes a las que pueda accederse independientemente. Por ejemplo, un carácter es indivisible, mientras que la cadena de caracteres "Hola" no. Cuando hablamos de que un conjunto de valores está ordenado, queremos decir que los operadores relacionales estándares (mayor que, menor que, igual, etc.) pueden aplicarse a los mismos.

Para declarar una variable basta con indicar el tipo de la misma y su nombre. La declaración termina con el punto y coma. Los siguientes ejemplos muestran declaraciones válidas:

```
int num;
unsigned x,y;
long double masaPlanetaria;
char car = 'c', letra;
short int corto;
```

Como puede apreciarse en el último ejemplo, es posible dar un valor inicial a una variable a la vez que se declara. Así, la sintaxis BNF de una declaración de variable es:

```
<decvar> ::= <tipo> <ident> [= <literal>] {, <ident> [= <literal>]};
```

La declaración de constantes es muy similar a la de variables, con dos diferencias fundamentales: hay que usar la palabra reservada *const* para indicar que se está definiendo una constante, y la asignación de un valor inicial no es opcional, sino obligatoria:

```
<decconst> ::= const <tipo> <ident> = <literal>;
```

Ejemplos de declaraciones de constantes válidas[5] son los siguientes:

```
const int HORAS = 24;
const double NUMERO_AVOGADRO = 6.023e+23;
const short ALUMNOS = 100;
const char ACEPTAR = 'S';
```

[5] Aunque no es una imposición sintáctica del lenguaje, como regla de estilo se emplean en este caso mayúsculas para designar a las constantes.

En algunos compiladores se permite que no se indique el tipo de una constante, asumiéndose que es entera por defecto. **Siempre indicaremos explícitamente el tipo de las constantes que definamos.**

Tanto las declaraciones de variables como de constantes pueden realizarse de dos maneras: *global y local*. El primer modo consiste en situar dicha declaración al principio del código, justo después de las inclusiones. El segundo modo consiste es situarlas dentro del programa principal, en la zona delimitada por las llaves. Ejemplos:

GLOBAL

```
int num, x, y;

int main()
{
```

LOCAL

```
int main()
{
        int num, x, y;
```

La diferencia entre ambas formas de declaración será evidente cuando en temas posteriores se introduzca la noción de función. De momento, la regla práctica que seguiremos será definir siempre variables y constantes de manera local.

C++ permite que las declaraciones locales se realicen en cualquier punto del algoritmo situado entre el comienzo ({) y el final (}) del mismo. **Siempre declararemos los objetos locales inmediatamente después de la llave de apertura.**

5. ACCIONES BÁSICAS Y EXPRESIONES

Dentro del cuerpo del algoritmo principal se indicarán las acciones que conducirán a la resolución del problema para el que dicho algoritmo fue confeccionado. La acción más básica que se puede realizar es la asignación. Su sintaxis es:

```
<asignación> ::= <ident> = <expresión>;
<expresión>  ::= <literal> | <ident> | <op_unario> <expresión> |
                 <expresión> <op_binario> <expresión> | (<expresión>)
```

El caso más simple es aquél en el que se asigna un valor literal a una variable, o el valor de una variable a otra, e.g.,

```
int main()
{
        int num1, num2;

        num1 = 10;
        num2 = num1;
```

En general, se empleará una expresión para asignar valores a las variables. Las posibles expresiones que se pueden utilizar dependerán en principio del tipo de los objetos involucrados en las mismas. Sin embrago, debe tenerse en cuenta que C++ no es un lenguaje con una comprobación de tipos muy fuerte. Ello quiere decir que no se controla el tipo de los objetos que forman parte de una operación. Por ejemplo, es lógico suponer que para sumar dos objetos, éstos han de ser de un tipo numérico (enteros o reales, en cualquiera de sus variantes). Sin embargo, esto no se comprueba, permitiéndose sin ir más lejos que se sumen dos caracteres. Lo que ocurre internamente es que se produce una operación de conversión (casting) mediante la cual los valores de tipo carácter se convierten en valores numéricos. Dicha conversión se puede producir entre cualesquiera dos tipos. De hecho podemos forzar a que ocurra indicándolo explícitamente. Para ello es preciso indicar el tipo al que queremos realizar la conversión, y el objeto que queremos convertir, este último entre paréntesis. Por ejemplo, si *letra* es un objeto de tipo **char** y queremos convertir su valor a entero indicaremos **int**(*letra*). Esto no modifica el tipo ni el valor de *letra*, sino que permite obtener un valor de conversión que podremos mostrar en pantalla, emplear dentro de una expresión más compleja, etc.

En los casos que sea preciso realizaremos las operaciones de conversión de manera explícita. **Nunca dejaremos que el sistema las realice de manera implícita.**

C++ proporciona operadores[6] para realizar las operaciones más usuales, como pueden ser las siguientes:

- *Operaciones aritméticas:* se proporcionan los operadores binarios +, –, *, / y % con el significado de suma, resta, multiplicación, división (entera o en punto flotante dependiendo de los operandos) y resto o módulo respectivamente. Asimismo, existe el operador unario – para la inversión de signo. Ejemplos:

```
int num1, num2, num3;

...
num1 = -10*num2;
num2 = num1%7;
num3 = num1/num2;
```

- *Operaciones relacionales:* se proporcionan los operadores binarios ==, !=, >, <, >=, <= con el significado respectivo de comparación de igualdad, desigualdad, mayor que, menor que, mayor o igual que y menor o igual que. El resultado de la operación relacional será un valor lógico, e.g.,

[6] Puede revisar el material de operadores en donde se profundiza un poco más acerca de estos y su implementación.

```
int num1, num2;
bool b1, b2;

...
b1 = num2 == num1;
b2 = (num1-num2) >= (num2*2);
```

- *Operaciones lógicas:* se proporcionan los operadores binarios && y || con el significado de conjunción y disyunción lógica respectivamente. También se dispone del operador unario ! con el significado de negación. Ejemplos:

```
int num1, num2;
bool b1, b2;

...
b1 = !(num2 == num1); // equivale a num2!=num1
b2 = (num1>num2) && (num2<0);
```

C++ proporciona una sintaxis especial en el caso bastante frecuente en el que el valor de una variable use como operando en una expresión aritmética ó logica cuyo resultado se asigna a la propia variable. Dicha sintaxis permite simplificar asignaciones del tipo *v = v <op> E*, permitiendo expresarlas como *v <op>= E*. Por ejemplo:

```
int x, y, z;

...
x *= x+y/z; // equivale a x = x * (x+y/z);
z += 1;     // equivale a z = z + 1;
```

Precisamente en relación con este último ejemplo, C++ proporciona una forma sintáctica aún más simple mediante el uso de los denominados operadores de incremento (++) y decremento (--). Éstos son operadores unarios que sólo pueden aplicarse sobre variables (no sobre expresiones) y que modifican el valor de la misma, incrementándola o decrementándola según corresponda. Por ejemplo:

```
int x, y, z;

...
z++;     // equivale a z = z + 1 o a z += 1
--x;     // equivale a x = x - 1 o a x -= 1
```

Tanto el ++ como el -- pueden ponerse antes o después de la variable que deseamos incrementar o decrementar. La diferencia entre ambas formas sintácticas estriba en su uso dentro de expresiones más complejas: si el operador se coloca antes de la variable, ésta se incrementa (o decrementa) y es dicho valor modificado el que se emplea en la expresión; si el operador se sitúa después, se usa el valor actual de la variable en la expresión y luego se modifica. Por ejemplo:

```
int x, y, z;

...
x = --y * z++;   // equivale a y = y - 1
                 //             x = y * z;
                 //             z = z + 1;
```

Con el fin de mejorar la legibilidad del código –y a pesar de que es sintácticamente correcto– **nunca emplearemos los operadores de incremento o decremento dentro de expresiones.**

En el caso de que tengamos una expresión compleja con varios operadores, C++ proporciona reglas de precedencia que permiten determinar cuál es el orden en el que se aplican los operadores. Dichas reglas son las siguientes:

1. Los operadores unarios se aplican antes que los binarios. Si hay más de un operador unario aplicado sobre la misma variable o subexpresión, el orden de aplicación es de derecha a izquierda.

2. Los operadores binarios se aplican según el siguiente orden de precedencia[7]:

   - Nivel 1 (mayor precedencia): *, /, &
   - Nivel 2: +, -
   - Nivel 3: <, <=, >, >=
   - Nivel 4: ==, !=
   - Nivel 5: &&
   - Nivel 6: ||
   - Nivel 7 (menor precedencia): =, *= , +=, /=, -=, %=, &=, |=

   En caso de que aparezcan varios operadores con igualdad de precedencia, se evalúan de izquierda a derecha. Estas reglas pueden alterarse mediante el uso de paréntesis.

[7] Para una mejor explicación y mayor profundización consulte el material sobre operadores en C++

INTRODUCCIÓN A C++

6. ENTRADA Y SALIDA

La entrada y salida (E/S en lo sucesivo) de información la gestionaremos en C++ mediante la funcionalidad que nos proporciona la biblioteca *iostream*, ya mencionada anteriormente. La palabra *stream* significa "flujo" o "corriente" en inglés, y nos da una indicación de cómo funcionan los aspectos de E/S: los dispositivos de salida (e.g., el monitor) se modelan como sumideros a los que va llegando un flujo de información; del mismo modo, los dispositivos de entrada (e.g., el teclado) son fuentes de las que surge un flujo de información. Tanto en un caso como en otro, lo único que necesitamos son operadores para insertar información en el flujo de salida, o extraer información del flujo de entrada. La biblioteca *iostream* nos proporciona dichos operadores así como unos identificadores estándar para los flujos más comunes.

Cuando incluimos la biblioteca `iostream` disponemos automáticamente del flujo `cin` (asociado a la entrada a través de teclado), y el flujo `cout` (asociado a la salida a través de pantalla). Sobre dichos flujos podemos aplicar respectivamente las operaciones de extracción (>>) y de inserción (<<). Empezando por este último, imaginemos el siguiente código:

```
#include <iostream>

using namespace std;

int main()
{
    int x = 10, y = -1;
    char c = 'A';
    double d = 10.124567;

    cout << x << y << endl
         << c << d << "Se acabó";
}
```

La ejecución del mismo produciría en pantalla la salida:

```
10-1
A10.124567Se acabó
```

Como puede apreciarse, la gestión de la salida de datos es muy cómoda y homogénea. Otro tanto ocurre con la gestión de la entrada de datos. En este caso, simplemente tenemos que extraer los datos que vayamos precisando del flujo `cin`:

```
#include <iostream>

using namespace std;

int main()
{
    int x, y;
    char c;
    double d;

    cin >> x >> y >> c >> d;
}
```

En el código que se muestra, se leen de teclado los valores de `x, y, c` y `d` sucesivamente. Así pues, en general tendremos la siguiente sintaxis para la E/S en C++:

<entrada> ::= cin >> <ident> {>> <ident>};

<salida> ::= cout << (<expresión> | endl) {<< (<expresión> | endl)};

El siguiente programa muestra un ejemplo de uso combinado de operaciones de E/S. En este caso se trata de leer dos números y mostrar su suma en pantalla:

```
#include <iostream>

using namespace std;

int main()
{
    int a, b; // los números que queremos sumar

    cout << "Introduzca dos números: " << endl;
    cin  >> a >> b;
    cout << "Su suma es " << a + b << endl;
}
```

Nótese un hecho importante: en la última línea aparecen combinados los operadores de inserción (<<) y de suma (+). El funcionamiento del programa es correcto, ya que el operador de inserción tiene una precedencia menor que la suma. Por ello, primero se realiza esta última, y es el resultado lo que se inserta en el flujo de salida. Si volvemos a la lista de niveles de precedencia mostrada al final de la sección anterior, los operadores de flujo se situarían en un nivel intermedio entre el 2 (suma y resta) y el 3 (operaciones relacionales).

Para finalizar este tema, debe mencionarse el hecho de que es posible alterar el formato de impresión de los datos en pantalla mediante el uso de la funcionalidad que proporciona la biblioteca *iomanip*. Mediante la misma es posible indicar el número de decimales de precisión con el que queremos escribir un número en punto flotante, el número de espacios que vamos a emplear para

escribir un dato, caracteres de relleno, etc. Esto se realiza mediante la inserción en el flujo de salida de modificadores, que afectarán a los datos que se introduzcan a continuación. Alguno de estos modificadores son los siguientes:

- `setprecision()`. Para indicar el número de dígitos significativos en un dato en punto flotante. Afecta a todos los datos que se introduzcan con posterioridad.

- `setw()`. Permite indicar el número de espacios que se emplearán para escribir un dato, alineando al mismo a la derecha dentro de dicho espacio. Si el espacio requerido es mayor que el indicado, el modificador se ignora. Sólo afecta al dato que se indica a continuación.

- `setfill()`: Para especificar el carácter de relleno que se empleará para los espacios no usados al escribir un dato (según un modificador setw()).

A continuación se muestra un ejemplo:

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
     const double a1 = 1.1;
     const double a2 = 10.12;
     const double a3 = 101.179;

     cout << setprecision(2) << a3 << endl;
     cout << setprecision(4) << a3 << endl;
     cout << setprecision(5) << a3 << endl;
     cout << "a1" << setw(10) << setfill('.') << a1 << endl;
     cout << "a2" << setw(10) << setfill('.') << a2 << endl;
     cout << "a3" << setw(10) << setfill('.') << a3 << endl;
}
```

Este código produciría la siguiente salida en pantalla:

```
1e+02
101.2
101.18
a1.......1.1
a2.....10.12
a3....101.18
```

7. REFERENCIAS

[1] D. d. L. y. C. d. l. Computación, 2005-2006. [En línea]. Available: http://es.scribd.com/doc/94274637/tema2. [Último acceso: 08 11 2012].