UNIVERSIDAD DE MALAGA
DPTO. DE LENGUAJES Y C. DE LA COMPUTACION
E.T.S. DE INGENIERIA INFORMATICA

FUNDAMENTOS DE LA PROGRAMACIÓN

TEMA II

INTRODUCCIÓN A UN LENGUAJE DE PROGRAMACIÓN

II.1. Introducción a C++
- *II.1.1. La Evolución de C++*
- *II.1.2. Ejemplo de programa en C++.*
- *II.1.3. Elementos básicos de C++.*
- *II.1.4 Esquema general de un programa en C++*

II.2. Tipos de datos simples.
- *II.2.1. Tipos simples predefinidos.*
- *II.2.2. Tipos simples definidos por el programador.*
- *II.2.3. Operadores.*
- *II.2.4. Conversiones de tipos.*

II.3. Constantes, variables y asignaciones.
II.4. Entrada y salida básicas.
II.5. Flujo de control.
II.6. Expresiones lógicas o booleanas.
II.7. Estructuras de selección.
- *II.7.1. Estructura if.*
- *II.7.2. Estructura switch.*

II.8. Estructuras de iteración.
- *II.8.1. Estructura while.*
- *II.8.2. Estructura do while.*
- *II.8.3. Estructura for*
- *II.8.4 Diseño de Bucles. Concepto de invariante*

II.9. Control de errores y excepciones.
II.10. Errores frecuentes.

II.1 INTRODUCCIÓN A C++

II.1.1 La evolución de C++

¿De dónde viene el nombre y el lenguaje C++?. C++ tiene sus orígenes en el lenguaje de programación C (el cual a su vez se derivó del lenguaje B, y éste del lenguaje BCPL; no existe un lenguaje A). El símbolo "++" es el operador de incremento de C, y al ponerlo en el nombre de C++ se quiere enfatizar que éste es una extensión del primero. No se le denominó D (y así seguir el orden alfabético) porque C++ es una extensión de C, y no un intento de remediar problemas quitando características del mismo.

El lenguaje C, creado por Dennis Ritchie de AT&T Bell Laboratories en la década de los setenta para escribir y mantener el sistema operativo UNIX, es un lenguaje peculiar porque es un lenguaje de alto nivel con muchas características de un lenguaje de bajo nivel, y en ello radican tanto sus ventajas como sus deficiencias. C es una opción excelente para escribir programas de sistemas (sistemas operativos, compiladores, ...), pero las características y el grado de libertad que lo hacen adecuado para ese tipo de aplicaciones, puede ocasionar, si no se ponen los remedios oportunos, programas difíciles de entender, con todos los inconvenientes que ello conlleva.

Con objeto de superar éstas y otras deficiencias de C, Bjarne Strousptrup, también de AT&T Bell Laboratories, creó C++ a principios de la década de los ochenta. Strousptrup diseñó C++ de modo que: 1) fuera un C mejor, 2) soportara la Abstracción de Datos y 3) soportara el paradigma de la Programación Orientada a Objetos. Este paradigma de programación se abordará en el segundo cuatrimestre de este curso.

Además del lenguaje C, C++ toma características de otros muchos lenguajes. Así, los conceptos relacionados con la orientación a objetos los toma de Simula67, los aspectos relacionados con las plantillas los recoge de Ada, y el mecanismo de manejo de excepciones está inspirado por Ada, Clu y ML.

II.1.2 Ejemplo de programa en C++

En las siguientes secciones de este tema se explicarán con detalle todas las características de C++ que se necesitan para escribir programas como el de la figura 1, pero para tener una idea rápida de cómo funciona un programa de C++, a continuación describiremos brevemente el funcionamiento de este programa en particular.

```
#include <iostream>
using namespace std;
const float CTMS_PULGADA = 2.54;

int main()
{
	float centimetros,pulgadas;

	cout << "Programa para convertir pulgadas a centimetros\n";
	cout << "Introduzca la medida en pulgadas: ";
	cin >> pulgadas;
	centimetros = pulgadas * CTMS_PULGADA;
	cout << "Esa medida en centimetros es: ";
	cout << centimetros << endl;

	return 0;
}
```

Figura 1. Ejemplo de programa en C++.

El principio y el final del programa contienen algunos detalles de los que no tenemos que ocuparnos todavía. El programa se inicia con:

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


int main()
{
```

El programa termina con:

```
	return 0;
}
```

La línea

```
	const float CTMS_PULGADA = 2.54;
```

es la declaración de una constante denominada `CTMS_PULGADA` que será utilizada en el programa. Esta constante es de tipo real.

La línea:

```
	float centimetros,pulgadas;
```

es la declaración de las variables `centimetros` y `pulgadas`, ambas de tipo real.

El resto de líneas son las instrucciones o sentencias que le indican al computador que realice un determinado trabajo. Las instrucciones que comienzan con `cin` o `cout` son operaciones de entrada y salida de datos. La palabra `cin`, que se pronuncia "c in", se utiliza para las entradas. Las instrucciones que comienzan por `cin` indican al computador qué debe hacer cuando se introduce información a través del teclado. La palabra `cout`, que se pronuncia "c out", se utiliza para las salidas, es decir, para enviar información del programa a

la pantalla. La letra `c` está ahí porque viene de la palabra "console". Los símbolos << y >> indican la dirección en la que se transfieren los datos y se llaman "poner en" y "obtener de", respectivamente. Así por ejemplo, la línea:

```
cout << "Programa para convertir pulgadas a centimetros\n";
```

podría leerse como: poner `"Programa para convertir pulgadas a centimetros\n"` en `cout` (palabra que representa la pantalla). Esto hace que el texto encerrado entre comillas aparezca en la pantalla. El carácter especial `'\n'` al final de la cadena entrecomillada indica que se inicie una nueva línea en la pantalla tras escribir el texto. Por su parte, la línea:

```
cin >> pulgadas;
```

podría leerse como: obtener la cantidad `pulgadas` de `cin` (palabra que representa el teclado). Esto hace que el número tecleado se almacene en la variable `pulgadas`. Para que la entrada tenga efecto el usuario deberá pulsar Retorno de Carro (la tecla Entrar o Intro) tras escribir la cantidad.

Por su parte, la línea:

```
centimetros = pulgadas * CTMS_PULGADA;
```

es una asignación (no una equivalencia como en matemáticas), utilizada para almacenar en la variable `centimetros` el resultado de evaluar la expresión aritmética `pulgadas * CTMS_PULGADA`, es decir, el resultado de multiplicar el contenido de la variable `pulgadas` por `2.54`.

Las dos últimas líneas son de nuevo operaciones de salida. En la línea:

```
cout << centimetros << endl;
```

el símbolo `endl` provoca un salto de línea. Observar que esta instrucción muestra por pantalla el contenido de la variable `centimetros` no la palabra "`centimetros`".

Finalmente, la figura 2 muestra un ejemplo de ejecución del programa. La cantidad que aparece en negrita (42) es la introducida por el usuario.

```
Programa para convertir pulgadas a centimetros
Introduzca la medida en pulgadas: 42
Esa medida en centimetros es: 106.68
```

Figura 2. Ejemplo de ejecución del programa de la figura 1

II.1.3 Elementos básicos de C++

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

- Palabras reservadas. Son un conjunto de palabras que tienen un significado predeterminado para el compilador, los pondremos en negrita y sólo pueden ser utilizadas con dicho sentido (por ejemplo: **`main`**, **`double`**). Estas palabras reservadas se escriben siempre en minúscula.
- Identificadores. Son nombres elegidos por el programador para representar entidades (tipos, constantes, variables, subprogramas, ...) en el programa. Un identificador es una secuencia de letras y dígitos, siendo el primer carácter una letra (el _ cuenta como una letra), y pueden tener cualquier longitud. En C++, las letras minúsculas se consideran diferentes de las mayúsculas.
- Constantes literales. Son valores que aparecen explícitamente en el programa, y podrán ser numéricos, caracteres y cadenas (por ejemplo: `2.54, "Esa medida en centimetros es: "`).
- Operadores. Símbolos con significado propio según el contexto en el que se utilicen (por ejemplo: `/`, `>>`, `<<`).
- Delimitadores. Símbolos que indican comienzo o fin de una entidad (por ejemplo: `(,),{,}`).
- Comentarios y espacios en blanco. Los comentarios en C++ se expresan de dos formas diferentes. Para comentarios cortos en línea utilizaremos `//` que indica comentario hasta el final de la línea. Para comentarios largos (de varias líneas) utilizaremos `/*` que indica comentario hasta `*/`. Los espacios en blanco, tabuladores, nueva línea, retorno de carro, avance de página y comentarios son ignorados por el compilador, excepto en el sentido en que separan componentes.

II.1.4 Esquema general de un programa en C++

```
/* Comentario indicando el nombre y objetivo del programa*/
Inclusión de módulos de biblioteca necesarios
Declaraciones y definiciones de constantes, tipos y subprogramas
int main()
{
        Declaraciones de variables
        Secuencia de sentencias
        return Estado;
}
```

Figura 3. Esquema general de un programa en C++.

La figura 3 muestra el esquema general de un programa escrito en C++. Al principio aparecerán, normalmente, unas líneas de inclusión de las definiciones de los módulos de biblioteca que nuestro programa necesita. En el ejemplo de la sección II.1.2, la línea:

```
#include <iostream>
```

hace que el fichero `iostream`, que contiene las definiciones de la biblioteca de entrada/salida, se incluya en el programa. Este fichero será necesario para poder realizar instrucciones de entrada y de salida.

Posteriormente se realizarán declaraciones y definiciones de constantes, de tipos y de subprogramas necesarios. El orden en el que se declaren o definan no importa siempre que se mantenga la siguiente premisa: es obligatorio la declaración de las entidades que se manipulen en el programa antes de que sean utilizadas. De cualquier forma, lo más lógico será situar primero las constantes y los tipos, y después los subprogramas, los cuales se tratarán en el tema III.

Finalmente aparecerá el algoritmo principal, llamado `main`, que será el que se ejecute cuando comience la ejecución del programa. Lo primero que aparecerá en dicho algoritmo (y esto se mantendrá también para los subprogramas) será la declaración de las variables necesarias en el mismo. Posteriormente, figurará la secuencia de sentencias que definen las acciones a ejecutar. En C++, la unidad básica de acción es la sentencia, y expresamos la composición de sentencias como una secuencia de sentencias terminadas cada una de ellas por el carácter “punto y coma” (;). Al finalizar, el algoritmo principal `main` devolverá un número entero, que será el código de retorno que se pasa al sistema operativo para indicar el estado en que terminó la computación (un número 0 indica terminación normal).

II.2 TIPOS DE DATOS SIMPLES

Las instrucciones que conforman un programa manipulan datos u objetos, los cuales tienen tres características:

- Nombre o identificador para referirse al objeto en el programa
- Valor del objeto, el cual puede ser constante o variar a lo largo de la ejecución del programa
- Tipo del objeto, que tiene las siguientes características:
  - Define el conjunto de valores que puede tomar un determinado objeto
  - Determina las operaciones que se pueden aplicar a un objeto
  - Define el espacio que será necesario reservar en memoria para albergar a un objeto
  - Define la interpretación del valor almacenado en memoria

En C++, al igual que en la mayoría de los lenguajes de programación, los tipos de los datos manipulados en un programa se clasifican en:

- Tipos simples, formados por valores o elementos indivisibles, es decir, que no se pueden descomponer en partes a las que poder acceder individualmente
- Tipos estructurados, cuyos elementos sí se pueden descomponer en partes a los que se puede acceder independientemente

Los tipos de datos estructurados de C++ se abordarán en el tema IV. Aquí estudiaremos los tipos de datos simples, los cuales a su vez se pueden clasificar en dos grupos:

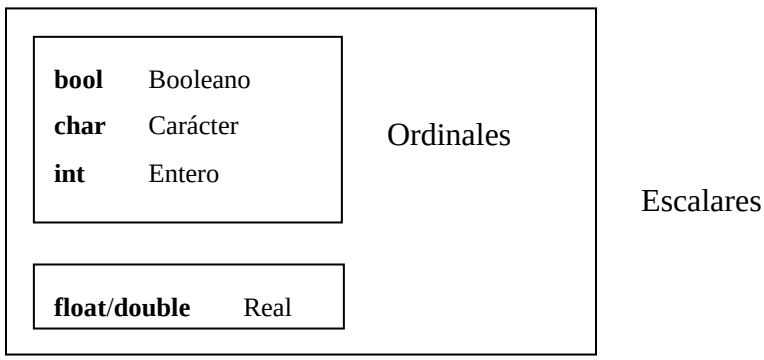
- Tipos simples predefinidos, aportados por C++ y que el programador puede utilizar sin tener que definirlos
- Tipos simples definidos por el programador

Ambos grupos serán abordados en las dos secciones siguientes.

II.2.1 Tipos simples predefinidos

Los tipos simples predefinidos en C++ son los siguientes:

bool char int float/double



Todos estos tipos citados tienen 2 propiedades:

a) Atómicos: Formados por elementos indivisibles.

b) Ordenados: Formados por elementos ordenados, esto es, le son aplicables los operadores relacionales (==, >, <, >=, <=, !=).

Los tipos de datos con estas 2 propiedades se denominan tipos Escalares.

Todos estos tipos excepto los reales tienen otra propiedad adicional:

c) Cada valor tiene un predecesor y un sucesor únicos (excepto el primero y el último, respectivamente).

Los tipos escalares con la propiedad c) se denominan tipos Ordinales.

El tipo bool

Se utiliza para representar valores lógicos o booleanos, es decir, los valores "verdadero" o "falso". Las constantes lógicas en C++ que representan dichos valores son `true` y `false`, respectivamente. Un valor lógico suele almacenarse en el tamaño de palabra direccionable más pequeño posible (normalmente 1 byte [8 bits]).

El tipo char

Se utiliza para representar los caracteres, es decir, símbolos alfanuméricos, de puntuación, espacios, control, etc. Una constante carácter se representará mediante el símbolo correspondiente del carácter entre comillas simples (por ejemplo, 'a', 'B', ';'). Normalmente utilizan un espacio de almacenamiento de 1 byte y puede representar 256 posibles valores diferentes (tabla ASCII).

El tipo int

Se utiliza para representar los números enteros. Un entero suele almacenarse en el tamaño de palabra definida por el procesador. Normalmente es de 4 bytes.

No obstante, dicho tamaño se puede modificar con objeto de representar un rango de valores menor mediante el modificador **short** (normalmente 2 bytes) o para representar un rango de valores mayor mediante el modificador **long** (normalmente 4 bytes).

Además también puede ser modificado para representar sólo números naturales, utilizando para ello el modificador **unsigned**.

Un entero constante se representa mediante una secuencia de dígitos, precedida por el símbolo "-" para los negativos, y posiblemente terminada con uno (o los dos) de los caracteres "U" (unsigned) y "L" (long). Por ejemplo:

123, -1520, 30000L, 50000UL

También es posible representar números enteros en base Hexadecimal (16) y base Octal (8). Para ello se utiliza el prefijo 0x y 0 respectivamente. Por ejemplo:

0x10B3FC23, 0751

Los tipos float y double

Ambos se utilizan para representar los números reales, diferenciándose en el rango de valores que representan y en la precisión de los mismos (número de dígitos significativos). El tipo **double** (normalmente 8 bytes) se utiliza para representar números reales de "doble precisión" y el tipo **float** (normalmente de 4 bytes) para representar la "simple precisión". El tipo double también puede ser modificado mediante el modificador long (**long double**) para representar "cuádruple precisión" (utilizando normalmente 10 bytes).

Las constantes reales se pueden representar de dos formas: notación de punto fijo (secuencia de dígitos [precedida por "-" para los negativos] con un punto decimal separando la parte entera de la decimal) y notación de punto flotante (secuencia de dígitos con o sin punto decimal seguida del símbolo "e" [que significa "por diez elevado a"], tras el que aparecerá un número entero [exponente]). Por ejemplo:

```
3.1415
-1e12          // representa –1 x 10^12
5.3456e-5      // representa 1.3456 x 10^-5
```

La tabla 1 muestra, a modo de resumen, el tamaño usado, el rango de valores y la precisión para cada uno de los tipos numéricos vistos. Los valores de la tabla son una muestra para dar una idea general de las diferencias entre los distintos tipos, pero dichos valores pueden variar de una máquina a otra. Para ver el tamaño (en bytes) que ocupa un determinado tipo en memoria, podemos aplicarle el siguiente operador: `sizeof(tipo)`.

Nombre del tipo	**Memoria que usa**	**Rango de valores**	**Precisión**	**Operadores**
short (también llamado **short int**)	2 bytes	-32.768 a 32.767	(no aplica)	Relacionales Aritméticos
unsigned short (también llamado **unsigned short int**)	2 bytes	0 a 65.535	(no aplica)	Relacionales Aritméticos
int	4 bytes	-2.147.483.648 a 2.147.483.647	(no aplica)	Relacionales Aritméticos
unsigned (también llamado **unsigned int**)	4 bytes	0 a 4.294.967.295	(no aplica)	Relacionales Aritméticos
long (también llamado **long int**)	4 bytes	-2.147.483.648 a 2.147.483.647	(no aplica)	Relacionales Aritméticos

unsigned long (también llamado **unsigned long int**)	4 bytes	0 a 4.294.967.295	(no aplica)	Relacionales Aritméticos
float	4 bytes	$3.4x10^{-38}$ a $3.4x10^{38}$	7 dígitos	Relacionales Aritméticos
double	8 bytes	$1.7x10^{-308}$ a $1.7x10^{308}$	15 dígitos	Relacionales Aritméticos
long double	10 bytes	$3.4x10^{-4932}$ a $3.4x10^{4932}$	19 dígitos	Relacionales Aritméticos

Tabla 1. Tipos numéricos

Al principio del apartado indicamos que el tipo también define la interpretación del valor almacenado en memoria. Supongamos que una variable tiene un valor de "01000001". Si la variable es de tipo entero la interpretación del valor será 65. Sin embargo, si esta variable es de tipo carácter la intepretación que haremos de ese valor es de la letra 'A' (valor 65 de la tabla ASCII).

Otro aspecto a tener en cuenta es cuando se intenta almacenar en una variable un valor que excede el número de bits reservado para almacenar valores de ese tipo. En ese caso se produce lo que se conoce como desbordamiento u *overflow*. Por ejemplo si tenemos un espacio reservado de 4 bits y queremos sumar "0001" y "1000", el resultado "10001" dará un desbordamiento (solo tenemos un espacio de 4 bits para almacenar el valor) por lo que el valor almacenado en la suma será erroneo.

Finalmente, en los tipos reales (float, double, …) es posible que se produzca una pérdida de precisión puesto que el número de bits reservados para la parte fraccionaria es limitado. Por ejemplo, si la precisión es de 4 digitos y sumamos 9.900, 1.000 y -0.999 el resultado dependerá del orden de las operaciones:

$$(X+Y)+Z = 10.900 + (-0.999) = 9.910$$

$$X+(Y+Z) = 9.900 + 0.001 = 9.901$$

II.2.2 Tipos simples definidos por el programador

El lenguaje C++ sólo tiene un tipo simple que pueda ser definido por el programador, el tipo enumerado.

El tipo enumerado

Además de los tipos simples predefinidos, el programador puede también definir y usar nuevos tipos simples que definan mejor las características de las entidades u objetos manipulados por el programa. Así, dicho tipo se definirá en base a una enumeración de los posibles valores que pueda tomar el objeto asociado. A este tipo se le llama enumerado.

Por ejemplo, si quisiéramos definir una variable para que represente una situación de un problema en el que hay cinco posibles colores, (rojo, verde, amarillo, azul y naranja) podríamos hacer lo siguiente:

```
enum Colores {
          rojo,
          verde,
          amarillo,
          azul,
          naranja
};
```

La definición de nuevos tipos enumerados se realiza usando la palabra reservada `enum`, un nombre para identificar el nuevo tipo definido (`Colores` en nuestro ejemplo), y la enumeración separada por comas de los posibles valores (identificadores) que lo define. Esta definición, al igual que sucederá con la definición de tipos estructurados (que abordaremos en el tema IV), se realizará en la zona marcada para tal fin en la figura 3, donde se presentó el esquema general de un programa en C++. La utilización de estos tipos permiten incrementar la legibilidad de los programas, aunque también pueden utilizarse los tipos predefinidos.

Un mismo identificador no puede aparecer en las enumeraciones de dos definiciones de tipo enumerado distintas.

El orden de los identificadores en la definición del tipo enumerado determina el valor numérico asociado a cada uno. Así al primer identificador de la lista se le asocia el valor 0, al segundo el valor 1, y así sucesivamente.

Los valores de tipo enumerado no pueden leerse ni escribirse directamente. Es cuestión del programador el implementar las operaciones adecuadas.

II.2.3 Operadores

Los operadores que se pueden aplicar a los datos de los tipos estudiados en las secciones anteriores son lo siguientes (ordenados por orden de precedencia o prioridad; primera línea mayor predecencia, última línea menor precedencia):

! -	unario	asociativo de derecha a izquierda
* / %	binarios	asociativos de izquierda a derecha
+ -	binarios	asociativos de izquierda a derecha
< <= > >=	binarios	asociativos de izquierda a derecha
== !=	binarios	asociativos de izquierda a derecha
&&	binario	asociativo de izquierda a derecha
| |	binario	asociativo de izquierda a derecha

Por ejemplo, en la expresión 3 * 4 + 8, se aplicará primero el * y después el +.

El significado de estos operadores es el siguiente:

Aritméticos

- valor	menos unario
valor * valor	producto
valor / valor	división (entera y real)
valor % valor	modulo o resto (no para reales)
valor + valor	suma
valor – valor	resta

Relacionales

valor < valor	comparación menor
valor <= valor	comparación menor o igual
valor > valor	comparación mayor
valor >= valor	comparación mayor o igual
valor == valor	comparación igualdad
valor != valor	comparación desigualdad

Lógicos

! valor	negación lógica
valor && valor	and lógico
valor | | valor	or lógico

II.2.4 Conversiones de tipos

Es posible que nos interese realizar operaciones con datos de tipos diferentes. C++ realiza conversiones de tipo (“castings”) automáticas, de tal forma que el resultado de la

operación sea del tipo más amplio de los implicados en ella. La siguiente lista determina el orden de "amplitud" de los tipos (de menor a mayor):

char, short, enumerado
int
unsigned int
long
unsigned long
float
double
long double

Así, por ejemplo, un valor de tipo enumerado se convertirá automáticamente a un `int` para llevar a cabo operaciones en las que aparecen mezclados enteros y enumerados. Otro ejemplo: un valor de tipo `int` se convertirá a un `double` si en la expresión aparece algún dato de tipo `double`.

No obstante, también es posible realizar conversiones de tipo explícitas, las cuales son necesarias en algunos casos, por ejemplo para convertir un valor de tipo entero a un enumerado. Para ello, se escribe el tipo al que queremos convertir y entre paréntesis el valor (o entidad con el valor) original a convertir. Por ejemplo:

```
int('a')        // devuelve el entero 97 (ordinal de 'a')
int(rojo)       // devuelve el entero 0
int(amarillo)   // devuelve el entero 2
Colores(1)      // devuelve el color verde
char(65)        // devuelve el carácter 'A'
double(2)       // devuelve el real 2.0
int(3.6)        // devuelve el entero 3
```

II.3 CONSTANTES, VARIABLES Y ASIGNACIONES

Las **constantes** no cambian de valor durante la ejecución del programa. Pueden aparecer como "constantes literales" y como "constantes simbólicas". Las primeras son aquellas cuyo valor aparece directamente en el programa, por ejemplo ('a', 123, 5.3456e-5, ...). Las segundas son aquellas cuyo valor se asocia a un nombre y se utiliza éste para representarla a lo largo del programa.

Las constantes simbólicas se declaran indicando la palabra reservada `const` seguida por su tipo, el nombre simbólico (o identificador) con el que nos referiremos a ella y el valor asociado tras el operador de asignación (=). Ejemplos de constantes simbólicas:

```
const char TERMINADOR = '.';
```

```
const int MAXIMO = 5000;
const unsigned ELEMENTO = 1000;
const double PI = 3.141592;
const colores MI_COLOR = azul;
```

Las **variables** pueden cambiar su valor durante la ejecución del programa. Se declaran especificando su tipo y el nombre simbólico (o identificador) con el que nos referiremos a ella. Se le podrá asignar un valor inicial en la declaración. Si no se hace así, dicha variable tendrá un valor inicial inespecificado. Por ejemplo:

```
int contador = 0;     // contador tiene valor inicial 0
double total = 5.0;   // total tiene valor inicial 5.0
char c;               // c con valor inicial inespecificado
```

La **sentencia de asignación** más simple de todas consiste en asignar a una variable el valor de una expresión calculada en base a los operadores mencionados anteriormente.

```
variable = expresion;
```

Por ejemplo:

```
int resultado;
resultado = 30 * MAXIMO + 1;
```

Además, C++ proporciona una sintaxis especial para simplificar asignaciones del tipo

```
v = v <operación> valor;
```

escribiéndolas como

```
v <operación>= valor;
```

Por ejemplo:

```
variable += expresion;      // variable = variable + expresion;
variable – = expresion;     // variable = variable – expresion;
variable *= expresion;      // variable = variable * expresion;
variable /= expresion;      // variable = variable / expresion;
variable %= expresion;      // variable = variable % expresion;
```

Cuando la expresión implica un incremento o decremento de uno, se puede escribir:

```
++variable;    // variable = variable + 1;
– –variable;   // variable = variable – 1;

variable++;    // variable = variable + 1;
variable– –;   // variable = variable – 1;
```

Notas:

- Las sentencias de asignación anteriores se pueden utilizar en otras muy diversas formas, pero nosotros no las utilizaremos debido a que dificultan la legibilidad y aumentan las posibilidades de cometer errores de programación.
- No existen operaciones aritméticas definidas para los tipos enumerados. Sin embargo, la conversión de un tipo enumerado a un tipo entero se realiza automaticamente, aunque no ocurre automaticamente la conversión inversa. Por ejemplo, si `color` es una variable de tipo `Colores`:

```
++color;                        // ERROR operación no definida
color += 3;                     // ERROR asignación no válida
color = Colores(color + 3);     // OK
```

II.4 ENTRADA Y SALIDA BÁSICAS

Para poder realizar entrada/salida básica de datos es necesario incluir la siguiente biblioteca escribiendo la siguiente sentencia al comienzo del programa:

```
#include <iostream>
```

La **salida** de datos se realiza utilizando el operador `<<` sobre la entidad `cout` especificando el dato a mostrar. Por ejemplo:

```
cout << contador;
```

escribirá en la salida estándar (pantalla) el valor de `contador`. También es posible escribir varios valores sin necesidad de escribir varias veces la entidad `cout`:

```
cout << "Contador" << contador;
```

Si queremos escribir un salto de línea lo podemos hacer de dos formas:

```
cout << "Contador" << contador << "\n";
cout << "Contador" << contador << endl;
```

Normalmente utilizaremos la segunda forma. El manipulador `endl` denota fin de línea (end of line). El `\n` no se suele utilizar solo, sino formando parte de una cadena de caracteres, como se vió en el ejemplo de la figura 1.

La **entrada** de datos se realiza mediante el operador `>>` sobre la entidad `cin` especificando la variable donde almacenar el valor de entrada. Por ejemplo:

```
cin >> contador;
```

almacenará el valor leido de la entrada estándar (teclado) en la variable `contador`.

Es posible leer varios valores simultáneamente sin necesidad de escribir varias veces la entidad `cin`:

```
cin >> minimo >> maximo;
```

La entrada de un dato se realiza de la siguiente forma:

- ✓ Saltando caracteres en blanco, tabuladores y saltos de línea hasta encontrar un carácter diferente a éstos.
- ✓ Después se leen caracteres hasta encontrar un carácter en blanco, un tabulador o un salto de línea (en el caso de lectura de un valor de tipo char, sólo se leerá un carácter).
- ✓ Los caracteres leídos se convierten al tipo esperado (se produce error si los datos no son de ese tipo) (ver sección II.9). En el caso de lectura de un valor de tipo char, no es necesaria ninguna conversión.

Para el ejemplo anterior, si `minimo` y `maximo` son dos variables de tipo `int`, la entrada se puede realizar:

```
12 45
```

es decir, separando los dos números con uno o varios espacios en blanco, o bien

```
12
45
```

es decir, un valor en cada línea (el separador en este caso es el salto de línea).

Ningún dato de entrada o de salida en un programa C++ se obtiene o envía directamente del/al hardware, sino que se realiza a través de "buffers" de entrada y de salida controlados por el Sistema Operativo y son independientes de nuestro programa.

Así, cuando se pulsa alguna tecla, los datos correspondientes a las teclas pulsadas se almacenan en una zona de memoria intermedia: **el "buffer" de entrada**. Cuando un programa realiza una operación de entrada de datos, accede al "buffer" de entrada y obtiene los valores allí almacenados si los hubiera, o esperará hasta que los haya (se pulsen una serie de teclas seguida por Entrar o Intro). Una vez obtenidos los datos asociados a las teclas pulsadas, se convertirán a un valor del tipo de la variable especificada por la operación de entrada, asignándole dicho valor a la misma.

Por otra parte, cuando se va a mostrar alguna información de salida, dichos datos no van directamente a la pantalla, sino que se convierten a un formato adecuado para ser impresos, y se almacenan en una zona de memoria intermedia denominada "**buffer" de salida**, de donde el Sistema Operativo tomará la información para ser mostrada por pantalla (normalmente cuando se muestre un salto de línea, se produzca una operación de entrada de datos, etc).

A las entidades `cin` y `cout` se les denomina **flujos**. El primero es el flujo de entrada estándar y el segundo el flujo de salida estándar. Además existe otro flujo que se puede utilizar directamente: `cerr`, que es el flujo de errores estándar.

Los operadores >> y << son los más simples para realizar entrada y salida y sólo son aplicables a los tipos de datos simples predefinidos y a las cadenas de caracteres.

Existen otras forma de realizar la entrada/salida como, por ejemplo, usando la operación get/put:

```
cin.get(c);          // lee un carácter de teclado y lo almacena en c
```

Esta operación, al contrario que el operador >>, no salta los espacios en blanco, tabuladores y saltos de línea antes de llevar a cabo su acción (leer un carácter). Esto es importante tenerlo en cuenta si lo que deseamos hacer es procesar un texto en el que hay que tratar también dichos caracteres (ej. contar los espacios en blanco que hay en un texto).

Por otro lado, podemos mostrar por pantalla el contenido `c` (de tipo char) con:

```
cin.put(c)
```

II.5 FLUJO DE CONTROL

El efecto deseado con un programa se consigue mediante la ejecución de una secuencia de instrucciones o sentencias. Hasta ahora hemos considerado las instrucciones de asignación y las que permiten efectuar entrada/salida básica, por lo que el orden de ejecución de las instrucciones (flujo de control) coincide con el orden en que están dispuestas en el programa, comenzando por la primera y continuando en secuencia hasta llegar a la última.

Bloques. Un bloque es una unidad de ejecución mayor que la sentencia, y permite agrupar una secuencia de sentencias como una unidad. Para ello, enmarcamos la secuencia de sentencias entre dos llaves, formando un bloque.

```
{
    sentencia_1;
    sentencia_2;
      .....
    sentencia_n;
}
```

Es posible anidar bloques, aunque no haremos uso de esta característica de forma habitual.

```
{
    sentencia_1;
    sentencia_2;
    {
        sentencia_3;
        sentencia_4;
        .....
    }
    .....
    sentencia_n;
}
```

Dentro de un bloque se podrán declarar variables o constantes, en tal caso, serán visibles desde el punto de la declaración hasta el final del bloque en que se definen. Aunque C++ da libertad al respecto, cuando deseemos declarar variables o constantes en nuestros programas, siempre lo haremos al principio de un bloque (ver figura 3), evitando hacerlo en puntos intermedios del mismo. Seguiremos el siguiente formato:

```
{
    declaración_1;
     ....
    declaración_n;

    sentencia_1;
      .....
    sentencia_m;
}
```

Sin embargo, la composición secuencial de sentencias no es suficiente para resolver todos los problemas porque:

1. Puede ser necesario resolver problemas que exijan una toma de decisión
2. Puede que se necesite repetir un cojunto de instrucciones un número de veces.

C++ dispone de dos tipos de estructuras que permiten alterar el flujo de control, las estructuras de selección (permiten toma de decisiones) y las estructuras de iteración (permiten la repetición de acciones). Ambas estructuras dependen de la evaluación de expresiones lógicas o booleanas.

II.6 EXPRESIONES LÓGICAS O BOOLEANAS

Una expresión booleana o lógica es una afirmación que una vez evaluada puede ser verdadera o falsa (**true** o **false**).

En cualquiera de las estructuras que permiten alterar el flujo de control del programa habrá que especificar las condiciones bajo las cuales dicho flujo continúa por un punto u otro del programa. La siguiente tabla muestra los operadores de que dispone C++ para construir expresiones lógicas.

Operador C++
==
!=
<, >, >=, <=
&&
||
!

Una expresión lógica se puede construir:

1. Definiendo una variable lógica (**`bool`** `v;`) y consultado su valor (`v`) en un instante dado (**`true`** o **`false`**).
2. Utilizando los operadores relacionales que permiten comparar dos objetos del mismo tipo y el resultado de esa comparación dará lugar a uno de los dos posibles valores lógicos (**`true`** o **`false`**). Veamos algunos ejemplos (`n,m,x` son variables enteras):

   `3 < 8`

   `n ≥ 4`

   `n * m ≤ x + 18`

3. Utilizando los operadores lógicos o booleanos que actúan sólo sobre expresiones booleanas

Expr. 1	Expr. 2	&&	||	! Expr. 1
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Algunos ejemplos de expresiones que utilizan operadores lógicos:

`v,w` son variables lógicas
`x, y, z` son variables enteras

```
v && w
v || true
(x == 3) &&  (z !=4)
(((x + z) - 4) != 4) || w
! (v &&  (x ≤  y))
```

C++ , al igual que la mayoría de los lenguajes de programación, evalúa las expresiones booleanas "en cortocircuito". Esto significa que la expresión se evalúa de izquierda a derecha y llegado el punto en el que ya se conoce el resultado de la expresión completa (el resto de la misma no tiene influencia en el resultado), no se continúa evaluando.

Por ejemplo: `(j > 0) && (100/j < 2)`. Si `j` contiene un valor negativo o cero, la expresión se evalúa a `false`, porque `(j > 0)` es `false` (y no se llega a evaluar la expresión `(100/j < 2)`).

Supongamos la siguiente expresión. ¿Como se interpretaría? :

```
bool x,y;
v =  ! x &&  y;
```

Si x e y son **false** :

1. v es **true** si primero actúa &&
2. v es **false** si primero actúa !

Esto implica que es necesario establecer una prioridad (precedencia) entre los operadores booleanos al igual que entre los aritméticos. Por ejemplo 3+2*4 es equivalente a 3+(2*4).

Convencionalmente de más a menos prioridad :

! && ||

Si se desea alterar esta prioridad convencional, se usan paréntesis.

Si mezclamos operadores booleanos con operadores relacionales y con operadores aritméticos, la prioridad sería la que se mostró en la sección II.2.3:

¡ En caso de duda usar paréntesis!

Ejemplos (y cuestiones) de uso de expresiones booleanas:

a) `bool valor;`

¿cómo se puede poner de otra forma la condición o pregunta:

`valor == false`?,

¿ y `valor == true`?

b)

```
bool prueba, valor ;
          prueba = valor == false
```

es equivalente a??

c)

```
bool cierto;
int x, y;
```

```
cierto =  (x < y) && (y < x);        es equivalente a??
cierto =  (x <= y) || (y <= x);      es equivalente a ??
```

d) **bool** x, y;

i) x != y

ii) (x || y) && ! (x && y)

¿Relación entre i) e ii) ?

e) **bool** p, q, r;

Evaluar:

```
i) ((p ||  r) &&  (q || r))== ((p && q) ||  r)
ii) ((p &&  r) ||  (q &&  r)) == ((p ||  q) &&  r)
```

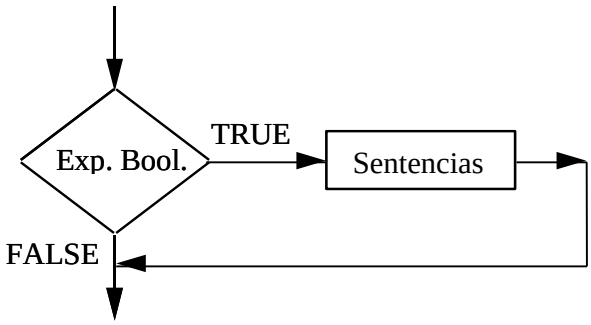
II.7 ESTRUCTURAS DE SELECCIÓN

Se utilizan para alterar el flujo de control, permitiendo seleccionar una de entre varias alternativas dependiendo de condiciones lógicas. La alternativa seleccionada hará que el flujo de control alcance una instrucción o bloque de instrucciones concreto. C++ dispone de dos tipos de estructuras de selección, `if` y `switch`.

II.7.1 Estructura "`if`"

En su versión más simple la estructura `if` permite seleccionar la ejecución o no de una determinada secuencia de sentencias dependiendo de que la expresión lógica de control se evalúe a `true` o a `false`. Tiene el siguiente formato:

```
if (expresion lógica) {
        Secuencia de sentencias
}
```



Veamos un ejemplo que comprueba si un número es positivo:

```
if (dato<0) {
      cout << "Incorrecto";
}
```

Veamos un ejemplo de uso de la sentencia `if`. A continuación mostraremos un programa que lee un número de paquete seguido de un peso en kilos, y que escribe a la salida el número de paquete, seguido de "CLASE1" si el peso del paquete es menor de 32 kilos, "CLASE2" si va de 32 a 128 kilos y "CLASE3" para más de 128 kilos (más adelante veremos cómo se debe codificar realmente este programa utilizando "anidamiento").

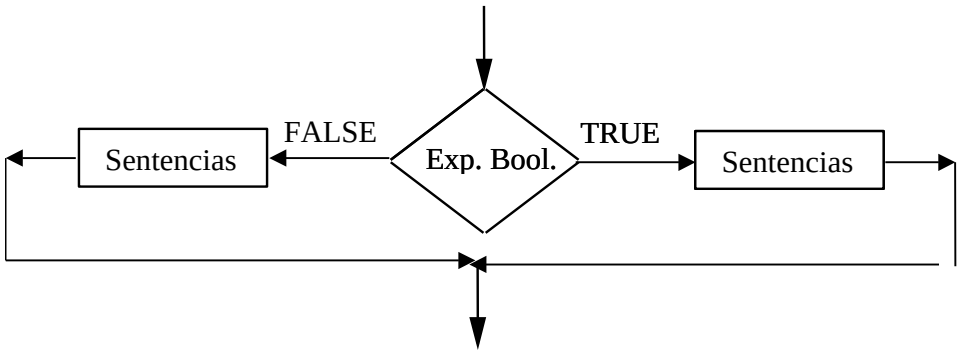
```
#include <iostream>
using namespace std;

int main () {
      unsigned num;
      float peso;

      cout << "Introduzca el número de paquete:";
      cin >>num;
      cout<<"Introduzca el peso del paquete:";
      cin >>peso;
      if (peso < 32) {
            cout << "El paquete"<< num << "es de CLASE1";
      }
      if ((peso >= 32)&&(peso<=128)) {
            cout << "El paquete"<< num<< "es de CLASE2";
      }
      if(peso > 128) {
            cout << "El paquete" << num << "es de CLASE3";
      }
      return 0;
}
```

La estructura `if` admite una versión más general en la que se puede especificar una secuencia de sentencias alternativa para ser ejecutada cuando la condición de control se evalúe a `false`. Tiene el siguiente formato:

```
if (expresion logica) {
      Secuencia de sentencias 1
} else {
      Secuencia de sentencias 2
}
```



Podemos completar el ejemplo anterior para que en el caso de que sea positivo indique que el número es correcto

```
if (dato<0) {
      cout << "Incorrecto";
} else {
      cout << "Correcto";
}
```

Si la secuencia de sentencias consta de una única instrucción no es necesario que esté enmarcada entre llaves. No obstante, por cuestiones de estilo, y para evitar errores de programación que se suelen producir al prescindir de las llaves, nosotros siempre usaremos llaves para delimitar las sentencias de las distintas alternativas.

Anidamiento. Una secuencia de sentencias consta de una o más sentencias, cada una de ellas acabada en punto y coma. Por tanto, nada impide que una de estas sentencias sea a su vez una sentencia de selección. Cuando ello ocurre hablamos de sentencias de selección anidadas. En el siguiente ejemplo, dada una variable "dia" de tipo enumerado, cuyo contenido representa un día de la semana, se muestran mensajes relativos al día representado.

```
if ((Lunes <= dia) && (dia <= Domingo)) {
      cout << "Día valido";
      if  (dia <= Viernes) {
            cout << "Día laborable";
      } else {
            cout << "Día no laborable";
      }
} else {
      cout << "Día no válido";
}
```

Frecuentemente aparecen en un programa situaciones en las que se desea seleccionar una alternativa de entre varias. Como ejemplo, considérese un programa para escribir en pantalla un mensaje diferente correspondiente a un número que representa una calificación numérica. El fragmento de código que escribe el mensaje por pantalla dependiendo de la nota quedaría:

```
if (Nota == 10) {
      cout << "Matrícula de Honor";
} else {
      if (Nota >= 9) {
            cout << "Sobresaliente";
      } else {
            if (Nota >= 7) {
                  cout << "Notable";
            } else {
                  if (Nota >= 5) {
                        cout << "Aprobado";
                  } else {
                        cout << "Suspenso";
                  }
            }
      }
}
```

El comportamiento que pretendemos describir con esta instrucción quedaría más claramente marcado si disponemos el texto de forma que se destaquen claramente las diferentes alternativas y las sentencias que seleccionan. En estas situaciones se usa una disposición del texto de la siguiente forma:

```
if (Nota == 10) {
      cout << "Matrícula de Honor";
} else if (Nota >= 9) {
      cout << "Sobresaliente";
} else if (Nota >= 7) {
      cout << "Notable";
} else if (Nota >= 5) {
      cout << "Aprobado";
} else {
      cout << "Suspenso";
}
```

```
if (expresion logica) {
      Secuencia de sentencias 1
} else if (expresion logica) {
      Secuencia de sentencias 2
} else if (expresion logica) {
      Secuencia de sentencias 2
} …
…
} else {
      Secuencia de sentencias n
}
```

De igual forma el ejemplo anterior de la lectura de paquetes y sus pesos podría quedar de la siguiente forma:

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

int main () {
      unsigned num;
      float peso;

      cout << "Introduzca el número de paquete:";
      cin >>num;
      cout<<"Introduzca el peso del paquete:";
      cin >>peso;
      if (peso < 32) {
            cout << "El paquete"<< num << "es de CLASE1";
      } else if (peso<=128) {
            cout << "El paquete"<< num<< "es de CLASE2";
      } else {
            cout << "El paquete" << num << "es de CLASE3";
      }
      return 0;
}
```

II.7.2 Estructura switch

Aunque la sentencia `if` de C++ es suficiente para describir cualquier comportamiento que requiera selección de alternativas en un programa, hay situaciones muy frecuentes que justifican una estructura especial que las represente fielmente. Ello es así cuando en el programa se desea seleccionar una de entre varias alternativas, dependiendo de que el valor resultante de evaluar una determinada expresión de control coincida con uno de entre varios valores. Por ejemplo, si dada una variable de tipo enumerado cuyo valor indica un día de la semana, deseamos mostrar en la salida su nombre, mediante instrucciones `if` se podría hacer:

```
if (dia == Lunes) {
      cout << "Lunes";
} else if (dia == Martes) {
      cout << "Martes";
} else if (dia == Miercoles) {
      cout << "Miercoles";
} else if (dia == Jueves) {
      cout << "Jueves";
} else if (dia == Viernes) {
      cout << "Viernes";
} else if (dia == Sabado) {
      cout << "Sabado";
} else if (dia == Domingo) {
      cout << "Domingo";
} else {
      cout << "ERROR. DIA  NO VALIDO";
}
```

pero alternativamente podemos usar la estructura `switch` de la siguiente forma:

```
switch (dia) {
case Lunes:
        cout << "Lunes";
        break;
case Martes:
        cout << "Martes";
        break;
case Miercoles:
        cout << "Miercoles";
        break;
case Jueves:
        cout << "Jueves";
        break;
case Viernes:
        cout << "Viernes";
        break;
case Sabado:
        cout << "Sabado";
        break;
case Domingo:
        cout << "Domingo";
        break;
default:
        cout << "ERROR. DIA NO VALIDO";
        break;
}
```

La sintaxis de la sentencia switch es:

```
switch (selector) {
case (etiqueta1) :Secuencia de sentencias 1
                  break;
case (etiqueta2) :Secuencia de sentencias 2
                  break;
case (etiqueta3) :Secuencia de sentencias 3
                  break;
…
case (etiquetan) :Secuencia de sentencias n
                  break;
default:
                  Secuencia de sentencias n
                  break;
```

Como se puede ver en la definición, la instrucción `switch` contiene selector (en el ejemplo, simplemente la variable `dia`) y una serie de etiquetas, precedidas de la palabra reservada `case`. La expresión de control debe ser de un tipo ordinal y las etiquetas deben ser expresiones constantes del mismo tipo que el selector. Cuando se ejecuta esta instrucción, en primer lugar se evalúa la expresión de control o selector, y el resultado se compara sucesivamente con cada etiqueta. Si una etiqueta coincide con dicho resultado, el flujo de control pasa a la primera sentencia asociada a la etiqueta, y a partir de ese momento serán ejecutadas todas las sentencias hasta el fin del `switch` o hasta que se ejecute una sentencia `break`.

La ejecución de una sentencia `break` hace que el flujo de control pase al final del bloque (en este caso, al final del `switch`), y su uso o no, depende del comportamiento que el programador desee expresar.

Si el objetivo buscado con la instrucción `switch` es (como en nuestro ejemplo) seleccionar una de entre varias alternativas excluyentes entre sí, una vez ejecutadas las sentencias asociadas a la alternativa seleccionada, es necesario situar la sentencia `break` para que, una vez localizada la alternativa deseada y tras ejecutar las sentencias correspondientes, no se continúen ejecutando las sentencias asociadas a los valores que la siguen. Así, por ejemplo, si no hubiéramos puesto las sentencias `break` y la variable "`dia`" hubiera contenido el valor Viernes, el resultado del ejemplo sería mostrar por pantalla los mensajes "Viernes", "Sabado", "Domingo" y "ERROR. DIA NO VÁLIDO", que no es lo que deseamos en este caso.

Es posible que distintas etiquetas seleccionen el mismo grupo de sentencias. Si

queremos representar esta situación basta con situar todas las etiquetas relacionadas de forma consecutiva, de la siguiente forma:

```
switch (selector) {
      case etiqueta1:
      case etiqueta2:
      case etiqueta3:
                        secuencia de sentencias1;
                        break;
      case etiqueta4:
                        secuencia de sentencias2;
                        break;
              ....
              ....
      default:
                        secuencia de sentencias3;
                        break;

}
```

Donde se indica que en caso de que el `selector` tenga el valor `etiqueta1`, `etiqueta2` o `etiqueta3`, deseamos que se ejecute la `secuencia de sentencias1`.

II.8 ESTRUCTURAS DE ITERACIÓN

Se utilizan para expresar que deseamos ejecutar de forma repetida una sentencia o grupo de sentencias. El número de repeticiones dependerá de la evaluación de una expresión lógica. El número de repeticiones puede ser:

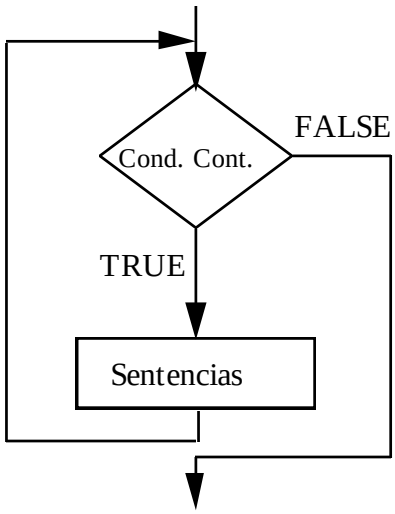
- No predeterminado. El número de repeticiones o iteraciones depende de alguna condición que se va modificando con las distintas repeticiones
- Fijo o predeterminado antes de empezar a repetir. El número de repeticiones o iteraciones se conoce a priori.

Explicaremos los tres esquemas de iteración (bucles) que admite C++.

II.8.1 Estructura while

Esta estructura de iteración pertenece al grupo de las construcciones con un número de repeticiones no predeterminado. El formato de la estructura `while` viene dado por:

```
while (condición de control) {
      Secuencia de sentencias       // Cuerpo del bucle
}
```



Cuando se ejecuta esta sentencia, en primer lugar se evalúa la expresión de control. Si el resultado es `true`, entonces el flujo de control entra en el cuerpo del bucle, ejecutándose todas las sentencias que lo forman. Seguidamente el flujo de control vuelve al principio del `while`, repitiéndose el proceso hasta que la expresión de control sea evaluada a `false`, en cuyo caso el flujo de control continúa por la primera instrucción que siga al bucle.

Como puede observarse, la expresión de control se comprueba cada vez antes de que el cuerpo del bucle sea ejecutado. El cuerpo se ejecuta mientras se cumpla la condición de control. Si la condición es `false`, entonces el cuerpo no se ejecuta. Hay que hacer notar que, si la condición es `true` inicialmente, la sentencia `while` no terminará (bucle infinito) a menos que en el cuerpo del bucle se modifique de alguna forma la condición de control. Por otra parte, de acuerdo al comportamiento indicado, si inicialmente la condición de control se evalúa a `false`, el cuerpo del bucle no se ejecutará ni siquiera una vez. Por tanto, esta estructura repetitiva permitirá expresar situaciones en las que deseamos que el cuerpo del bucle se repita cero o más veces.

Por ejemplo, si queremos leer una serie de números enteros y calcular su suma, terminando cuando se lea un número negativo, podemos hacer:

```
/*
 * Autor:
 * Fecha:                Versión: 2.0
 *
 * Programa para el cálculo de la suma de números leídos de teclado
 */
      #include <iostream>
      using namespace std;

      int main () {
            int sum,i;
            sum = 0;
```

```
            cout << "Introduce un numero ";
            cin >> i;
            while (i >= 0) {
                  sum = sum + i;
                  cout << " Introduce un numero ";
                  cin >>  i;
            }
            cout << "La suma es " << sum << endl;
            return 0;
      }
```

Veamos otro ejemplo. Un programa que calcula el Máximo Común Divisor (MCD) de dos números mediante el algoritmo de Euclides. Dados dos números, el MCD se alcanza cuando llegamos al mismo número restando el menor del mayor.

```
/*
 * Autor:
 * Fecha:                Versión: 2.0
 *
 * Programa para el cálculo del MCD de dos números leídos de teclado
 * Se usa el algoritmo de Euclides.
 */
#include <iostream>
using namespace std;
int main()
{
    int num1,num2;

    cout << "Teclee dos números: ";
    cin >> num1 >> num2;
    while (num1 != num2) {
        if (num1 > num2) {
            num1=num1 - num2;
        } else {
            num2=num2-num1;
        }
    }
    cout << "El máximo común divisor es:" << num1;
    return 0;
}
```

Veamos otro ejemplo de uso de esta estructura:

```
/*
 * Autor:
 * Fecha:                Versión: 1.0
 *
 * Programa para el cálculo del MCD de dos números leídos de teclado
 * Este enfoque hace una prueba exhaustiva de las posibilidades comprobando
 * números hasta que encontremos el mayor número que divide a los dos
 * números leídos por teclado.
 */
#include <iostream>
using namespace std;
int main()
{
    int num1, num2;
    int cand;   // candidato a MCD probado en cada iteración

    cout << "Teclee dos números: ";
    cin >> num1 >> num2;
```

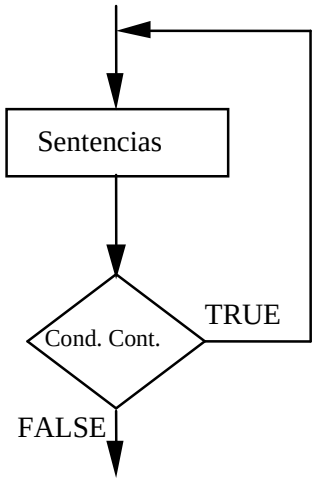
```
    if (num1 < num2) {
        cand = num1;
    } else {
        cand = num2;
    }

    while (((num1 % cand) != 0) || ((num2 % cand) != 0)) {
        cand--;
    }
    cout << "El máximo común divisor es:" << cand;
    return 0;
}
```

II.8.2 Estructura do ... while

Al igual que la estructura del apartado anterior, esta construcción también se utiliza para diseñar bucles cuyas sentencias se repetirán un número no predeterminado de veces. Su formato es:

```
do {
      Secuencia de sentencias;            // Cuerpo del bucle
} while (condición de control);
```



En esta estructura, primero se ejecuta el cuerpo del bucle y después se evalúa la condición de control para determinar si deseamos repetir o no una vez más la ejecución del cuerpo. Este comportamiento seguirá hasta que en el momento de evaluar la condición de control se obtenga como resultado `false`.

Como puede observarse, puesto que la expresión de control se evalúa después de ejecutar el cuerpo del bucle, en esta estructura el cuerpo del bucle se ejecutará siempre al menos una vez. Aunque es menos general que el bucle `while`, se adapta a numerosas situaciones típicas en programación. Por ejemplo, si deseamos leer un número hasta

asegurarnos que el valor introducido es mayor que cero, podríamos haber optado por una estructura `while` de la siguiente forma:

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

int main () {

      int numero;
      cout << "Introduzca un número mayor que cero";
      cin  >> numero;
      while (numero <= 0) {
                  cout << "Introduzca un número mayor que cero";
                  cin  >> numero;
      };
      return 0;
}
```

Pero, ya que la petición del número debe ser ejecutada siempre al menos una vez, la estructura "`do while`" se adapta mejor al problema que queremos resolver, quedando por tanto más natural de la siguiente forma:

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

int main () {
      int numero;
      do {
            cout << "Introduzca un número mayor que cero ";
            cin  >> numero;
      } while (numero <= 0);
      return 0;
}
```

II.8.3 Estructura for

Aunque es una estructura muy flexible y ofrece diferentes posibilidades, lo más adecuado es usarla cuando se conoce de forma predeterminada el número de iteraciones que se realizarán. Su formato es:

```
for (inicialización; condición de control; incremento) {
                  Secuencia de sentencias;          // Cuerpo del bucle

}
```

En esta estructura, tras la palabra reservada `for` aparece la cabecera del bucle. Dicha cabecera consta de tres componentes, que reflejan la forma en que evolucionará el flujo de control del bucle. Para ello, se basa en el uso de una "variable de control". La componente inicialización define la variable de control y almacena su valor inicial (es posible declarar

dicha variable en la propia inicialización y en ese caso la visibilidad de la variable se restringe al bucle for que se especifica), la componente incremento indica cómo se va actualizando dicha variable al final de cada iteración y la componente condición de control es una expresión lógica que determina cuándo acabará el bucle.

El comportamiento en detalle de la estructura `for` puede describirse fácilmente mediante la siguiente estructura `while`:

```
Inicialización;
while (condición de control) {
      Secuencia de sentencias
      Incremento
}
```

Como se puede observar, al principio, y una sola vez se ejecuta la componente de inicialización, que será una instrucción en la que se le da valor inicial a la variable de control del bucle, posteriormente se comprueba al comienzo de cada iteración la condición de control, y *mientras* ésta sea cierta, se ejecutará la secuencia de sentencias que forman el cuerpo del bucle `for`, ejecutándose al final de cada iteración la componente de incremento, que será una instrucción en la que se especifique la forma en que va modificando la variable de control tras cada iteración del bucle.

Por ejemplo, si deseamos mostrar por pantalla los N primeros números naturales, donde N es un valor leído por teclado, podríamos usar una estructura `while` de la siguiente forma

```
/*
 * Autor:
 * Fecha:                Versión: 1.0
 *
 * Programa para mostrar n números naturales
 */

#include <iostream>
using namespace std;

int main () {
      unsigned i,N;

      cout << "Introduzca un numero ";
      cin >> N;
      i = 1;
      while (i <= N) {
            cout << i << endl;
            i++;
      }
      return 0;
}
```

pero al conocer el número de iteraciones que realizará el bucle será mas adecuado si utilizamos la estructura `for` de la siguiente forma:

```
/*
 * Autor:
 * Fecha:                Versión: 1.0
 *
 * Programa para mostrar n números naturales (Uso bucle for)
 */

#include <iostream>
using namespace std;

int main () {

      unsigned N;

      cout << "Introduzca un numero ";
      cin >> N;
      for (unsigned i = 1; i <= N; i++) {
            cout << i << endl;
      }
      return 0;
}
```

En este ejemplo, la variable de control "i" se declara en la propia inicialización del "for", por lo que la visibilidad de "i" se restringe a dicho bucle (no es accesible fuera). Esto es una técnica común y es lo más natural.

De hecho, podemos considerar la estructura `for` como una adaptación de la estructura `while`, que es especialmente apropiada para aquellas situaciones en las que se puede distinguir claramente en el bucle aquellas instrucciones que representan el comportamiento repetitivo de aquellas que añadimos para controlar el flujo de control. La estructura `for` permite reunir toda la información relativa al control del flujo del bucle en la cabecera, con lo que se mejora la legibilidad del programa, ya que cuando un programador lea un bucle `for`, accede fácilmente a la información que determina la evolución del flujo de control en el mismo.

Aunque nuestro ejemplo es muy simple, se ve claramente que en el bucle `for` basta leer la cabecera `(unsigned i = 1; i <= N; i++)` para saber que el flujo de control hará que el cuerpo del bucle se ejecute exactamente N veces; sin embargo, para obtener esa información del bucle `while` es necesario estudiar instrucciones previas (que pueden no estar justo al lado como en este caso) para saber el valor inicial de la variable `i`, y analizar con detalle el cuerpo del bucle para saber en que momento, y de qué forma se modifica dicha variable (en este caso es fácil porque el bucle es simple, aunque puede ser más difícil si el bucle representa una tarea más complicada).

Veamos otro ejemplo que calcula el primer número perfecto mayor de 28. Un número perfecto es aquel cuyo valor es igual a la suma de sus divisores (sin contar a él mismo).

```
/*
 * Autor:
 * Fecha:                Versión: 1.0
 *
 * Programa para el cálculo del primer número perfecto mayor que 28
 */
#include <iostream>
using namespace std;
int main()
{
    int intento;
    int suma;

    intento = 29;
    do {
        suma = 1;
        for (int cont = 2; cont < intento; cont++)  {
            if ((intento % cont) == 0) {
                suma = suma + cont;
            }
        }
        if (suma != intento) {
            intento++;
        }
    } while  (suma!=intento);

    cout << "Numero perfecto mayor que 28 = " << intento;
    return 0;
}
```

Como puede observase en el ejemplo, no existe ninguna restricción en cuanto a las sentencias que se pueden incluir en el cuerpo de un bucle. Así, es correcto definir otros bucles dentro, a lo que se denomina anidamiento de bucles. Es importante señalar que, en el anidamiento, el primer bucle en comenzarse es el último en terminarse. Esto es:

```
while (condicion) {
        …
        do {
              …
        } while ( cond);
        …
}
```

Este ejemplo en cambio no sería correcto:

```
while (condicion) {
        …
        do {
              …
        …
}
        } while ( cond);
```

Es importante hacer notar que, para poder aprovechar la legibilidad que proporciona el uso de la estructura `for`, es necesario que el cuerpo del bucle no afecte a la variable de control, pues en otro caso, la información proporcionada en la cabecera sería incompleta y por tanto, engañosa.

C++ da mucha libertad en el uso de la instrucción `for`, y permite entre otras cosas modificar la variable de control dentro del bucle. Sin embargo, abusar de dicha libertad puede ocasionar pérdida de legibilidad en el programa (como se comentó en el párrafo anterior), con todos los inconvenientes que ello ocasiona. Por este motivo, nosotros restringiremos la forma de usar esta estructura, limitándonos al formato comentado, en el que la cabecera siempre contiene una sentencia de asignación, la expresión lógica puede tener cualquier formato válido y la única modificación posible de la variable de control tiene lugar en la componente de incremento de la cabecera.

La variable de control puede ser declarada en la cabecera de la sentencia. De esa forma, la visibilidad de la variable se restringe al bucle (no es accesible fuera de éste) como un mecanismo que incida en que se trata de una variable de control de ese bucle.

II.8.4 Diseño de bucles. Concepto de invariante de un bucle

Para el diseño correcto de un bucle se deben considerar los siguientes aspectos:

- Diseño del flujo de control, determinando:
  - Condición de terminación del bucle.
  - La forma de inicializar y actualizar esa condición.
- Diseño del procesamiento interior del bucle, determinando:
  - El proceso que se repite
  - La forma de iniciarlo y actualizarlo.

El invariante de un bucle establece su comportamiento. Se expresa mediante un conjunto de condiciones que se deben cumplir antes de entrar por primera vez en el bucle, que se mantienen después de cada iteración y que por tanto se siguen cumpliendo cuando éste acaba. Es importante destacar:

- El diseño de un bucle puede simplificarse estableciendo antes su invariante.
- No confundir el invariante con la condición de control de un bucle.

Veamos cómo se podría diseñar e implementar un bucle basándonos en el establecimiento del invariante. Para ello utilizaremos el ejemplo del Cálculo del Factorial de un número natural n

leído por teclado.

Primer nivel de refinamiento. Si realizamos un primer esfuerzo para el diseño de la solución del problema, la primera versión de algoritmo podría quedar**:**

int main () {

- Leer número n.
- Si n es cero el resultado será uno. Si no, generar y multiplicar acumulativamente todos los números naturales comprendidos entre el uno y el número leido.
- Escribir el resultado.

}

En la segunda acción aparece un proceso iterativo cuyo invariante (siguiendo su definición) podría ser:

1) $n \geq 1$.
2) Una variable `contador` almacenará los distintos números naturales generados sucesivamente.
3) Una variable `fact` (`fact` $\geq$ 1) contendrá el producto acumulativo de todos los números naturales comprendidos entre uno y el valor de contador.

$$fact = \prod_{1}^{contador} i$$

4) De la expresión anterior deducimos que contador debe tener como valor final `n` para que fact contenga el factorial de `n`. Por otro lado, como n puede ser uno, el valor inicial de `contador` debe ser como máximo 1. Y como mínimo debe ser también uno puesto que `fact` $\geq$ 1.

El bucle quedaría de la siguiente forma:

```
int main () {
     unsigned fact,contador, n;
     …..
     fact= 1;
     contador =  1;
     while (contador < n) {
          contador =  contador + 1;
          fact =  fact * contador;
```

```
        }
        …
}
```

Segundo nivel de refinamiento. Una vez diseñado el bucle completaríamos el programa codificando el resto de pasos indicados en el primer nivel de refinamiento.

```
int main() {
      unsigned n, fact, contador;
      cout << "Introduzca el numero n: ";
      cin >> n;
      if (n == 0) {
            fact =  1;
      }else
```

```
            fact =  1;
            contador =  1;
```

Inv: $((contador >= 1) \wedge (contador <= n) \wedge (fact = \prod_{1}^{contador} i))$

```
            while (contador < n) {
                  contador =  contador + 1;
                  fact =  fact * contador;
```

Inv: $((contador >= 1) \wedge (contador <= n) \wedge (fact = \prod_{1}^{contador} i))$

```
            }
```

Inv: $((contador >= 1) \wedge (contador <= n) \wedge (fact = \prod_{1}^{contador} i))$

```
      }
      cout << "El factorial es: " << fact<< endl;
      return 0;
}
```

II.9 Control de Errores y Excepciones

Durante la ejecución de un programa se pueden dar situaciones inesperadas (malfuncionamiento del programa) o errores que desencadenarán su finalización.

Las sentencias condicionales se utilizan tradicionalmente para el control de estos errores. Por ejemplo, el siguiente fragmento de código controlará un error debido a una posible división por cero:

```
if (denominador==0) {
     cout << "Error división por cero" << endl;
} else {
     valor =numerador/denominador;
}
```

Por otro lado, las estructurs de iteración pueden utilizarse para control de errores obligando al usuario a repetir una acción si éste no la realiza correctamente. Supongamos que el usuario debe introducir un número positivo. El uso de un bucle puede obligarle a repetir la acción hasta que lo haga correctamente:

```
/*
 * Autor:
 * Fecha:                Versión: 1.0
 *
 *
 */

#include <iostream>
using namespace std;

int main () {
      int num;

      do {
            cout << "Introduzca un numero positivo";
            cin >> num;
      } while (num <0);
      // continua el programa
      return 0;
}
```

Sin embargo, existen otros mecanismos para manejar los errores inesperados, las **excepciones**, que se utilizan para indicar la aparición de estos errores. El procedimiento es muy simple, cuando se produce el error se eleva una excepción utilizando la palabra reservada **throw** que es seguida de un argumento que puede ser de cualquier valor. Veamos como utilizar dicha sentencia en el primero de los ejemplos anteriores:

```
if (denominador==0) {
      throw "Error división por cero" << endl;
} else {
      valor =numerador/denominador;
}
```

Cuando se ejecuta la sentencia `throw` el control del programa se pasa a otra parte conocida como **manejador de la excepción** y tiene que formar parte de una construcción `try/catch`. El manejador permitirá al programador dar una solución alternativa al error (dar un mensaje informativo e incluso resolver el error) y poder continuar la ejecución del programa. Este mecanismo se estudiará en el segundo cuatrimeste.

II.10 ERRORES FRECUENTES

- Usar = en lugar de == para efectuar una comparación de igualdad. El operador = expresa asignación de una valor a una variable, y en C++ se permite que forme parte de una expresión, por lo que el compilador no avisa de un error en caso de confundir uno por otro.

- Usar & para el AND lógico, que realmente es &&. El compilador no avisa del error porque & es un operador válido, aunque no hace lo que deseamos.
- Construir expresiones con operadores relacionales que no expresan lo que deseamos. C++ es muy flexible a la hora de construir expresiones, por lo que es posible que el compilador acepte expresiones que no se comportan como esperamos. Un ejemplo típico viene dado por la expresión `(0 <= X <= 100)`. Deseamos expresar la pregunta de si el valor de X está comprendido entre 0 y 100, sin embargo C++ interpreta algo totalmente distinto (que no entraremos a analizar). La expresión correcta es `(0 <= X) && (X <= 100)`. Para evitar estas situaciones, recomendamos que siempre que usemos una expresión compuesta usemos paréntesis en cualquier subexpresión que contenga.
- Olvidar la instrucción `break` al final de la alternativa seleccionada en una etiqueta de una instrucción `switch`. En tal caso, después de ejecutar la alternativa no se continúa por la instrucción que siga al `switch`, sino que se ejecutan las sentencias asociadas a la siguiente alternativa.
- Construir expresiones lógicas compuestas que no expresan realmente lo que deseamos expresar. Es frecuente usar el operador || cuando deberíamos usar el operador && y viceversa.
- Errores de ejecución por no tener en cuenta que las expresiones lógicas se evalúan en cortocircuito. Por ejemplo, la instrucción `V = ((A / B) > C) && (B != 0)` genera un error de ejecución si `B` vale cero, mientras que si aprovecho el cortocircuito, dicho error no se produce `V = (B != 0) && ((A/ B) > C)`.