

C++ básico mediante ejemplos



Fundamentos de programación para ingenieros

Julio Garralón Ruiz

Departamento de Lenguajes y Ciencias de la Computación

Universidad de Málaga

España

6 de octubre de 2016



CC Attribution-NonCommercial-ShareAlike (CC BY-NC-SA)

This license lets others remix, tweak, and build upon your work non-commercially, as long as they credit you and license their new creations under identical terms. To see a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/deed.en_EN or send a request to: Creative Commons, 171 Second Street, Suite 300 San Francisco, California 94105, USA.

You are free to:

Share Copy and redistribute the material in any medium or format.

Adapt Remix, transform, and build upon the material.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution – You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



Non commercial – You may not use the material for commercial purposes.



ShareAlike – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions – You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Índice general

1	C++ básico	1
1.1	Sentencias y datos	1
1.2	El programa principal	1
1.3	Salida de datos en pantalla	2
1.4	La Memoria	4
1.4.1	Variables y la sentencia de asignación	4
1.4.2	Constantes simbólicas	5
1.5	Cálculos aritméticos	6
1.6	Entrada de datos del usuario	8
1.7	Funciones definidas por el programador	10
1.8	Tipos de datos escalares	12
1.8.1	Números enteros	13
1.8.2	Números reales	13
1.8.3	Caracteres	14
1.8.4	Lógicos	15
1.9	Otras cuestiones aritméticas	16
1.9.1	Operadores de incremento y decremento	16
1.9.2	Operadores de asignación aritmética	16
1.9.3	Precedencia de operadores	17
1.9.4	Compatibilidad de tipos	18
1.9.5	La biblioteca matemática	18
1.10	Registros simples	19
1.11	Unidades léxicas	21
2	Decisiones	25
2.1	Expresiones lógicas	25
2.2	Bloques y sentencias compuestas	29
2.3	Sentencias de selección	29
2.3.1	La sentencia <code>if</code>	29
2.3.2	Sentencias <code>if</code> anidadas	31
2.3.3	Sentencias <code>if</code> múltiples	32
2.4	Ejemplos con funciones	34
2.4.1	La sentencia <code>switch</code>	36
2.4.2	El operador condicional	37
3	Bucles	41
3.1	El bucle genérico <code>while</code>	41
3.2	El bucle controlado por contador <code>for</code>	43
3.3	El bucle post-test <code>do-while</code>	45
3.4	Bucles anidados	46
4	Subprogramas	51
4.1	El concepto de llamada	51

4.2	Parámetros y procedimientos	53
4.3	Reglas de ámbito	56
4.3.1	Ámbitos de subprogramas	57
4.3.2	Ámbitos de datos y efectos laterales	57
4.4	Paso de parámetros	60
4.5	Cuestiones semánticas	60
4.6	Diseño modular	62
4.6.1	Diseño ascendente y descendente	62
5	Datos compuestos	65
5.1	Tipos definidos por el programador	65
5.2	Registros	65
5.3	Arrays	69
5.4	Matrices	76
5.5	Cadenas de caracteres	81
5.6	Anidamientos	82
A	Palabras reservadas de C++	95
B	Conjunto de caracteres ASCII	97
C	Tipos predefinidos de C++	99
C.1	Números enteros	99
C.2	Números reales	99
C.3	Caracteres	100
C.4	Lógicos	100
D	Secuencias de escape de C++	101
E	Bibliotecas estándares	103
E.1	Entrada/salida básica de C++	103
E.2	La biblioteca de cadenas de caracteres de C++	104
E.3	La biblioteca estándar de C	106
E.4	La biblioteca matemática de C	106
E.5	Biblioteca de límites de números enteros y reales de C	106

Índice de listados

Uso de la biblioteca de entrada/salida <code>iostream</code>	2
Caracteres nueva línea	2
Definición de variables	4
Asignaciones simples	5
Inicialización de variables	5
Definición de constantes simbólicas	6
Cálculos simples	6
División de tipo real	7
Uso de la biblioteca matemática <code>cmath</code>	7
Mensajes de entrada de datos al usuario	8
Entrada de datos de usuario de tipo entero	9
Entrada de datos de usuario de tipo real	9
Uso básico de funciones	11
Desbordes en operaciones aritméticas con naturales	13
Salida de números con la biblioteca <code>iomanip</code>	14
Concatenación de caracteres con el tipo <code>string</code>	14
Uso de <code>cin.get</code> para leer blancos	15
Operadores de incremento y decremento	16
Uso del operador de asignación aritmética <code>+=</code>	16
Uso del operador de asignación aritmética <code>*=</code>	17
Uso de la biblioteca matemática <code>cmath</code>	18
Uso de struct para agrupar variables de tipo simple	20
Uso de registros como parámetros a funciones	20
Identificación de unidades léxicas	21
Control de errores aritméticos	25
Definición y uso de funciones lógicas	28
Bloques	29
Sentencia <code>if</code> doble	29
Sentencia <code>if</code> simple	30
Anidamientos de sentencias <code>if</code>	31
Selecciones múltiples con <code>elses</code> anidados	32
La sentencia <code>if else</code> múltiple	32
Decisiones con una función lógica	34
Un programa completo con funciones simples	35
La sentencia <code>switch</code> básica	36
Casos múltiples en la sentencia <code>switch</code>	36
El operador condicional	37
Bucle <code>while</code> simple controlado por centinela	41
Bucle <code>while</code> simple controlado por contador	42
Sentencias de selección en el cuerpo de un bucle	42
Bucle <code>for</code> simple	43
Resultados intermedios de un bucle	44
Bucle <code>do-while</code> simple	45
Bucle anidado simple	47

Bucles anidados acoplados	47
Funciones con parámetros de entrada	54
Parámetros de salida en procedimientos	54
Parámetros de entrada/salida	55
Parámetro real promocionado al tipo del parámetro formal	60
Parámetro real truncado al tipo del parámetro formal	61
Uso básico de registros	67
Errores de rango al acceder a arrays	71
Comprobación de errores de rango en arrays	71
Uso básico de arrays modernos	74
Matrices cuadradas de longitud fija	77
Arrays bidimensionales de caracteres	77
Anidamiento de un array dentro de un registro	82
Anidamiento de un array incompleto usando marcas	84
Anidamiento de un array de registros dentro de otro registro	88

Prólogo

Estos apuntes son simplemente una serie de ejemplos que presentan los conceptos fundamentales del lenguaje de programación de propósito general C++. Estos conceptos incluyen el concepto de variable, asignación, sentencias condicionales, bucles, subprogramas y uso de estructuras de datos mediante arrays y registros. Están pensados para alumnos de primer curso de los distintos grados de informática e ingenierías donde se impartan asignaturas de programación. Las posibilidades de C++ como lenguaje *orientado a objetos* no se presentan aquí.

Se recomienda editar y ejecutar los ejemplos en un ordenador, y una vez entendidos, hacer los ejercicios en el orden en que se proponen, ya que se han seleccionado de forma que presentan los conceptos de programación con dificultad gradual creciente.

Málaga, 4 de octubre de 2016
Julio Garralón Ruiz

Capítulo 1

C++ básico

1.1. Sentencias y datos

Una *sentencia* es la unidad mínima *sintáctica* que representa una *acción* en un lenguaje de propósito general de alto nivel como C++. Las sentencias se componen de *unidades léxicas* indivisibles, las cuales a su vez se componen de combinaciones de caracteres ASCII¹.

Un *dato* es la unidad *léxica* fundamental que representa información *textual* o *numérica*. En última instancia, un dato es un valor individual almacenado en una *posición de memoria*.

1.2. El programa principal

Código fuente mínimo en C++.

```
int main()
{
}
```

Los comentarios en C++ no forman parte del programa ejecutable generado.

```
/*
    Esto es un comentario multilinea fuera del programa principal main().
    Los comentarios deben ir encerrados entre los pares de caracteres
    mostrados al principio y al final de este parrafo en el mismo orden.
*/
int main()
{
    // Esto es un comentario de una sola linea que se indica por una barra
    // doble / inclinada al principio de cada linea.
    // El cuerpo del main() encierra todas las sentencias del programa entre
    // los caracteres { y }.
    // Los comentario pueden ir tanto dentro del cuerpo del main, como fuera.
}
```

¹Véase la sección 1.11

1.3. Salida de datos en pantalla

El texto literal dentro de un programa C++ se escribe entre comillas, bien simples, cuando son caracteres individuales, o bien dobles, cuando son cadenas de caracteres de longitud variable. El siguiente ejemplo (incompleto) lo ilustra.

```
int main()
{
    cout << "This is text.";
    cout << "In C++, text is a character string.";
    cout << "The following are single characters.";
    cout << 'a';      // an ASCII letter
    cout << '7';      // an ASCII digit
    cout << '+';      // a special ASCII character
    cout << ';';      // only the ; inside the quotes is printed
    cout << 'H' << 'o' << 'l' << 'a'; // several items within a single cout
}
```

incomplete

El punto y coma es la unidad léxica que determina el final de una sentencia.
--

El uso de las sentencias `cout` y `cin` necesita incluir la definición de la biblioteca estándar de entrada/salida `iostream` mediante la directiva del compilador `include`, la cual se sustituye por el código fuente que incluye todas las definiciones de la biblioteca.

```
#include <iostream> // allows the use of cout
using namespace std; // search iostream in standard folders

int main()
{
    cout << "This is text.";
    cout << "In C++, text is a character string.";
    cout << "The following are single characters.";
    cout << 'a';      // an ASCII letter
    cout << '7';      // an ASCII digit
    cout << '+';      // a special ASCII character
    cout << ';';      // only the ; inside the quotes is printed
    cout << 'H' << 'o' << 'l' << 'a'; // several items within a single cout
}
```

Las líneas que empiezan con el símbolo # <i>no</i> forman parte de C++, son <i>directivas del preprocesador de C++</i> , y por tanto no llegan a ser compiladas, sino que desaparecen justo <i>antes</i> de la compilación.

Los *espacios en blanco* (' '), *tabuladores* ('\t'), y *saltos de línea* ('\n' o bien `endl`) también hay que imprimirlos explícitamente con la sentencia `cout`.

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

int main()
{
    cout << "This is text." << endl;
    cout << "In C++, text is a character string." << endl;
    cout << "The following are single characters." << endl;
}
```

```
cout << 'a' << ' ';    cout << '7' << ' ';
cout << '+' << ' ';    cout << ';' << ' ' << endl;
cout << 'H' << 'o' << 'l' << 'a' << endl;
}
```

Los números literales no se escriben entre comillas. El siguiente ejemplo incluye números enteros y reales. Los números reales se pueden escribir tanto en *coma fija* como en *coma flotante*.


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

int main()
{
    // poor output: cluttered numbers on screen
    cout << 12456 << -3413;    // integer numbers
    cout << 12.56 << -0.0004;  // real numbers in fixed point format
    cout << 1.3e25 << -0.4e35; // huge numbers in floating point format
    cout << 1.3e-15;           // close to zero number in floating point format
    // better: separated with whitespaces
    cout << 12456 << ' ' << -3413 << ' ' << 12.56 << ' ' << -0.0004 << ' '
        << 1.3e25 << ' ' << -0.4e35 << ' ' << d1.3e-15 << endl;
}
```

Diferentes tipos de datos se pueden mezclar en una sola sentencia `cout`.

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

int main()
{
    // mixing textual and numerical output
    cout << "Pi number: " << 3.1415926535 << endl;
        << "E number: " << 2.7182818284 << endl;
    cout << '2' << '+' << '2' << '=' << '4' << endl;
}
```

 Las sentencias se pueden dividir en varias líneas para mejorar su legibilidad, siempre que las unidades léxicas se mantengan indivisibles.

Ejercicios

1. Escribe un programa que muestre en pantalla tu nombre completo. Después amplíalo para escribir también tu edad de dos formas, (1) como caracteres, y (2) como un número.
2. Escribe un programa que imprima un pequeño cuadrado en pantalla de 4×4 asteriscos. Después amplíalo para que pinte, al lado, el triángulo que aparece a la derecha usando los espacios en blanco que sean necesarios.

```
****      *
****      **
****      ***
****      ****
```

1.4. La Memoria


1.4.1. Variables y la sentencia de asignación

Una *variable* es una posición de memoria principal identificada con un nombre simbólico que le asigna el programador. Dicho *identificador* debe elegirse de forma que recuerde fácilmente su contenido.

La sentencia de asignación es la sentencia más importante de los lenguajes de propósito general como C++, ya que representa el movimiento de datos en memoria. Su sintaxis utiliza el operador de asignación, que es el símbolo =:

`variable_izquierda = valor_derecha;`

La sentencia de asignación evalúa la *expresión* de la derecha (*rvalue*) y almacena el resultado en la variable de la izquierda (*lvalue*).

 ¡Cuidado! El contenido previo de la variable de la izquierda se pierde tras la asignación.

El almacenamiento de valores en variables, ya sean literales o calculados, se realiza con la sentencia de asignación, tal como pretende el siguiente ejemplo.

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

```
int main()
```

```
{
    counter_1 = 0;
    counter_2 = 0;
    radius_1 = 0.5;
    radius_2 = 1.75;
}
```

incomplete


Todas las variables deben ser *definidas* antes de ser utilizadas indicando su tipo delante de su identificador.

Las variables que almacenan números enteros se definen con la palabra reservada **int**, y las que almacenan números reales con la palabra reservada **float**.

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

```
int main()
```


```
{
    int counter_1, counter_2;    // integer numbers definition
    float radius_1, radius_2;   // real numbers definition
    counter_1 = 0;
    counter_2 = 0;
    radius_1 = 1.0;
    radius_2 = 0.5;
}
```

 Se pueden definir todas las variables que el programador desee, prácticamente no hay límite, pero es conveniente mantener el código limpio de definiciones de variables innecesarias.

Para ver el contenido de las variables en pantalla, se utiliza la sentencia `cout`.

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


int main()
{
    float radius_1, radius_2, radius_3;
    cout <<radius_1 <<' ' <<radius_2 <<' ' <<radius_3 <<endl;
}
```

 Antes de asignar el primer valor a una variable, ésta contiene un valor aleatorio que suele ser inútil o *basura*.

Los datos que almacenan las variables se pueden copiar a otras variables, es decir, se pueden trasladar o duplicar de un lugar de la memoria a otro.

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

int main()
{
    float radius_1, radius_2, radius_3;
    cout <<radius_1 <<' ' <<radius_2 <<' ' <<radius_3 <<endl;
    radius_1 = 1.0;
    radius_2 = radius_1;
    radius_3 = radius_2;
    cout <<radius_1 <<' ' <<radius_2 <<' ' <<radius_3 <<endl;
}
```

 Recuerda que después de cada asignación se pierde el contenido de la variable de la izquierda, pero los valores de la derecha permanecen inalterados en memoria.

El valor de la derecha de una asignación debe ser del mismo tipo o compatible con el tipo de la variable de la izquierda.

Las variables se pueden *inicializar* justo en la definición.

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

int main()
{
    float radius_1 = 1.1;    // initialized definition
    float radius_2 = 2.2;    // initialized definition
    float radius_3 = radius_2; // initialized definition
    cout <<radius_1 <<' ' <<radius_2 <<' ' <<radius_3 <<endl;
}
```

1.4.2. Constantes simbólicas


Las posiciones de memoria que no deben cambiar nunca su contenido se definen como *constantes simbólicas* con la palabra reservada **const**.

```

#include <iostream>
using namespace std;

int main()
{
    const float Pi_number = 3.1415926;    // cannot be changed
    const float E_number = 2.7182818;    // cannot be changed
    const float GoldenRatio = 1.618034;  // cannot be changed
    cout <<"Pi = " <<Pi_number <<endl;
    cout <<"E = " <<E_number <<endl;
    cout <<"Golden ratio = " <<GoldenRatio <<endl;
}

```

 En C++ es habitual identificar las constantes simbólicas con letras mayúsculas.

Ejercicios

1. Escribe un programa que intercambie el contenido de dos variables previamente inicializadas. Imprime los contenidos de las variables antes y después del intercambio en el mismo orden.
2. Escribe un programa que defina 4 variables, les asigne 4 valores diferentes, y rote sus contenidos. Para comprobar que funciona, el programa debe sacar en pantalla los contenidos de las variables antes de empezar los movimientos de datos y al final del programa en el mismo orden.

1.5. Cálculos aritméticos

Las *expresiones* son bien datos simples, llamados *operandos*, o bien combinaciones bien formadas de éstos y ciertos símbolos que actúan como *operadores*.

Las expresiones en C++ combinan datos del mismo tipo o *compatibles* en *notación infija*. Los operadores permitidos incluyen los aritméticos habituales +, -, *, /, y los paréntesis, los cuales dan prioridad a las operaciones que encierran. En ausencia de ellos, los operadores aditivos son menos prioritarios que los multiplicativos.

Las expresiones constituyen la parte de la derecha de la sentencia de asignación, permitiendo realizar cálculos para luego almacenarlos en la variable de la izquierda.

```


#include <iostream>
using namespace std;

int main()
{
    int num1=1, num2=1;
    int num3, num4, num5;
    num3 = num1 + num2;
    num4 = num2 + num3;
    num5 = num3 + num4;
    cout <<"First 5 Fibonacci numbers: "
         <<num1 <<' ' <<num2 <<' ' <<num3 <<' ' <<num4 <<' ' <<num5 <<endl;
}

```

Es muy habitual cambiar el valor de una variable utilizando el valor original que tenía. Es decir, la variable de la derecha puede aparecer en la expresión de la izquierda.

```
counter = counter + 1; // incremento en 1
x = x * x;             // elevar al cuadrado
```

 Recuerda que el operador = no es el operador de *igualdad*, sino un símbolo que representa una transferencia de datos en memoria.

El operador / puede representar la división entera si sus 2 operandos son números enteros, o la división real cuando al menos uno de sus 2 operandos es un dato de tipo real.

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

int main()
{
    const float Pi = 3.1415926535;
    const float EarthRadius = 6371; // kilometers
    float volume, surface;
    surface = 4 * Pi * EarthRadius * EarthRadius;
    volume = 4.0/3.0 * Pi * EarthRadius * EarthRadius * EarthRadius;
    cout <<"Earth surface: " <<surface <<endl;
    cout <<"Earth volume: " <<volume <<endl;
}
```

Otro operador aritmético muy habitual es el *módulo*, representado por el símbolo %. Produce como resultado el resto de la división entera, mientras que / devuelve el cociente de la división entera (siempre y cuando los dos operandos sean de tipo entero):

```
1 / 2   vale  0
1 % 2   vale  1
```

Ejemplos útiles para extraer los dígitos de un número:

```
x % 10    devuelve sólo las unidades de x
x % 100   devuelve sólo las decenas de x
x % 1000  devuelve sólo las centenas de x
x / 10    elimina las unidades de x
x / 100   elimina las decenas de x
x / 1000  elimina las centenas de x
```

En las expresiones pueden aparecer *llamadas* a funciones matemáticas si se incluye la definición de la biblioteca matemática estándar de C `cmath` (véase el apéndice E). El siguiente ejemplo pretende calcular las soluciones reales de una ecuación de segundo grado.

```
#include <iostream> // cout, endl
#include <cmath>     // sqrt
using namespace std;

int main()
{
    float a, b, c; // 2nd. degree equation coefficients
    float root1, root2; // roots to be calculated
    float discr; // intermediate result
    a = 1.0;
    b = 2.0;
    c = -3.0;
    discr = b*b - 4*a*c;
```

```

root1 = -b + sqrt(discr) / 2*a;    flawed
root2 = -b - sqrt(discr) / 2*a;    flawed
cout <<"Root 1: " <<root1 <<endl;
cout <<"Root 2: " <<root2 <<endl;
}

```

Cuidado con la precedencia de operadores. Las expresiones correctas son del ejemplo anterior son:

```

root1 = (-b + sqrt(discr)) / (2*a);
root2 = (-b - sqrt(discr)) / (2*a);

```

1.6. Entrada de datos del usuario

Un programa C++ no es muy útil si su salida siempre es la misma. Para darle mayor utilidad debe utilizar datos de usuario introducidos en tiempo de ejecución. Se puede *leer de teclado* números, caracteres individuales o cadenas de caracteres. El siguiente ejemplo ilustra la entrada de una cadena de caracteres en una variable textual de tipo *string*.

```

#include <iostream>
using namespace std;

int main()
{
    string first_name, last_name;
    cin >>last_name;
    cin >>first_name;
    cout <<"Your full name is " <<first_name <<' ' <<last_name <<endl;
}

```

incomplete

👁 [Los contenidos previos de las variables especificadas en una sentencia `cin` se pierden!]

Siempre es conveniente avisar al usuario qué dato debe introducir (el usuario no tiene por qué conocer las instrucciones que ejecuta un programa).

```

#include <iostream>
using namespace std;

int main()
{
    string first_name, last_name;
    cout <<"Your last name: ";    // requesting input
    cin >>last_name;              // accepting (or reading) input
    cout <<"Your first name: ";  // requesting more input
    cin >>first_name;            // accepting (or reading) input
    cout <<"Your full name is " <<first_name <<' ' <<last_name <<endl;
}

```

👁 [En tiempo de ejecución el dato tecleado no llega al espacio de memoria del programa, es decir, a sus variables, hasta que el usuario pulsa la tecla *Return*. Además, en general todos los caracteres blancos anteriores se ignoran.]

Ejemplo con datos de usuario de tipo entero.


```

#include <iostream>
using namespace std;

int main()
{
    int a, b, result;
    cout <<"Enter two integer numbers to be operated: ";
    cin >>a >>b;    // two inputs with a single sentence
    result = a + b;
    cout <<"Their sum is "<<result <<endl;
    result = a * b;
    cout <<"And their product is "<<result <<endl;
}

```

Ejemplo con datos de usuario de tipo real.

```

#include <iostream>
using namespace std;

int main()
{
    float a, b, c;        // 2nd. degree equation coefficients
    float root1, root2;   // roots to be calculated
    float discr;          // intermediate result
    cout <<"Coefficients of 2nd. degree equation: ";
    cin >>a >>b >>c;
    discr = b*b - 4*a*c;
    root1 = (-b + sqrt(discr)) / (2*a);
    root2 = (-b - sqrt(discr)) / (2*a);
    cout <<"Root 1: " <<root1 <<endl;
    cout <<"Root 2: " <<root2 <<endl;
}

```

El último ejemplo puede fallar. ¿Por qué?

Ejercicios

1. Escribe un programa que escriba en pantalla la tabla de multiplicar del 1 al 10 de un número entero n elegido por el usuario. Usa el carácter tabulador '\t' para alinear los números en columnas. Por ejemplo, si el usuario desea la tabla del 12, la salida debe tener un formato similar al siguiente:

```

12 x  1 =  12
12 x  2 =  24
12 x  3 =  36
...

```

2. Escribe un programa que lea dos números enteros de teclado y los sume, los reste, los multiplique, y calcule tanto el cociente como el resto de su división entera.
3. Escribe un programa que, utilizando solo dos variables, calcule la suma de 5 números enteros leídos por teclado e imprima el resultado final.
4. La aceleración de caída g en la superficie de un planeta se calcula con la fórmula

$$g = \frac{GM}{r^2},$$

donde $G = 6,6742 \times 10^{-11} \text{Nm}^2/\text{kg}^2$ es la constante de gravitación universal, M es la masa del planeta, y r la distancia del objeto al centro del planeta. Con los siguientes datos podrías calcular la aceleración de caída g en los astros indicados:

Planeta	Masa (kg)	Radio (m)
Tierra	$5,97 \times 10^{24}$	$6,378 \times 10^6$
Marte	$6,42 \times 10^{23}$	$3,397 \times 10^6$
Luna	$7,35 \times 10^{22}$	$1,737 \times 10^6$
Júpiter	$1,90 \times 10^{27}$	$71,492 \times 10^6$
Sol	$1,99 \times 10^{30}$	$6,96 \times 10^8$

Escribe un programa que calcule el peso P (en newtons) de un objeto en la superficie de diferentes planetas según la fórmula $P = mg$, donde m es la masa del objeto en kilogramos, dato que introduce el usuario por teclado, y g la constante de aceleración en la superficie del planeta. Almacena en constantes simbólicas las aceleraciones en los distintos astros utilizando las expresiones apropiadas en C++.

5. Escribe un programa que calcule el tiempo t en segundos que tarda un objeto en llegar al suelo desde varias distancias: 1 metro, 10 metros, 100 metros, 1 kilómetro y una distancia introducida por el usuario. Despeja t en la fórmula $e = \frac{1}{2}gt^2$, y utiliza $g = 9,81 \text{m/s}^2$.
6. La ley de Coulomb dice que la fuerza, F , que actúa entre dos esferas con cargas eléctrica q_1 y q_2 , separadas una cierta distancia r , puede obtenerse con la fórmula

$$F = k \frac{q_1 q_2}{r^2},$$

donde k es la *constante de Coulomb*, que vale $8,99 \cdot 10^9 \text{Nm}^2/\text{C}^2$. Escribe un programa que calcule las fuerzas de atracción de 2 cargas cuyo valor se introduce por teclado y están separadas las siguientes distancias: 1 milímetro, 1 centímetro, 1 metro, 100 metros, 1 kilómetro, y una distancia que introduzca el usuario por teclado en tiempo de ejecución. ¿Observas alguna forma en que el usuario podría hacer fallar el programa?

7. Sabiendo que la receta del “pastel de manzana danés” para 4 personas necesita

675 gramos de manzanas,
75 gramos de mantequilla,
150 gramos de azúcar,
100 gramos de migas de pan,
150 mililitros de leche,

escribe un programa que calcule y saque en pantalla los ingredientes necesarios de un pastel para un número variable de personas que se lee por teclado. Define como constantes simbólicas los valores literales que utilices.

1.7. Funciones definidas por el programador

En C++ se puede *encapsular* varias operaciones que calculan un solo valor en *funciones*, las cuales son *subprogramas* que trabajan con datos de entrada *parametrizados*. Para ejecutar un subprograma se debe realizar una *llamada* con los parámetros adecuados. Cuando la ejecución llega al punto de llamada, entonces el *control* de la ejecución pasa al subprograma, quedándose pendiente la ejecución del subprograma que lo invoca. Cuando el subprograma invocado termina de ejecutarse, el control *retorna* de nuevo a la sentencia que seguía a la llamada. Vemos que una llamada a un subprograma es la primera sentencia que altera el flujo secuencial habitual de las instrucciones de un programa.

De momento vamos a definir y llamar a subprogramas que sólo admiten parámetros de entrada y calculan un solo valor de *retorno*. Como todos los datos, el valor que retorna una función debe ser de un

tipo determinado. Estos subprogramas se conocen en C++ como *funciones*². Las reglas sintácticas para utilizar funciones en C++ se resumen en el siguiente cuadro.

1. La definición de cada subprograma se realiza una sola vez antes y fuera del cuerpo del `main`. Se compone de una cabecera o *interfaz de e/s* que, además del nombre de la función, especificará el tipo del resultado de la función, y el tipo de cada parámetro de entrada.
2. Se pueden realizar múltiples *llamadas* con parámetros diferentes siempre que el número de parámetros y el tipo de cada uno sea compatible con su especificación siguiendo el orden de la interfaz de e/s.
3. La llamada, puesto que es un valor a calcular, puede aparecer en cualquier expresión que admita su tipo como compatible.
4. Un subprograma puede a su vez llamar a otros subprogramas que se hayan definido previamente.
5. El resultado calculado se devuelve con la sentencia **return** y será la última sentencia que se ejecuta de la función, porque además devuelve el control de ejecución al subprograma que invocó a la función. El resultado devuelto sustituye en tiempo de ejecución a la llamada dentro de la expresión que contenga la llamada.



Recuerda que la ejecución de un programa empieza y termina en el cuerpo del subprograma principal `main()`. Aunque se defina completamente un subprograma, si no se realiza ninguna llamada, no se ejecutará nunca.

El siguiente ejemplo reescribe el anterior que calculaba la suma y el producto de dos números.

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

// definition of function Sum() of type int
int Sum(int number1, int number2) // input/output interface
{
    int result = number1 + number2;
    // the result type returned must be that of the function
    return result;
}

// definition of function Product() of type int
int Product(int number1, int number2) // input/output interface
{
    int result = number1 * number2;
    // the result type returned must be that of the function
    return result;
}

int main()
{
    int a, b, added, multiplied;
    cout << "Enter two integer numbers to be operated: ";
    cin >> a >> b;
    added = Sum(a, b); // calling point to Sum(...)
    multiplied = Product(a, b); // calling point to Product(...)
```

²Ya que recuerdan a las funciones matemáticas univaluadas.

```
cout <<"Their sum is "<<added <<"and their product "<<multiplied <<endl;
}
```

Ejercicios

1. Escribe las interfaces de entrada-salida apropiadas para funciones que calculen:
 - (1) el máximo de tres números reales;
 - (2) la distancia entre dos puntos del plano;
 - (3) el número de dígitos de un número entero;
 - (4) cuántos números son mayores que cero de tres dados;
 - (5) cuántas vocales hay en cuatro caracteres de entrada; y
 - (6) el valor real de un número racional (que consta de un numerador y un denominador).
2. Reescribe el ejercicio 4 de la sección anterior que calcula el peso de un objeto en diferentes planetas definiendo una función que calcula el peso de un objeto en un planeta genérico cualquiera. El programa debe realizar una llamada a esta función para cada planeta. Toda la entrada-salida de datos debe realizarse en el programa principal.
3. Reescribe el ejercicio 5 que calcula la caída de un objeto utilizando una función que realice el cálculo del peso. Llama a esta función desde el principal dos veces, cada una con valores de altura diferentes que se leen de teclado. Toda la entrada-salida de datos debe realizarse en el programa principal.
4. Reescribe el ejercicio 6 que calcula la fuerza de atracción entre dos cargas eléctricas utilizando una función que realice el cálculo. Toda la entrada-salida de datos debe realizarse en el programa principal.
5. Escribe una función que calcule la distancia euclídea entre dos puntos del plano definidos por sus coordenadas 2D: (x_1, y_1) y (x_2, y_2) según la fórmula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

El programa principal leerá de teclado las coordenadas de dos puntos y llamará a la función 3 veces para calcular y sacar en pantalla:

- (1) la distancia entre ellos;
- (2) la distancia entre el primer punto y el origen; y
- (3) la distancia entre el segundo punto y el origen.

1.8. Tipos de datos escalares

El tipo de un dato determina el dominio de posibles valores que puede tomar, y las operaciones que se pueden realizar con él. C++ dispone de los tipos de datos escalares *predefinidos* siguientes, algunos de los cuales difieren simplemente en la extensión de su dominio (véase la sección C del apéndice para más información):

- Números enteros: `short`, `int`, `long`, `unsigned`.
- Números reales: `float`, `double`.
- Caracteres individuales: `char`.

- Cadenas de caracteres: `string`.
- Valores lógicos o *booleanos*: `bool`.

Aparte de los operadores específicos para cada uno de ellos, todos los datos escalares pueden asignarse con el operador de asignación `=`, y comparados con los operadores relacionales `<`, `<=`, `>`, `>=`, y de igualdad `==`, `!=`.

Además los números enteros y los caracteres individuales son *ordinales*, esto es, cada valor tiene exactamente un sucesor y un predecesor, excepto el primero y el último. Estos tipos de datos son adecuados para las comparaciones de igualdad, a diferencia de los números reales cuyo resultado depende de la precisión de la máquina.

Todos los datos de tipos predefinidos pueden mostrarse en pantalla y ser leídos de teclado directamente con las sentencias `cout` y `cin` de la biblioteca de entrada/salida estándar (aunque los valores mostrados para los datos de tipo `bool` serán simplemente el valor 0 si el dato vale `false` y 1 si vale `true`).

1.8.1. Números enteros

Los números naturales se definen con la palabra reservada `unsigned` o con la combinación `unsigned int`. No pueden contener números negativos por lo que no son adecuados cuando se van a realizar restas u otras operaciones que puedan producir resultados negativos.

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

int main()
{
    unsigned number1, number2, result;
    cout << "Enter 2 naturals to be subtracted: ";
    cin >> number1 >> number2;
    result = number1 - number2;    flawed
    cout << "Difference: " << result << endl;
}
```

Los números enteros definidos con la palabra reservada `int` sí pueden contener números negativos y pueden utilizarse con todos los operadores aritméticos básicos `{+, -, *, /, %}`. En realidad, `short` y `long` son *modificadores* de tipo que especifican un número de bytes menor o mayor respectivamente que el número de bytes reservado para un dato de tipo `int` (véase la sección C del apéndice para más detalles).

1.8.2. Números reales

En C++ los números reales pueden tener diferente precisión y diferente dominios según el número de palabras de memoria utilizadas para almacenarlos:

- *Precisión simple*: una palabra de memoria, **`float`**.
- *Precisión doble*: dos palabras, **`double`**.
- *Precisión extendida*: cuatro palabras, **`long double`**.

Los operadores específicos son los aritméticos reales `+`, `-`, `*` and `/`.

Los números reales se pueden escribir en *coma fija*, como 12.34, -7.567 or .54, o en *coma flotante*, como en 1.5e2 ($= 1,5 \times 10^2$) or 7.5e-7 ($= 7,5 \times 10^{-7}$). Sin embargo, internamente se almacenan siempre en coma flotante.

La formato de la salida en pantalla se puede cambiar utilizando el *manipulador* `setprecision` de la sentencia `cout` definido en la biblioteca de extensión de la entrada/salida `iomanip`, tal como ilustra el siguiente ejemplo.

```
#include <iostream>    // cout, cin, endl
#include <iomanip>      // fixed, setprecision()
using namespace std;

int main()
{
    const float GravitationalK = 6.67384e-11; // m3 kg^-1 s^-2

    // by default, large or small numbers are shown with floating format
    cout <<"Gravitational Constant (floating point format):\n"
          <<"By default:\t" <<GravitationalK <<endl;
    // and you can change the number of significant digits printed on screen
    cout <<"Precision 4:\t" <<setprecision(4) <<GravitationalK <<endl;

    // fixed point format can also be used...
    cout <<"Gravitational Constant (fixed point format):\n" <<fixed;
    cout <<"By default:\t" <<GravitationalK <<endl;
    // or with arbitrary precision (counting from the decimal point
    cout <<"Precision 2:\t" <<setprecision(2) <<GravitationalK <<endl
          <<"Precision 16:\t" <<setprecision(16) <<GravitationalK <<endl;
}
```

La salida del ejemplo debe ser muy parecida a la siguiente:

```
Gravitational Constant (floating point format):
By default:      6.67384e-11
Precision 4:     6.674e-11
Gravitational Constant (fixed point format):
By default:      0.0000
Precision 2:     0.00
Precision 16:    0.000000000000667384
```

1.8.3. Caracteres

Cada carácter, bien individual, o bien formando parte de una cadena de caracteres, se almacena como un entero que ocupa un byte que coincide con su código ASCII.

Las variables de caracteres individuales se definen con la palabra reservada **char**, y las cadenas con el tipo `string` de la biblioteca estándar de C++ `string`. Las cadenas de caracteres son tipos compuestos, no escalares, pero esta biblioteca de C++ permite hacer muchas operaciones con ellas como si fueran datos individuales. Tanto los caracteres individuales como las cadenas pueden compararse con los operadores relacionales y de igualdad utilizando para ello la ordenación de la tabla ASCII. Además, las cadenas pueden utilizar el operador `+` para concatenar caracteres.

Los literales caracteres individuales se escriben entre comillas simples, mientras que las cadenas entre comillas dobles.

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

int main()
{
    char a, b, c;
```

```

string together;

cout <<"Enter any three characters: ";
cin >>a >>b >>c;
// you cannot concatenate single characteres as in a+b+c
together = a;
together = together + b;
together = together + c;
cout <<"The concatenation of " <<a <<" , " <<b <<" and " <<c
    <<" is " <<together <<endl;
}

```

Cuando leemos caracteres simples con la sentencia `cin` en general se ignoran los blancos que preceden al carácter, pero este comportamiento es diferente si leemos con la sentencia `cin.get(...)` de la biblioteca `iostream`, tal como muestra el ejemplo siguiente.

```

#include <iostream>
using namespace std;

int main()
{
    char character;
    cout <<"Enter a character: ";
    cin.get(character);
    cout <<"Your character is \" <<character <<\" \" <<endl;
}

```

1.8.4. Lógicos

El dominio del tipo *booleano* son simplemente dos valores, **false** y **true**. Es el tipo del resultado producido por una expresión lógica.

```

int number;
bool greaterthan10;
...
greaterthan10 = number>10;
// now greaterthan10 holds true or false
// depending on the contents of number
...

```

Veremos más sobre este tipo de datos en el capítulo siguiente.

Ejercicios

1. Escribe un programa que muestre por pantalla el número de bytes que ocupa cada uno de los siguientes tipos de datos. Para ello utiliza la función predefinida `sizeof(tipo)` de C++ que devuelve el número de bytes que ocupa los datos del tipo que se le pasa como parámetro de entrada:

char	int	float
short	long	double
unsigned int	long long	long double

Operador de asignación	Sentencia equivalente
<code>var += expr</code>	<code>var = var + expr</code>
<code>var -= expr</code>	<code>var = var - expr</code>
<code>var *= expr</code>	<code>var = var * expr</code>
<code>var /= expr</code>	<code>var = var / expr</code>
<code>var %= expr</code>	<code>var = var % expr</code>

Tabla 1.1: Operadores de asignación aritmética.

1.9. Otras cuestiones aritméticas

1.9.1. Operadores de incremento y decremento

Estos operadores se utilizan mucho en C++ sobre variables de tipo entero. El de *incremento* `++` suma 1 al contenido de una variable, y el de *decremento* `--` resta 1.

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

int main()
{
    int number, original;
    cout <<"Input an integer number: ";
    cin >>number;
    cout <<"The previous three integer numbers are: ";
    original = number;
    number--;    cout <<number <<" ";
    number--;    cout <<number <<" ";
    number--;    cout <<number <<endl;
    cout <<"The following three integer numbers are: ";
    number = original;
    number++;    cout <<number <<" ";
    number++;    cout <<number <<" ";
    number++;    cout <<number <<endl;
}
```

1.9.2. Operadores de asignación aritmética

Estos operadores también actualizan el contenido de una variable numérica, pero evitan repetir su identificador en la parte derecha de una asignación. La tabla 1.1 enumera estos operadores y su sentencia de asignación equivalente.

El siguiente ejemplo utiliza el operador de asignación suma.

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

int main()
{
    // in the Fibonacci series each number is the sum of the two previous
    int last2=1, last=1;           // the first two are always 1, 1
    int fib = last2 + last;
    cout <<"Fibonacci series: " <<last2 <<' ' <<last <<' ' <<fib <<' ';
```


Precedencia	Operadores	Asociatividad
más alta	()	izquierda a derecha
	++ --	derecha a izquierda
	* / %	izquierda a derecha
	+ -	izquierda a derecha
más baja	= += -= *= /= %=	derecha a izquierda

Tabla 1.2: Reglas de precedencia de los operadores aritméticos.

```

last2 = last; last = fib; fib += last2; cout <<fib <<' ';
last2 = last; last = fib; fib += last2; cout <<fib <<' ';
last2 = last; last = fib; fib += last2; cout <<fib <<' ';
last2 = last; last = fib; fib += last2; cout <<fib <<' ';
last2 = last; last = fib; fib += last2; cout <<fib <<"...\n";
// ... endlessly ...
}

```

Y el siguiente el de asignación multiplicación.

```

#include <iostream>
using namespace std;

int main()
{
    float x, ratio;
    cout <<"Starting term of a geometric series: ";
    cin >>x;
    cout <<"Ratio: ";
    cin >>ratio;
    cout <<x <<' '; x *= ratio;
    cout <<x <<' '; x *= ratio;
    cout <<x <<' '; x *= ratio;
    cout <<x <<' '; x *= ratio;
    cout <<x <<"...\n";
}

```

1.9.3. Precedencia de operadores

Las reglas de precedencia aritmética en C++ se resumen en las siguientes:

- Los operadores multiplicativos son más prioritarios que los aditivos. La tabla 1.9.3 los ordena de mayor a menor precedencia.
- En el mismo nivel de precedencia, y a excepción de los operadores de asignación, se evalúan de izquierda a derecha.
- Los operadores de asignación son los menos prioritarios.

Los siguientes ejemplos se muestran con sus expresiones equivalentes usando paréntesis:

```

a /= a - b / c + d    ≡    a = (a / ((a - (b / c)) + d))
b *= a / b + c % d    ≡    b = (b * ((a / b) + (c % d)))
a - b /= c % d        →    error

```

1.9.4. Compatibilidad de tipos

En general, cuando se mezclan tipos *incompatibles* en una expresión, se producen *errores semánticos*. No obstante, C++ permite mezclar algunos tipos si puede tener sentido. A continuación se dan ejemplos útiles.

Conversión implícita. Promoción a número real.

```
float money, price_per_hour;
money = hours * price_per_hour; // hours promociona a float
```

Conversión implícita. Conversión a número entero, se pierde precisión.

```
int hours;
hours = money / price_per_hour; // domina tipo de la variable destino
```

Conversión implícita. A división entera o real.

```
float num_real; int num_entero;
num_real = 1/2; // = 0.0
num_real = 1.0/2; // = 0.5
num_entero = 1.0/2.0; // = 0, domina tipo de la variable destino
```

Conversión explícita. Obtención del código ASCII de un carácter.

```
char a;
cout <<"El codigo ASCII de " <<a <<" es " <<int(a) <<endl;
```

Conversiones explícitas. A división entera.

```
float time_in_minutes; int hh, mm;
hh = int(time_in_minutes) / 60;
mm = int(time_in_minutes) % 60;
```

👁 El tipo de la variable izquierda de una asignación no puede cambiarse, ni siquiera temporalmente.

1.9.5. La biblioteca matemática

La biblioteca matemática estándar de C `cmath`, permite hacer llamadas a funciones trigonométricas, calcular logaritmos, exponenciales, o potencias reales, entre otras. El siguiente ejemplo muestra como calcular el seno y coseno del ángulo doble.

```
#include <iostream>
#include <cmath> // cos(), sin()
using namespace std;

int main()
{
    const double Pi = 4*atan(1.0);
    double degrees, radians, sine2, cosine2, angle2;
    cout <<"Enter an angle in degrees: ";
    cin >>degrees;
    radians = degrees*Pi/180;
    sine2 = 2*sin(radians)*cos(radians);
    cosine2 = cos(radians)*cos(radians) - sin(radians)*sin(radians);
    cout <<"The sine of the double angle is " <<sine2
```

```

    <<" and the cosine " <<cosine2 <<endl;
}

```

Ejercicios

1. Escribe el resultado que produciría cada una de las siguientes sentencias, teniendo en cuenta su definición inicializada: `int d=2, v=50, n=10, t=5; string cadena="Hola";`

- | | |
|-------------------------------------|--|
| (1) <code>n / t + 3</code> | (6) <code>18 / -t</code> |
| (2) <code>v / t + n - 10 * d</code> | (7) <code>v % 3</code> |
| (3) <code>v + n / t * d</code> | (8) <code>cadena + ", "+ "que tal!"</code> |
| (4) <code>v + n / t + d</code> | (9) <code>cadena + '.'</code> |
| (5) <code>d / 57</code> | (10) <code>d *= n + 1</code> |

2. Escribe un programa que calcule la media de 5 números enteros leídos de teclado.
3. Escribe un programa que lea de teclado 4 caracteres y muestre en pantalla sus códigos ASCII.
4. Usando los operadores de asignación, escribe un programa que calcule y muestre en pantalla los factoriales de los 10 primeros números naturales. Después amplía tu programa para incluir factoriales números naturales mayores hasta obtener errores de *desborde*.
5. Escribe un programa que lea de teclado un número de 5 dígitos como si fuera un solo número entero e imprima en pantalla la suma de sus dígitos. (Sugerencia: utiliza los operadores resto y cociente de la división entera.)
6. Escribe un programa que lea de teclado un número de segundos arbitrariamente grande y responda con el tiempo equivalente en formato `hh:mm:ss`. Por ejemplo, si se introduce 12345 segundos, la salida debe ser `3:25:45`.
7. Diseña una función que devuelva un valor aproximado del número e según la fórmula:

$$e = \left(1 + \frac{1}{x}\right)^x,$$

donde x es un número real de entrada a la función. El programa principal debe leer x de teclado y llamar a la función 3 veces con valores de entrada $x/100$, x , y $100x$ y escribir en pantalla esas 3 aproximaciones al número e . Utiliza el tipo `double` y el manipulador `setprecision(ndigitos)` para escribir los valores en pantalla controlando el número de dígitos significativos.

Utiliza el manipulador `setprecision(ndigitos)` de la biblioteca estándar `iomanip` si quieres mejorar la precisión mostrada en pantalla con la sentencia `cout`.

1.10. Registros simples

Además de variables, el programador puede *definir tipos* nuevos, no predefinidos, los cuales suelen ser *tipos compuestos*. Un uso muy simple y útil son *registros* (**struct** en C++) que almacenan varias variables de tipo simple. Para ello, utilizamos la palabra reservada **typedef** que nos permite primero definir el tipo, y una vez definido el tipo, podemos definir las variables.

Por ejemplo, los números racionales no existen en C++, pero se pueden definir agrupando en un registro el numerador y denominador de una fracción.

```

#include <iostream>
using namespace std;

typedef struct {
    int numerator;    // field 1
    int denominator; // field 2
} TRational; // new compound type

int main()
{
    TRational q1, q2, sum; // three rationals, two integers inside each
    cout <<"Enter a rational number (two integers): ";
    cin >>q1.numerator >>q1.denominator;
    cout <<"Enter another rational number (two integers): ";
    cin >>q2.numerator >>q2.denominator;
    sum.numerator = q1.numerator*q2.denominator + q1.denominator*q2.numerator;
    sum.denominator = q1.denominator*q2.denominator;
    cout <<"Their sum is " <<sum.numerator<<"/"<<sum.denominator <<endl;
}

```

Pero, cuidado, en este ejemplo no podríamos haber escrito la sentencia de suma directamente:

```

TRational q1, q2, sum;
sum = q1 + q2;

```

ya que el operador primitivo de la suma no existe en C++ para números racionales. Estas operaciones con tipos no predefinidos suelen programarse en forma de subprogramas como muestra el siguiente ejemplo.

```

#include <iostream>
using namespace std;

typedef struct {
    int numerator;    // field 1
    int denominator; // field 2
} TRational; // new compound type

TRational SumQ(TRational q1, TRational q2)
{
    TRational qs;
    qs.numerator = q1.numerator*q2.denominator +
                  q1.denominator*q2.numerator;
    qs.denominator = q1.denominator*q2.denominator;
    return qs;
}

int main()
{
    TRational r1, r2, result; // three rationals
    cout <<"First rational number to sum up (two integers): ";
    cin >>r1.numerator >>r1.denominator;
    cout <<"Second rational number (two integers): ";
    cin >>r2.numerator >>r2.denominator;
    result = SumQ(r1, r2); // compatible function call
    cout <<"Their sum is " <<result.numerator<<"/"<<result.denominator <<endl;
}

```

Observa que los parámetros pasados a una función pueden ser de tipos no predefinidos. También el tipo del valor devuelto por una función puede ser un tipo definido por el programador.

☞ Salvo las cadenas de caracteres, las variables de tipo compuesto no pueden escribirse en pantalla o leerse de teclado de forma conjunta. Hay que acceder individualmente a cada componente.

Ejercicio

1. Define un `TPunto2D` para almacenar las 2 coordenadas (x, y) de un punto. Escribe una función que acepte 2 puntos como parámetros de entrada y devuelva la distancia entre ellos. El programa principal leerá las coordenadas de ambos puntos y sacará el resultado en pantalla.

1.11. Unidades léxicas

Las unidades mínimas de información completa en un programa C++ son grupos de caracteres ASCII que no contienen espacios entre ellos que se llaman *unidades léxicas*. En el siguiente ejemplo están todas las que debes saber reconocer, ya que los compiladores suelen hacer referencia a ellas en sus mensajes de error.

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

int main()
{
    const float Pi=3.141592654;
    float radius, area;
    // input
    cout << " Radius: ";
    cin >> radius;
    // calculation
    area = Pi*radius*radius;
    // output
    cout << " Area: " << area << endl;
}
```

Palabras reservadas: `int`, `main`, `const`, `float`.

Literales: `3.141592654`, `" Radius: "`, `" Area: "`.

Identificadores: `Pi`, `radius`, `area`.

Identificadores estándares: `iostream`, `cout`, `cin`, `endl`.

Operadores: `=`, `*`, `<<`, `>>`.

Delimitadores: comentarios, `(`, `)`, `{`, `}`, `;`.

Ejercicios de repaso

1. Escribe las siguientes fórmulas matemáticas en forma de expresiones utilizando sólo las primitivas básicas y las funciones de la biblioteca estándar de C++. Todas las letras representan números reales, excepto i , que es el número imaginario; y los pares del tipo (x, y) son números complejos. Algunas de ellas no se pueden escribir, indica cuáles son.

$$\begin{array}{lll}
(1) & \sqrt{x_1^2 + x_2^2} & (3) \quad (a, b) + (x, y) \quad (5) \quad \cos a + i \operatorname{sen} b \\
(2) & \frac{-b - \sqrt{b^2 - 4ac}}{2a} & (4) \quad |a \log_{10} x + b \ln x| \quad (6) \quad \lceil 1 + n^2 \rceil
\end{array}$$

2. Escribe un programa que escriba en pantalla la figura de la derecha.

```

      *
    * *
  *   *
*     *
  *   *
    * *
      *

```

3. Escribe un programa que lea de teclado un número de 5 dígitos como si fuera un solo número entero e imprima sus dígitos separados por blancos. Después, amplíalo para construir aritméticamente un número entero con los mismos dígitos, pero invertidos. ¿Qué ocurre si el número no es de 5 cifras?
4. Diseña una función que calcule y devuelva el *índice de masa corporal* de un humano según la fórmula:

$$IMC = \frac{\text{peso}}{\text{altura}^2},$$

donde el *peso* en kilogramos y la altura en *metros* son parámetros de entrada a la función. Llama a la función 3 veces desde un programa principal que lea de teclado el peso y altura de una persona que quiere adelgazar. La primera llamada utilizará el peso real introducido, la segunda 5 kilos menos, y la tercera 10 kilos menos. (Nota: se asume que el IMC de una persona sana suele estar entre 18.5 y 24.9).

5. Define un tipo `TPunto2D` y utilízalo para realizar este ejercicio. Escribe una función con los parámetros de entrada necesarios que calcule la pendiente de una línea que pasa por 2 puntos (x_1, y_1) y (x_2, y_2) , la cual está dada por la ecuación

$$m = \frac{y_2 - y_1}{x_2 - x_1}.$$

Llamando a la anterior función, escribe un programa que calcule el valor de la pendiente de una línea que conecta dos puntos introducidos por teclado, y las pendientes de las líneas que conectan cada uno de los puntos con el origen.

Diseña después una función que reciba como parámetros la pendiente m y las coordenadas de un punto del plano, (x_0, y_0) , y escriba en pantalla la ecuación de la línea que los une con el formato:

$$y = m(x - x_0) + y_0,$$

donde x e y son simplemente caracteres que representan las coordenadas de un punto genérico. Llama a esta función desde el programa anterior para sacar los resultados en pantalla.

6. Diseña una función que calcule el punto medio de una línea que conecta 2 puntos del plano (x_1, y_1) y (x_2, y_2) según la fórmula:

$$P_m = \left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2} \right).$$

Pruébala con un programa principal que llame a la función 4 veces con los pares de puntos de la tabla adjunta.

Después realiza el ejercicio definiendo un tipo nuevo para almacenar las coordenadas de un punto del plano, `TPunto2D`, simplificando así la interfaz de la función.

(3,7)	(8,12)
(2,10)	(12,6)
(2,3)	(2,1)
(4,3)	(2,3)

7. Según la serie de MacLaurin la función seno se puede aproximar por:

$$\text{Seno}(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots,$$

donde x es el ángulo en radianes. La serie será más precisa cuanto mayor sea el número de términos. Escribe dos funciones `Seno2 (x)` y `Seno4 (x)` que calculen el seno de un ángulo con 2 y 4 términos respectivamente. Compara su precisión llamándolas desde un programa principal para calcular los senos de los ángulos $\pi/6$, $\pi/4$, $\pi/3$, $\pi/2$, π y 2π . Llama también a la función `sin(x)` de la biblioteca matemática estándar para los mismos ángulos y compara los resultados.

Capítulo 2

Decisiones

2.1. Expresiones lógicas

En este capítulo y el siguiente vamos a ver ejemplos con sentencias que permiten a un programa tomar *decisiones*, es decir, las sentencias que dotan de *inteligencia* a un programa. Son por tanto sentencias que alteran el flujo de control secuencial habitual de un programa, ya que existe la posibilidad de no ejecutar algunas sentencias. A continuación se presentan varias situaciones donde es necesario tomar decisiones y se hace patente la necesidad de nuevas sentencias.

En el siguiente ejemplo se necesita controlar un posible error aritmético ¿cuál es?

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

int main()
{
    float a, b, c;          // 2nd. degree equation coefficients
    float root1, root2;     // roots to be calculated
    float discr;            // intermediate result
    cout <<"Coefficients of 2nd. degree equation: ";
    cin >>a >>b >>c;
    discr = b*b - 4*a*c;
    root1 = (-b + sqrt(discr)) / (2*a);
    root2 = (-b - sqrt(discr)) / (2*a);
    cout <<"Root 1: " <<root1 <<endl;
    cout <<"Root 2: " <<root2 <<endl;
}
```

En el siguiente ejemplo, incompleto, se necesita ejecutar solo una sentencia o un grupo de ellas entre dos posibles.

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

int main()                incomplete
{
    int a, b;             // first rational number a/b
    int c, d;             // second rational number c/d
    int a2, b2;           // a/b reduced to a common denominator with c/d
    int c2, d2;           // c/d reduced to a common denominator with a/b
```

```

cout <<"Enter a rational number (two integers): ";
cin >>a >> b;
cout <<"Another one to be compared with the first: ";
cin >>c >> d;
// reduce to common denominator
a2 = a*d;
b2 = b*d;
c2 = c*b;
d2 = d*b;
// only one of the two following statements should be executed
...
cout <<a<< '/' <<b << " is greater than " <<c<< '/' <<d<<endl;
...
cout <<a<< '/' <<b << " is not greater than " <<c<< '/' <<d<<endl;
...
}

```

En el siguiente hay que elegir una sentencia o grupo de ellas entre varias de ellas.

```

int main()           incomplete
{
    float grade;
    cout <<"Enter a grade between 0 and 10: ";
    cin >>grade;
    ...
    cout <<" outstanding\n"; // only if 9 <= grade <=10
    ...
    cout <<" mention\n";      // only if 7 <= grade < 9
    ...
    cout <<" passed\n";        // only if 5 <= grade < 7
    ...
    cout <<" failed\n";        // only if 0 <= grade < 5
    ...
}

```

Tanto las *sentencias de decisión*, como las *sentencias de repetición* que veremos en el siguiente capítulo, se basan en una *condición de control*, que es una expresión *lógica* o *booleana* que toma uno de dos posibles valores: *verdadero* o *falso*. A ambos tipos de sentencias se las conoce también como *sentencias de control*, ya que pueden alterar el flujo secuencial del *control de ejecución*. Veamos primero cómo construir expresiones lógicas, cuyo tipo es **bool**.

Las más habituales son las *comparaciones* que se contruyen con los *operadores relacionales* primitivos de C++. El resultado se basan en el orden interno del dominio de cada tipo:

```

x < 1e-2 // float x; true si x < 1/100, false si no
a < 0    // int a;   true si a es negativo, false si no
c == ' ' // char c;  true si c es un espacio, false si no
c >= 'A' // char c;  true si c es un caracter ASCII mayor que 'A'

```

Los operadores lógicos **and**, **or**, **not**, que habitualmente se escriben en C/C++ como &&, ||, y ! respectivamente, permiten construir condiciones compuestas:

```

// float a, discr;
a!=0 && discr<0 // true si (a no es cero) Y (discr es negativo)
// char c;
!(c<'A' || c>'Z') // true si c es una letra mayuscula
c>='A' && c<='Z' // true si c es una letra mayuscula

```

Categoría	Operadores	Al mismo nivel de precedencia
Prioridad	()	izquierda a derecha
Unarios	++ -- + - !	derecha a izquierda
Multiplicativos	* / %	izquierda a derecha
Aditivos	+ -	izquierda a derecha
Entrada/salida	>> <<	izquierda a derecha
Relacionales	<= >=	izquierda a derecha
Igualdad	== !=	izquierda a derecha
lógico and	&&	izquierda a derecha
lógico or		izquierda a derecha
Operador condicional	?:	derecha a izquierda
Asignaciones	= += -= *= /= %=	derecha a izquierda

Tabla 2.1: Operadores agrupados por precedencia decreciente.

Cuidado con la precedencia de los operadores lógicos. La tabla 2.1 incluye la precedencia de estos operadores.


El resultado de una expresión lógica también se puede almacenar en una variable (de tipo **bool**).

```
// float x, y; int a, b; bool ok;
ok = 23.0*x <= sqrt(y*y) || a>b && ok
ok = 23.0*x + sqrt(y*y) || a>b && ok
```

En el segundo de los ejemplos anteriores hay un error de tipo semántico ¿cuál?. En el primero, sin errores, el orden de evaluación podría ser:

- (1) 23.0*x
- (2) y*y
- (3) sqrt()
- (4) <=
- (5) a>b
- (6) &&
- (7) ||
- (8) =

A estas *sentencias de decisión*, además de las *sentencias de repetición* que veremos en el siguiente capítulo, se las denomina *sentencias de control* porque se basan en una *condición de control*. El funcionamiento básico es:

 Si en tiempo de ejecución la condición de control es *verdadera*, hay ejecución de la sentencia (o grupo de ellas), si no, no se ejecuta (o se ejecuta otro grupo diferente).

Funciones lógicas

Las condiciones más elaboradas conviene *encapsularlas* en funciones que devuelven un resultado de tipo **bool**. Las llamadas a estas funciones también pueden hacer el papel de condición de control. El siguiente ejemplo muestra primero la definición de la función y en el principal aparece la llamada, la cual valdrá **true** o **false**.

```

#include <iostream>
using namespace std;

bool IsEven(int a)
{
    return a%2 == 0;
}

int main()           incomplete
{
    int number;
    bool it_is_even;
    ...
    it_is_even = IsEven(number);
    ...
}

```

Evaluación en cortocircuito

La evaluación de las expresiones lógicas compuestas se evalúan por orden de izquierda a derecha, y termina prematuramente cuando se puede determinar el resultado final sin necesidad de evaluar el resto de la expresión. Esto es útil en expresiones que contienen varias subexpresiones enlazadas con el operador **and** o con el operador **or**. En los ejemplos siguientes se muestran en color más claro las subexpresiones que no llegan a evaluarse por no ser necesario. La primera termina valiendo `false`, y la segunda `true`.

```

// float a=1, b=-1, c;
a>=0 && b>=0 && b*b>4*a*c && sqrt(b*b-4*a*c)>0
// char c1='e', c2='B', c3='5';
c1<'a' || c1>'z' || c2<'Z' || c2>'A' || c1>c2 || c2>c3

```

Ejercicios

1. Escribe funciones lógicas que devuelvan `true` cuando...
 - a) un número sea múltiplo de 5.
 - b) dos números reales se diferencien en menos del valor 10^{-12} .
 - c) un carácter sea una letra, ya sea minúscula o mayúscula.
 - d) un carácter sea alfanumérico.
 - e) un carácter sea una vocal, ya sea minúscula o mayúscula.
 - f) un carácter sea un dígito hexadecimal.
 - g) un carácter sea imprimible, es decir, que no sea un blanco (espacio, tabulador, nueva línea o retorno de carro) ni un carácter de control (los 32 primeros caracteres ASCII).
2. Escribe una función lógica que devuelva `true` cuando un número entero sea par, y `false` cuando sea impar.
3. Escribe una función lógica que devuelva `true` cuando 2 números reales no difieran en más de una tolerancia dada arbitrariamente pequeña.

2.2. Bloques y sentencias compuestas

En C++ las sentencias se pueden agrupar entre llaves { } sin que ello altere la ejecución secuencial del programa.

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

int main()
{
    // definitions
    const float Pi=3.141592654;
    float radius, area;
    {
        cout << " Radius: ";
        cin >> radius;
    }
    {
        area = Pi*radius*radius;
        cout << " Area: " << area << endl;
    }
}
```

Los bloques se pueden utilizar en cualquier sitio de un programa C++, pero su mayor utilidad es para agrupar las sentencias que forman las sentencias *de control* compuestas, las cuales sí alteran la ejecución secuencial habitual de un programa.

2.3. Sentencias de selección

Hay tres tipos de sentencias de selección:

- a) La *selección simple* ejecuta opcionalmente un grupo de sentencias.
- b) La *selección doble* ejecuta un grupo de sentencias de dos posibles.
- c) La *selección múltiple* ejecuta un grupo de sentencias entre varias posibles seleccionada según el valor que tome una única expresión tipo ordinal.

C++ proporciona tres mecanismos de selección: (1) la sentencia **if**, la más general; (2) la sentencia **switch**; y (3) el operador condicional **?:**.

2.3.1. La sentencia **if**

Ejemplo de selección doble.

```
#include <iostream>
#include <cmath> // sqrt
using namespace std;


int main()
{
    float a, b, c; // 2nd. degree equation coefficients
    float root1, root2; // roots to be calculated
    float discr; // intermediate result
```

```

cout <<"Coefficients of 2nd. degree equation: ";
cin >>a >>b >>c;
discr = b*b - 4*a*c;
if (discr<0) {
    // this is the true part
    cout <<"Imaginary roots\n";
}
else {
    // this is the false part
    root1 = (-b + sqrt(discr)) / (2*a);
    root2 = (-b - sqrt(discr)) / (2*a);
    cout <<"Root 1: " <<root1 <<endl;
    cout <<"Root 2: " <<root2 <<endl;
}
}

```

El ejemplo anterior tiene todavía una deficiencia ¿sabrías decir cuál es? Si no es así, ten en cuenta el siguiente consejo.

 Las posibles divisiones por cero deben comprobarse *antes* de llevarse a cabo (con ayuda de sentencias de decisión).

La parte **else** de la sentencia **if** es opcional, dando lugar a selecciones simples, como se muestra en el siguiente ejemplo.


```

#include <iostream>
using namespace std;

int main()
{
    int number, greatest;

    cout <<"Enter four integer numbers: ";
    cin >>greatest;    // the first is the greatest until now
    cin >>number;       // read the next
    if (number > greatest) {
        greatest = number;    // now the second is the greatest
    }
    cin >>number;       // read the next losing previous number
    if (number > greatest) {
        greatest = number;    // now the third is the greatest
    }
    cin >>number;       // read the next losing previous number
    if (number > greatest) {
        greatest = number;    // the last one is the greatest
    }
    cout <<"The greatest value is " <<greatest <<endl;
}

```

 Observa que las sentencias de selección no cambian los contenidos de la memoria, aunque las sentencias que agrupan, *anidadas* en bloques, sí podrían hacerlo.

2.3.2. Sentencias **if** anidadas

No hay restricciones en las sentencias que pueden ir dentro de un bloque, por tanto se pueden *anidar* sentencias **if** dentro de sentencias **if**.

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

int main()
{
    int number1, number2, number3, number4;

    // read four integers
    cout <<"Enter four integer numbers: ";
    cin >>number1 >>number2 >>number3 >>number4;

    // make comparisons until finding out which is the greatest
    if (number1 > number2) {
        if (number1 > number3) {
            if (number1 > number4) {
                cout <<"The greatest is the first number\n";
            }
            else {
                cout <<"The greatest is the fourth number\n";
            }
        }
        else {
            if (number3 > number4) {
                cout <<"The greatest is the third number\n";
            }
            else {
                cout <<"The greatest is the fourth number\n";
            }
        }
    }
    else {
        if (number2 > number3) {
            if (number2 > number4) {
                cout <<"The greatest is the second number\n";
            }
            else {
                cout <<"The greatest is the fourth number\n";
            }
        }
        else {
            if (number3 > number4) {
                cout <<"The greatest is the third number\n";
            }
            else {
                cout <<"The greatest is the fourth number\n";
            }
        }
    }
}
```

👁 Presta atención a la *indentación*, que consiste en sangrar el código fuente varios espacios hacia el interior de forma consistente a lo largo de todo un programa para mejorar su legibilidad.

2.3.3. Sentencias **if** múltiples

Las selecciones múltiples se pueden construir anidando sentencias **if** en las partes **else**.

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

int main()
{
    int number;
    cout <<"Enter an integer number: ";
    cin >>number;
    if (number < 10) {
        cout <<"The number has only one digit\n";
    }
    else {
        if (number < 100) {
            cout <<"The number has two digits\n";
        }
        else {
            if (number < 1000) {
                cout <<"The number has three digits\n";
            }
            else {
                if (number < 10000) {
                    cout <<"The number has four digits\n";
                }
                else {
                    cout <<"The number has more than four digits\n";
                }
            }
        }
    }
}
```

No obstante, es más habitual escribir los anidamientos sin escribir las llaves de las partes **else**.

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

int main()
{
    int number;
    cout <<"Enter an integer number: ";
    cin >>number;
    if (number < 10) {
        cout <<"The number has only one digit\n";
    }
    else if (number < 100) {
        cout <<"The number has two digits\n";
    }
    else if (number < 1000) {
```



```

    cout <<"The number has three digits\n";
}
else if (number < 10000) {
    cout <<"The number has four digits\n";
}
else {
    cout <<"The number has more than four digits\n";
}
}

```

Ejercicios

1. Escribe un programa que responda cuál es el mayor y el menor de una serie de cinco números leídos de teclado.
2. Escribe un programa que lea 5 números enteros positivos y responda cuántos son pares.
3. Suponga que el nivel de grado de los estudiantes que no han terminado la universidad se determina con base a la siguiente tabla:

Número de créditos obtenidos	Grado
Menor que 32	Primer año
32 a 63	Segundo año
64 a 95	Tercer año
96 ó más	Último año

Utilizando esta información, escriba un programa que acepte el número de créditos que ha acumulado un estudiante y determine en qué año del grado se encuentra.

4. Escribe una función lógica que devuelva `true` para años bisiestos y `false` para los que no lo sean. Un año es bisiesto si cumple una de las dos siguientes condiciones:
 - El año es divisible por 400.
 - El año es divisible por 4, pero no por 100.

Por ejemplo, el año 1996 es bisiesto, el 2000 y el 2004 también, pero el año 1900, 1995 y el 2100 no lo son.

Pruébala con un programa que determine si un año leído de teclado es bisiesto.

5. Una empresa maneja códigos numéricos de 4 dígitos con el formato POOC siguiente:
 - El primero P es el *código de una provincia*.
 - El segundo y tercero OO el *código de operación*.
 - El último C el *dígito de control*, el cual debe coincidir con el resto de la división entera de OO por P.

Utilizando el operador módulo % y cociente / de la división entera de C++, escriba un programa que lea de teclado un número de cuatro dígitos, y después extraiga e imprima en pantalla las tres partes separadas. Por ejemplo, para la entrada 1234, la salida sería:

```

Codigo de provincia: 1
Codigo de operacion: 23
Digito de control: 4

```

El programa debe dar el mensaje de error adecuado cuando el número leído no tenga cuatro dígitos, o cuando el dígito de control C no coincide con el resto de dividir el código de operación por el código de provincia.

2.4. Ejemplos con funciones

El siguiente ejemplo utiliza una función lógica que encapsula las condiciones para determinar si un alumno cumple unos requisitos mínimos para calcular su nota final.

```
#include <iostream>
#include <iomanip>    // output format: fixed, setw, setprecision
using namespace std;

/*
 * Find out whether minimum grades are met.
 */
bool Minimums(float test1, float test2, float test3,
              float prac1, float prac2, float prac3, float exam)
{
    const float MinGrade = 1.0;
    const float MinExam = 2.5;
    bool ok;
    if (exam >= MinExam) {
        if (test1 >= MinGrade && test2 >= MinGrade && test3 >= MinGrade &&
            prac1 >= MinGrade && prac2 >= MinGrade && prac3 >= MinGrade) {
            ok = true;
        }
        else {
            ok = false;
        }
    }
    else {
        ok = false;
    }
    return ok;
}

int main()
{
    const float ExamWeigh = 0.6;
    const float TestWeigh = 0.2;
    const float PracWeigh = 0.2;
    float test1, test2, test3, tests;
    float practice1, practice2, practice3, practice;
    float exam, final;
    cout << "3 tests grades (0-10): ";
    cin >> test1 >> test2 >> test3;
    cout << "3 practices grades (0-10): ";
    cin >> practice1 >> practice2 >> practice3;
    cout << "Final exam grade (0-10): ";
    cin >> exam;
    tests = (test1 + test2 + test3) / 3;
    practice = (practice1 + practice2 + practice3) / 3;
    if (Minimums(test1, test2, test3, practice1, practice2, practice3, exam)) {
        final = exam*ExamWeigh + tests*TestWeigh + practice*PracWeigh;
    }
    else {
        final = exam*ExamWeigh;
    }
}
```

```

}
tests *= TestWeigh;
practice *= PracWeigh;
exam *= ExamWeigh;
cout <<fixed; // fixed point output format for numbers
cout <<setw(23) <<"-----\n";
cout <<setw(18) <<"Tests weighed: " <<setprecision(2) <<tests <<endl;
cout <<setw(18) <<"Practice weighed: " <<setprecision(2) <<practice <<endl;
cout <<setw(18) <<"Exam weighed: " <<setprecision(2) <<exam <<endl;
cout <<setw(18) <<"Final grade: " <<final <<endl;
}

```

El siguiente ejemplo clacula el salario total de un empleado de una empresa. El salario depende de su formación académica, y la antigüedad en la empresa. Además, se pagan las horas extras.

```

#include <iostream>
using namespace std;

const float BasicSalary = 1000;
const float MediumSalary = 1500;
const float HighSalary = 2000;
const float FiveYearsService = 100;
const float OvertimeHour = 25;

float Salary(int years, int level)
{
    float payment;
    if (years>2 && level==3 || years>10 && level==2) {
        payment = HighSalary;
    }
    else if (level==3 || years>2 && level==2 || years>10 && level==1) {
        payment = MediumSalary;
    }
    else {
        payment = BasicSalary;
    }
    return payment;
}

float OvertimeBonus(int hours)
{
    // every overtime hour is 25 euros paid
    return hours * OvertimeHour;
}

int main()
{
    int years_of_service, academic_level, overtime_hours;
    float total_wage;

    // calculate basic salary
    cout <<"Enter years of service: ";
    cin >>years_of_service;
    cout <<"Academic level (1:primary; 2:secondary; 3:graduate): ";
    cin >>academic_level;

```

```

if (academic_level>3) {
    // wrong level, assume basic
    academic_level = 1;
}
total_wage = Salary(years_of_service, academic_level);
// add overtime hours
cout <<"Overtime hours: ";
cin >>overtime_hours;
total_wage += OvertimeBonus(overtime_hours);
// final result
cout <<"Total salary is " <<total_wage <<endl;
}

```

2.4.1. La sentencia **switch**

La sentencia **switch** simplifica algunas selecciones múltiples, aquéllas en las que las comparaciones dependen del valor que toma una única expresión. Como restricción, esa expresión debe ser de tipo ordinal, es decir, un número entero o un carácter individual.

```

#include <iostream>
using namespace std;

int main()
{
    unsigned day;
    cout <<"Number of the day of the week: ";
    cin >>day;
    switch (day) {
        case 1: cout <<"Monday\n"; break;
        case 2: cout <<"Tuesday\n"; break;
        case 3: cout <<"Wednesday\n"; break;
        case 4: cout <<"Thursday\n"; break;
        case 5: cout <<"Friday\n"; break;
        case 6: cout <<"Saturday\n"; break;
        case 7: cout <<"Sunday\n"; break;
        default: cout <<"Number should be between 1 and 7\n";
    }
}

```

Observa el uso de la sentencia **break** al final de cada caso. Sin ella, cuando uno de los casos es verdadero, se ejecutarían secuencialmente tanto las sentencias de este caso como las del resto de casos. No obstante, esta característica de C++ puede resultar muy útil, como se aprecia en el siguiente ejemplo.

```

#include <iostream>
#include <cstdlib> // toupper()
using namespace std;

int main()
{
    char letter;
    cout <<"Letter? ";
    cin >>letter;
    toupper(letter); // convert to uppercase
    switch (letter) {

```

```

    case 'A': case 'E': case 'I': case 'O': case 'U':
    cout << "A vowel\n";
    break;
    case 'B': case 'C': case 'D': case 'F': case 'G': case 'H': case 'J':
    case 'K': case 'L': case 'M': case 'N': case 'P': case 'Q': case 'S':
    case 'T': case 'V': case 'W': case 'X': case 'Y': case 'Z':
    cout << "A consonant\n";
    break;
    default:
    cout << "Not a letter\n";
}
}

```



Recuerda que sólo expresiones de tipo *ordinal* pueden utilizarse como la expresión de selección de una sentencia `switch`.

2.4.2. El operador condicional

El operador condicional permite hacer asignaciones abreviadas basadas en una condición evitando el uso de la sentencia `if`. Trata de averiguar su funcionamiento con el siguiente ejemplo que calcula la resistencia equivalente de dos resistores que pueden estar en serie o en paralelo.

```

#include <iostream>
using namespace std;

main()
{
    float r1, r2, r; char configuration;
    cout << "Enter two resistor values: ";
    cin >> r1 >> r2;
    cout << "Series or parallel (s/p): ";
    cin >> configuration;
    // any other answer different from 's' or 'p' will be considered 's'
    r = configuration=='p' || configuration=='P' ? r1*r2/(r1+r2) : r1+r2;
    cout << "Equivalent resistance: " << r << endl;
}

```

Como mostraba la tabla 2.1, este operador tiene una precedencia muy baja, lo que evita a menudo el uso de paréntesis.

Ejercicios de repaso

1. Escribe una función que halle el determinante de una matriz 2×2 :

$$D = \begin{vmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{vmatrix} = m_{11}m_{22} - m_{12}m_{21}$$

Con ayuda de esta función, escribe un programa que acepte como entrada los coeficientes a , b , c , d , e , f , de un sistema de ecuaciones lineales 2×2 ,

$$\begin{cases} ax + by = c \\ dx + ey = f, \end{cases}$$

y después calcule sus soluciones con la *regla de Cramer*:

$$x = \frac{\begin{vmatrix} c & b \\ f & e \end{vmatrix}}{\begin{vmatrix} a & b \\ d & e \end{vmatrix}} \quad y = \frac{\begin{vmatrix} a & c \\ d & f \end{vmatrix}}{\begin{vmatrix} a & b \\ d & e \end{vmatrix}}$$

Prueba el programa con dos ecuaciones linealmente independientes, es decir, que cumplan que $ae \neq bd$. Después pruébalo con dos ecuaciones linealmente dependientes ($ae = bd$), y modifica tu programa para dar al usuario el mensaje de error adecuado cuando no haya solución.

2. Escribe una función que calcule la resistencia que ofrece un conductor cilíndrico a la corriente eléctrica de acuerdo con la fórmula

$$R = \rho \frac{l}{A},$$

donde ρ es su *resistividad*, l la longitud del conductor, y A el área de su sección transversal. Con ayuda de esta función, escribe un programa que lea de teclado los datos de un conductor cilíndrico fabricado con uno de los 4 materiales que se muestran en la tabla adjunta. Los datos a leer serán su letra inicial (c, a, p, o h); su resistividad; su longitud; y su sección transversal. Utiliza una sentencia `switch` para discriminar los 4 posibles casos según el material.

Material	ρ
cobre	$16,8 \cdot 10^{-9}$
aluminio	$25,4 \cdot 10^{-9}$
plata	$15,9 \cdot 10^{-9}$
hierro	$97,1 \cdot 10^{-9}$

3. Codifica un programa que se comporte como una calculadora simple que realice las 5 operaciones básicas con números enteros. El usuario primero elegirá una de las 5 posibles operaciones tecleando el carácter correspondiente +, -, *, /, o %, y después introducirá los 2 operandos. Una posible ejecución sería:

```
Operacion (+ - * / \%): /
Operando 1: 5
Operando 2: 2
Resultado: 2
```

4. ¿Cómo escribirías con el operador condicional una asignación a una variable `h`, el valor de otra variable `a` sólo si se cumple la condición (`a > b`)?
5. Teniendo en cuenta que la asociatividad del operador condicional es de derecha a izquierda, reescribe la siguiente asignación condicional anidada con una sentencia `if` equivalente.

```
h = (a > b) ? (b < c) ? a : b : b;
```

6. Define una función de tipo carácter, `LetraDNI(...)`, que devuelva la letra de control de un DNI español. La letra correspondiente se obtiene con el resto de la división entera del número del DNI por 23, el cual es el número de orden (del 0 a 22) de la siguiente lista de letras: T, R, W, A, G, M, Y, U, P, D, X, B, N, J, Z, S, Q, V, H, L, C, K, y E. Por ejemplo, al dividir el número 12345678 por 23, se obtiene el resto 14, al cual le corresponde la letra Z, por lo que el DNI completo sería 12345678Z.

Realiza un programa que lea por teclado el número de un DNI y responda con el DNI completo correspondiente.

7. Escribe un programa que resuelva completamente una ecuación de segundo grado

$$ax^2 + bx + c = 0,$$

incluyendo las soluciones complejas si fuera el caso. Los tres coeficientes a , b y c se leerán de teclado. Por ejemplo, para la ecuación $x^2 - 3x + 2 = 0$, la entrada y salida serían:

Coeficientes: 1 -3 2
Soluciones reales: 2, 1

Si el coeficiente cuadrático a fuera 0, debe calcularse una única solución real. Y si tanto el coeficiente cuadrático como el lineal b fueran 0, se debe responder con el mensaje adecuado.

En el caso de raíces complejas, hay que calcular por separado las partes real e imaginaria, dando la solución en el formato $a + i b$. Por ejemplo, para $x^2 - 2x + 5 = 0$, la ejecución sería:

Coeficientes: 1 -2 5
Soluciones complejas conjugadas: 1 +i 2, 1 -i 2

Si alguna de las partes, real o imaginaria, son cero, no debería mostrarse esa parte:

Coeficientes: 1 0 1
Soluciones complejas conjugadas: +i 1, -i 1

Capítulo 3

Bucles

La *potencia* real de un ordenador se manifiesta en su capacidad para repetir acciones rápidamente, aunque sean en general acciones muy simples. Las sentencias que permiten expresar la repetición de acciones son los llamados *bucles*, y, al igual que las sentencias de selección, son un tipo de sentencias de control, las cuales ya sabemos que se basan en una *condición de control*. En este caso, la condición de control es la que determina cuándo se detiene la ejecución de las acciones a repetir, las cuales constituyen el *cuerpo* del bucle. La condición de control se puede evaluar después o antes de cada repetición o *iteración*. Los bucles que evalúan la condición de control *antes* de cada iteración son los bucles *pre-test*, y los que lo hacen *después* de cada iteración son los bucles *post-test*. En C++ disponemos de dos tipos de bucles pre-test, el **while** y el **for**. Además, como bucle post-test, C++ también dispone del bucle **do-while**.

Hay que tener en cuenta que cualquiera de tres bucles puede usarse para repetir cualquier grupo de acciones, pero cada uno es más apropiado para algunos tipos de repeticiones, evitando alguna sentencia de selección adicional.

3.1. El bucle genérico **while**

El bucle **while** es el más flexible de C++. Es un bucle pre-test y su condición de control se evalúa antes de cada iteración. El cuerpo de sentencias a repetir se escribe dentro de un bloque a continuación de la condición de control. Mientras la condición se evalúe a **true**, el cuerpo se ejecutará completamente. El siguiente ejemplo suma números mientras el usuario introduzca números distintos del cero. Cuando introduzca un cero, el bucle se detiene y el programa continúa con la sentencia escrita a continuación del bucle.

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

int main()
{
    int number, total=0;
    cout <<"A number to add (0 to finish): ";
    cin >>number;
    total = number;
    while (number!=0) {
        cout <<"Another one to add (0 finish): ";
        cin >>number;
        total += number;
    }
    cout <<"Total is " <<total <<endl;
```

```
}
```

- 👁 Hay que asegurarse de que la condición de control tomará en algún momento el valor `false`, si no, sería un bucle *infinito* y *colgaría* el programa.

Los bucles **while** suelen ser bucles que no pueden calcular el número de repeticiones en el momento justo en que empiezan. Se dicen que son bucles *no-deterministas*, y a veces *controlados por centinela*. En el ejemplo anterior el centinela era el valor cero.

No obstante, la flexibilidad de la sentencia **while** también permite utilizarse para bucles *deterministas*, que son aquéllos con un número de repeticiones que se puede calcular al empezar el bucle. También se dice que son bucles *controlados por contador*. La naturaleza del problema a resolver es el que determina cuándo un bucle es determinista y cuándo no. A veces, como el ejemplo anterior, se puede utilizar ambas alternativas modificando ligeramente el código.

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

int main()
{
    const int HowMany=5;
    int number, counter=0, total=0;
    cout <<"Enter "<<HowMany<<" integer numbers to add: ";
    cin >>number;
    counter++;
    total += number;
    while (counter < HowMany) {
        cout <<"Another one ("<<HowMany-counter<<" left): ";
        cin >>number;
        counter++;
        total += number;
    }
    cout <<"Total is " <<total <<endl;
}
```

Dentro de un bucle puede ir cualquier tipo de sentencia, por ejemplo sentencias de decisión, tal como muestra el siguiente ejemplo, que selecciona los números pares para ser sumados.

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

int main()
{
    int number, total=0;
    cout <<"Even numbers to add (0 to finish): ";
    cin >>number;
    total = number;
    while (number!=0) {
        cout <<"Another one to add (0 finish): ";
        cin >>number;
        if (number%2 == 0) {    // never trust on the user
            total += number;
        }
    }
    cout <<"Total of even numbers is " <<total <<endl;
}
```

```
}
```

Ejercicios

1. Escribe un programa que lea números enteros de teclado hasta introducir el valor 0 y muestre en pantalla el mayor y el menor de todos ellos.
2. Escribe un programa que lea números enteros de teclado hasta introducir el valor 0 y muestre en pantalla la media aritmética de los números pares y la media aritmética de los números impares de forma separada.
3. Escribe un programa que lea un texto de teclado, carácter a carácter, hasta introducir un punto. El programa debe llevar la cuenta del número de comas, letras, vocales y el número total de caracteres a la entrada. Los resultados se imprimirán en pantalla al final.
4. Utilizando sumas y restas sucesivas, escribe un programa que realice la división entera entre dos números enteros, sacando en pantalla tanto el cociente como el resto.

3.2. El bucle controlado por contador **for**

Un *contador* no es más una variable que incrementa o decrementa su valor, muy frecuentemente dentro un bucle. La condición de control de los bucles **for** es una expresión que vigila el valor de un contador hasta que llega a un valor final, momento en el que termina el bucle. Esa expresión suele ser un simple contador que se llama *variable de control* del bucle. Aunque un bucle **while** puede estar controlado por un contador, el bucle **for** dispone de una sintaxis más apropiada para ello.

La primera línea de un bucle **for** contiene la inicialización de la variable de control, que se realiza una sola vez justo antes de que comience el bucle; la propia condición de control, que se evalúa justo antes de cada iteración y permite iterar mientras se evalúe a **true**; y la sentencia de actualización de la variable de control, que se realiza justo después de cada iteración.

Observa la sintaxis en el siguiente ejemplo que cuenta el número de apuestas diferentes dentro de una quiniela de fútbol española.

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


int main()
{
    const unsigned NumMatches=14;
    unsigned i, n1s=0, nXs=0, n2s=0, invalids=0;
    char result;    // to hold '1', 'X', '2'

    cout <<"Bet on "<<NumMatches<<" football matches (1/X/2)\n";
    for (i=1; i<=NumMatches; i++) {
        cout <<"Match number " <<i <<": ";
        cin >> result;
        switch (result) {
            case '1': n1s++; break;
            case 'x':
            case 'X': nXs++; break;
            case '2': n2s++; break;
            default: invalids++;
        }
    }
}
```

```

    }
}
if (n1s>0) { cout <<n1s <<" home win(s)\n"; }
if (nXs>0) { cout <<nXs <<" tie(s)\n"; }
if (n2s>0) { cout <<n2s <<" visitor win(s)\n"; }
if (invalids>0) { cout <<invalids <<" invalid result(s)\n"; }
}

```

 La variable de control puede utilizarse dentro del cuerpo del bucle cuantas veces se desee, pero no debe ser modificada dentro del cuerpo del bucle.

En los bucles es muy útil aprovechar cálculos intermedios de iteraciones anteriores, tal como ilustra el siguiente ejemplo que calcula la serie geométrica

$$\sum_{k=0}^{\infty} r^k = 1 + r + r^2 + r^3 + r^4 \dots = \frac{1}{1-r}$$


```

#include <iostream>
using namespace std;


int main()
{
    float series=0, term=1, ratio;
    unsigned n;
    cout <<"Number of geometric terms: ";
    cin >>n;
    cout <<"Ratio: ";
    cin >>ratio;
    for (unsigned i=0; i<n; i++) {
        series += term; // update current value
        term *= ratio; // prepare next term
    }
    cout <<"Approximated value: " <<series <<endl;
}

```

Nótese que se podría haber llamado a la función `pow()` de la biblioteca matemática estándar, pero esto hubiera aumentado el coste computacional en cada iteración.

 Desde el momento en que se usan bucles, cobra importancia el coste computacional, por lo que hay que prestar atención al número de operaciones que se realizan en cada iteración. No obstante, es preferible un código claro y bien estructurado que un código “demasiado optimizado”.

Otra observación del ejemplo anterior es la definición de la variable de control, que se puede hacer en la propia sentencia **for**, en la parte de su inicialización, pero hay que tener presente lo siguiente:

 Fuera de la sentencia **for**, la variable de control no existe si se define dentro de ella.

Ejercicios

1. Escribe un programa que acepte 2 números enteros y saque en pantalla todos los números intermedios, incluidos ellos mismos. Después, modifica el programa para mostrar sólo los números impares.

- Utilizando bucles, escribe un programa que saque en pantalla la tabla de multiplicación de 1 al 10 de cualquier número entero que elija el usuario. Usa la secuencia de escape del tabulador `'\t'` para alinear los números en columnas.
- Escribe un programa que lea n números reales de teclado con un bucle `for`, y saque en pantalla el mayor y el menor de todos ellos.
- Escribe un programa que, dados los coeficientes de un polinomio de grado n ,

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0,$$

y un punto x , evalúe el polinomio en ese punto.

- El valor aproximado de π se puede obtener con el desarrollo en serie:

$$4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right)$$

Utilizando esta fórmula, escriba un programa que calcule el valor de π utilizando 5, 100 y 100000 términos de la serie.

- Escribe un programa que imprima en pantalla los n primeros números de Fibonacci. Es esta serie, partiendo de los 2 primeros números que valen 1, cada número es suma de los 2 anteriores: 1, 1, 2, 3, 5, 8, 13... El número n se leerá previamente por teclado.

3.3. El bucle post-test `do-while`

Cuando se requiere que al menos se ejecute la primera iteración de un bucle, el bucle `do-while` debe ser considerado, ya que realiza una iteración antes de evaluar la condición de control. Después de la primera iteración, se realizarán iteraciones mientras la condición de control se evalúe a `true`.

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

int main()
{
    int number, total=0;
    do {
        cout <<"A number to add (0 to finish): ";
        cin >>number;
        total += number;
    } while (number!=0);
    cout <<"Total is " <<total <<endl;
}
```

El bucle `do-while` es habitual en la entrada de datos controlada por centinela.

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

int main()
{
    float price, amount, total_amount;
    unsigned n=0, how_many;
    do {
        n++;
    } while (n <= how_many);
}
```

faulty

```

    cout <<"Price in euros per item " <<n <<" (0 to exit): ";
    cin >>price;
    cout <<"Number of items: ";
    cin >>how_many;
    total_amount += price*how_many;
} while (price != 0);
cout <<"Total amount: " <<total_amount <<" euros\n";
}

```

¿Notas alguna posible deficiencia en el ejemplo? Es fácil de solucionar, aunque requiere un sentencia **if** adicional.

Ejercicios

1. Modifica el programa anterior de la quiniela para forzar al usuario a teclear solo uno de los tres posibles signos.
2. Escribe un programa que simule el comportamiento de una calculadora simple para realizar repetitivamente las operaciones aritméticas básicas (+, -, *, /, %) sobre dos números reales hasta que el usuario introduzca la letra f. Un ejemplo de ejecución podría ser:

```

Operacion (+ - * / f): *
Primer operando: 24
Segundo operando: 3
Resultado: 72

Operacion (+ - * / f): /
Primer operando: 1
Segundo operando: 2
Resultado: 0.5

Operacion (+ - * / f): +
Primer operando: 2.4
Segundo operando: 0.3
Resultado: 2.7

Operacion (+ - * / % f): f

```

3. Utilizando una función lógica `EsBisiesto(..)`, escribe un programa que fuerce al usuario a introducir una fecha *gregoriana* correcta (desde el año 1582), teniendo en cuenta los días de febrero en los años bisiestos.

3.4. Bucles anidados

Los bucles también pueden anidarse dentro de otros bucles, de forma que una iteración del bucle externo ejecuta completamente todas las iteraciones del bucle interno, el cual volverá a ejecutarse completamente en la siguiente iteración del bucle externo. Su necesidad surge naturalmente en problemas bidimensionales, tal como muestra el siguiente ejemplo.

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

```
int main()
{
    unsigned i, j, width, height;
    cout <<"Width and height of rectangle: ";
    cin >>width >>height;
    for (i=0; i<height; i++) {
        for (j=0; j<width; j++) {
            cout <<'*';
        }
        cout <<endl;
    }
}
```

Si el usuario introduce una anchura w y una altura h , ¿cuántas veces se ejecuta la sentencia que pinta un solo asterisco?

Los anidamientos pueden ser muy flexibles cuando las acciones del bucle interno utilizan valores intermedios calculados en el bucle exterior. El siguiente ejemplo aprovecha esa flexibilidad utilizando el valor de la variable de control del bucle externo en la condición de control del bucle interno.

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

```
int main()
{
    unsigned i, j, height;
    cout <<"Height of triangle: ";
    cin >>height;
    for (i=1; i<=height; i++) {
        for (j=1; j<=i; j++) {
            cout <<'*';
        }
        cout <<endl;
    }
}
```

Ahora, si el usuario introduce una anchura w y una altura h , ¿cuántas veces se ejecuta la sentencia que pinta un solo asterisco?

Salida para un tamaño 5×8 :

```
*****
*****
*****
*****
*****
```

Output for height 5:

```
*
**
***
****
*****
```

Ejercicios

1. Escribe un programa que muestre en pantalla los subíndices del triángulo inferior izquierda de un matriz bidimensional, sin incluir la diagonal principal, tal como se muestra a la derecha para una matriz 5×5 .
2. Escribe un programa que muestre en pantalla la letra C con una longitud característica. Por ejemplo, para la longitud 5, el programa mostrará la figura ajunta de la derecha.

```
(2,1)
(3,1) (3,2)
(4,1) (4,2) (4,3)
(5,1) (5,2) (5,3) (5,4)
```

```
CCCCC
C
C
C
CCCCC
```

3. Escribe un programa que muestre en pantalla una montaña de números de una altura dada. La figura de la derecha muestra la montaña para una altura igual a 5. Ayudándote del operador módulo %, amplía el programa para alinear también montañas de una altura mayor que 9.

```

1
121
12321
1234321
123454312

```

Ejercicios de repaso

1. Escriba un programa que invierta aritméticamente los dígitos de un número positivo entero con número variable de cifras. Por ejemplo, si se introduce el número 8735, debe mostrar el número 5378; y si se introduce el 123456, se mostrará 654321.
2. Con ayuda de una función que halle el *máximo común divisor* de 2 números, escribe un programa que lea de teclado el numerador y denominador de un número racional y escriba en pantalla el mismo número racional su forma conónica. Puedes utilizar un tipo definido `TRacional` para agrupar el numerador y denominador en un solo registro.
3. Escribe una función que halle el *mínimo común múltiplo* de 2 números. Un programa principal leerá de teclado el numerador y denominador de dos números racionales para compararlos, para lo cual utilizará la función anterior para reducirlos a común denominador. Puedes utilizar un tipo definido `TRacional` para agrupar el numerador y denominador en un solo registro.
4. Escribe un programa que cuente el número de palabras que hay en un texto que se introduce por teclado y acaba en un punto. Se considerará palabra cualquier secuencia de letras y dígitos, y podrá aparecer cualquier otro carácter imprimible entre ellas.
5. Realiza un programa que lea valores de resistencias electrónicas (en ohmios, Ω) conectadas bien en paralelo

$$\frac{1}{1/R_1 + 1/R_2 + 1/R_3 + \dots}$$

o bien en serie

$$R_1 + R_2 + R_3 + \dots$$

El usuario elegirá la configuración escribiendo la letra 'p' o la letra 's'; después aceptará el número de resistores, n ; y después de hacer los cálculos necesarios responderá con el valor de la resistencia equivalente.

6. La biblioteca estándar `cstdlib` proporciona un generador de números enteros aleatorios, `rand()`, generados en el rango $[0, \text{RAND_MAX}]$ (`RAND_MAX` es un valor grande constante definido en la propia biblioteca). El generador no es más que una lista de números que, para sea realmente aleatoria, debe empezar a su vez en una posición aleatoria o *semilla*, porque si no, siempre generaría los mismos primeros números. Para ello, el generador suele inicializarse con una sentencia que dependa de un valor aleatorio, como es el momento en el que se ejecuta el programa:

```
srand(time(NULL)); // #include <ctime>
```

Usando las sentencias comentadas y un bucle `do-while` escribe un programa que genere un número aleatorio entre 1 y 100 (utiliza el operador módulo % para asegurarte de que está en ese rango). A continuación el usuario, sin ver cuál es el número oculto, debe introducir números hasta que acierte. Al final el programa debe indicar el número de intentos realizado.

7. Sin usar funciones para calcular potencias o factoriales, ya sean estándares o definidas por el programador, escribe las siguientes funciones para calcular su valor en un punto x según la serie de MacLaurin dada. El número de términos de la serie vendrá determinado por la diferencia entre los dos últimos términos, la cual no excederá de una tolerancia dada, por ejemplo 10^{-12} . Comprueba antes que el punto x se encuentra en el intervalo adecuado cuando éste no sea infinito. (Pista: utiliza en cada iteración del bucle el término calculado en la iteración previa.)

$$\begin{aligned}
\frac{1}{1-x} &= 1 + x + x^2 + x^3 + x^4 + \dots \text{for } -1 < x < 1 \\
\frac{1}{1+x} &= 1 - x + x^2 - x^3 + x^4 - \dots \text{for } -1 < x < 1 \\
\ln(1+x) &= x - x^2/2 + x^3/3 - x^4/4 + \dots \text{for } -1 < x \leq 1 \\
\arctan(x) &= x - x^3/3 + x^5/5 - x^7/7 + \dots \text{for } -1 < x < 1 \\
\exp(x) &= 1 + x + x^2/2! + x^3/3! + x^4/4! + \dots \text{for } -\infty < x < \infty \\
\cos(x) &= 1 - x^2/2! + x^4/4! - x^6/6! + \dots \text{for } -\infty < x < \infty \\
\sin(x) &= x - x^3/3! + x^5/5! - x^7/7! + \dots \text{for } -\infty < x < \infty
\end{aligned}$$

8. Diseña una función que calcule el *coeficiente binomial*

$$C_n^m = \frac{n!}{m!(n-m)!}$$

donde n y m son 2 números enteros y cumplen que $n \geq m$.

Después diseña otra función que, dados 2 números reales a y b y un número entero positivo n , calcule la *fórmula binomial*:

$$\begin{aligned}
(a+b)^n &= \sum_{k=0}^n \binom{n}{k} a^{n-k} b^k \\
&= \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \binom{n}{2} a^{n-2} b^2 + \dots + \binom{n}{n} a^0 b^n,
\end{aligned}$$

Desde un programa principal que lea de teclado valores para a , b y n , llama a ésta última función con estos valores e imprime el resultado. ¿Notas alguna deficiencia para valores moderadamente grandes de n ? Si es así, ¿qué crees que está pasando?

9. Sea p la probabilidad (entre 0 y 1) de que un evento X ocurra una sola vez. La probabilidad de que dicho evento ocurra i veces en n experimentos sigue la fórmula

$$Pr(X_i) = \binom{n}{i} p^i (1-p)^{n-i},$$

Escribe una función que calcule la potencia entera de un número real. Después, con ayuda de esta función, escribe otra que, dada la probabilidad p de que ocurra un evento una vez, y un número de experimentos n , calcule la probabilidad de que el evento ocurra i veces mediante la fórmula anterior. Escribe un programa completo para probarlo todo.

10. Escribe un programa que determine el valor de π mediante el algoritmo de Gauss-Legendre, en el cual se inicializa $a_0 = 1$, $b_0 = 1/\sqrt{2}$, $t_0 = 1/4$, $p_0 = 1$, y se itera mediante:

$$\begin{aligned}
a_{i+1} &= \frac{a_i + b_i}{2} \\
b_{i+1} &= \sqrt{a_i b_i} \\
t_{i+1} &= t_i - p_i (a_i - a_{i+1})^2 \\
p_{i+1} &= 2p_i
\end{aligned}$$

Finalmente,

$$\pi \approx \frac{(a_i + b_i)^2}{4t_i}.$$

Capítulo 4

Subprogramas

Una de las consideraciones más importantes que un programador debe tener en cuenta es evitar la repetición de código. Con ello se consigue:

- 1) reducir el número de errores (*fiabilidad*),
- 2) oportunidades para reutilizar código (*reusabilidad*),
- 3) facilidad de mantenimiento, y
- 4) mayor legibilidad.

Para ello es fundamental *estructurar* bien el código, dividirlo en trozos de código que resuelvan *subtareas* bien definidas. Estos trozos de código son los *subprogramas*¹. Además, para ganar flexibilidad los subprogramas se pueden *parametrizar*, de forma que las mismas operaciones se pueden ejecutar sobre distintos *datos de entrada*. Los datos de entrada y de salida de un subprograma es lo que se conoce como la *interfaz de entrada/salida* del subprograma.

4.1. El concepto de llamada

Para evitar la repetición de un trozo de código que se tiene que escribir muchas veces en distintos puntos de un programa, se *encapsula* definiéndolo una sola vez y se *invoca* todas las veces que sea necesario. Una vez que la ejecución llega al punto de la llamada, el control pasa al subprograma que se ejecuta hasta que termina y finalmente *retorna* el control al punto del programa donde se había llamado, continuando la ejecución con la siguiente sentencia a la llamada. El concepto *llamada-retorno* se ilustra en la figura 4.1.



Un subprograma no *tiene vida* mientras no sea *llamado* desde algún punto del programa en tiempo ejecución.

Los subprogramas podrían ser simplemente una *expansión de código* que se podrían llamar múltiples veces evitando así la repetición de código, como muestra el siguiente esquema.

¹También llamados *módulos*, *rutinas* o *subrutinas*.

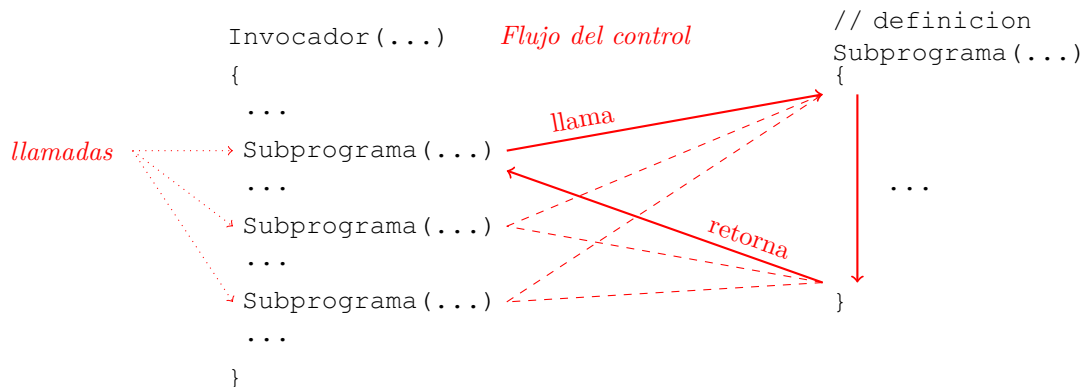


Figura 4.1: Concepto *llamada-retorno*.

```
float MeanGrade()    // function definition
{
    float grade1 = 3.0;
    float grade2 = 2.0;
    float grade3 = 8.5;
    return grade1*0.2 + grade2*0.2 + grade3*0.6;
}

int main()
{
    ...
    global_grade = MeanGrade(); // first call
    ...
    if (MeanGrade() > 5.0) {    // second call
        ...
    }
    ...
    cout <<MeanGrade();        // third call
    ...
}
```

Pero observa que no hay *parámetros de entrada* lo que hace que la expansión de código sea muy rígida. La parametrización del mismo subprograma permite realizar el *mismo cálculo* sobre *diferentes datos*.

```
float MeanGrade(float grade1, float grade2, float grade3) // definition
{
    return grade1*0.2 + grade2*0.2 + grade3*0.6;
}

int main()
{
    float exam1, exam2, exam3;
    ...
    for (int i=0; i<NumberOfStudents; i++) {
        ...
        cin >>exam1 >>exam2 >>exam3;    // user input data
        ...
        // input parameters change at each iteratio
    }
}
```

```

    cout <<MeanGrade(exam1, exam2, exam3);
    ...
}
...
}

```

4.2. Parámetros y procedimientos

Los parámetros que aparecen en cada llamada son los *argumentos reales*, mientras que los correspondientes en la *cabecera* de un subprograma son los parámetros *formales* o *ficticios*. En el ejemplo anterior, `grade1`, `grade2`, `grade3` son los parámetros formales, mientras que `exam1`, `exam2`, `exam3` son los parámetros reales correspondientes.

La cabecera de un subprograma contiene la *lista de parámetros formales* donde se especifica el número de parámetros necesario y el tipo de cada uno de ellos. Cada llamada debe contener el mismo número de parámetros especificado en esta lista, y cada uno del tipo indicado.

Es el orden en esta lista el que establece el vínculo entre el parámetro real de cada llamada y el correspondiente formal de la definición, y no los identificadores utilizados.

Por otro lado, en general, todo (sub)programa tiene datos de salida y datos de entrada. Es fundamental especificar para cada parámetro si es un dato de *entrada* o es un dato de *salida*. Hasta ahora hemos definido un tipo de subprogramas, las funciones, que sólo tienen parámetros de entrada, y la salida era el valor devuelto por la sentencia **return**, simulando de alguna manera el comportamiento de las funciones matemáticas que son *univaluadas*. El tipo de este valor devuelto también debe especificarse como tipo de la propia función en su cabecera.

Los parámetros reales de *entrada* pueden ser cualquier expresión del mismo tipo que el correspondiente parámetro formal siempre que sean del mismo tipo o compatible. Además, la expresión donde aparece la llamada debe ser compatible con el tipo del valor que devuelve la función. Por otro lado, deben proporcionar información válida al subprograma.

Pero en programación es habitual la necesidad de devolver más de un valor de salida, lo que no se puede hacer con la sentencia **return**. A veces incluso no es necesario devolver ningún valor de salida, por ejemplo cuando un subprograma simplemente imprime cálculos en pantalla. Por estas razones los subprogramas también admiten *parámetros de salida*, que en la llamada van a ser variables que van a contener resultados al final de la ejecución del subprograma.

Los parámetros reales de *salida* deben ser variables, nunca valores constantes o u otras expresiones, aunque sean del *mismo* tipo. No es necesario que estén inicializadas con ningún valor.

Habiendo parámetros de salida, ya no es estrictamente necesaria la sentencia **return** ni especificar ningún tipo para el propio subprograma, lo cual se hará con la palabra reservada **void** al principio en su cabecera sustituyendo al tipo que se especificaba para las funciones. A este tipo de subprogramas que no representan ningún valor se les denomina *procedimientos*.

Las dos siguientes funciones convierten temperaturas de/a grados celsius a/de grados fahrenheit. Utilizan el mecanismo de salida de la sentencia **return**. Nota el tipo `double` al frente de la definición de cada función.

```
double ToCelsius(double fahrenheit)
{
    return 5.0/9.0*(fahrenheit-32);
}
```

```
double ToFahrenheit(double celsius)
{
    return celsius*9.0/5.0 + 32;
}
```

El mecanismo de salida *return* hace posible llamadas *dentro* de expresiones de tipo compatible:

```
temp = ToCelsius(temp0) - (alt - alt0) * 6.4;
```


Pero el mismo ejemplo se puede realizar con procedimientos que no devuelven nada como subprograma, sino que utilizan parámetros de salida. Las llamadas ahora van a ser distintas.

```
void ToCelsius(double fahrenheit, // input parameter
               double &celsius)  // output parameter
{
    celsius = 5.0/9.0*(fahrenheit-32);
}

void ToFahrenheit(double celsius, // input parameter
                  double &fahrenheit) // output parameter
{
    fahrenheit = celsius*9.0/5.0 + 32;
}
```

Observa el “tipo” void en la cabecera de cada subprograma, y el símbolo & que precede a los parámetros de salida. Ahora la llamada no puede aparecer dentro de ninguna expresión ya que no devuelve nada. Son los parámetros de salida los que se utilizan después de que termine la llamada.

```
ToCelsius(temp0, temp1); //temp1 will hold the result
temp = temp1 - (alt - alt0) * 6.4;
```

 Las llamadas a *procedimientos* no pueden aparecer dentro de expresiones porque no devuelven nada, tal como indica su “tipo” void (que en inglés significa *vacío*).

Recuerda que los parámetros reales de salida de las llamadas deben referenciar variables, posiciones de memoria que puedan albergar valores de salida. Las siguientes llamadas a los anteriores procedimientos son ilegales:

```
ToCelsius(temp0, temp*2); //temp*2 no es una posición de memoria
ToCelsius(temp0, 12.3); //12.3 no puede contener otros valores
```

Las siguientes sin embargo son legales:

```
ToCelsius(temp0-100.0, temp); //temp si es una variable
ToCelsius(77.0, temp); //y puede contener resultados
```

¿Es posible que haya parámetros que proporcionen información válida a un subprograma a la vez que almacenen algún resultado de salida? La respuesta es afirmativa, son los *parámetros de entrada/salida*. En general pueden verse como parámetros que actualizan su valor dentro del subprograma.

Los *parámetros de entrada/salida* deben tener valores inicializados correctamente antes de la llamada y ser capaces de almacenar un resultado de salida. Por eso, al igual que los parámetros reales de salida, deben ser variables.

Los siguientes ejemplos son los mismos que los anteriores utilizando parámetros de entrada/salida.

```
void ToCelsius(double &temperature)    // input-output parameter
{
    temperature = 5.0/9.0*(temperature-32);
}

void ToFahrenheit(double &temperature) // input-output parameter
{
    temperature = temperature*9.0/5.0 + 32;
}
```

Al igual que se hace con los parámetros de entrada, antes de la llamada hay que poner valores válidos en los parámetros de entrada/salida.

```
temp0 = 72.0; //inicializacion del parametro de entrada/salida
ToCelsius(temp0); //actualizacion del parametro
temp = temp0 - (alt - alt0) * 6.4; //uso del nuevo valor
```

Ejercicios

1. Escribe un procedimiento que intercambie el contenido de 2 variables de tipo entero. Con ayuda de éste, escribe otro, `Order2`, que acepte 2 parámetros de entrada/salida y los recoloque de forma ascendente, es decir, que el más pequeño quede en el primer parámetro, y el más grande en el segundo. Pruébalo con un programa que lea 2 números de teclado y los saque en pantalla de forma ordenada.
2. Escribe otro procedimiento, `Order3`, que acepte 3 parámetros de entrada/salida y los recoloque de forma ascendente sin utilizar ninguna sentencia de decisión, sino llamando al procedimiento `Order2` del ejercicio anterior. Pruébalo con un programa completo.
3. A la vista de los ejercicios anteriores, escribe otro programa que llame a un procedimiento, `Order4`, para ordenar ascendentemente 4 números leídos de teclado.
4. Con ayuda de un tipo registro, `TRacional`, escribe un procedimiento que intercambie el contenido de 2 números racionales, donde cada número racional son 2 números enteros, el numerador y el denominador. Pruébalo con un programa completo.
5. Sin utilizar tipos definidos por el programador, escribe subprogramas para
 - a) leer de teclado un número racional (numerador y denominador);
 - b) sumar 2 números racionales devolviendo numerador y denominador por separado; y
 - c) mostrar en pantalla un número racional en formato a/b.

Escribe un programa principal para leer 2 números racionales, sumarlos, y sacar el resultado en pantalla en el formato *numerador/denominador*.

6. Modifica el ejercicio anterior utilizando el tipo definido `TRacional` para almanenar el numerador y denominador de un número racional.

```

void Subprograma_A( ... )
{
    // Este subprograma es visible para el resto de subprogramas.
    // Incluso puede llamarse a si mismo recursivamente.
    // Sin embargo, no puede llamar a ninguno de los definidos despues de el.
    ...
}
void Subprograma_B( ... )
{
    // Este subprograma es visible a si mismo, a Subprograma_C, y a main.
    // Solo puede llamar a Subprograma_A.
    ...
}
void Subprograma_C( ... )
{
    // Este subprograma es visible a si mismo y a main.
    // Puede llamar a Subprograma_A y a Subprograma_B.
    ...
}
int main()
{
    // Este subprograma no puede ser llamado por ninguno
    // (es responsabilidad del sistema operativo o entorno de programacion).
    // Puede llamar a todos los subprogramas definidos antes que el.
    ...
}

```

Figura 4.2: Ámbitos de subprogramas en C++.

4.3. Reglas de ámbito

Sabemos que los datos, ya sean variables o constantes, los subprogramas y los tipos no predefinidos, se referencian con un nombre o identificador que elige el programador (a no ser que sea estándar). Pero hay que tener claro en qué parte de un programa se puede utilizar el identificador. Además, nos podemos preguntar si podemos utilizar el mismo identificador para entidades diferentes. Esto lo determina las dos siguientes reglas. La *regla de ámbito* dice:

El *ámbito* o *alcance* de un identificador empieza en el punto del código fuente donde se define, y termina al final del bloque que lo define (*alcance de bloque*). Si la definición está fuera de cualquier bloque, su ámbito termina al final del archivo fuente (*alcance de archivo*).

Una clara consecuencia de esta regla es que cualquier identificador debe definirse *antes* de ser utilizado.

Con los identificadores de tipos definidos por el programador no suele haber problemas porque suelen tener alcance de archivo y diferentes identificadores, pero con datos y subprogramas hay que tener más cuidado porque pueden definirse en distintos bloques con el mismo identificador. Cuando más de un dato o subprograma se definen con el *mismo* identificador y sus ámbitos se *solapan*, hay que aplicar la *regla de visibilidad*:

Teniendo en cuenta que un identificador no puede utilizarse más de una vez en el mismo bloque, si en cualquier punto de un programa dos o más datos o subprogramas se han definido con el mismo identificador, la entidad que prevalece es la *más interna*, es decir, el dato o subprograma accesible es el que se ha definido en el bloque más interno, y los demás están *ocultos* por aquél.


```

// Observa el punto y coma al final del prototipo!
void Subprograma_A( ... ); // el ambito de Subprograma_A empieza aqui
void Subprograma_C( ... ); // y el de Subprograma_C aqui

int main()
{
    // main sigue sin poderse llamar desde ningun otro subprograma,
    // sin embargo, puede llamar a Subprograma_A y Subprograma_C,
    // pero no a Subprograma_B.
    ...
}
void Subprograma_A( ... )
{
    // Subprograma_C puede ser llamado, pero no Subprograma_B.
    ...
}
void Subprograma_B( ... )
{
    // Se puede llamar a Subprograma_A y a Subprograma_C.
    ...
}
void Subprograma_C( ... )
{
    // Se puede llamar a Subprograma_A y a Subprograma_B.
    ...
}

```

Figura 4.3: Uso de prototipos en C++.

4.3.1. Ámbitos de subprogramas

En cuanto a los ámbitos de subprogramas, en C/C++ es fácil decidir cuándo un subprograma es accesible, ya que no es necesario aplicar nunca la regla de visibilidad por la siguiente restricción:

En C/C++ no se puede anidar subprogramas. Todos, incluido `main`, tienen *alcance de archivo*.

La figura 4.2 ilustra la accesibilidad de 4 subprogramas definidos en un programa C++, incluyendo el principal.

Por otro lado, los ámbitos de subprogramas pueden anticiparse utilizando los llamados *prototipos*.

El prototipo de un subprograma replica su *cabecera*, constituida por su tipo, su identificador y su lista de parámetros formales. Sirve para *adelantar* y extender su ámbito.

La figura 4.3 ilustra su uso. El uso de prototipos es engorroso porque hay que mantener replicadas la cabecera del subprograma y el prototipo, y si se ordenan correctamente los subprogramas no es necesario utilizarlos, a no ser que se utilice recursividad indirecta, que es una técnica de programación que no se explica en estos apuntes.

4.3.2. Ámbitos de datos y efectos laterales

En C/C++ se distinguen datos *locales*, los que tienen alcance de bloque, y datos *globales*, que tienen alcance de archivo porque se definen fuera de cualquier bloque, fuera incluso de los cuerpos de los

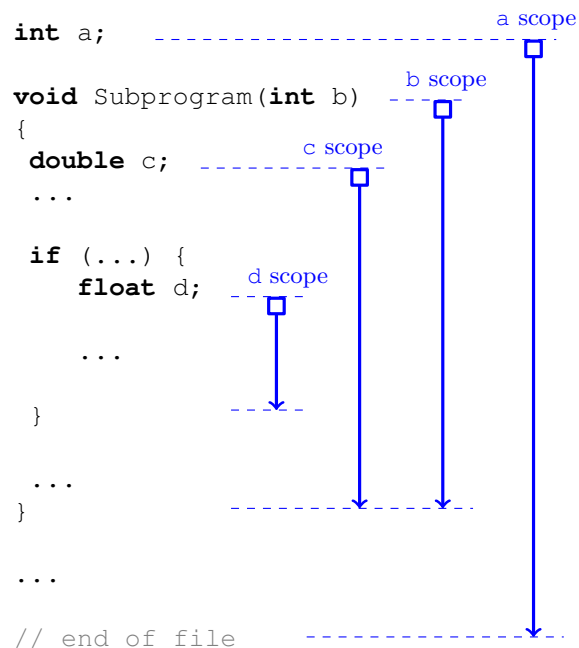


Figura 4.4: Solapamiento de ámbitos.

subprogramas. Una diferencia importante entre ellos es que los datos locales solo existen en la memoria cuando se está ejecutando el bloque que los contiene, por eso en C/C++ también se les llama *automáticos*. Pero los datos globales existen durante toda la ejecución del programa, y se dice que son *estáticos*.

La figura 4.4 muestra los ámbitos de varias variables. En este caso no hay que aplicar la regla de visibilidad porque todas tienen nombres diferentes y son visibles en su ámbito. La figura 4.5 sin embargo muestra un escenario donde las variables más internas ocultan a las más externas con el mismo identificador.

Es una buena técnica de programación limitar los ámbitos de las variables a las zonas de código donde se van a utilizar, evitando así accesos en bloques ajenos a su definición que puedan modificar su contenido de forma accidental. Pero el problema potencial realmente grave ocurre cuando definimos variables globales, ya que tienen alcance de archivo y cualquier subprograma puede acceder y modificar su contenido, dando lugar a *efectos laterales* que pueden provocar errores muy difíciles de localizar. La utilidad de las variables globales es histórica² y su uso se limita a otros escenarios de programación donde las variables locales no son la solución.

✎ Aunque no están prohibidas en C/C++, el uso de variables globales debe ser evitado siempre que sea posible.

Ejercicio

1. Responde a las siguientes preguntas relacionadas con el código fuente adjunto.

²Cuando se empezaba a programar con lenguajes de alto nivel en los años 70 todos los datos eran globales.

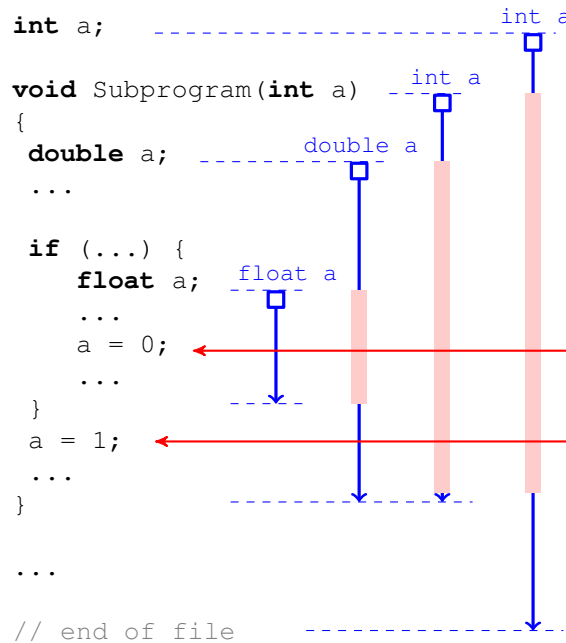


Figura 4.5: Ocultación de variables.

```

1  float x;
2
3  void SubA(float &y)
4  {
5      double x;
6      x = 2.5;
7      if (x > y) {
8          y = 0;
9      }
10     SubB(a);
11 }
12
13 void SubB(int b)
14 {
15     double a;
16     a = x*b;
17     if (a > b) {
18         bool b;
19         b = true;
20     }
21     cout <<"b=" <<b <<endl;
22     SubA(b);
23 }
24
25 int main()
26 {
27     cout <<"x="; cin >>x;
28     SubB(x);
29     SubA(x);
30     cout <<"a=" <<a <<endl;
31 }

```

- a) Especifica qué clase de alcance (de archivo o de bloque) tienen los ámbitos de cada una de las siguientes definiciones y qué rangos de líneas ocupan:

```

float x;
int a;
double x;
int b;
bool b;
void SubA(...)
void SubB(...)

```

- b) Especifica el rango de líneas e identificadores involucrados en la aplicación de la regla de visibilidad si ésta ha lugar en algún punto.
- c) Especifica las líneas donde se produzcan accesos ilegales a variables.
- d) Especifica las líneas donde aparezcan llamadas a subprogramas ilegales.
- e) Especifica las líneas donde ocurran efectos laterales o potencialmente pueden ocurrir.

Mecanismo de paso	Por valor	Por valor constante	Por referencia	Por ref. constante
Sintaxis	(por omisión)	<code>const</code>	<code>&</code>	<code>const ... &</code>
Parámetro entrada	dato simple	dato simple	n.a.	datos compuestos
Parámetro salida	n.a.	n.a.	todos	n.a.
Parámetro e/s	n.a.	n.a.	todos	n.a.

Tabla 4.1: Paso de parámetros adecuado en C++ según tipo de datos.

4.4. Paso de parámetros

C++ proporciona los siguientes mecanismos para la implementación del *paso de parámetros*, esto es, cómo se transfieren los datos en memoria entre el subprograma llamado y el que lo invoca.

- 1) *Paso por valor*, adecuado para parámetros de entrada en general. Los contenidos del parámetro real se copian en el espacio de memoria del subprograma, donde se trabajará con la copia, preservando el contenido del parámetro real original. Para los parámetros de entrada también es apropiado el *paso por valor constante*, en cuyo caso el compilador ni siquiera permite modificar el valor de la copia. El paso constante se indica con la palabra reservada **const** delante del tipo del correspondiente parámetro formal.
- 2) *Paso por referencia*, adecuado para parámetros de salida. En este caso el contenido del parámetro real no se copia, sino que se trabaja con el original directamente, de tal modo que cualquier modificación dentro del subprograma se reflejará en el contenido del parámetro real posterior a la llamada. Por supuesto, para que esto sea posible, el parámetro real de la llamada *debe* ser una variable, no cualquier expresión. El paso por referencia se indica con el símbolo `&` delante del identificador del parámetro formal correspondiente.
- 3) *Paso por referencia constante*, adecuado para parámetros de entrada de gran tamaño. Al igual que en el caso anterior, no hay copia del parámetro real, evitando el uso de memoria innecesario y mejorando la eficiencia. Al ser constantes, por otro lado, el compilador garantiza que no hay modificaciones accidentales del parámetro real.

La tabla 4.1 resume la idoneidad de cada mecanismo de paso de parámetros en C++ según el tipo de parámetro.

4.5. Cuestiones semánticas

Cuando el parámetro es de salida o entrada/salida, el parámetro real debe ser del *mismo* tipo que el formal, no permitiéndose conversiones, ya que el dato sobre el que trabaja el subprograma es el propio parámetro real.

Pero cuando el parámetro es de entrada, el parámetro que se pasa puede ser de tipo compatible al que indica el parámetro formal, dándose la posibilidad de realizarse las mismas conversiones, implícitas o explícitas, que ocurren en la sentencia de asignación. Hay que tener en cuenta que el subprograma trabajará con el tipo indicado en la lista de parámetros formales. En el ejemplo siguiente, el parámetro que recibe el subprograma se *promociona* a `double`.

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

double Square_double(double x) // expects a real number parameter
{
    return x*x;
}
```

Tipo de datos
long double
double
float
unsigned long [int]
long [int]
unsigned [int]
int
unsigned short [int]
short [int]
unsigned char
[signed] char
bool

Figura 4.6: Jerarquía de promoción de los tipos predefinidos. (Los corchetes indican que es opcional.)

```

}

int main()
{
    int inumber, squared;
    cout <<"An integer number to real square: ";
    cin >>inumber;
    squared = Square_double(inumber); // sends a integer number parameter
    cout <<"The square is " <<squared <<endl;
}

```

Observa que las conversiones también se aplican en la asignación de un valor que devuelve una función. En el mismo ejemplo, se pierde la posible precisión del valor devuelto por la función al asignarse a una variable entera. La tabla 4.6 enumera los tipos predefinidos por orden de mayor a menor “precisión” que C++ utiliza para promocionar valores de distinto tipo.

Convertir valores, ya sea implícita o explícitamente, puede resultar en valores incorrectos de salida. En el ejemplo siguiente, si el valor de entrada vale 2.5, la salida valdría incorrectamente 4 por trabajar con precisión entera.

```

#include <iostream>
using namespace std;


int Square_int(int x) // x cannot hold any fractional part
{
    return x*x;
}

int main()
{
    float rnumber, squared;
    cout <<"A real number to int square: ";
    cin >>rnumber;
    squared = Square_int(rnumber); // sends a real number
    cout <<"The truncated square is " <<truncated <<endl;
}

```


4.6. Diseño modular

El término *diseño modular* o *diseño estructurado* hace referencia a


 técnicas de diseño de software que separan la funcionalidad de un programa en *módulos* independientes, intercambiables y reutilizables, de tal manera que cada uno contiene todo lo necesario para ejecutar sólo un aspecto de la funcionalidad deseada.

Una de las principales razones que causó la *crisis del software* de los años 60 fue la ausencia de lenguajes de alto nivel que permitieran escribir grandes programas organizándolos con pequeños subprogramas, lo que llevó a las conocidas deficiencias para reutilizar código, localizar errores o mejorar su legibilidad.


Al diseñar módulos o subprogramas, lo más importante son las interfaces de entrada/salida de cada uno de ellos. Si no son interfaces claras, hay alta probabilidad de que la tarea no esté bien definida, o quizá de que no hay oportunidad para *modularizar*.

 Lo primero y más importante al escribir un subprograma es definir claramente su interfaz de entrada/salida.

Muy relacionado con la definición de la interfaz es el hecho de que un subprograma debe realizar una tarea bien definida, es decir, el subprograma debe tener una *alta cohesión*.

 Trata de dividir el código en tareas bien definidas, y evita la repetición de código.

Además, las interfaces deben ser lo más pequeñas posibles, siempre que contengan toda la información necesaria, y no más, para realizar su función. El *acoplamiento* entre módulos indica la fuerza de interconexión entre las distintas unidades de un programa. Este acoplamiento debe reducirse al mínimo para favorecer la reutilización de código y el mantenimiento general del programa.

 Diseña subprogramas con una *alta cohesión* y reduce al mínimo el *acoplamiento* entre ellos.

Otro concepto relacionado es el *ocultamiento de la información*. Los detalles de la implementación de un subprograma no deben ser visibles fuera de él, de forma que pueda ser invocado teniendo en cuenta simplemente cuál es la información mínima que necesita de entrada y cuál la salida. Cualquier resultado intermedio debe ser interno al subprograma y tratado con ayuda de variables locales. De hecho, la implementación de un subprograma (los detalles) debe poder cambiarse sin necesidad de cambiar su interfaz de entrada/salida si ésta está bien definida. Se logra así lo que se llama la *separación* de la *especificación* de la *implementación*.

4.6.1. Diseño ascendente y descendente

Abordar el diseño modular de un problema de programación moderadamente grande se puede hacer de dos formas. Una de ellas es empezar por los detalles de las subtareas más claras que son independientes y/o tienen poca dependencia del resto. En este caso estaríamos diseñando de abajo hacia arriba, lo que se llama *diseño ascendente*. El otro enfoque, más común, especialmente con grandes proyectos de programación, es empezar por los módulos principales, sin detenerse en los detalles menos significativos al principio. Una vez diseñadas sus interfaces y su implementación a groso modo, se *refinan*, posiblemente subdividiendo en más módulos, hasta llegar a los detalles finales. De esta forma estamos diseñando de arriba hacia abajo, lo que se conoce como *diseño descendente*.

El uso de *diagramas de llamadas* como el de la figura 4.7 ayuda a aplicar el diseño modular. Estos diagramas simplemente resaltan todas las llamadas a subprogramas que contiene un programa.

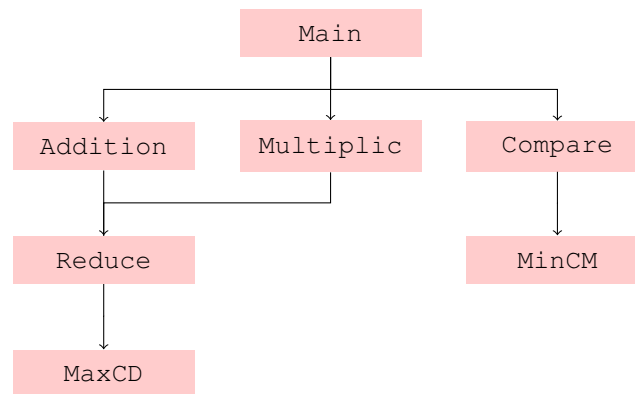


Figura 4.7: Diagrama de llamadas.

Ejercicios de repaso

1. Escribe una función lógica, `EsPrimo`, que determine si un número natural es primo. Llama a esta función desde otra, `EscribirPrimos`, que escriba en pantalla todos los números naturales primos que haya en un rango dado.

Un programa principal llamará a `EscribirPrimos` para escribir en pantalla los números primos que haya entre los primeros 1000 números naturales. Después amplíalo para que muestre todos los números primos que haya en un rango especificado por el usuario.

2. Dos números son *amigos* cuando cada uno coincide con la suma de los *divisores propios* del otro (un número no es divisor propio de sí mismo). Escribe una función, `SumaDivisores`, que calcule la suma de sus divisores propios (utiliza una función lógica para decidir si un divisor es primo). Después escribe otra función lógica, `Amigos`, que acepte como entrada 2 números enteros y devuelva `true` o `false` según sean o no amigos. Por ejemplo, son números amigos el 220 y el 284 porque los divisores del primero son 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 y 110 que suman 284, y los divisores del segundo son 1, 2, 4, 71 y 142, que suman 220.

Amplía después el programa para que saque en pantalla todos los números que son amigos dentro de un rango dado que el usuario introduce por teclado. Para ello tienes que generar con un bucle anidado todas las posibles combinaciones de 2 números naturales en ese rango.

3. Escribe un procedimiento que reciba como parámetros de entrada la parte real y la parte imaginaria de un número complejo y lo pase a forma polar devolviéndolo en 2 parámetros de salida que contengan el módulo y argumento correspondientes. Asegúrate de que el argumento queda en el rango $[0, 2\pi]$.

Escribe un programa principal que lea de teclado repetitivamente números complejos (dos números reales cada uno) e imprima en pantalla el módulo y argumento de cada uno hasta que se introduzca el número complejo (0,0).

4. Utilizando el diseño descendente, programa el proceso de ortogonalización de Gram-Schmidt para vectores 3D. Para ello, sigue estos pasos:

a) Define el tipo `TVector3D` para almacenar las coordenadas (x, y, z) de un vector 3D (también podrías hacerlo utilizando 3 variables reales independientes).

b) Define las interfaces de e/s de los siguientes módulos independientes:

Leer3D Leer un vector 3D de teclado.

Escribir3D Escribir un vector 3D en pantalla.

PEscalar3D Producto escalar de 2 vectores 3D.

Diferencia3D Diferencia de 2 vectores 3D.

Suma3D Suma de vectores 3D.

Escalado3D Multiplicación de los componentes de un vector 3D por un scalar común.

- c) Implementa y comprueba uno por uno los subprogramas previamente definidos con un programa principal simple.
- d) Utilizando 2 de los subprogramas anteriores, define e implementa un subprograma que calcule la proyección de un vector 3D \vec{v} sobre otro \vec{u} según la fórmula:

$$Proj(\vec{v}, \vec{u}) = \frac{\vec{v} \cdot \vec{u}}{\vec{u} \cdot \vec{u}} \vec{u}$$

- e) Define e implementa el proceso de ortogonalización de Gram-Schmidt en un subprograma que tome 3 vectores 3D linealmente independientes como entrada, $\vec{v}_1, \vec{v}_2, \vec{v}_3$, y genere 3 vectores 3D ortogonales $\vec{u}_1, \vec{u}_2, \vec{u}_3$ como salida según las fórmulas:

$$\begin{aligned}\vec{u}_1 &= \vec{v}_1 \\ \vec{u}_2 &= \vec{v}_2 - Proj(\vec{v}_2, \vec{u}_1) \\ \vec{u}_3 &= \vec{v}_3 - Proj(\vec{v}_3, \vec{u}_1) - Proj(\vec{v}_3, \vec{u}_2)\end{aligned}$$

- f) Finalmente, utilizando los subprogramas anteriores, escribe un programa principal que acepte 3 vectores linealmente independientes por teclado e imprima en pantalla 3 vectores ortogonales que generen el mismo espacio 3D mediante el proceso de Gram-Schmidt.

Soluciones al ejercicio 1 de la sección 4.3

- (a)

float x;	Alcance de archivo, líneas: 1–31
float y;	Alcance de bloque, líneas: 3–11
double x;	Alcance de bloque, líneas: 5–11
int b;	Alcance de bloque, líneas: 13–23
double a;	Alcance de bloque, líneas: 15–23
bool b;	Alcance de bloque, líneas: 18–20
void SubA(...);	Alcance de archivo, líneas: 3–31
void SubB(...);	Alcance de archivo, líneas: 13–31
- (b) **double** x oculta a **float** x desde la línea 5 a la línea 11.
bool b oculta a **int** b desde la línea 18 a la línea 20.
- (c) El identificador a de la línea 30 no referencia ninguna variable.
- (d) La llamada a SubB en la línea 10 de SubA es ilegal, SubB() no es accesible. Y el argumento x de la llamada SubA(x) es ilegal porque no es de tipo int (es un argumento de salida).
- (e) La lectura de teclado `cin >>x` en la línea 27 es un efecto lateral, y la llamada Sub(x) en la línea 28 es un efecto lateral potencial (podrá modificarse la variable global x).

Capítulo 5

Datos compuestos

5.1. Tipos definidos por el programador

A lo largo de estos apuntes hemos visto cómo las expresiones aritméticas dotan de capacidad de *cálculo* a nuestros programas, las sentencias de decisión les dotan de *inteligencia*, los bucles de *potencia*, y los subprogramas de capacidad *organizativa de operaciones*. En este capítulo veremos cómo podemos organizar en un programa C++ la *información*, completando los requerimientos fundamentales que debe tener un lenguaje de alto nivel de propósito general.

Ya conocemos los tipos *predefinidos* de C++ para gestionar unidades de información pequeñas:

```
float x, y;
bool ok;
const double Pi=3.1415926535;
```

También conocemos los tipos *definidos por el programador* sencillos que agrupaban pocas variables de tipo simple, introducidos en la sección 1.10:

```
typedef struct { int hh, mm, ss; } THour;
THour begin, coffebreak, finish;
THour breakfast, lunch, dinner;
```

La figura 5.1 ilustra todos los tipos que proporciona C++ al programador, de los cuales sólo los *tipos enumerados* no se ven en estos apuntes por no ser esenciales. Como puede verse en la figura, a excepción de los enumerados, los tipos definidos por el programador son de tipo compuesto. Y aunque el tipo compuesto `string` está definido en la biblioteca estándar `string`, a efectos prácticos podemos considerarlo predefinido. En este capítulo vamos a ver cómo utilizar los tipos compuestos para organizar grandes cantidades de información de forma eficiente. Estos datos son los *registros* y los *arrays*.

5.2. Registros

Los *registros*, llamados `struct` en C++, son simplemente colecciones *heterogéneas* de datos relacionados entre sí, cada uno de los cuales se denomina *campo*. Cada campo es como una variable dentro de otra más grande, con su tipo y su identificador. Pero todos comparten un nombre común, lo que nos da la posibilidad de manejar el dato compuesto como una unidad, por ejemplo para pasarlo como un solo parámetro a un subprograma. Tal como se observa en el siguiente ejemplo, una vez definido el tipo registro, se pueden definir tanto variables como constantes simbólicas del nuevo tipo.

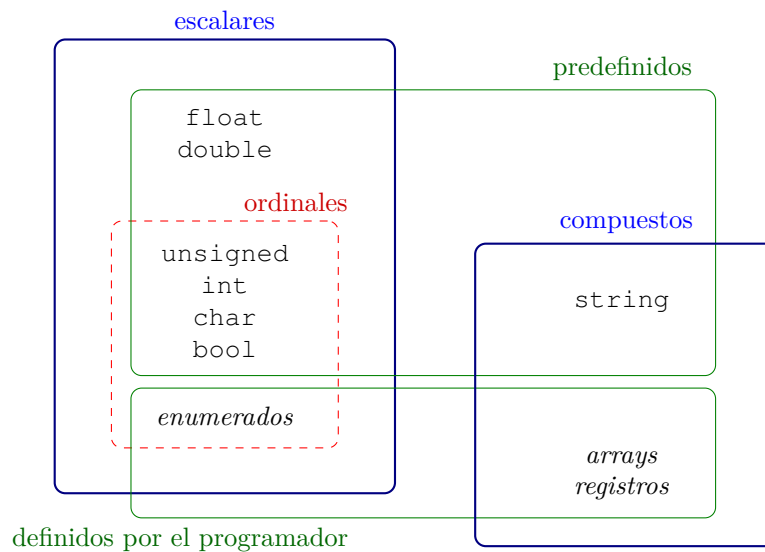


Figura 5.1: Tipos en C++.

```
typedef struct {
    int day, month, year;
} TDate; // el nuevo tipo
TDate birth_date, join_date; // variables registro
TDate opening_date = {29,7,2010}; // registros inicializados
const TDate ConstitutionDate={6,12,1978}; // constante simbolica
```

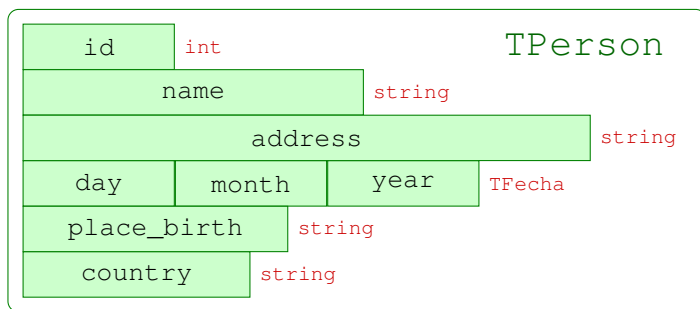
Tradicionalmente la definición del tipo registro no se hacía con la sentencia **typedef**, por lo que es muy habitual encontrar la siguiente definición equivalente. Nótese la posición del identificador del tipo nuevo tras la palabra reservada **struct**.

```
struct TDate {
    int day, month, year;
};
```

No hay restricciones en cuanto al número o tipo de campos. Un registro incluso puede contener variables de tipos registro definidos anteriormente, produciéndose un *anidamiento de datos*. El siguiente ejemplo define un tipo para almacenar todos los datos relevantes de una persona.

```
typedef struct {
    int id_card;
    string name;
    string address;
    TDate birth;
    string place_birth;
    string nationality;
} TPerson; // the type

// the data
TPerson employee, boss;
const TPerson Me = {
    123456789,
    "Juan Garcia",
    "Remindme Street",
    {1,1,1980},
    "Any Place",
    "Spanish"
};
```




El identificador del registro se utiliza para referirnos a él como un todo, mientras que para acceder a cada campo utilizamos operador '.' entre el identificador del registro y el del campo concreto. El siguiente ejemplo muestra cómo pasar todo el registro a un subprograma en un único parámetro, y cómo acceder a sus campos individualmente.

```

void WriteDate(const TDate &date)
{
    cout <<date.day<< '/' <<date.month<< '/' <<date.year;
}

```

 Un campo de un registro se puede usar exactamente de la misma forma que una variable individual del mismo tipo.

Las siguientes operaciones con registros completos son legales en C++:

- Inicialización con valores para cada campo.
- Asignación de todo el registro en una sola sentencia.
- Paso como parámetro, tanto de entrada como de salida.
- Uso como valor de retorno de una función.

El siguiente ejemplo ilustra todas estas operaciones.

```

#include <iostream>
using namespace std;

typedef struct {
    unsigned day, month, year;
} TDate; // new record type

void ReadDate(TDate &date) // output parameter: pass by reference
{
    bool ok;
    do {
        cin >>date.day>>date.month>>date.year;
        if (date.month <= 12) {
            switch (date.month) {
                case 1: case 3: case 5: case 7: case 8: case 10: case 12:
                    ok = date.day>0 && date.day<=31;
                    break;
                case 4: case 6: case 9: case 11:
                    ok = date.day>0 && date.day<=30;
                    break;
            }
        }
    } while (!ok);
}

```

```

        case 2:
            ok = date.day>0 && date.day<=28;
            break;
        default: // case 0
            ok = false;
    }
    if (!ok)
        cout <<"Invalid day or month. Another date (dd mm yy): ";
    }
    else {
        ok = false;
        cout <<"The month must be between 1 and 12. Another one (dd mm yy): ";
    }
} while (!ok);
}

void WriteDate(const TDate &date) // input parameter: pass by constant ref.
{
    cout <<date.day<< '/' <<date.month<< '/' <<date.year;
}

// compares two dates (notice the return type)
TDate MoreRecent(const TDate &d1, const TDate &d2)
{
    TDate recent;
    if (d1.year>d2.year ||
        d1.year==d2.year && d1.month>d2.month ||
        d1.year==d2.year && d1.month==d2.month && d1.year>d2.year) {
        recent = d1; // a whole record assignment
    }
    else {
        recent = d2; // a whole record assignment
    }
    return recent; // returning as a function value
}

int main()
{
    const TDate UnixBirth = {1, 1, 1970}; // whole record initialization
    TDate date1, date2, later;
    cout <<"Comparing dates with Unix birth 1/1/1970 (dd mm yy): ";
    ReadDate(date1);
    cout <<"Introduce another one (dd mm yy): ";
    ReadDate(date2);
    later = MoreRecent(date1,date2); // assigning a whole returned record
    WriteDate(MoreRecent(UnixBirth,later)); // passing a whole returned record
    cout <<" is more recent." <<endl;
}

```

Como los tipos registro no están predefinidos en C++, las siguientes operaciones son ilegales en C++ mientras no sean programadas:

- Valores literales no pueden asignarse fuera de la definición del registro:

```
date = {2, 3, 1987};
```

- Salida en pantalla de todo el registro como si fuera una sola variable:

```
cout << date;
```

- Lectura de teclado como si fuera una sola variable:

```
cin << date;
```

- Uso de operadores relacionales o aritméticos sobre registros completos:

```
if (date1 < date2) ...  
... date1 + date2...
```

Ejercicios

1. Define un tipo, TComplejo, para almacenar 4 números reales que representan (1) la parte real, (2) la parte imaginaria, (3) el módulo, y (4) el argumento principal de un número complejo (a, b) . Utilizando este tipo, escribe los dos siguientes procedimientos y llámalos desde un programa principal:

LeerComplejo. Lee un número complejo de teclado. Un parámetro de entrada de tipo lógico especificará si se deben leer las partes real e imaginaria, o bien el módulo y argumento.

EscribirComplejo. Escribe un número complejo en pantalla. Un parámetro de entrada de tipo lógico indicará si se muestran las partes real e imaginaria, o bien el módulo y argumento.

2. Utiliza el tipo TComplejo para los parámetros de entrada/salida de los dos siguientes subprogramas:

ModificaModArg. Actualiza el módulo ρ y el argumento principal ϕ de un número complejo a partir de sus partes real e imaginaria (a, b) según las fórmulas:

$$\rho = \sqrt{a^2 + b^2}, \quad \phi = \tan^{-1} \frac{b}{a}.$$

ModificaReIm. Actualiza las partes real e imaginaria (a, b) de un número complejo a partir de su módulo ρ y su argumento principal ϕ según las fórmulas:

$$a = \rho \cos(\phi), \quad b = \rho \sin(\phi).$$

Comprueba su funcionamiento con un programa principal que lea de teclado las partes real e imaginaria de un número complejo y calcule su módulo y argumento. Después vuelve a calcular las partes real e imaginaria a partir del módulo y argumento calculados para compararlas con los valores originales leídos de teclado.

3. Utilizando los subprogramas de los ejercicios anteriores, escribe funciones para calcular la suma y producto de dos números complejos. Escribe un programa completo adecuado para probarlo todo.

5.3. Arrays

Un *array* es una colección *homogénea* de datos, esto es, todos del mismo tipo, como por ejemplo listas de números, de cadenas de caracteres, o incluso de registros. El tipo de cada *componente* es el *tipo base*, y el tipo de todo el array es el *tipo array*. Todos los datos del array comparten el mismo nombre, la *variable array*, pero los componentes no tienen nombre específico, sino que se identifican de forma única por medio de su posición en el array, la cual se llama *índice*. Los índices son obligatoriamente números naturales, y en C++ el primer componente siempre tiene el índice 0, tal como se ilustra en la figura 5.2.

Además del tipo base, la otra característica fundamental de un array es su *tamaño*, es decir, su número de componentes. Dependiendo del momento en que se pueda especificar, hay dos clases de arrays, a saber:

Array de enteros									
125	-15	-98	267	-16	-95	153	137	186	-98
0	1	2	3	4	5	6	7	8	9

Array de caracteres									
'h'	'0'	'1'	'a'	'.'	','	'6'	'H'	','	'?'
0	1	2	3	4	5	6	7	8	9

Array de valores lógicos						
true	false	false	false	true	true	false
0	1	2	3	4	5	6

Figura 5.2: Tres arrays con tipos base diferentes: int, char y bool.

Arrays estáticos Su número de elementos lo define el programador en *tiempo de edición* del programa, por lo que debe ser una expresión constante que estime el tamaño máximo del array de forma anticipada. Este tamaño es la *longitud física*. Esta longitud no se puede cambiar durante la ejecución del programa.

Arrays dinámicos Su longitud se puede especificar en *tiempo de ejecución* de forma más precisa en base a los datos de entrada del programa. Además, esta longitud puede cambiar a lo largo de la ejecución del programa según se necesite.

En estos apuntes no vamos a ver los arrays dinámicos. En cuanto a los estáticos, en C++ disponemos de los arrays tradicionales al estilo del lenguaje C, y los arrays modernos del estándar 2011 de C++, los cuales están definidos en la biblioteca estándar array. La diferencia fundamental entre ellos es la capacidad de ser asignados completamente, existente en los modernos, pero de la que carecen los arrays tradicionales del lenguaje C.

Definición de arrays

La definición de un array estático debe incluir el tipo base de cada componente y su longitud máxima. Una vez definido el tipo, se puede definir tanto variables array como arrays de constantes, tal como ilustra el siguiente ejemplo.

```
#include <array>          // necesario para arrays del estandar 2011
const unsigned MaxStudents = 10; // longitud fisica del array
typedef array<float, MaxStudents> TMarks; // el tipo array
const TMarks AllPassed = {{ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }};
TMarks partials = {{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};
TMarks calculus, algebra;
```

Obsérvese las llaves dobles { { } } para la inicialización de los arrays modernos en el estándar 2011.

Acceso a elementos individuales

El acceso a un componente del array debe incluir, además del nombre del array, el índice del elemento al que se quiere acceder. Se puede utilizar la sintaxis de los arrays tradicionales entre corchetes []:


```
miarray[indice]
```

o bien la sintaxis moderna, típica de la orientación a objetos, que llama a la función `at()`, la cual puede comprobar posibles errores del rango:

miarray.at(indice)

Los siguientes dos ejemplos acceden al primer elemento del array, con la sintaxis tradicional el de la izquierda, y con la función `at()` el de la derecha:

```
cin >>marks[0];          cin >>marks.at(0);
marks[0] = marks[0]*0.25; marks.at(0) = marks.at(0)*0.25;
cout <<marks[0];         cout <<marks.at(0);
```

 Una vez que se ha especificado un elemento del array con el índice correcto, se puede utilizar exactamente igual que una variable individual del mismo tipo.

Errores de rango

Los errores más habituales que se cometen al utilizar arrays son los molestos *errores de rango*, que se producen cuando se especifica un índice fuera de los límites físicos del array. El siguiente ejemplo tiene un error de rango potencial, ¿sabrías dónde?

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

const unsigned Length = 12;
typedef array <int, Length> T12;

int main()          flawed
{
    T12 DaysOf2014 = {{ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }};
    unsigned nmonth;
    cout <<"Choose a 2014 month (1..12): ";
    cin >>nmonth;
    cout <<"Month " <<nmonth <<" has "<<DaysOf2014[nmonth-1] <<" days\n";
}
```

No obstante, estos errores pueden capturarse de muy diversas formas con sentencias adicionales, como puede observarse en el siguiente listado que corrige el error potencial anterior.

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

const unsigned Length = 12;
typedef array <int, Length> T12;

int main()
{
    T12 DaysOf2014 = {{ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }};
    unsigned nmonth;
    do {
        cout <<"Choose a 2014 month (1..12): ";
        cin >>nmonth;
    } while (nmonth==0 || nmonth>12);
    cout <<"Month " <<nmonth <<" has "<<DaysOf2014[nmonth-1] <<" days\n";
}
```

La función `at()` de los arrays modernos del estándar 2011 tiene la capacidad de capturar errores de rango automáticamente, pero requiere el uso de excepciones, las cuales no se explican en estos apuntes. En estos apuntes siempre utilizaremos el acceso tradicional con el operador `[]`.

Por otra parte, los arrays estáticos no suelen estar completamente llenos con valores válidos, sino que sólo una parte del array físico está siendo utilizada. Es decir, la longitud física no tiene por qué coincidir con la *longitud lógica*, la cual puede variar en tiempo de ejecución. Recuerda:



Siempre que se escriba una expresión entre corchetes `[]` para acceder a elementos de un array hay que asegurarse que este valor siempre estará en el rango adecuado, tanto físico como lógico.

Ejercicios

1. Sin utilizar bucles, escribe un programa que lea 4 números de teclado y los almacene en un array. Después muestra el contenido del array en el orden de sus posiciones. A continuación, rótalos una posición de forma que cada uno quede en la posición siguiente a la suya, y el último quede en la primera posición. Finalmente, vuelve a mostrar en orden los elementos del array con los contenidos ya rotados.
2. Realiza el ejercicio anterior con bucles utilizando la variable de control para obtener los índices.
3. Sin utilizar bucles, escribe un programa que lea 4 pares de caracteres y se almacenen alternativamente en dos arrays de 4 posiciones, es decir, el primero tecleado, el tercero, el quinto y el séptimo en uno de los arrays, y el resto en el otro. Después compara las posiciones iguales componente a componente de ambos arrays con el fin de escribir en pantalla los mayores de cada par introducido.
4. Modifica el ejercicio anterior de forma que las comparaciones y la salida en pantalla se realicen con un bucle.
5. Usando bucles, escribe un programa que lea 4 números de teclado en un array y los copie uno a uno en otro array. Saca en pantalla el contenido del segundo.
6. Utilizando bucles, escribe un programa que lea 5 números reales y los guarde en un array. Después calcula las siguientes fórmulas y escribe los resultados en pantalla:

$$S = \sum_{k=1}^5 x_k \quad C = \sum_{k=1}^5 x_k^2 \quad M = \max_{1 \leq k \leq 5} |x_k|$$

Asignación completa de arrays

La característica más importante de los arrays estáticos modernos del estándar 2011 es la capacidad de copiar un array completo en otro con una sola asignación. La copia es física, de forma que se copia tanto las posiciones válidas como las que están sin usar del array. Esta operación crucial, que está prohibida en los arrays tradicionales, permite a su vez el paso de arrays completos como parámetros, y ser devueltos como único valor de una función. El siguiente ejemplo explota estas posibilidades.


```

typedef array <float, 3> TVector3D;
...
// prototipo de la funcion
TVector3D Adicion3D(const TVector3D v, const TVector3D w);

// llamada al subprograma
const TVector3D I = {{ 1, 0, 0 }};
TVector3D a, b, suma;
...
b = I;    // asignacion completa de un array
suma = Adicion3D(a, b); // valor devuelto por una funcion
...

```

Sin embargo, tanto el paso por valor de un array como su devolución como valor de una función puede ser costosa computacionalmente si el array es muy grande. Para estos casos es preferible el paso de arrays por referencia, tanto si es de salida como si es de entrada. En este último caso se debe pasar por referencia constante. La alternativa más eficiente al ejemplo anterior sería:

```

// prototipo del subprograma
void Adicion3D(const TVector3D &v, const TVector3D &w, TVector &result);
...
// llamada
Adicion3D(b, c, suma);

```

Arrays incompletos

Ya hemos mencionado que es muy habitual que no se conozca en tiempo de compilación¹ el tamaño realmente necesario de un array, por eso se suele sobreestimar y usar sólo una parte del array, de forma que los arrays están *incompletos*. Es decir, la *longitud lógica* normalmente es menor que la *longitud física*. El siguiente array de enteros muestra esta situación:

longitud lógica = 7							longitud física = 10		
125	-15	-98	267	-16	-95	153	-	-	-
0	1	2	3	4	5	6	7	8	9

En estos casos es fundamental guardar en alguna variable la longitud lógica del array, y es habitual pasar como dos parámetros independientes el array incompleto y su longitud, ya sea de entrada ya sea de salida:

```

// prototipos
void UtilizarNotas(const TMarks &notas, unsigned longitud); // de entrada
void ModificarNotas(TMarks &notas, unsigned &longitud); // de salida
// llamadas
UtilizarNotas(notas, cuantas);    // TMarks notas; unsigned cuantas
ModificarNotas(notas, cuantas);  // TMarks notas; unsigned cuantas

```

Otra solución es usar un tipo registro para almacenar cada array con su longitud lógica, permitiendo su paso a subprogramas como parámetro único. En este caso estamos anidando un tipo compuesto, el array, dentro de otro tipo compuesto, el registro. En la sección 5.6 se verán ejemplos usando esta estrategia.

A continuación vemos un ejemplo pasando la longitud como un parámetro aparte.

¹Cuando el programador edita el programa.

```

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

// array type of students grades
const unsigned MaxStudents = 100;
typedef array <float, MaxStudents> TGrades;

void ReadGrades(TGrades &grades, unsigned &n)
{
    unsigned i=0;
    cout <<"How many students: ";
    cin >>n;
    do {
        i++;
        cout <<"Grade student " <<i <<": ";
        cin >> grades[i-1];
    } while (i<n);
}

void WriteGrades(const TGrades &grades, const unsigned n,
                 const float threshold)
{
    unsigned i=0;
    for (i=0; i<n; i++) {
        if (grades[i]>=threshold) {
            cout <<"Student " <<i <<": " <<grades[i] <<endl;
        }
    }
}

float MeanGrades(const TGrades &grades, const unsigned n)
{
    float sum=0;
    unsigned i=0;
    do {
        sum += grades[i++];
    } while (i<n);
    return sum/n;
}

int main()
{
    TGrades marks;
    float minimum;
    unsigned howmany;
    ReadGrades(marks, howmany);
    cout <<"The mean grade is " <<MeanGrades(marks, howmany) <<endl;
    cout <<"\nMinimum grade to pass: ";
    cin >>minimum;
    WriteGrades(marks, howmany, minimum);
}

```

Manipulación completa de arrays

Como hemos ido viendo a lo largo de esta sección, las operaciones legales que podemos realizar sobre la totalidad de un array son las siguientes:

- Inicialización de variables o constantes simbólicas:

```
const TipoEnteros UnArray = {{ 12, 13, -34, 78, ... }} //constante
TipoEnteros miarray = {{ 12, 13, -34, 78, ... }} //variable
```
- Asignación completa:

```
TipoEnteros a, b; //arrays de enteros
a = b; //asignacion completa
```
- Paso como parámetro:

```
MiSubprograma(miarray);
```
- Devolver un array como valor de retorno de una función:

```
miarray = MiFuncion(...);
```

No es legal en cambio:

- Asignación completa de un literal array fuera de su sentencia de definición:
~~```
a = \{\{ 3, 12, 45, ... \}\};
```~~
- Entrada/salida completa de arrays:  
~~```
cin >> a;
cout << a;
```~~

Ejercicios

1. Define un tipo array, TNumeros, para almacenar 4 números enteros. Utilizando bucles, escribe un programa que lea de teclado 4 números y los almacene en el array. Para ello utiliza una función LeerNumeros(...) que los lea de teclado y los ponga en un parámetro de salida. Con el array como parámetro de entrada, después llama a funciones adecuadas que realicen los siguientes cálculos con sus componentes:
 - a) La media.
 - b) El mayor.
 - c) El menor.
 - d) La cuenta de los pares.
 - e) La cuenta de los impares.
2. Rediseña el ejercicio anterior para un máximo de 100 números. El usuario introducirá números enteros por teclado hasta teclear un 0.
3. Escribe un programa que lea una serie de números hasta un 0 y saque en pantalla sólo los números mayores que la media.
4. Escribe un programa que lea N números reales y calcule, por medio de una función, su desviación típica, la cual se define como:

$$\sigma = \sqrt{\frac{\sum_{i=1}^N (x_i - \bar{x})^2}{N}},$$

donde x_i es cada número introducido, y \bar{x} la media de ellos. N se leerá por teclado.

5. Dada la siguiente tabla constante, `LetrasNIF[]`, y sabiendo que la letra correspondiente a un DNI es la que está en la posición igual al resto de la división entera del número de DNI por 23, escribe una función que acepte el número de DNI y devuelva su letra NIF.

```
const char LetrasNIF[] = "TRWAGMYFPDXBNJZSQVHLCKE";
```

5.4. Matrices

Al igual que los campos de un registro, el tipo base de un array puede ser cualquiera. Si el tipo base es un tipo escalar o un tipo registro que no contiene arrays, tenemos un *array unidimensional*. Pero podemos utilizar como tipo base de un array otro tipo array definido anteriormente, con lo que estaríamos *anidando* arrays para obtener *arrays multidimensionales*. Por ejemplo, una *matriz* de números reales se puede definir así:

```
const unsigned NRows=4;
const unsigned NColumns=4;
typedef array<float, NColumns> TRow;
typedef array<TRow, NRows> TMatrix;
TMatrix a, b;      // dos matrices
const TMatrix Id = {{ // una matriz constante
    {{ 1, 0, 0, 0 }},
    {{ 0, 1, 0, 0 }},
    {{ 0, 0, 1, 0 }},
    {{ 0, 0, 0, 1 }}
}};
```

Y un tipo array 3D compuesto por 3 matrices 4×8 sería:

```
const unsigned NRows=4;
const unsigned NColumns=8;
const unsigned NLayers=3;
typedef array<float, NColumns> TRow;
typedef array<TRow, NRows> TLayer;
typedef array<TLayer, NLayers> T3DTable;
```

Para acceder a una componente individual de un *array bidimensional* debemos proporcionar dos índices, uno para la *fila* y otro para la *columna*. El siguiente ejemplo muestra el contenido de la primera fila de una matriz:

```
for (unsigned i=0; i<NColumns; i++) {
    cout <<a[0][i] <<' ';
}
```

Y este otro el contenido de la primera columna:

```
for (unsigned i=0; i<NRows; i++) {
    cout <<a[i][0] <<' ';
}
```

También se puede acceder a diferentes partes de un array multidimensional proporcionando los índices adecuados, y teniendo en cuenta su orden. Por ejemplo, si `a` es una matriz del tipo `TMatrix` definido anteriormente, `a[2]` sería toda la tercera fila. Sin embargo, con el tipo `TMatrix` no es posible referenciar una columna con una sola expresión porque el tipo columna no está definido, y expresiones como `a[][2]` no tienen sentido. Sí podríamos definir una matriz como un array de columnas, pero entonces no podríamos referenciar las filas con una sola expresión. El siguiente ejemplo ilustra el uso de matrices. Nota que en todos los ejemplos vistos de matrices la longitud física coincide con la lógica.

```

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

const unsigned Length = 3; // physical length
typedef array <float, Length> TVector;
typedef array <TVector, Length> TMatrix;

void WriteMat(const TMatrix &m)
{
    int i, j;
    for (i=0; i<Length; i++) {
        for (j=0; j<Length; j++) {
            cout << m[i][j] << ' ';
        }
        cout << endl;
    }
}

TMatrix AddMat(const TMatrix &a, const TMatrix &b)
{
    TMatrix c;
    int i, j;
    for (i=0; i<Length; i++) {
        for (j=0; j<Length; j++) {
            c[i][j] = a[i][j] + b[i][j];
        }
    }
    return c;
}

int main()
{
    TMatrix m1 = {{ {1,2,3}}, {4,5,6}}, {{1,1,1}} };
    TMatrix m2 = {{ {1,0,0}}, {1,1,1}}, {{0,1,0}} };
    TMatrix ms;
    cout << "First matrix:\n";
    WriteMat(m1);
    cout << "Second matrix:\n";
    WriteMat(m2);
    cout << "Their sum:\n";
    ms = AddMat(m1, m2);
    WriteMat(ms);
}

```

El tipo base de una matriz también puede ser char. Estudia el siguiente programa que almacena una partida de ajedrez en una matriz de caracteres 8×8 . Cada casilla contiene un carácter que representa una pieza del juego o la casilla vacía. El programa simplemente pide al usuario movimientos, y muestra a continuación el estado actual de la partida.

```

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

typedef array <char, 8> TRow;

```

```

typedef array <TRow,8> TChessboard;
typedef struct { unsigned row, col; } TSquare;

// White pieces are uppercase, black pieces lowercase
const char WKing = 'K';    const char BKing = 'k';
const char WQueen = 'Q';   const char BQueen = 'q';
const char WRook = 'R';    const char BRook = 'r';
const char WBishop = 'B';  const char BBishop = 'b';
const char WKnight = 'N';  const char BKnight = 'n';
const char WPawn = 'P';    const char BPawn = 'p';
const char None = ' ';

void ShowMatch(const TChessboard &match)
{
    int i, j;
    // first column numbering line
    cout << ' ';
    for (j=0; j<8; j++) {
        cout << ' ' << j+1;
    }
    cout << endl;
    for (i=0; i<8; i++) {
        cout << i+1 << ' '; // line number
        for (j=0; j<8; j++) {
            cout << match[i][j] << ' ';
        }
        cout << i+1 << endl; // another line number
    }
    cout << ' ';
    // last column numbering line
    for (j=0; j<8; j++) {
        cout << ' ' << j+1;
    }
    cout << endl;
}

void ShowHorizLine(unsigned ncells, bool borders)
{
    string chunk = borders ? "|---" : " ---";
    cout << "    ";
    for (unsigned i=0; i<ncells; i++) {
        cout << chunk;
    }
    if (borders) cout << '|';
    cout << endl;
}

void ShowNumbersLine()
{
    cout << "    ";
    for (unsigned j=1; j<=8; j++) {
        cout << " " << j << ' ';
    }
    cout << endl;
}

```

```

}

void ShowChessboard(const TChessboard &match)
{
    int i, j;
    // first column numbering line
    ShowNumbersLine();
    ShowHorizLine(8, false);
    for (i=0; i<8; i++) {
        cout << ' ' << i+1 << " |"; // line number
        for (j=0; j<8; j++) {
            cout << ' ' << match[i][j] << " |";
        }
        cout << ' ' << i+1 << endl; // another line number
        ShowHorizLine(8, false);
    }
    cout << ' ';
    // last numbering line
    ShowNumbersLine();
}

bool ExistsSquare(const TSquare square)
{
    return square.row>0 && square.row<=8 && square.col>0 && square.col<=8;
}

void MovePiece(TChessboard &match)
{
    TSquare from, to;
    cout << "From square (row column)? ";    cin >> from.row >> from.col;
    if (!ExistsSquare(from)) {
        cout << "Square (" << from.row << ', ' << from.col << ") does not exists!\n";
    }
    else if (match[from.row-1][from.col-1]==None) {
        cout << "In square (" << from.row << ', ' << from.col << ") there is nothing!\n";
    }
    else {
        cout << "To square (row column)? ";    cin >> to.row >> to.col;
        if (!ExistsSquare(to)) {
            cout << "Square (" << to.row << ', ' << to.col << ") does not exists!\n";
        }
        else {
            match[to.row-1][to.col-1] = match[from.row-1][from.col-1];
            match[from.row-1][from.col-1] = None;
        }
    }
}

int main()
{
    TChessboard mymatch = {{
        {{ BKing, BKnight, BBishop, BQueen, BKing, BBishop, BKnight, BKing }},
        {{ BPawn, BPawn, BPawn, BPawn, BPawn, BPawn, BPawn, BPawn }},
        {{ None, None, None, None, None, None, None, None }}},

```

```

    {{ None,    None,    None,    None,    None,    None,    None,    None }},
    {{ None,    None,    None,    None,    None,    None,    None,    None }},
    {{ None,    None,    None,    None,    None,    None,    None,    None }},
    {{ WPawn,   WPawn,   WPawn,   WPawn,   WPawn,   WPawn,   WPawn,   WPawn }},
    {{ WRook,   WKnight, WBishop, WQueen, WKing,   WBishop, WKnight, WRook }}
};
char yes_no;
cout << "Initial position:\n\n";
// ShowMatch(mymatch);
ShowChessboard(mymatch);
do {
    cout << "\nWant to move piece (s/n)? ";
    cin >> yes_no;
    if (yes_no=='s' || yes_no=='S') {
        MovePiece(mymatch);
        //ShowMatch(mymatch);
        ShowChessboard(mymatch);
    }
} while (yes_no!='n' && yes_no!='N');
}

```

Ejercicios

1. Define el tipo TMatriz4x4 como un array bidimensional de 4x4 números reales. Utilizando ese tipo, escribe los siguientes subprogramas:

EsSimetrica. Determina si cada elemento (i, j) es igual al elemento (j, i) :

$$a_{ij} = a_{ji}, \quad \forall 1 \leq i, j \leq 4$$

SumaColumna. La suma de todos los valores absolutos de una columna:

$$\sum_{i=0}^4 |a_{ij}|$$

SumaFila. La suma de todos los valores absolutos de una fila:

$$\sum_{j=0}^4 |a_{ij}|$$

Norma1. El máximo de todas las sumas de los valores absolutos de una columna:

$$\max_{1 \leq j \leq 4} \sum_{i=0}^4 |a_{ij}|$$

NormaInf. El máximo de todas las sumas de los valores absolutos de una fila:

$$\max_{1 \leq i \leq 4} \sum_{j=0}^4 |a_{ij}|$$

Mult. El producto de 2 matrices.

2. Amplía el programa de ajedrez para simular el movimiento del caballo. El programa sucesivamente debe:

- (1) Preguntar al usuario por la posición y color del caballo atacante, chequeando posibles errores de rango.
- (2) Preguntar al usuario por la posición destino, chequeando también posibles errores de rango o movimientos no permitidos.
- (3) Ejecutar el movimiento del caballo si todo es correcto y mostrar el nuevo contenido de la partida.

5.5. Cadenas de caracteres

Ya sabemos que en C++ podemos almacenar texto de longitud arbitraria en cadenas de caracteres de tipo `string`. Observa que estas cadenas son en realidad arrays dinámicos porque la longitud física y la lógica coinciden, y el programador puede llamar indistintamente a las funciones `size()` o `length()` para obtener esta longitud. Además, con ayuda de la biblioteca `string`, las siguientes operaciones son legales:

- Definición de constantes simbólicas:
`const string MiNombre = "Jeremias Smith Pitarte";`
- Definición de variables inicializadas:
`string secretario="Dimitriades Canterbury Sanz";`
- Asignación de literales constantes o variables completas:
`empleado = "Peter Man Fred";`
`jefe = secretario;`
- Salida de una cadena completa:
`cout <<MiNombre;`
- Entrada de una cadena completa que no contenga blancos:
`cin >>jefe;`
- Entrada de una cadena completa que puede contener blancos:
`cin.getline(jefe);`
- Paso como parámetro:
`MiSubprograma(empleado)`
- Devolución como valor de retorno de una función:
`empleado = MiFuncion(...);`
- Concatenación de cadenas o caracteres individuales:
`boss = "Yosef "+empleado;`
`plural = "coche '+'s';`
- Comparación alfabética con los operadores relacionales:
`if (empleado>= jefe) ...`

Además, la clase `string` proporciona otras funciones para trabajar con cadenas, como por ejemplo búsqueda de caracteres, inserción, borrado, etc. El apéndice E.2 muestra algunas posibilidades. Al ser `string` una clase de C++ orientada al objeto, las llamadas a sus subprogramas siguen la sintaxis típica de la orientación a objetos de C++.

Ejercicio

1. Escribe un programa que cuente el número de vocales (minúsculas y mayúsculas) de un texto leído de teclado y terminado en un punto. Se debe leer todo el texto en una sola variable de tipo `string` línea a línea con la función `getline`. Una vez leído, el programa buscará y contará las vocales utilizando una de las funciones de búsqueda de la biblioteca estándar `string`.

5.6. Anidamientos

Puesto que C++ no impone restricciones en cuanto al tipo base de un array, tenemos una gran flexibilidad para definir complejas *estructuras de datos* utilizando *anidamientos* de arrays y registros. A continuación ilustramos las más habituales.

Anidamiento dentro de un registro

Una de las estructuras más simples son registros anidados dentro de otros registros. Utilizando el símil de relaciones entre los componentes de una familia, el acceso a un campo individual se logra especificando todos sus registros *ascendentes* con el operador punto `'.'`. La sintaxis de acceso sería:

```
padre.hijo.nieto.biznieto.tataranieto ...
```

El siguiente subprograma, por ejemplo, usa el tipo TDate anidado dentro del tipo TPerson de la sección 5.2 para imprimir en pantalla el día, mes y año de nacimiento de una persona.

```
void WriteDate(const TPerson &p)
{
    cout <<p.birth.day<< '/' <<p.birth.month<< '/' <<p.birth.year;
}
```

También podemos anidar arrays dentro de un registro. La sintaxis de acceso a un componente del array seguirá la sintaxis:

```
registro.arraydedatos[indice]
```

El siguiente programa extiende el de la sección 5.3 que almacenaba las notas de estudiantes, pero esta vez el número de estudiantes, el cual determina la longitud lógica del array, se almacena junto al array en un registro, de forma que ya no es necesario pasar esa longitud como parámetro independiente a los subprogramas. Nota que para evitar copias en memoria innecesarias, los arrays de entrada se pasan por referencia constante.

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

// array type
const unsigned MaxStudents = 100;
typedef array <float, MaxStudents> TMarks;
// array with actual length
typedef struct {
    TMarks marks;
    unsigned length;
} TGrades;

void ReadGrades(TGrades &grades)
{
    unsigned n, i=0;
    cout <<"How many students: ";
    cin >>n;
    do {
        i++;
        cout <<"Grade student " <<i <<": ";
```

```

    cin >> grades.marks[i-1];
} while (i<n);
grades.length = n;
}

float MeanGrade(const TGrades &grades)
{
    float sum=0;
    unsigned i;
    for (i=0; i<grades.length; i++) {
        sum += grades.marks[i];
    }
    return sum/grades.length;
}

float MaxGrade(const TGrades &grades)
{
    float maximum=0;
    unsigned i;
    for (i=0; i<grades.length; i++) {
        if (grades.marks[i]>maximum)
            maximum = grades.marks[i];
    }
    return maximum;
}

int main()
{
    TGrades grades;
    ReadGrades(grades);
    cout <<"The mean grade is " <<MeanGrade(grades) <<endl;
    cout <<"The maximum grade is " <<MaxGrade(grades) <<endl;
}

```

Anidamientos dentro de un array

Una estructura muy habitual son los arrays de registros. Por ejemplo, un array unidimensional de datos personales.

| | | | | | | |
|-----------|------------|------------|-----------|-----------|-----|-----------|
| 23415678X | 234182378M | 14415678G | 86715678J | 78675678P | ... | 56116678Z |
| Saturnino | Hurraca | Restituto | Elisenda | Eufemia | ... | Indalina |
| Diaz | Saenz | Lopez | Perez | Gonzalez | ... | Martinez |
| 9/6/1955 | 28/12/2009 | 12/10/2007 | 3/8/2001 | 18/8/1965 | ... | 2/6/1988 |

La definición del registro anidado debe hacerse *antes* que la definición del registro o array anfitrión:

```

typedef struct {
    int day, month, year;
} TDate;

typedef struct {
    string id;
    string first_name, last_name;
    TDate birth;
}

```

```


} TPerson;

const unsigned NPeople = 1000; // la longitud fisica
typedef array <TPerson, NPeople> TPeople; // el tipo array
TPeople plantilla; // el array

```

La sintaxis de acceso a campos de un registro individual dentro de un array es:

```
miarray[indice].campo
```

 Como norma general para una escritura correcta de accesos a estructuras de datos arbitrariamente complejas, después de un dato de tipo array se escribirá el operador corchete [], y después de un dato de tipo registro se escribirá el nombre de uno de sus campos.

El siguiente procedimiento accede a campos de registros contiguos de un array. Observa que es responsabilidad de la llamada que la longitud lógica *howmany* sea correcta.

```

void WritePeopleNames(const TPeople &everyone, const unsigned howmany)
{
    for (unsigned i=0; i<howmany; i++) {
        cout <<everyone[i].first_name <<' ' <<everyone[i].last_name <<endl;
    }
}

```

Cuando se trabaja con arrays incompletos, las componentes vacías se pueden manejar con estrategias muy diversas. Dos estrategias habituales son:

- Mantener las componentes *contiguas*, lo que implica que hay que guardar la longitud lógica del array en algún sitio. Esta es la estrategia que hemos venido utilizando hasta ahora.
- Permitir que las componentes vacías estén mezcladas con las que contienen valores válidos, en cuyo caso ya no tiene sentido la longitud lógica, y los bucles que recorren el array deben utilizar como límite la longitud física. En este caso, las componentes vacías deben *marcarse* de alguna manera con algún valor especial que contengan, valor que servirá para detectarlas con la sentencia de selección adecuada dentro de los bucles.

El siguiente programa utiliza esta segunda estrategia, y los componentes vacíos se marcan con la cadena vacía "" en el campo identificador de cada registro. Contabilizando los registros no marcados también se puede determinar el número de elementos del array.

```

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

// individual data type
typedef struct {
    int day, month, year;
} TDate;
typedef struct {
    string id;
    string first_name, last_name;
    TDate birth;
} TPerson;
// the array of data type
const unsigned MaxPeople = 10;

```

```

// an empty id (") marks a non-existing person
typedef array <TPerson, MaxPeople> TPeople;

void ReadPerson(TPerson &p)
{
    cout <<"Id: ";
    cin >>p.id; // no spaces inbetween
    cout <<"First name: ";
    cin >>ws; // skip last user <enter> from keyboard buffer
    getline(cin, p.first_name); // read until \n
    cout <<"Last name: ";
    // no need to remove last \n, already read by last getline
    getline(cin, p.last_name); // read until \n
    cout <<"Date of birth (dd mm yyyy): ";
    cin >>p.birth.day>>p.birth.month>>p.birth.year;
}

void WritePerson(const TPerson &p)
{
    cout <<"Id: " <<p.id <<endl
        <<"First name: " <<p.first_name <<endl
        <<"Last name: " <<p.last_name <<endl
        <<"Date of birth (dd/mm/yyyy): "
        <<p.birth.day<< '/' <<p.birth.month<< '/' <<p.birth.year <<endl;
}

void WritePerson1Line(const TPerson &p)
// without labels, in 1 line
{
    cout <<p.id << ", "
        <<p.first_name << ", "
        <<p.last_name << ", "
        <<p.birth.day<< '/' <<p.birth.month<< '/' <<p.birth.year<<endl;
}

unsigned NPeople(const TPeople &people)
{
    unsigned n=0;
    for (unsigned i=0; i<MaxPeople; i++) {
        if (!people[i].id.empty()) {
            n++;
        }
    }
    return n;
}

void WritePeople(const TPeople &people)
{
    unsigned i;
    for (i=0; i<MaxPeople; i++) {
        //empty is defined in the string library
        if (!people[i].id.empty()) {
            WritePerson1Line(people[i]);
        }
    }
}

```

```

    }
}

bool SearchPerson(const TPeople &people, const string wanted,
                 TPerson &who, unsigned &position)
// return true if found, false otherwise
{
    bool found=false;
    unsigned i=0;
    while (!found && i<MaxPeople) {
        // strings can be compared with primitive operators
        if (people[i].id == wanted) {
            who = people[i];
            position = i;
            found = true;
        }
        else {
            i++;
        }
    }
    return found;
}

bool InsertPerson(TPeople &people, const TPerson &person)
// return true if there has been enough space, false otherwise
{
    bool inserted;
    unsigned i=0;
    // search for an empty element
    while (!people[i].id.empty()) {
        i++;
    }
    // check if the maximum number of elements have been reached
    if (i<MaxPeople) {
        people[i] = person;
        inserted = true;
    }
    else {
        inserted = false;
    }
    return inserted;
}

void RemovePerson(TPeople &people, const unsigned position)
// premise: the position to remove has been previously checked
{
    // just mark it as empty
    people[position].id = "";
}

/*
 * Array of records with personal data:
 *     A new record is inserted, and an old one is removed,
 *     as desired by the user
 */

```

```

*
* Unused records are marked with an empty identifier "",
* and can be mixed among the used ones.
*/
int main()
{
    TPerson another, who;
    string id;
    unsigned where;
    // the order of initialized data must conform the order of definition
    TPeople everyone = {{ // data
        { "1A", "Saturnino", "Diaz",    { 9/ 6/1955} },
        { "", "Noname", "Noone",    { 9/ 6/1955} }, //empty
        { "2B", "Hurraca", "Saenz",    {28,12,2009} },
        { "3C", "Restituto", "Lopez",    {12,10,2007} },
        { "4D", "Elisenda", "Perez",    { 3, 8,2001} },
        { "", "Anyone", "Someone",    { 3, 8,2001} }, // empty
        { "5E", "Eufemia", "Gonzalez", {18, 8,1965} },
        { "6F", "Indalina", "Martinez", { 2, 6,1988} }
    }};
    cout <<"Everyone includes " <<NPeople(everyone)<<" people:\n";
    WritePeople(everyone);

    // accept data of a new person
    cout <<"\nA new person to insert\n";
    ReadPerson(another);
    // try to insert the new data
    if (!InsertPerson(everyone, another)) {
        cout <<"Sorry! Not enough space for a new person\n";
    }
    else {
        cout <<"Ok, inserted\n";
    }

    // search and remove a person
    cout <<"\nIdentity of person to remove: "; cin >>id;
    if (SearchPerson(everyone, id, who, where)) {
        cout <<"\nThe following person is to be removed:\n";
        WritePerson(who);
        RemovePerson(everyone, where);
    }
    else {
        cout <<"Sorry! Identificator "<<id<<" not found\n";
    }
    cout <<"\nNow everyone includes:\n";
    WritePeople(everyone);
}

```

Anidamiento de un array de registros dentro de otro registro

El siguiente programa es el mismo ejemplo que el anterior, pero utilizando la estrategia de mantener contiguos los registros válidos. Además, la longitud lógica se almacena anidando ésta junto al array de datos en otro registro mayor que contiene la estructura de datos completa. Ahora la sintaxis de acceso a

uno de los campos de un registro individual sería:

```
estructura.arraydedatos[indice].campo
```

Si queremos referenciar un registro individual completo, simplemente omitimos el último operador punto '.' y el nombre del campo individual:

```
estructura.arraydedatos[indice]
```

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

// individual data type
typedef struct {
    int day, month, year;
} TDate;
typedef struct {
    string id;
    string first_name, last_name;
    TDate birth;
} TPerson;
// the array of data type
const unsigned MaxPeople = 10;
typedef array <TPerson, MaxPeople> TData;
// the whole type
typedef struct {
    unsigned npeople;
    TData data;
} TPeople;

void ReadPerson(TPerson &p)
{
    cout <<"Id: ";
    cin >>p.id; // no spaces inbetween
    cout <<"First name: ";
    cin >>ws; // skip last user <enter> from keyboard buffer
    getline(cin, p.first_name); // read until \n
    cout <<"Last name: ";
    // no need to remove last \n, already read by last getline
    getline(cin, p.last_name); // read until \n
    cout <<"Date of birth (dd mm yyyy): ";
    cin >>p.birth.day>>p.birth.month>>p.birth.year;
}

void WritePerson(const TPerson &p)
{
    cout <<"Id: " <<p.id <<endl
        <<"First name: " <<p.first_name <<endl
        <<"Last name: " <<p.last_name <<endl
        <<"Date of birth (dd/mm/yyyy): "
        <<p.birth.day<< '/' <<p.birth.month<< '/' <<p.birth.year <<endl;
}
```



```

void WritePerson1Line(const TPerson &p)
// without labels, in 1 line
{
    cout <<p.id <<" "
         <<p.first_name <<" "
         <<p.last_name <<" "
         <<p.birth.day<<'/'<<p.birth.month<<'/'<<p.birth.year<<endl;
}

void WritePeople(const TPeople &people)
{
    unsigned i;
    for (i=0; i<people.npeople; i++) {
        WritePerson1Line(people.data[i]);
    }
}

bool SearchPerson(const TPeople &people, const string wanted,
                 TPerson &who, unsigned &position)
// return true if found, false otherwise
{
    bool found=false;
    unsigned i=0;
    while (!found && i<people.npeople) {
        // strings can be compared with primitive operators
        if (people.data[i].id == wanted) {
            who = people.data[i];
            position = i;
            found = true;
        }
        else {
            i++;
        }
    }
    return found;
}

bool InsertPerson(TPeople &people, const TPerson &person)
// return true if there has been enough space, false otherwise
{
    bool inserted;
    // the logical length cannot exceed the available physical space
    if (people.npeople < MaxPeople) {
        people.data[people.npeople] = person;
        people.npeople++; // another record is busy
        inserted = true;
    }
    else {
        inserted = false;
    }
    return inserted;
}

```

```

void RemovePerson(TPeople &people, const unsigned position)
// premise the position to remove has been previously checked
{
    // just put the last one on top of the one to remove
    TPerson last = people.data[people.npeople-1];
    people.data[position] = last;
    people.npeople--;
}

/*
 * Array of records with personal data:
 *     A new record is inserted, and an old one is removed, as desired by the user
 *
 * Unused records are kept at the end,
 * maintaining the valid ones contiguous at the beginning.
 */
int main()
{
    TPerson another, who;
    string id;
    unsigned where;
    // the order of initialized data must conform the order of definition
    TPeople everyone = {
        6, // npeople
        {{ // data
            { "1A", "Saturnino", "Diaz",    { 9/ 6/1955} },
            { "2B", "Hurraca",  "Saenz",    {28,12,2009} },
            { "3C", "Restituto", "Lopez",    {12,10,2007} },
            { "4D", "Elisenda",  "Perez",    { 3, 8,2001} },
            { "5E", "Eufemia",   "Gonzalez", {18, 8,1965} },
            { "6F", "Indalina",  "Martinez", { 2, 6,1988} }
        }}
    };
    cout <<"Everyone includes:\n";
    WritePeople(everyone);

    // accept data of a new person
    cout <<"\nA new person to insert\n";
    ReadPerson(another);
    // try to insert the new data
    if (!InsertPerson(everyone, another)) {
        cout <<"Sorry! Not enough space for a new person\n";
    }
    else {
        cout <<"Ok, inserted\n";
    }

    // search and remove a person
    cout <<"\nIdentity of person to remove: "; cin >>id;
    if (SearchPerson(everyone, id, who, where)) {
        cout <<"\nThe following person is to be removed:\n";
        WritePerson(who);
        RemovePerson(everyone, where);
    }
}

```

```

else {
    cout <<"Sorry! Identificator "<<id<<" not found\n";
}
cout <<"\nNow everyone includes:\n";
WritePeople(everyone);
}

```

Ejercicios de repaso

1. Define un tipo array para almacenar hasta un máximo de 100 enteros, TNumeros, teniendo en cuenta que la longitud lógica debe actualizarse correctamente cada vez que se modifica. Para ello, almacena esta longitud junto al array de datos en un registro de tipo TZs. Usando esta estructura de datos, escribe subprogramas para las operaciones que se enumeran a continuación.

LeerZs Función para leer números de teclado en una estructura TZs hasta introducir un cero. El 0 final no se guardará.

EscribirZs Procedimiento que escribe en pantalla los n primeros números almacenados en el array de una estructura TZs.

AnadirZs Función que añade los n primeros números de una estructura TZs al final de otra estructura TZs destino. Debe devolver `true` si los datos del array fuente han cabido totalmente en el array destino, y `false` si no ha sido así.

EliminarZ Procedimiento que elimina el valor indicado en una cierta posición del array de una estructura TZs. Para evitar muchos movimientos, puedes guardar el valor de la última posición en la que se elimina.

VoltearZs Procedimiento que da la vuelta al contenido del array de una estructura TZs de forma que el primero debe quedar el último, el segundo el penúltimo, y así sucesivamente hasta llegar al medio. Sólo debe tenerse en cuenta los elementos que están dentro de la longitud lógica. No debe utilizarse otro array temporal para guardar resultados intermedios.

Utilizando los subprogramas anteriores que necesites, escribe un programa principal que lea dos vectores de teclado, añada el segundo al final del primero, le dé la vuelta a éste, y finalmente lo escriba en pantalla. Después de escribirlo en pantalla, vacíalo eliminando uno a uno todos sus valores, y vuelve a escribirlo vacío en pantalla.

2. Desarrolla un programa que defina un array para un máximo de 100 números enteros y lo mantenga ordenado al realizar las siguientes operaciones:

EstaOrdenado Función lógica que devuelve `true` si el array está correctamente ordenado, y `false` si no lo está.

InsertarOrdenado Función lógica que inserta un número entero en la posición adecuada manteniendo el orden ascendente del array. Apóyate en otro subprograma **AbrirHueco** que desplace todos los números un lugar a la derecha a partir de una posición dada. Se debe devolver `true` si ha habido espacio para el número insertado y `false` si no ha habido espacio.

Eliminar Procedimiento que elimina un elemento del array. Por supuesto, se deben mantener contiguos los elementos después de la eliminación.

Escribe un programa principal que lea números enteros repetitivamente y los inserte de forma ordenada en un array cuyo contenido inicial es

{0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100}.

La lectura de números acabará con el valor centinela 0, el cual no se almacena. Después elimina del array los elementos que están en las posiciones pares. Finalmente, haz una llamada a

EstaOrdenado para confirmar que el array efectivamente está ordenado mostrando su contenido con otro procedimiento Escribir. Si no es así, se debe imprimir un mensaje de error interno (el cual no debería mostrarse nunca si está bien hecho).

3. Escribe un programa que lea de teclado líneas de texto hasta llegar a un punto. Las líneas de texto se irán añadiendo a una variable de tipo `string`. El programa reemplazará todas las cadenas “EUP” del texto por la cadena “Politecnica” utilizando la función `replace` de la biblioteca estándar `string`.
4. Define dos tipos de datos para almacenar matrices de números reales de dimensiones 2×2 y 3×3 . Utilizando estos tipos, define e implementa los siguientes subprogramas:

Det2x2. Función que devuelve el determinante de una matriz 2×2 .

Det3x3. Función que devuelve el determinante de una matriz 3×3 .

Menor. Función que devuelve el *menor complementario* (i, j) de una matriz 3×3 . El menor complementario del elemento (i, j) es el determinante de la matriz 2×2 que resulta de eliminar la fila i y la columna j de la matriz original.

MCofactores. Función que devuelve la *matriz de cofactores* de una matriz 3×3 . Cada elemento (i, j) de esta matriz es el menor complementario del elemento (i, j) de la matriz original, con la salvedad de que el signo se cambia para todos los elementos cuya suma de índices $i + j$ sea impar.

Traspuesta. Función que devuelve la *traspuesta* de una matriz 3×3 , esto es, cada elemento (i, j) pasa a ser el (j, i) .

Adjunta. Función que devuelve matriz *adjunta* de una matriz 3×3 , que se define como la traspuesta de su matriz de cofactores.

Inversa. Función que devuelve la *inversa* de una matriz 3×3 A , que se define como:

$$A^{-1} = \frac{1}{|A|} \text{Adj}(A)$$

Utilizando estas funciones, escribe un programa completo que lea de teclado una matriz 3×3 y escriba en pantalla su inversa.

5. Un programa lee las distancias en kilómetros que separan todas las estaciones de una línea de tren. La línea de tren no puede contener más de 20 estaciones, pero puede contener menos, marcando con un cero la última estación. Escribe el programa para que saque en pantalla una matriz triangular con las distancias entre cualquier par de estaciones. Por ejemplo, si se introducen las distancias 2.25, 3.25, 4.5, 2.0, se sacará la matriz:

| | 1 | 2 | 3 | 4 | 5 |
|---|------|------|-----|---|---|
| 1 | 0 | | | | |
| 2 | 2.25 | 0 | | | |
| 3 | 5.5 | 3.25 | 0 | | |
| 4 | 10 | 7.75 | 4.5 | 0 | |
| 5 | 12 | 9.75 | 6.5 | 2 | 0 |

6. Basado en el concepto matemático de *conjunto*, define el tipo `ConjuntoZ` para almacenar números enteros distintos entre el 0 y el 100. La implementación debe utilizar un array de valores lógicos de longitud física 101, de forma que si un número particular pertenece al conjunto, la posición que tiene como índice ese número debe estar a `true`, y si el número no pertenece al conjunto, esa posición debe estar a `false`.

Implementa después las operaciones típicas entre conjuntos que se enumeran a continuación utilizando el tipo `ConjuntoZ`:

Vacio. Devuelve el conjunto vacío.

Cardinalidad. Devuelve el número de elementos de un conjunto.

Pertenece. Devuelve `true` si un determinado número pertenece al conjunto, y `false` si no.

Incluido. Devuelve `true` si todos los elementos de un conjunto están dentro de otro.

Incluir. Incluye un número dentro de un conjunto.

Excluir. Excluye un elemento del conjunto, si es que está.

Union. Devuelve la unión de dos conjuntos.

Interseccion. Devuelve la intersección de dos conjuntos.

Utilizando estos subprogramas siempre que sea posible, y dos subprogramas adicionales para leer de teclado un conjunto de números y para mostrarlo en pantalla, escribe un programa que (1) lea dos conjuntos de teclado; (2) imprima en pantalla la unión e intersección de ellos; y (3) determine si uno está incluido en el otro.

7. Define un tipo registro, `TUnElectrodomestico`, para almacenar la siguiente información sobre el consumo mensual de un electrodoméstico eléctrico:

- Nombre (alfabético);
- Potencia (vatios);
- Número de minutos en funcionamiento.

Define una estructura de datos, `TElectrodomesticos`, para contener la información de hasta un máximo de 100 electrodomésticos en una casa.

Escribe una función, `CosteConsumo`, que calcule el costo por el consumo de todos los electrodomésticos de una casa. Un parámetro de entrada será la estructura de datos con toda la información necesaria, mientras que un array de números reales de salida, que deberá definirse convenientemente, contendrá los costes individuales por aparato. Por ejemplo, siendo el precio de cada kilowatio-hora de 0.1364 euros, el coste del consumo de un aparato de 500 vatios de potencia que ha estado en funcionamiento durante 390 minutos (= 6,5 horas) en un mes será de

$$C = \frac{500}{1000} \times 6,5 \times 0,1364.$$

Escribe un programa que inicialice una estructura `TElectrodomesticos` con valores para 10 aparatos de una casa. Después de llamar a la función `CosteConsumo` con los parámetros adecuados, el programa debe mostrar en pantalla el coste total del consumo mensual de todos los aparatos de la casa.

Utilizando algún algoritmo de ordenación, amplía el programa para que imprima en pantalla los costes individuales por aparato ordenados de mayor a menor coste.

Apéndice A

Palabras reservadas de C++

Palabras reservadas más habituales de C++. No pueden ser usadas como identificadores.

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

Apéndice B

Conjunto de caracteres ASCII

| carácter | código | carácter | código | carácter | código | carácter | código |
|----------|--------|----------|--------|----------|--------|----------|--------|
| <NUL> | 0 | ! | 33 | A | 65 | a | 97 |
| <SOH> | 1 | " | 34 | B | 66 | b | 98 |
| <STX> | 2 | # | 35 | C | 67 | c | 99 |
| <ETX> | 3 | \$ | 36 | D | 68 | d | 100 |
| <EOT> | 4 | % | 37 | E | 69 | e | 101 |
| <ENQ> | 5 | & | 38 | F | 70 | f | 102 |
| <ACK> | 6 | ' | 39 | G | 71 | g | 103 |
| <BEL> | 7 | (| 40 | H | 72 | h | 104 |
| <BS> | 8 |) | 41 | I | 73 | i | 105 |
| <HT> | 9 | * | 42 | J | 74 | j | 106 |
| <LF> | 10 | + | 43 | K | 75 | k | 107 |
| <VT> | 11 | , | 44 | L | 76 | l | 108 |
| <FF> | 12 | - | 45 | M | 77 | m | 109 |
| <CR> | 13 | . | 46 | N | 78 | n | 110 |
| <SO> | 14 | / | 47 | O | 79 | o | 111 |
| <SI> | 15 | 0 | 48 | P | 80 | p | 112 |
| <DLE> | 16 | 1 | 49 | Q | 81 | q | 113 |
| <DC1> | 17 | 2 | 50 | R | 82 | r | 114 |
| <DC2> | 18 | 3 | 51 | S | 83 | s | 115 |
| <DC3> | 19 | 4 | 52 | T | 84 | t | 116 |
| <DC4> | 20 | 5 | 53 | U | 85 | u | 117 |
| <NAK> | 21 | 6 | 54 | V | 86 | v | 118 |
| <SYN> | 22 | 7 | 55 | W | 87 | w | 119 |
| <ETB> | 23 | 8 | 56 | X | 88 | x | 120 |
| <CAN> | 24 | 9 | 57 | Y | 89 | y | 121 |
| | 25 | : | 58 | Z | 90 | z | 122 |
| <SUB> | 26 | ; | 59 | [| 91 | { | 123 |
| <ESC> | 27 | < | 60 | \ | 92 | | 124 |
| <FS> | 28 | = | 61 |] | 93 | } | 125 |
| <GS> | 29 | > | 62 | ^ | 94 | ~ | 126 |
| <RS> | 30 | ? | 63 | _ | 95 | | 127 |
| <US> | 31 | @ | 64 | ` | 96 | | |
| <SP> | 32 | | | | | | |

Tabla B.1: Códigos ASCII en decimal.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|-------|-------|------|---|---|---|---|-------|
| 0 | <NUL> | <DLE> | <SP> | 0 | @ | P | ` | p |
| 1 | <SOH> | <DC1> | ! | 1 | A | Q | a | q |
| 2 | <STX> | <DC2> | " | 2 | B | R | b | r |
| 3 | <ETX> | <DC3> | # | 3 | C | S | c | s |
| 4 | <EOT> | <DC4> | \$ | 4 | D | T | d | t |
| 5 | <ENQ> | <NAK> | % | 5 | E | U | e | u |
| 6 | <ACK> | <SYN> | & | 6 | F | V | f | v |
| 7 | <BEL> | <ETB> | ' | 7 | G | W | g | w |
| 8 | <BS> | <CAN> | (| 8 | H | X | h | x |
| 9 | <HT> | |) | 9 | I | Y | i | y |
| A | <LF> | <SUB> | * | : | J | Z | j | z |
| B | <VT> | <ESC> | + | ; | K | [| k | { |
| C | <FF> | <FS> | , | < | L | \ | l | |
| D | <CR> | <GS> | - | = | M |] | m | } |
| E | <SO> | <RS> | . | > | N | ^ | n | ~ |
| F | <SI> | <US> | / | ? | O | _ | o | |

Tabla B.2: Códigos ASCII en hexadecimal.

Apéndice C

Tipos predefinidos de C++

Los dominios especificados a continuación son típicos de una máquina con una longitud de palabra de 32 bits.

C.1. Números enteros

| tipo | número de bytes | dominio |
|--------------------|-----------------|-------------------------|
| short | 2 | $[-2^{16} - 1, 2^{16}]$ |
| int | 4 | $[-2^{32} - 1, 2^{32}]$ |
| long | 4 | $[-2^{32} - 1, 2^{32}]$ |
| long long | 8 | $[-2^{64} - 1, 2^{64}]$ |
| unsigned short | 2 | $[0, 2^{16}]$ |
| unsigned | 4 | $[0, 2^{32}]$ |
| unsigned long | 4 | $[0, 2^{64}]$ |
| unsigned long long | 8 | $[0, 2^{64}]$ |

Operadores ordenados por orden de precedencia descendente.

| | |
|-----------------|-------------------|
| multiplicativos | \star / $\%$ |
| aditivos | $+$ $-$ |
| relacionales | $<$ $<=$ $>$ $>=$ |
| igualdad | $==$ $!=$ |
| asignación | $=$ |

C.2. Números reales

| tipo | número de bytes | valor abs. mín. | valor abs. máx. |
|-------------|-----------------|-----------------|-----------------|
| float | 4 | $1,175^{-38}$ | $3,403^{+38}$ |
| double | 8 | $2,225^{-308}$ | $1,798^{+308}$ |
| long double | 16 | $3,362^{-4932}$ | $1,190^{+4932}$ |

Operadores ordenados por orden de precedencia descendente.

| | |
|-----------------|-----------|
| multiplicativos | * / |
| aditivos | + - |
| relacionales | < <= > >= |
| igualdad | == != |
| asignación | = |

C.3. Caracteres

| tipo | número de bytes | dominio |
|---------------|-----------------|-------------------|
| char | 1 | $[-2^7, 2^7 - 1]$ |
| unsigned char | 1 | $[0, 2^8 - 1]$ |

Por orden de precedencia descendiente se pueden utilizar los siguientes.

| | |
|--------------|-----------|
| relacionales | < <= > >= |
| igualdad | == != |
| asignación | = |

Y los operadores para cadenas de tipo `string`.

| | |
|---------------|-----------|
| concatenación | + |
| relacionales | < <= > >= |
| igualdad | == != |
| asignación | = |

C.4. Lógicos

| tipo | número de bytes | dominio |
|------|-----------------|---------------|
| bool | 1 | {false, true} |

Operadores permitidos.

| | |
|--------------|-----------|
| relacionales | < <= > >= |
| igualdad | == != |
| asignación | = |

Apéndice D

Secuencias de escape de C++

| Hex. | Dec. | Secuencia de escape | Significado |
|------------|------|---------------------|--|
| 07 | 7 | \a | campana |
| 08 | 8 | \b | retroceso |
| 09 | 9 | \t | tabulador |
| 0A | 10 | \n | nueva línea |
| 0B | 11 | \v | tabulador vertical |
| 0C | 12 | \f | nueva página |
| 0D | 13 | \r | retorno de carro |
| 22 | 34 | \" | comillas dobles |
| 27 | 39 | \' | apóstrofe |
| 3F | 63 | \? | interrogación |
| 5C | 92 | \\ | barra invertida |
| cualquiera | | \ooo | código ASCII octal |
| cualquiera | | \xhh | código ASCII hexadecimal precedido por x |

Apéndice E

Bibliotecas estándares

E.1. Entrada/salida básica de C++

```
#include <iostream>
```

Entrada estándar (teclado o fichero)

Lectura de texto y números saltando blancos iniciales:

```
cin >>variable1 >>variable2 >> ...
```

Los números octales leídos se preceden por un 0, y los hexadecimales por el par 0x.

Lectura de caracteres sin saltarse blancos iniciales. Dos alternativas:

```
char ch;  
ch = cin.get()  
cin.get(ch)
```

Lectura de una línea de texto completa o hasta max caracteres en una cadena:

```
getline(cin, str)  
cin.getline(str)  
cin.getline(str, max)
```

Saltarse blancos residuales de entradas previas (útil para llamar a `getline` tras leer con `>>`):

```
cin >>ws
```

Ignorar caracteres hasta un carácter residual de nueva línea:

```
cin.ignore(max_a_ignorar, '\n')
```

Salida estándar (pantalla o fichero)

Salida de expresiones numéricas y de caracteres:

```
cout <<expresion1 <<expresion2 <<...
```

Actualización de la salida (normalmente se espera hasta un carácter de nueva línea):

```
cout <<flush
```

Salida de un carácter *fin de línea*. Dos alternativas:

```
cout <<endl  
cout <<'\n'
```

Los *manipuladores* son funciones complementarias de la biblioteca `iomanip` que permiten una entrada/salida mejor formateada (especialmente salida). En general, cambian estados *persistentes* en el sentido de que permanecen en efecto hasta que se realizan nuevas llamadas. Las que no son persistentes sólo afectan a la operación de entrada/salida siguiente.

```
#include <iomanip>
```

Entrada estándar (teclado o fichero)

Interpretar siguiente número como decimal o hexadecimal:

```
cin >>dec >>numero
cin >>hex >>numero
```

Salida estándar (pantalla o fichero)

Salida de números enteros en formato decimal (por omisión), octal o hexadecimal:

```
cout <<oct <<expresion_entera
cout <<hex <<expresion_entera
cout <<dec <<expresion_entera
```

Salida de números reales en coma fija, científica, o en el formato más corto:

```
cout <<fixed <<expresion_real
cout <<scientific <<expresion_real
cout <<automatic <<expresion_real
```

Alineamiento a la izquierda o a la derecha:

```
cout <<left <<expresion
cout <<right <<expresion
```

Establecimiento de la anchura a w caracteres (no es persistente, por omisión 0):

```
cout <<setw(w) <<expresion_numerica
```

Establecimiento del número de dígitos significativos a p (por omisión 6):

```
cout <<setprecision(p) <<expresion_real
```

Establecimiento del carácter de relleno a c (por omisión el espacio):

```
cout <<setfill(c) <<expresion_numerica
```

E.2. La biblioteca de cadenas de caracteres de C++

```
#include <string>
```

En las siguientes tablas, las variables s, s1, y s2, se suponen de tipo `string`; las variables i, n, r, p, e ini, de tipo `int`; y la variable c, de tipo `char`.

Definiciones

Variables o constantes simbólicas inicializadas o no:

```
string s, s1, s2;
string s2="Este valor se puede cambiar.";
const string s3="!`Este valor no se puede cambiar!";
```

Incluso se puede inicializar con el contenido de otra cadena ya inicializada:

```
string s(s2); //s es la nueva
```

Entrada/salida

Entrada de teclado o salida en pantalla:

```
cin >>s; //sin incluir blancos
getline(cin, s); //1\'linea completa incluyendo blancos
cout <<s;
```

Tamaño

Longitud lógica:

```
s.length()
s.size()
```


Asignación

Asignación de cadenas completas o caracteres individuales:

```
s = "cadena";  
s = s2;  
s[i] = c; //sin comprobar errores de rango  
s.at(i) = c; //comprobando errores de rango
```

Longitud

Acceso a una posición particular:

```
c = s[i]; //sin comprobar rangos  
c = s.at(i); //comprobando rangos  
c = s.at(s.length()-1); //ultimo caracter
```

Selección de una subcadena:

```
s.substring(pos) //desde s[pos] al final  
s.substring(0, len) //desde el principio hasta s[len-1]  
s.substring(pos, len) //desde s[pos] a s[pos+len-1]
```

Comparaciones

Operadores relacionales y de igualdad. Por ejemplo:

```
s == s2  
s < s2 //segun la ordenacion ASCII)
```

Comparación estilo lenguaje C:

```
s.compare(s2) //0 si s==s2; -1 si s<s2; 1 si s>s2
```

Modificación

Concatenación, adición e inserción:

```
s += s2; //s2 asignada al final de s  
s = s1+s2; //concatenacion de s1 y s2 asignada a s  
s = c; //caracter c asignado al final de s  
s.append(s1+s2); //adicion de s1+s2 al final de s  
s.append(r, c); //caracter c asignado r veces al final de s  
s.insert(ini, s2); //insercion de s2 en la posicion ini de s  
s.insert(ini, r, c); //c insertado r veces en posicion ini de s
```

Eliminación:

```
s.erase(ini); //Eliminacion de todos los caracteres de s desde ini  
s.erase(ini, n); //Eliminacion de n caracteres de s desde ini
```

Búsqueda y sustitución

Búsqueda del carácter o cadena sc en s:

```
pos = s.find(sc); //la busqueda empieza en el principio  
pos = s.find(sc, ini); //la busqueda empieza en posicion ini
```

Búsqueda en s de cualquier carácter de sc y retorno de la posición:

```
pos = s.find_first_of(sc); //busqueda empieza en el principio  
pos = s.find_first_of(sc, ini); //empieza en posicion ini
```

Búsqueda en s de cualquier carácter que no esté en sc y retorno de la posición:

```
pos = s.find_first_not_of(sc); //busqueda desde el principio  
pos = s.find_first_not_of(sc, ini); //desde posicion ini
```

Sustitución de n caracteres de s desde la posición ini...

```
s.replace(ini,n,s2); //sustitucion por s2  
s.replace(ini,n,r,c); //sustitucion por r copias de c
```

Todas las funciones de búsqueda y sustitución devuelven `string::npos` si falla la búsqueda.

E.3. La biblioteca estándar de C

La biblioteca estándar de C `stdlib` incluye utilidades de propósito general como generación de números aleatorios, búsqueda y sustitución en arrays, o conversión de datos entre distintos tipos. Aquí sólo mencionamos cuatro de ellas.

```
#include <stdlib>
```

| | |
|------------------------------|---|
| <code>rand()</code> | Devuelve un número pseudo-aleatorio entre 0 y <code>RAND_MAX</code> . |
| <code>srand(seed)</code> | Inicializa el generador aleatorio con un valor entero. |
| <code>abs(n)</code> | Valor absoluto de un número entero. |
| <code>system(comando)</code> | Invoca la línea de comandos del sistema para ejecutar un <i>comando</i> . |

E.4. La biblioteca matemática de C

```
#include <cmath>
```

| | |
|------------------------------|---|
| <code>sqrt(x)</code> | Devuelve \sqrt{x} . |
| <code>exp(x)</code> | Devuelve e^x . |
| <code>log(x)</code> | Devuelve $\log_e(x)$. |
| <code>log10(x)</code> | Devuelve $\log_{10}(x)$. |
| <code>pow(base, expo)</code> | Devuelve $base^{expo}$. |
| <code>fabs(x)</code> | Devuelve $ x \in \mathcal{R}$. |
| <code>floor(x)</code> | Devuelve $\lfloor x \rfloor$. |
| <code>ceil(x)</code> | Devuelve $\lceil x \rceil$. |
| <code>sin(alfa)</code> | Devuelve $\sin(\alpha)$. |
| <code>cos(alfa)</code> | Devuelve $\cos(\alpha)$. |
| <code>tan(alfa)</code> | Devuelve $\tan(\alpha)$. |
| <code>asin(x)</code> | Devuelve $\sin^{-1}(x)$ en el rango $[-\pi/2, \pi/2]$. |
| <code>acos(x)</code> | Devuelve $\cos^{-1}(x)$ en el rango $[0, \pi]$. |
| <code>atan(x)</code> | Devuelve $\tan^{-1}(x)$ en el rango $[-\pi/2, \pi/2]$. |
| <code>atan(y, x)</code> | Devuelve $\tan^{-1}(y/x)$ en el rango $[-\pi, \pi]$. |

Todas las funciones devuelven un número real en precisión doble. Las funciones trigonométricas trabajan con ángulos en radianes.

E.5. Biblioteca de límites de números enteros y reales de C

```
#include <climits>
```

Incluye la definición como constantes simbólicas de algunos límites de números enteros que son dependientes de máquina. Por ejemplo, los valores máximo y mínimo de los distintos tipos enteros: `INT_MIN`, `INT_MAX`, `SHRT_MIN`, `SHRT_MAX`, `LONG_MIN`, `LONG_MAX`...

```
#include <ctype>
```

Incluye la definición como constantes simbólicas de algunos límites de números reales que son dependientes de máquina. Por ejemplo los valores máximo y mínimo de los distintos tipos reales `FLT_MIN`, `FLT_MAX`, `DBL_MIN`, `DBL_MAX`... y otros relacionados con el formato de coma flotante, como las longitudes de mantisas y exponentes: `FLT_MANT_DIG`, `FLT_MAX_EXP`, `DBL_MANT_DIG`, or `DBL_MAX_EXP`.

Índice alfabético

- ámbito, 56
- alcance, 56
 - archivo, 56
 - bloque, 56
- anidamiento, 29
- array
 - índice, 69
 - acceso fuera de rango, 71
 - asingación completa, 72
 - dinámico, 69
 - estático, 69
 - incompleto, 73
 - longitud
 - física, 73
 - lógica, 73
 - matrices, 76
 - columna, 76
 - fila, 76
 - multidimensional, 76
 - unidimensional, 69, 76
- ASCII
 - caracteres, 97
 - conjunto de caracteres, 97
- asignación, 4
- biblioteca, 7, 103
 - cfloat, 106
 - climits, 106
 - cmath, 18, 106
 - cstdlib, 106
 - entrada/salida, 103
 - iostream, 103
 - matemática, 7, 18
 - `string`, 104
- bloque, 29
- bool
 - función, 27
- bucles, 41
- bucles anidados, 46
 - acoplados, 47
- carácter, 100
 - cadena, 100
 - individual, 100
- casting, 18
- centinela, 42
- cin
 - get, 15
- comentarios, 1
- comparaciones, 26
- condiciones, 25
 - compuestas, 26
- constante, 5
 - const, 5
 - simbólica, 5
- contador, 6, 43
- conversión, 18
 - explícita, 18
 - implícitas, 18
- cout, 2
- dato, 1
- definición
 - subprograma, 51
- directiva
 - `#`, 2
 - `#include`, 2
- efectos laterales, 58
- entrada, 8
 - usuario, 8
- error
 - fuera de rango, 71
 - semánticos, 18
- errores
 - control de, 25
- espacios, 2
- estructura de datos, 82
- evaluación en cortocircuito, 28
- expresión, 6
 - aritmética, 6
 - precedencia, 17
 - tipo booleano, 15
 - tipo carácter, 14
 - tipo lógico, 15
- formato
 - coma fija, 13
 - coma flotante, 13
- función, 10
 - definición, 10

- entrada/salida, 10
 - estándar, 18
 - interfaz, 10
 - lógica, 27
 - parámetros, 10
- function
 - void, 53
- identificador, 4
- indentación, 31
- iomanip, 14
- llamada, 51
- memoria principal, 4
- número, 99
 - entero, 99
 - natural, 99
- números, 99
 - real, 99
 - precisión doble, 99
 - precisión extendida, 99
 - precisión simple, 99
- operador, 6
 - acceso [], 70
 - acceso at(), 70
 - aritmético entero, 13
 - asignación aritmética, 16
 - concatenación, 14
 - condicional, 37
 - decremento, 16
 - igualdad, 13
 - incremento, 16
 - lógico, 26
 - precedencia, 8
 - precedencia aritmética, 17
 - relacional, 13
- operadores
 - aritméticos reales, 13
- operando, 6
- ordinal, 13
- parámetro, 53
 - entrada, 53
 - entrada/salida, 54
 - ficticio, 53
 - formal, 53
 - promoción, 60
 - real, 53
 - salida, 53
- paso parámetros, 60
 - por referencia, 60
 - por referencia constante, 60
 - por valor, 60
- preprocesador, 2
- principal, 1
- procedimiento, 53
- punto y coma, 2
- raíz cuadrada, 7
- registro, 65
 - campo, 65
- regla
 - ámbito, 56
 - visibilidad, 56
- secuencias de escape, 101
- selección
 - doble, 29
 - múltiple, 32
 - opcional, 30
 - simple, 30
- selection
 - anidada, 31
 - examples, 34
- sentencia, 1
 - compuesta, 29
 - control, 29
 - repetición, 29
 - anidamiento, 46
 - do-while, 45
 - for, 43
 - while, 41
 - selección, 29
 - operador condicional ?:, 37
 - doble if, 29
 - múltiple if, 32
 - nested if, 31
 - simple if, 30
 - switch, 36
- fixed, 14
- setprecision, 14
- string, 81
- struct, 65
- subprograma, 51
 - ámbito, 57
- tipo, 4, 12, 99
 - bool, 25
 - false, 25
 - true, 25
 - bool, 15, 100
 - false, 100
 - true, 100
 - char, 14, 100
 - compatible, 18
 - compuesto, 19
 - matrices, 76
 - registro, 65

- string, 81
- struct, 65
- compuuesto
 - array, 69
- definido por el programmer, 65
- double, 13, 99
- int, 13
- escalar, 12
- float, 13, 99
- int, 99
- lógico, 25
- long double, 99
- unsigned, 13
- predefinido, 99
- predefinidos, 65
- registro, 19
- string, 14, 100
- struct, 19
- unsigned, 99
- tipos compatibles, 18
- typedef, 19
- unidades léxicas, 21
- variable, 4
 - ámbito, 57
 - definición, 4
 - global, 57
 - local, 57
- visibilidad, 56