

Enciclopedia del lenguaje **C++**

2ª edición

Programación orientada a objetos

Elementos del lenguaje

Estructura de un programa

Lógica de programación



Más de 270 ejemplos resueltos

ra-ma.es

Fco. Javier Ceballos Sierra

Puede descargarse el CD-ROM con las URL para obtener el software de desarrollo y las aplicaciones contenidas en el libro



Ra-Ma®

Enciclopedia del lenguaje C++

2ª edición

Fco. Javier Ceballos Sierra

Profesor titular de la
Escuela Politécnica Superior
Universidad de Alcalá

<http://www.fjceballos.es>





Enciclopedia del lenguaje C++. 2ª edición.

© Fco. Javier Ceballos Sierra

© De la edición: RA-MA 2009

MARCAS COMERCIALES: Las marcas de los productos citados en el contenido de este libro (sean o no marcas registradas) pertenecen a sus respectivos propietarios. RA-MA no está asociada a ningún producto o fabricante mencionado en la obra, los datos y los ejemplos utilizados son ficticios salvo que se indique lo contrario.

RA-MA es una marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso, ni tampoco por cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa ni de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro, sin autorización previa y por escrito de RA-MA; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes intencionadamente, reprodujeren o plagiasen, en todo o en parte, una obra literaria, artística o científica.

Editado por:

RA-MA Editorial

C/ Jarama, 3A, Polígono industrial Igarsa

28860 PARACUELLOS DEL JARAMA, Madrid

Teléfono: 91 658 42 80

Telefax: 91 662 81 39

Correo electrónico: editorial@ra-ma.com

Internet: www.ra-ma.es y www.ra-ma.com

ISBN: 978-84-7897-915-8

Depósito Legal: M-xxxxx-2009

Autoedición: Fco. Javier Ceballos

Filmación e impresión: ?.

Impreso en España

Primera impresión: Abril 2009

RESUMEN DEL CONTENIDO

PARTE 1. PROGRAMACIÓN BÁSICA	1
CAPÍTULO 1. INTRODUCCIÓN A C++	3
CAPÍTULO 2. ELEMENTOS DEL LENGUAJE C++	25
CAPÍTULO 3. ESTRUCTURA DE UN PROGRAMA	59
CAPÍTULO 4. ENTRADA Y SALIDA ESTÁNDAR	97
CAPÍTULO 5. SENTENCIAS DE CONTROL.....	127
CAPÍTULO 6. TIPOS ESTRUCTURADOS DE DATOS	165
CAPÍTULO 7. PUNTEROS, REFERENCIAS Y GESTIÓN DE LA MEMORIA .	225
CAPÍTULO 8. MÁS SOBRE FUNCIONES	281
PARTE 2. MECANISMOS DE ABSTRACCIÓN.....	333
CAPÍTULO 9. CLASES	335
CAPÍTULO 10. OPERADORES SOBRECARGADOS	421
CAPÍTULO 11. CLASES DERIVADAS	483
CAPÍTULO 12. PLANTILLAS	579
CAPÍTULO 13. EXCEPCIONES	623

CAPÍTULO 14. FLUJOS	661
PARTE 3. DISEÑO Y PROGRAMACIÓN	707
CAPÍTULO 15. ESTRUCTURAS DINÁMICAS	709
CAPÍTULO 16. ALGORITMOS	797
PARTE 4. APÉNDICES.....	813
A. NOVEDADES EN C++0x	815
B. LA BIBLIOTECA ESTÁNDAR DE C++.....	833
C. LA BIBLIOTECA DE C	837
D. ENTORNOS DE DESARROLLO	863
E. INSTALACIÓN DEL PAQUETE DE DESARROLLO	879
F. CÓDIGOS DE CARACTERES.....	883

CONTENIDO

PRÓLOGO.....	XXIII
PARTE 1. PROGRAMACIÓN BÁSICA	1
CAPÍTULO 1. INTRODUCCIÓN A C++.....	3
¿POR QUÉ APRENDER C++?	4
REALIZACIÓN DE UN PROGRAMA EN C++	5
Cómo crear un programa.....	6
¿Qué hace este programa?.....	7
Guardar el programa escrito en el disco.....	8
Compilar y ejecutar el programa.....	8
Biblioteca estándar de C++	8
Depurar un programa	10
EJERCICIO.....	11
DECLARACIÓN DE UNA VARIABLE	12
ASIGNAR VALORES.....	15
AÑADIR COMENTARIOS	16
EXPRESIONES ARITMÉTICAS	18
EXPRESIONES CONDICIONALES	19
ESCRIBIR NUESTRAS PROPIAS FUNCIONES.....	22
EJERCICIOS PROPUESTOS.....	23
CAPÍTULO 2. ELEMENTOS DEL LENGUAJE C++	25
PRESENTACIÓN DE LA SINTAXIS DE C++.....	25
CARACTERES DE C++	26

Letras, dígitos y carácter de subrayado	26
Espacios en blanco	26
Caracteres especiales y signos de puntuación	27
Secuencias de escape.....	27
TIPOS DE DATOS	28
Tipos primitivos	28
Tipos derivados	29
Enumeraciones	30
Clases	32
SINÓNIMOS DE UN TIPO.....	33
LITERALES	33
Literales enteros	34
Literales reales	35
Literales de un solo carácter.....	35
Literales de cadenas de caracteres.....	36
IDENTIFICADORES	36
PALABRAS CLAVE.....	36
DECLARACIÓN DE CONSTANTES SIMBÓLICAS	37
¿Por qué utilizar constantes?.....	38
DECLARACIÓN DE UNA VARIABLE	38
Iniciación de una variable	39
OPERADORES.....	39
Operadores aritméticos.....	39
Operadores de relación.....	40
Operadores lógicos.....	41
Operadores unitarios	42
Operadores a nivel de bits	43
Operadores de asignación	43
Operador condicional	46
Otros operadores	47
Operador global y de resolución de ámbito (::)	47
Operador sizeof.....	48
Operador coma.....	48
Operador dirección-de	49
Operador de indirección.....	49
Operador referencia a.....	50
PRIORIDAD Y ORDEN DE EVALUACIÓN	52
CONVERSIÓN ENTRE TIPOS DE DATOS.....	53
EJERCICIOS PROPUESTOS.....	56

CAPÍTULO 3. ESTRUCTURA DE UN PROGRAMA	59
PARADIGMAS DE PROGRAMACIÓN	59
ESTRUCTURA DE UN PROGRAMA C++	60
Directrices para el preprocesador	63
Inclusión incondicional	64
Definición de un identificador	64
Inclusión condicional	65
Definiciones y declaraciones	65
Sentencia simple	66
Sentencia compuesta o bloque	66
Funciones	67
Declaración de una función	67
Definición de una función	70
Llamada a una función	72
Función main	72
PASANDO ARGUMENTOS A LAS FUNCIONES	73
PROGRAMA C++ FORMADO POR VARIOS MÓDULOS	78
ÁMBITO DE UN NOMBRE	81
Nombres globales y locales	81
CLASES DE ALMACENAMIENTO DE UNA VARIABLE	83
Calificación de variables globales	83
Calificación de variables locales	85
ESPACIOS DE NOMBRES	87
Directriz using	89
EJERCICIOS RESUELTOS	91
EJERCICIOS PROPUESTOS	92
 CAPÍTULO 4. ENTRADA Y SALIDA ESTÁNDAR	 97
ENTRADA Y SALIDA	98
Flujos de salida	99
Flujos de entrada	101
ESTADO DE UN FLUJO	104
DESCARTAR CARACTERES DEL FLUJO DE ENTRADA	106
ENTRADA/SALIDA CON FORMATO	107
ENTRADA DE CARACTERES	112
CARÁCTER FIN DE FICHERO	113
CARÁCTER \n	115
ENTRADA DE CADENAS DE CARACTERES	117
EJERCICIOS RESUELTOS	117
EJERCICIOS PROPUESTOS	122

CAPÍTULO 5. SENTENCIAS DE CONTROL..... 127

SENTENCIA if	127
ANIDAMIENTO DE SENTENCIAS if	129
ESTRUCTURA else if	132
SENTENCIA switch	134
SENTENCIA while.....	137
Bucles anidados.....	140
SENTENCIA do ... while.....	142
SENTENCIA for.....	145
SENTENCIA break.....	149
SENTENCIA continue.....	150
SENTENCIA goto	150
SENTENCIAS try ... catch.....	152
EJERCICIOS RESUELTOS.....	153
EJERCICIOS PROPUESTOS.....	157

CAPÍTULO 6. TIPOS ESTRUCTURADOS DE DATOS 165

INTRODUCCIÓN A LAS MATRICES	166
MATRICES NUMÉRICAS UNIDIMENSIONALES.....	167
Definir una matriz	167
Acceder a los elementos de una matriz	168
Iniciar una matriz	169
Trabajar con matrices unidimensionales	170
Tipo y tamaño de una matriz.....	172
Vector.....	172
Acceso a los elementos	173
Iteradores.....	174
Tamaño	175
Eliminar elementos	175
Buscar elementos	175
Insertar elementos	175
Ejemplo.....	176
Matrices asociativas	178
Map	180
CADENAS DE CARACTERES.....	183
Leer y escribir una cadena de caracteres.....	184
String.....	185
Constructores	185
Iteradores.....	186
Acceso a un carácter	186

Asignación	186
Conversiones a cadenas estilo C	187
Comparaciones.....	187
Inserción.....	188
Concatenación.....	189
Búsqueda.....	189
Reemplazar	189
Subcadenas.....	189
Tamaño	190
Operaciones de E/S	190
MATRICES MULTIDIMENSIONALES.....	191
Matrices numéricas multidimensionales	191
Matrices de cadenas de caracteres.....	197
Matrices de objetos string	198
SENTENCIA for_each.....	201
ESTRUCTURAS	201
Definir una estructura.....	202
Matrices de estructuras.....	206
UNIONES	208
EJERCICIOS RESUELTOS	210
EJERCICIOS PROPUESTOS.....	218

CAPÍTULO 7. PUNTEROS, REFERENCIAS Y GESTIÓN DE LA MEMORIA. 225

CREACIÓN DE PUNTEROS	225
Operadores	227
Importancia del tipo del objeto al que se apunta.....	228
OPERACIONES CON PUNTEROS	228
Operación de asignación	229
Operaciones aritméticas	229
Comparación de punteros.....	231
Punteros genéricos	231
Puntero nulo	232
Punteros y objetos constantes	232
REFERENCIAS	233
Paso de parámetros por referencia	233
PUNTEROS Y MATRICES	235
Punteros a cadenas de caracteres.....	239
MATRICES DE PUNTEROS.....	241
Punteros a punteros	243
Matriz de punteros a cadenas de caracteres	246
ASIGNACIÓN DINÁMICA DE MEMORIA	248

Operadores para asignación dinámica de memoria	249
new	249
delete	251
Reasignar un bloque de memoria	252
MATRICES DINÁMICAS	254
PUNTEROS A ESTRUCTURAS	258
PUNTEROS COMO PARÁMETROS EN FUNCIONES	260
DECLARACIONES COMPLEJAS	264
EJERCICIOS RESUELTOS	265
EJERCICIOS PROPUESTOS	272
 CAPÍTULO 8. MÁS SOBRE FUNCIONES	 281
PASAR UNA MATRIZ COMO ARGUMENTO A UNA FUNCIÓN	281
Matrices automáticas	282
Matrices dinámicas y contenedores	284
PASAR UN PUNTERO COMO ARGUMENTO A UNA FUNCIÓN	286
PASAR UNA ESTRUCTURA A UNA FUNCIÓN	290
DATOS RETORNADOS POR UNA FUNCIÓN	293
Retornar una copia de los datos	293
Retornar un puntero al bloque de datos	295
Retornar la dirección de una variable declarada static	297
Retornar una referencia	299
ARGUMENTOS EN LA LÍNEA DE ÓRDENES	301
REDIRECCIÓN DE LA ENTRADA Y DE LA SALIDA	303
FUNCIONES RECURSIVAS	305
PARÁMETROS POR OMISIÓN EN UNA FUNCIÓN	307
FUNCIONES EN LÍNEA	309
MACROS	310
FUNCIONES SOBRECARGADAS	311
Ambigüedades	313
OPERADORES SOBRECARGADOS	314
PUNTEROS A FUNCIONES	315
EJERCICIOS RESUELTOS	320
EJERCICIOS PROPUESTOS	325
 PARTE 2. MECANISMOS DE ABSTRACCIÓN	 333
CAPÍTULO 9. CLASES	335
DEFINICIÓN DE UNA CLASE	335

Atributos	337
Métodos de una clase	338
Control de acceso a los miembros de la clase	339
Acceso público.....	340
Acceso privado.....	341
Acceso protegido	341
Clases en ficheros de cabecera	341
IMPLEMENTACIÓN DE UNA CLASE	345
MÉTODOS SOBRECARGADOS.....	348
PARÁMETROS CON VALORES POR OMISIÓN	350
IMPLEMENTACIÓN DE UNA APLICACIÓN	351
EL PUNTERO IMPLÍCITO this.....	352
MÉTODOS Y OBJETOS CONSTANTES.....	354
INICIACIÓN DE UN OBJETO.....	356
Constructor.....	358
Asignación de objetos	362
Constructor copia	363
DESTRUCCIÓN DE OBJETOS.....	364
Destructor.....	365
PUNTEROS COMO ATRIBUTOS DE UNA CLASE	366
MIEMBROS STATIC DE UNA CLASE	375
Atributos static	375
Acceder a los atributos static.....	377
Métodos static	378
ATRIBUTOS QUE SON OBJETOS	380
CLASES INTERNAS	382
INTEGRIDAD DE LOS DATOS	384
DEVOLVER UN PUNTERO O UNA REFERENCIA	386
MATRICES DE OBJETOS	387
FUNCIONES AMIGAS DE UNA CLASE	397
PUNTEROS A LOS MIEMBROS DE UNA CLASE	400
EJERCICIOS RESUELTOS	404
EJERCICIOS PROPUESTOS.....	419
 CAPÍTULO 10. OPERADORES SOBRECARGADOS.....	 421
SOBRECARGAR UN OPERADOR	421
UNA CLASE PARA NÚMEROS RACIONALES	428
SOBRECARGA DE OPERADORES BINARIOS.....	430
Sobrecarga de operadores de asignación.....	430
Sobrecarga de operadores aritméticos.....	432
Aritmética mixta.....	434

Sobrecarga de operadores de relación.....	436
Métodos adicionales.....	436
Sobrecarga del operador de inserción	437
Sobrecarga del operador de extracción	440
SOBRECARGA DE OPERADORES UNARIOS.....	442
Incremento y decremento.....	442
Operadores unarios/binarios.....	444
CONVERSIÓN DE TIPOS DEFINIDOS POR EL USUARIO	444
Conversión mediante constructores	446
Operadores de conversión	447
Ambigüedades.....	451
ASIGNACIÓN.....	451
INDEXACIÓN.....	453
LLAMADA A FUNCIÓN	454
DESREFERENCIA.....	456
SOBRECARGA DE LOS OPERADORES new y delete.....	458
Sobrecarga del operador new	458
Sobrecarga del operador delete	461
EJERCICIOS RESUELTOS	463
EJERCICIOS PROPUESTOS.....	480

CAPÍTULO 11. CLASES DERIVADAS..... 483

CLASES DERIVADAS Y HERENCIA	484
DEFINIR UNA CLASE DERIVADA	488
Control de acceso a la clase base	489
Control de acceso a los miembros de las clases	490
Qué miembros hereda una clase derivada	491
ATRIBUTOS CON EL MISMO NOMBRE.....	496
REDEFINIR MÉTODOS DE LA CLASE BASE	498
CONSTRUCTORES DE CLASES DERIVADAS.....	500
COPIA DE OBJETOS	503
DESTRUCTORES DE CLASES DERIVADAS.....	506
JERARQUÍA DE CLASES	506
FUNCIONES AMIGAS.....	514
PUNTEROS Y REFERENCIAS	516
Conversiones implícitas	517
Restricciones	519
Conversiones explícitas.....	520
MÉTODOS VIRTUALES	522
Cómo son implementados los métodos virtuales	526
Constructores virtuales.....	528

Destructores virtuales.....	530
INFORMACIÓN DE TIPOS DURANTE LA EJECUCIÓN	532
Operador <code>dynamic_cast</code>	532
Operador <code>typeid</code>	535
POLIMORFISMO.....	535
CLASES ABSTRACTAS	550
HERENCIA MÚLTIPLE.....	552
Clases base virtuales	556
Redefinición de métodos de bases virtuales.....	560
Conversiones entre clases	562
EJERCICIOS RESUELTOS	563
EJERCICIOS PROPUESTOS.....	575
 CAPÍTULO 12. PLANTILLAS	 579
DEFINICIÓN DE UNA PLANTILLA	580
FUNCIONES GENÉRICAS	582
Especialización de plantillas de función	586
Sobrecarga de plantillas de función	588
ORGANIZACIÓN DEL CÓDIGO DE LAS PLANTILLAS	590
Fichero único.....	590
Fichero de declaraciones y fichero de definiciones	591
Fichero único combinación de otros	593
CLASES GENÉRICAS.....	594
Declaración previa de una clase genérica	599
Especialización de plantillas de clase.....	599
Derivación de plantillas	605
Otras características de las plantillas.....	608
EJERCICIOS RESUELTOS	611
EJERCICIOS PROPUESTOS.....	620
 CAPÍTULO 13. EXCEPCIONES	 623
EXCEPCIONES DE C++	625
MANEJAR EXCEPCIONES	628
Lanzar una excepción.....	629
Capturar una excepción.....	629
Excepciones derivadas	631
Capturar cualquier excepción.....	632
Relanzar una excepción.....	633
CREAR EXCEPCIONES	633
Especificación de excepciones	634

Excepciones no esperadas	635
FLUJO DE EJECUCIÓN	637
CUÁNDO UTILIZAR EXCEPCIONES Y CUÁNDO NO	642
ADQUISICIÓN DE RECURSOS	643
Punteros inteligentes	649
EJERCICIOS RESUELTOS	653
EJERCICIOS PROPUESTOS	659

CAPÍTULO 14. FLUJOS..... 661

VISIÓN GENERAL DE LOS FLUJOS DE E/S	663
BÚFERES	664
VISIÓN GENERAL DE UN FICHERO	666
DESCRIPCIÓN DE LOS BÚFERES Y FLUJOS	670
Clase streambuf	670
Clase filebuf	671
Clase ostream	673
Clase istream	675
Clase iostream	678
Clase ofstream	679
Clase ifstream	681
Clase fstream	683
E/S UTILIZANDO REGISTROS	685
ESCRIBIR DATOS EN LA IMPRESORA	687
ABRIENDO FICHEROS PARA ACCESO SECUENCIAL	687
Un ejemplo de acceso secuencial	688
ACCESO ALEATORIO A FICHEROS EN EL DISCO	698
EJERCICIOS PROPUESTOS	703

PARTE 3. DISEÑO Y PROGRAMACIÓN..... 707

CAPÍTULO 15. ESTRUCTURAS DINÁMICAS..... 709

LISTAS LINEALES	710
Listas lineales simplemente enlazadas	710
Operaciones básicas	713
Inserción de un elemento al comienzo de la lista	714
Buscar en una lista un elemento con un valor x	715
Inserción de un elemento en general	716
Borrar un elemento de la lista	717
Recorrer una lista	718

Borrar todos los elementos de una lista	718
UNA CLASE PARA LISTAS LINEALES	719
Clase genérica para listas lineales	722
Consistencia de la aplicación	731
LISTAS CIRCULARES	733
Clase CListaCircularSE<T>.....	734
PILAS.....	739
COLAS.....	741
EJEMPLO	743
LISTA DOBLEMENTE ENLAZADA.....	746
Lista circular doblemente enlazada.....	746
Clase CListaCircularDE<T>.....	747
Ejemplo.....	754
ÁRBOLES.....	756
Árboles binarios	757
Formas de recorrer un árbol binario.....	759
ÁRBOLES BINARIOS DE BÚSQUEDA.....	761
Clase CARbolBinB<T>	762
Buscar un nodo en el árbol.....	766
Insertar un nodo en el árbol.....	767
Borrar un nodo del árbol	768
Utilización de la clase CARbolBinB<T>.....	771
ÁRBOLES BINARIOS PERFECTAMENTE EQUILIBRADOS.....	774
Clase CARbolBinE<T>.....	775
Utilización de la clase CARbolBinE<T>	783
CLASES RELACIONADAS DE LA BIBLIOTECA C++.....	786
Plantilla list	786
EJERCICIOS PROPUESTOS.....	789

CAPÍTULO 16. ALGORITMOS..... 797

ORDENACIÓN DE DATOS	797
Método de la burbuja	798
Método de inserción.....	801
Método quicksort	802
Comparación de los métodos expuestos.....	804
BÚSQUEDA DE DATOS	805
Búsqueda secuencial	805
Búsqueda binaria.....	805
Búsqueda de cadenas	806
CLASES RELACIONADAS DE LA BIBLIOTECA C++.....	810
Modo de empleo de los algoritmos	811

EJERCICIOS PROPUESTOS.....	812
PARTE 4. APÉNDICES.....	813
NOVEDADES DE C++0x.....	815
LA BIBLIOTECA ESTÁNDAR DE C++	833
LA BIBLIOTECA DE C.....	837
ENTORNOS DE DESARROLLO	863
INSTALACIÓN DEL PAQUETE DE DESARROLLO	879
CÓDIGOS DE CARACTERES	883
ÍNDICE	889

PRÓLOGO

Un programa tradicional se compone de procedimientos y de datos. Un programa orientado a objetos consiste solamente en objetos, entendiendo por objeto una entidad que tiene unos atributos particulares, los datos, y unas formas de operar sobre ellos, los métodos o procedimientos.

La programación orientada a objetos es una de las técnicas más modernas que trata de disminuir el coste del software, aumentando la eficiencia en la programación y reduciendo el tiempo necesario para el desarrollo de una aplicación. Con la programación orientada a objetos, los programas tienen menos líneas de código, menos sentencias de bifurcación, y módulos que son más comprensibles porque reflejan de una forma clara la relación existente entre cada concepto a desarrollar y cada objeto que interviene en dicho desarrollo. Donde la programación orientada a objetos toma verdadera ventaja es en la compartición y reutilización del código.

Sin embargo, no debe pensarse que la programación orientada a objetos resuelve todos los problemas de una forma sencilla y rápida. Para conseguir buenos resultados, es preciso dedicar un tiempo significativamente superior al análisis y al diseño. No obstante, éste no es un tiempo perdido, ya que simplificará enormemente la realización de aplicaciones futuras.

Según lo expuesto, las ventajas de la programación orientada a objetos son sustanciales. Pero también presenta inconvenientes; por ejemplo, la ejecución de un programa no gana en velocidad y obliga al usuario a aprenderse una amplia biblioteca de clases antes de empezar a manipular un lenguaje orientado a objetos.

Existen varios lenguajes que permiten escribir un programa orientado a objetos y entre ellos se encuentra C++. Se trata de un lenguaje de programación basa-

do en el lenguaje C, estandarizado (ISO/IEC – *International Organization for Standardization/International Electrotechnical Commission*) y ampliamente difundido. Gracias a esta estandarización y a la biblioteca estándar, C++ se ha convertido en un lenguaje potente, eficiente y seguro, características que han hecho de él un lenguaje universal de propósito general ampliamente utilizado, tanto en el ámbito profesional como en el educativo, y competitivo frente a otros lenguajes como C# de Microsoft o Java de Sun Microsystems. Evidentemente, algunas nuevas características que se han incorporado a C# o a Java no están soportadas en la actualidad, como es el caso de la recolección de basura; no obstante, existen excelentes recolectores de basura de C++, tanto comerciales como gratuitos, que resuelven este problema. Otro futuro desarrollo previsto es la ampliación de la biblioteca estándar para desarrollar aplicaciones con interfaz gráfica de usuario.

¿Por qué C++? Porque posee características superiores a otros lenguajes. Las más importantes son:

- *Programación orientada a objetos.* Esta característica permite al programador diseñar aplicaciones pensando más bien en la comunicación entre objetos que en una secuencia estructurada de código. Además, permite la reutilización del código de una forma más lógica y productiva.
- *Portabilidad.* Prácticamente se puede compilar el mismo código C++ en la casi totalidad de ordenadores y sistemas operativos sin apenas hacer cambios. Por eso C++ es uno de los lenguajes más utilizados y portados a diferentes plataformas.
- *Brevedad.* El código escrito en C++ es muy corto en comparación con otros lenguajes, debido a la facilidad con la que se pueden anidar expresiones y a la gran cantidad de operadores.
- *Programación modular.* El cuerpo de una aplicación en C++ puede construirse a partir de varios ficheros fuente que serán compilados separadamente para después ser enlazados todos juntos. Esto supone un ahorro de tiempo durante el diseño, ya que cada vez que se realice una modificación en uno de ellos no es necesario recompilar la aplicación completa, sino sólo el fichero que se modificó.
- *Compatibilidad con C.* Cualquier código escrito en C puede fácilmente ser incluido en un programa C++ sin apenas cambios.
- *Velocidad.* El código resultante de una compilación en C++ es muy eficiente debido a su dualidad como lenguaje de alto y bajo nivel y al reducido tamaño del lenguaje mismo.

El libro, en su totalidad, está dedicado al aprendizaje del lenguaje C++, de la programación orientada a objetos y al desarrollo de aplicaciones. Esta materia puede agruparse en los siguientes apartados:

- Programación básica
- Mecanismos de abstracción
- Diseño y programación

La primera parte está pensada para que en poco tiempo pueda convertirse en programador de aplicaciones C++. Y para esto, ¿qué necesita? Pues simplemente leer ordenadamente los capítulos del libro, resolviendo cada uno de los ejemplos que en ellos se detallan. La segunda parte abarca en profundidad la programación orientada a objetos.

En la primera parte el autor ha tratado de desarrollar aplicaciones sencillas, para introducirle más bien en el lenguaje y en el manejo de la biblioteca de clases de C++, que en el diseño de clases de objetos. No obstante, después de su estudio sí debe haber quedado claro que un programa orientado a objetos sólo se compone de objetos. Es hora pues de entrar con detalle en la programación orientada a objetos, segunda parte, la cual tiene un elemento básico: la *clase*.

Pero si el autor finalizara el libro con las dos partes anteriores, privaría al lector de saber que C++ aún proporciona mucho más. Por eso la tercera parte continúa con otros capítulos dedicados a la implementación de estructuras dinámicas, al diseño de algoritmos y a la programación con hilos.

Todo ello se ha documentado con abundantes problemas resueltos. Cuando complete todas las partes, todavía no sabrá todo lo que es posible hacer con C++, pero sí habrá dado un paso importante.

Esta obra fue escrita utilizando un compilador GCC para Win32 (un compilador C++ de la colección de compiladores GNU) que se adjunta en el CD-ROM que acompaña al libro. Se trata de un compilador de libre distribución que cumple la norma ISO/IEC, del cual existen versiones para prácticamente todos los sistemas operativos. Por lo tanto, los ejemplos de este libro están escritos en C++ puro, tal y como se define en el estándar C++, lo que garantizará que se ejecuten en cualquier implementación que se ajuste a este estándar, que en breve serán la totalidad de las existentes. Por ejemplo, el autor probó la casi totalidad de los desarrollos bajo el paquete Microsoft Visual Studio .NET, y también sobre la plataforma Linux, para conseguir un código lo más portable posible.

Agradecimientos

En la preparación de este libro quiero, en especial, expresar mi agradecimiento a **Manuel Peinado Gallego**, profesor de la Universidad de Alcalá con una amplia experiencia en desarrollos con C++, porque revisó la primera edición de este libro; y a **Óscar García Población, Elena Campo Montalvo, Sebastián Sánchez Prieto, Inmaculada Rodríguez Santiago y M^a Dolores Rodríguez Moreno**, que basándose en su experiencia docente me hicieron diversas sugerencias sobre los temas tratados. Todos ellos son profesores de Universidad, con una amplia experiencia sobre la materia que trata el libro.

Finalmente, no quiero olvidarme del resto de mis compañeros, aunque no cite sus nombres, porque todos ellos, de forma directa o indirecta, me ayudaron con la crítica constructiva que hicieron sobre otras publicaciones anteriores a ésta, y tampoco de mis alumnos, que con su interés por aprender me hacen reflexionar sobre la forma más adecuada de transmitir estos conocimientos; a todos ellos les estoy francamente agradecido.

Francisco Javier Ceballos Sierra

P A R T E

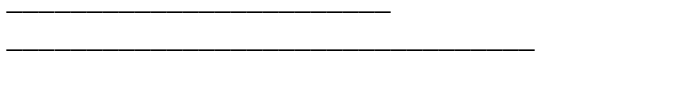
1

Programación básica

- Introducción a C++
- Elementos del lenguaje
- Estructura de un programa
- Entrada y salida estándar
- Sentencias de control
- Tipos estructurados de datos
- Punteros, referencias y gestión de la memoria
- Más sobre funciones

P A R T E

2

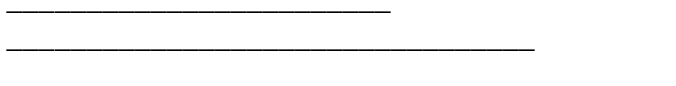


Mecanismos de abstracción

- Clases
- Operadores sobrecargados
- Clases derivadas
- Plantillas
- Excepciones
- Flujos

P A R T E

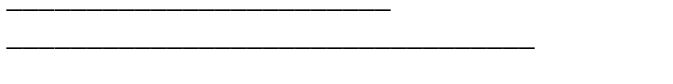
3



Diseño y programación

- Estructuras dinámicas
- Algoritmos

P A R T E 4



Apéndices

- Novedades en C++0x
- La biblioteca estándar de C++
- La biblioteca de C
- Entornos de desarrollo
- Instalación del paquete de desarrollo
- Códigos de caracteres

NOVEDADES DE C++0x

C++0x es el nombre de trabajo para el nuevo estándar del lenguaje de programación C++ que reemplazará al estándar ISO/IEC 14882 actual, publicado en 1998 (C++98) y actualizado 2003 (C++03), y que de ser aprobado a lo largo de este año 2009 pasará a llamarse C++09. Es 100% compatible con C++03. Las modificaciones introducidas afectan tanto a la biblioteca estándar como al lenguaje. Entre las nuevas características que se incluirán en este nuevo estándar destacamos las siguientes:

- Cambios en la biblioteca estándar independientes del lenguaje: por ejemplo, plantillas con un número variable de argumentos (*variadic*) y **constexpr**.
- Facilidades para escribir código: **auto**, **enum class**, **long long**, **nullptr**, ángulos derechos (>>) en plantillas o **static_assert**.
- Ayudas para actualizar y mejorar la biblioteca estándar: **constexpr**, listas de iniciadores generales y uniformes, referencias *rvalue*, plantillas *variadic* y una versión de la biblioteca estándar con todas estas características.
- Características relacionadas con la concurrencia: modelo de memoria multithread, **thread_local** o una biblioteca para realizar programación concurrente (hilos).
- Características relacionadas con conceptos: **concepts** (mecanismo para la descripción de los requisitos sobre los tipos y las combinaciones de los mismos lo que mejorará la calidad de los mensajes de error del compilador), sentencia **for** para iterar sobre un conjunto de valores y conceptos en la biblioteca estándar.
- Expresiones *lambda*.

La finalidad de todas estas nuevas características de C++ es mejorar el rendimiento de las aplicaciones durante su construcción y durante su ejecución, mejo-

rar la usabilidad y funcionalidad del lenguaje y proporcionar una biblioteca estándar más completa y segura.

INFERENCIA DE TIPOS

La inferencia de tipos asigna automáticamente un tipo de datos a una variable a partir de una expresión. Para ello, la variable es calificada **auto**. Esto es, el especificador **auto** es un marcador de posición para un tipo que se deduce de una determinada expresión:

```
auto var = expresión;
```

Por ejemplo, en la siguiente sentencia x tendrá el tipo **int** porque es el tipo de su valor de iniciación:

```
auto x = 15;
```

El uso de **auto** es tanto más útil cuanto más difícil sea conocer exactamente el tipo de una variable o expresión. Por ejemplo, considere la siguiente función genérica:

```
template<class T> void mostrarVector(const vector<T>& v)
{
    for (auto p = v.begin(); p != v.end(); ++p)
        cout << *p << "\n";
}
```

¿Cuál es tipo de p ? En este caso **auto** está reemplazando al tipo:

```
typename vector<T>::const_iterator
```

El antiguo significado de **auto** (variable local automática) es redundante y ya no es utilizado.

ÁNGULOS DERECHOS EN EL USO DE PLANTILLAS

Considere el siguiente ejemplo escrito en C++03:

```
var vector<vector<double>> v1;
```

Observamos que en C++03 era necesario dejar un espacio entre los ángulos sombreados. Ahora, en C++0x ya no es necesario:

```
var vector<vector<double>>> v1;
```


SENTENCIA **for** APLICADA A COLECCIONES

Es posible acceder a cada uno de los elementos de una colección utilizando la siguiente sentencia **for**:

```
for (auto var : colección)
```

Por ejemplo, la siguiente plantilla de función utiliza esta sentencia, primero para multiplicar por 2 cada uno de los elementos del vector pasado como argumento y después, para mostrar cada uno de los elementos del vector:

```
template<class T> void mostrarVector(const vector<T>& v)
{
    for(auto& x : v)
    {
        x *= 2;
    }

    for (auto x : v)
        cout << x << "\n";
}
```

LISTA DE INICIACIÓN

C++0x extiende el lenguaje para que las listas de iniciación que ya utilizábamos cuando definíamos una estructura o una matriz puedan utilizarse ahora también para iniciar otros objetos. Una lista de iniciación puede utilizarse en los siguientes casos, entre otros:

- Para iniciar una variable:

```
int x = {0};
vector<double> v = { 3.2, 2.1, 7.6, 5.4 };
list<pair<string, string>> capitales = { { "España", "Madrid" },
                                         { "Francia", "París" },
                                         { "Italia", "Roma" }
                                         };
```

- Para iniciar un objeto creado con **new**:

```
new vector<string>{"uno", "dos", "tres"}; // 3 elementos
```

- En una sentencia **return**:

```
return { "uno" }; // retorna una lista de un elemento
```

- Como argumento en una función:

```
fn({"uno","dos"}); // el argumento es una lista de dos elementos
```

ENUMERACIONES

Las enumeraciones tradicionales tienen el inconveniente de que sus elementos son convertidos implícitamente a **int**. Por ejemplo:

```
enum colores { rojo, verde, azul };
colores color = 2;    // error: conversión de 'int' a 'colores'
int miColor = azul;   // correcto: conversión de 'colores' a 'int'
```

Para solucionar este inconveniente C++0x ha añadido las enumeraciones fuertemente tipadas y delimitadas: **enum class**. Por ejemplo:

```
enum class colores { rojo, verde, azul };
int color = azul;           // error: azul fuera de ámbito
int miColor = colores::azul; // error: conversión 'colores' a 'int'
```

ENTERO MUY LARGO

C++0x ha añadido el tipo **long long** para especificar un entero de al menos 64 bits. Por ejemplo:

```
long long x = 9223372036854775807LL;
```

PUNTERO NULO

Desde los comienzos de C, la constante 0 ha tenido un doble significado: constante entera y puntero constante nulo, lo cual puede dar lugar a errores. Por ejemplo, supongamos las siguientes sobrecargas de la función *fn*:

```
void fn(char *);
void fn(int);
```

Una llamada como *fn(NULL)* (donde la constante NULL está definida en C++ como 0) invocaría a *fn(int)*, que no es lo que esperamos. Para corregir esto, el estándar C++0x ha añadido la constante **nullptr** para especificar un puntero nulo. Por ejemplo:

```
fn(nullptr);
```

EXPRESIONES CONSTANTES GENERALIZADAS

Una sentencia C++ requiere en muchas ocasiones una constante. Por ejemplo, cuando se declara una matriz, la dimensión especificada tiene que ser una constante; también, en una sentencia **switch**, los **case** deben ir seguidos por una constante. Esto quiere decir que el compilador en esos casos no permitirá nada que no sea una constante. Para dar solución a situaciones como la presentada a continuación, C++0x estándar añade la palabra reservada **constexpr**. En el ejemplo siguiente, si no utilizáramos **constexpr** para especificar que la función *operator /* devuelve una constante, el **case** tercero daría un error indicando que requiere una constante.

```
enum estadoFlujo { good, fail, bad, eof };

constexpr int operator|(estadoFlujo f1, estadoFlujo f2)
{
    return estadoFlujo(f1|f2);
}

void fn(estadoFlujo x)
{
    switch (x)
    {
        case bad:
            // ...
            break;
        case eof:
            // ...
            break;
        case fail|eof: // invoca a operator(fail, eof)
            // ...
            break;
        default:
            // ...
            break;
    }
}
```

REFERENCIAS rvalue y lvalue

Un expresión *lvalue* es aquella que puede ser utilizada en el lado izquierdo de una asignación y una expresión *rvalue* es aquella que puede ser utilizada en el lado derecho de una asignación. Esto es, cuando, por ejemplo, una variable *x* o un elemento *a[i]* de una matriz se presenta como el objetivo de una operación de asignación: *x = z*, o como el operando del operador incremento: *x++*, o como el operando del operador dirección: *&x*, nosotros utilizamos el *lvalue* de la variable o

del elemento de la matriz. Esto es, el *lvalue* es la localización o dirección de memoria de esa variable o elemento de la matriz. En caso contrario, cuando la variable x o el elemento $a[i]$ de una matriz se utilizan en una expresión: $z = x + 5$, nosotros utilizamos su *rvalue*. El *rvalue* es el valor almacenado en la localización de memoria correspondiente a la variable.

Sólo las expresiones que tienen una localización en la memoria pueden tener un *lvalue*. Así, en C/C++ esta expresión no tiene sentido: $(7 + 3)++$, porque la expresión $(7 + 3)$ no tiene un *lvalue*.

En otras palabras, durante la compilación se hace corresponder *lvalues* a los identificadores y durante la ejecución, en la memoria, se hacen corresponder *rvalues* a los *lvalues*.

Una vez aclarados los conceptos *lvalue/rvalue*, vamos a estudiar las “referencias *lvalue/rvalue*”. En C++, las referencias no **const** pueden ser vinculadas a *lvalues*, pero no a *rvalues*:

```
void fn(int& a) {}
int x = 0;
fn(x);      // x es un lvalue que se vincula con la referencia 'a'
fn(0);      // error: 0 es un rvalue
```

y las referencias **const** a *lvalues* o *rvalues*:

```
void fn(const int& a) {}
int x = 0;
fn(x);      // x es un lvalue vinculado con 'a'
fn(0);      // correcto: rvalue vinculado con 'a' (const int&)
```

¿Por qué no se puede vincular un *rvalue* no **const** a una referencia no **const**? Pues para no permitir cambiar los objetos temporales que son destruidos antes de que su nuevo valor (si se pudiera cambiar) pueda ser utilizado:

```
class C {}
C fn() { C x; return x; }
// ...
C z;
C& r1 = z;          // correcto: z es un lvalue
C& r2 = fn();        // error: objeto temporal (rvalue no const)
const C& r3 = fn();  // correcto: referencia const a un rvalue
```

La función *fn* devuelve un objeto temporal, el que se utilizará en la asignación. Devolver un objeto temporal copia de un objeto fuente (en el ejemplo copia de x) en lugar de utilizar el propio objeto fuente, tiene un coste: crear el objeto

temporal y destruirlo. Para dar solución a este problema y a otros similares, el nuevo estándar C++0x introduce el concepto de *referencia rvalue*.

Una *referencia rvalue* a un objeto de la clase *C* es creada con la sintaxis *C&&*, para distinguirla de la referencia existente (*C&*). La referencia existente se denomina ahora *referencia lvalue*. La nueva referencia *rvalue* se comporta como la referencia actual *lvalue* y además puede vincularse a un *rvalue*:

```
class C {};
C fn() { C x; return x; };
// ...
C a;
C& r1 = a;           // correcto: a es un lvalue
C& r2 = fn();         // error: fn() es un rvalue (objeto temporal)
```

```
C&& rr1 = fn();       // correcto: referencia rr1 a un objeto temporal
C&& rr2 = a;           // correcto: referencia rr2 a un lvalue
```

Una referencia *rvalue* y una *lvalue* son tipos distintos. Por lo tanto, podrán ser utilizadas, una y otra, para declarar versiones sobrecargadas de la misma función. Por ejemplo:

```
class C {};

void fn1(const C& x) {}; // #1: referencia lvalue
void fn1(C&& x) {};      // #2: referencia rvalue

C fn2() { C x; return x; };
const C cfn2() { C x; return x; };

int main()
{
    C a;
    const C ca;
    fn1(a);           // llama a #1 (lvalue)
    fn1(ca);          // llama a #1 (lvalue const)
    fn1(fn2());        // llama a #2 (rvalue)
    fn1(cfn2());       // llama a #1 (rvalue const)
}
```

La primera llamada a *fn1* utiliza la referencia *lvalue* (#1) porque el argumento es un *lvalue* (la conversión *lvalue* a *rvalue* es menos afín que la de *C&* a *const C&*). La segunda llamada a *fn1* es una coincidencia exacta para #1. La tercera es una coincidencia exacta para #2. Y la cuarta, no puede utilizar #2 porque la conversión de *const C&&* a *C&&* no está permitida; llama a #1 a través de una conversión de *rvalue* a *lvalue*.

Las normas de resolución de la sobrecarga indican que los *rvalues* prefieren *referencias rvalue* (una *referencia rvalue* se vincula a *rvalues* incluso si no están calificados **const**), que los *lvalues* prefieren *referencias lvalue*, que los *rvalues* pueden vincularse a una *referencia lvalue const* (por ejemplo, *const C&*), a menos que haya una *referencia rvalue* en el conjunto de sobrecargas, y que los *lvalues* pueden vincularse a una *referencia rvalue*, pero prefieren una *referencia lvalue* si la hay.

La razón principal para añadir *referencias rvalue* es eliminar copias innecesarias de los objetos, lo que facilita la aplicación de la *semántica de mover* (no copiar: *semántica de copiar*). A diferencia de la conocida idea de copiar, mover significa que un objeto destino roba los recursos del objeto fuente, en lugar de copiarlos o compartirlos. Se preguntará, ¿y por qué iba alguien a querer eso? En la mayoría de los casos, preferiremos la semántica de copia. Sin embargo, en algunos casos, hacer una copia de un objeto es costoso e innecesario. C++ ya implementa la semántica de mover en varios lugares, por ejemplo, con **auto_ptr** y para optimizar la operación de retornar un objeto.

Para aclarar lo expuesto, piense en dos objetos **auto_ptr** *a* y *b*. Cuando realizamos la operación *b = a*, lo que sucede es que el objeto *b* pasa a ser el nuevo propietario del objeto apuntado por *a*, y *a* pasa a no apuntar a nada y es destruido.

Según lo expuesto, es eficiente añadir a una clase nuevas versiones sobrecargadas del operador de asignación y del constructor copia que utilicen la semántica de mover, ya que son más eficientes que el operador de asignación y constructor copia tradicionales. Por ejemplo:

```
class C
{
public:
    C(){ /* ... */ }
    // Semántica de copiar
    C(const C& a){ /* ... */ };
    C& operator=(const C& a){ /* ... */ };

    // Semántica de mover
    C(C&& a){ /* ... */ };
    C& operator=(C&& a){ /* ... */ };
    // ...
};
```

PLANTILLAS *variadic*

En ciencias de la computación, se dice que un operador o una función es *variadic* cuando puede tomar un número variable de argumentos. Pues bien, C++0x incluye también plantillas con un número variable de parámetros.

Por ejemplo, la siguiente plantilla *P* puede aceptar cero o más argumentos de tipo:

```
template<typename... T> class P
{
    P(T... t) { }
};
```

A partir de esta plantilla, *P<int>* generará una clase más o menos así, suponiendo que el nombre de la nueva clase es *P_int*:

```
class P_int
{
    P_int(int t) { }
};
```

Y una expresión como *P<int, double>* generará una clase más o menos así, suponiendo que el nombre de la nueva clase es *P_int_double*:

```
class P_int_double
{
    P_int_double(int t1, double t2) { }
};
```

ENVOLTORIO PARA UNA REFERENCIA

C++0x aporta la plantilla **reference_wrapper<T>** para manipular referencias a objetos. Un objeto **reference_wrapper<T>** contiene una referencia a un objeto de tipo *T*. Esta plantilla, entre otros, proporciona los métodos:

```
reference_wrapper<T> ref(T& t);
T& get() const;
```

Ahora, con esta plantilla, entre otras cosas, podremos trabajar con vectores de referencias en lugar de con vectores de punteros. Por ejemplo:

```
class C { /* ... */ }
```

```
int main()
{
    vector<reference_wrapper<C>> v;
    C obj1(10); C obj2(20);
    v.push_back(ref(obj1)); v.push_back(ref(obj2));
    v[1].get() = C(25); // ahora obj2.n = 25
}
```

OPERADOR **decltype**

El operador **decltype** evalúa el tipo de una expresión. Por ejemplo:

```
const double&& fn();
int i;
struct S { double v; };
const S* p = new S();

decltype(fn()) v1;    // el tipo es const double&&
decltype(i) v2;      // el tipo es int
decltype(p->v) v3;    // el tipo es double
```

DECLARACIÓN DE FUNCIÓN

C++0x aporta una nueva declaración de función de la forma:

auto fn([parámetros])->tipo_retornado

El *tipo retornado* sustituirá a **auto**. Por ejemplo, la siguiente línea declara una función *f* que tiene un parámetro *x* de tipo **int** y devuelve un puntero a una matriz de 4 elementos de tipo **double**:

```
auto f(int x)->double(*)[4];
```

Podemos combinar este tipo de declaración con el operador **decltype** para deducir el tipo del valor retornado. Por ejemplo, el tipo del valor retornado por la siguiente función es *vector<T>::iterator*:

```
template <class T>
auto ultimo(vector<T>& v){ return v.end() } ->decltype(v.end());
```

PUNTEROS INTELIGENTES

Un objeto **auto_ptr** emplea el modelo de “propiedad exclusiva”. Esto significa que no se puede vincular más de un objeto **auto_ptr** a un mismo recurso. Para

asegurar la propiedad exclusiva, las operaciones de copiar y asignar del **auto_ptr** hacen que el objeto fuente entregue la propiedad de los recursos al objeto destino. Veamos un ejemplo:

```
void fn()
{
    // Crear sp1 con la propiedad del puntero a un objeto C
    auto_ptr<C> sp1(new C);
    { // Bloque interno: define sp2
        // Crear sp2 a partir sp1.
        auto_ptr<C> sp2(sp1);
        // La propiedad del puntero a C ha pasado a sp2
        // y sp1 pasa a ser nulo.
        sp2->m_fn(); // sp2 representa al objeto de la clase C
        // El destructor del sp2 elimina el objeto C apuntado
    }
    sp1->m_fn(); // error: sp1 es nulo
}
```

En el código anterior destacamos, por una parte, que después de la copia el objeto fuente ha cambiado, algo que normalmente no se espera. ¿Por qué funcionan así? Porque si no fuera así, los dos objetos **auto_ptr**, al salir de su ámbito, intentarían borrar el mismo objeto con un resultado impredecible; pero en nuestro ejemplo también ocurre que cuando *sp1* invoca a *m_fn* no puede hacerlo porque ya no tiene la propiedad del objeto *C* (es nulo); el resultado es un error de ejecución. Por otra parte, los objetos **auto_ptr** no se pueden utilizar con contenedores de la biblioteca STL porque estos pueden mover sus elementos y ya hemos visto lo que pasa cuando copiamos un objeto **auto_ptr** en otro.

Para afrontar estos problemas C++0x proporciona la plantilla **shared_ptr**. Igual que **auto_ptr**, la plantilla **shared_ptr** almacena un puntero a un objeto, pero **shared_ptr** implementa la semántica de la propiedad compartida: el último propietario del puntero es el responsable de la destrucción del objeto o de la liberación de los recursos asociados con el puntero almacenado. Un objeto **shared_ptr** está vacío si no posee un puntero. El ejemplo siguiente demuestra que los problemas que en la versión anterior introducía **auto_ptr** ahora han desaparecido:

```
void fn1()
{
    // Crear sp1 con la propiedad del puntero a un objeto C
    shared_ptr<C> sp1(new C);
    { // Bloque interno: define sp2
        // Crear sp2 a partir sp1.
        shared_ptr<C> sp2(sp1);
        // La propiedad del puntero a C es compartida entre sp2 y sp1.
        sp2->m_fn(); // sp2 representa al objeto de la clase C
        // El destructor del sp2 no elimina el objeto C apuntado
    }
```

```

    }
    spl->m_fn(); // spl representa al objeto de la clase C
    // El destructor del spl elimina el objeto C apuntado
}

```

Otra plantilla relacionada con **shared_ptr** es **weak_ptr**. Esta plantilla almacena una referencia débil a un objeto que ya está gestionado por un **shared_ptr**. Para acceder al objeto, un **weak_ptr** se puede convertir en un **shared_ptr** utilizando la función miembro **lock**.

A diferencia de **shared_ptr**, un **weak_ptr** no incrementa el contador de referencias del recurso compartido. Por ejemplo, si se tiene un **shared_ptr** y un **weak_ptr**, ambos vinculados a los mismos recursos, el contador de referencias es 1, no 2:

```

C* pObjC = new C;
shared_ptr<C> spl(pObjC); // el contador de referencias vale 1
weak_ptr<C> wpl(spl);    // el contador de referencias vale 1
shared_ptr<C> sp2(spl);  // el contador de referencias vale 2
cout << "El contador de spl/sp2 referencias es: "
    << spl.use_count() << endl;

```

Los objetos **weak_ptr** se utilizan para romper los ciclos en las estructuras de datos. Un ciclo es similar a un abrazo mortal en el *multithreading*: dos recursos mantienen punteros entre sí de manera que un puntero no puede ser liberado porque el otro recurso comparte su propiedad y viceversa. Pues bien, esta dependencia puede ser rota utilizando **weak_ptr** en lugar de **shared_ptr**.

DELEGACIÓN DE CONSTRUCTORES

La delegación de constructores proporciona un mecanismo por el cual un constructor puede delegar en otro para realizar la iniciación de un objeto. Esto es útil cuando las distintas sobrecargas de un constructor no se pueden refundir en una que utilice parámetros con valores por omisión y estemos obligados a definir dos constructores y, probablemente, a repetir el código de iniciación.

Según lo expuesto, las dos versiones de la clase A mostradas a continuación serían equivalentes:

```

class D
{
public:
    operator double() const;
};

```

```

class A
{
    private:
        int n;
        double d;
        void iniciar();
    public:
        A(int x = 0, double y = 0.0) : n(x), d(y) { iniciar(); }
        A(D& b) : d(b) { iniciar(); }
        // ...
};

class A
{
    private:
        int n;
        double d;
    public:
        A(int x = 0, double y = 0.0) : n(x), d(y) { /* iniciar */ }
        A(B& b) : A(0, b) {}
        // ...
};

```

CONVERSIONES EXPLÍCITAS

Una función de conversión puede ser explícita (**explicit**), en cuyo caso se requiere la intervención directa del usuario (especificando una conversión forzada) allí donde sea necesario utilizarla. Si no es explícita, las conversiones se realizarán sin la intervención del usuario (sin tener que especificar una conversión forzada). Por ejemplo:

```

class A { };

class B
{
    public:
        explicit operator A() const;
};

void fn(B b)
{
    A a1(b);      // correcto: iniciación directa
    A a2 = b;     // error: conversión B a A requerida
    A a3 = (A)b;  // correcto: conversión B a A forzada
}

```

EXPRESIONES LAMBDA

Una expresión *lambda* (también conocida como función *lambda*) es una función sin nombre definida en el lugar de la llamada. Como tal, es similar a un objeto función (un objeto que puede ser invocado como si de una función ordinaria se tratara). De hecho, las expresiones *lambda* se transforman automáticamente en objetos función. Entonces, ¿por qué no utilizar los objetos función directamente? Podría hacerse así, pero la creación de un objeto función es una tarea laboriosa: hay que definir una clase con miembros de datos, una sobrecarga del operador llamada a función y un constructor. A continuación, hay que crear un objeto de ese tipo en todos los lugares donde sea requerido.

Para demostrar la utilidad de las expresiones *lambda*, supongamos que necesitamos buscar el primer objeto *X* cuyo valor se encuentra dentro de un determinado rango. Utilizando los objetos función tradicionales se puede escribir una clase *F* de objetos función así:

```
class X
{
    double d;
public:
    X(double x = 0.0) : d(x) {}
    double dato() const { return d; }
};

class F
{
    double inf, sup;
public:
    F(double i, double s) : inf(i), sup(s) {}
    bool operator()(const X& obj)
    {
        return obj.dato() >= inf && obj.dato() < sup;
    }
};

int main()
{
    vector<X> v;
    v.push_back(X(1.0)); v.push_back(X(7.0)); v.push_back(X(15.0));
    double inf = 5.0, sup = 10.0;
    vector<X>::iterator resu;
    resu = std::find_if(v.begin(), v.end(), F(inf, sup));
    cout << (*resu).dato() << endl;
}
```

Obsérvese el tercer parámetro de la función **find_if** definida en *<algorithm>*. Se trata de un objeto función que define la función a aplicar sobre objetos de otra clase. En otras palabras, la clase *F* representa la clase del objeto función que el compilador generaría para una expresión *lambda* dada. Según esto, si en su lugar utilizamos una expresión *lambda* el código quedaría así:

```
class X
{
    double d;
public:
    X(double x = 0.0) : d(x) {}
    double dato() const { return d; }
};

int main()
{
    vector<X> v;
    v.push_back(X(1.0)); v.push_back(X(7.0)); v.push_back(X(15.0));
    double inf = 5.0, sup = 10.0;
    vector<X>::iterator resu;
    resu = std::find_if(v.begin(), v.end(),
        [&](const X& obj) -> bool {
            return (obj.dato() >= inf && obj.dato() <= sup);});
    cout << (*resu).dato() << endl;
}
```

Vemos que una expresión *lambda* empieza con el presentador *lambda*, [], que puede estar vacío (no depende de variables fuera del ámbito del cuerpo de la expresión *lambda*), puede incluir el símbolo = (depende de variables que serán pasadas por valor) o el símbolo & (depende de variables que serán pasadas por referencia). A continuación, entre paréntesis, se especifican los parámetros de la expresión *lambda*. Después el tipo del valor retornado, el cual se puede omitir si no hay valor retornado o si se puede deducir de la expresión. Y finalmente, está el cuerpo de la expresión *lambda*.

A partir de una expresión *lambda* se genera una clase de objetos función. Esto es, a partir de esta expresión *lambda* se generaría una clase análoga a la clase *F* expuesta anteriormente. En resumen, la expresión *lambda* y su correspondiente clase están relacionadas así:

- Las variables con referencias externas se corresponden a los datos miembros de la clase.
- La lista de parámetros *lambda* se corresponde con la lista de los argumentos pasados a la sobrecarga del operador () de llamada a función.

- El cuerpo de la expresión *lambda* se corresponde más o menos con el cuerpo de la sobrecarga del operador ().
- El tipo del valor retornado por la sobrecarga del operador () se deduce automáticamente de una expresión **decltype**.

CONCEPTO

Mecanismo que permite especificar claramente y de manera intuitiva las limitaciones de las plantillas, mejorando al mismo tiempo la capacidad del compilador para detectar y diagnosticar violaciones de estas limitaciones.

Los *conceptos* se basan en la idea de separar la comprobación de tipos en las plantillas. Para ello, la declaración de la plantilla se incrementará con una serie de restricciones. Cuando una plantilla sea instanciada, el compilador comprobará si la instanciación reúne todas las limitaciones, o los requisitos, de la plantilla. Si todo va bien, la plantilla será instanciada; de lo contrario, un error de compilación especificará que las limitaciones han sido violadas. Veamos un ejemplo concreto. Supongamos la plantilla de función *min* definida así:

```
template<typename T>
const T& min(const T& x, const T& y)
{
    return x < y ? x : y;
}
```

Es necesario examinar el cuerpo de *min* para saber cuáles son las limitaciones de *T*. *T* debe ser un tipo que tenga, al menos, definido el operador <. Utilizando conceptos, estos requisitos pueden ser expresados directamente en la definición *min* así:

```
template<LessThanComparable T>
const T& min(const T& x, const T& y)
{
    return x < y ? x : y;
}
```

En esta otra versión de *min* vemos que en lugar de decir que *T* es un tipo arbitrario (como indica la palabra clave **typename**), se afirma que *T* es un tipo *LessThanComparable*, independientemente de lo que pueda ser. Por lo tanto, *min* sólo aceptará argumentos cuyos tipos cumplan los requisitos del concepto *LessThanComparable*, de lo contrario el compilador mostrará un error indicando que los argumentos de *min* no cumplen los requisitos *LessThanComparable*.

¿Cómo se define un concepto? Pues se define utilizando la palabra clave **concept** seguida por el nombre del concepto y de la lista de parámetros de la plantilla. Por ejemplo:

```
auto concept LessThanComparable<typename T>
{
    bool operator<(T, T);
};
```

El código anterior define un concepto llamado *LessThanComparable* que establece que el parámetro *T* de una determinada plantilla debe ser un tipo que tiene definido el operador `<`.

PROGRAMACIÓN CONCURRENTE

Una gran novedad en el estándar C++0x es el soporte para la programación concurrente. Esto es muy positivo porque ahora todos los compiladores tendrán que ajustarse al mismo modelo de memoria y proporcionar las mismas facilidades para el trabajo con hilos (*multithreading*). Esto significa que el código será portable entre compiladores y plataformas con un coste muy reducido. Esto también reducirá el número de API. El núcleo de esta nueva biblioteca es la clase **std::thread** declarada en el fichero de cabecera `<thread>`.

¿Cómo se crea un hilo de ejecución? Pues creando un objeto de la clase **thread** y vinculándolo con la función que debe ejecutar el hilo:

```
void tareaHilo();
std::thread hilo1(tareaHilo);
```

La tarea que realiza el hilo puede ser también una función con parámetros:

```
void tareaHilo(int i, std::string s, std::vector<double> v);
// ...
std::thread hilo1(tareaHilo, n, nombre, lista);
```

Los argumentos pasados se copian en el hilo antes de que la función se invoque. Ahora bien, si lo que queremos es pasarlos por referencia, entonces hay que envolverlos utilizando el método **ref**. Por ejemplo:

```
void tareaHilo(string&);
// ...
std::thread hilo1(tareaHilo, ref(s));
```

También, en lugar de definir la tarea del hilo mediante una función, la podemos definir mediante un objeto función:

```
class CTareaHilo
{
public:
    void operator()();
};

CTareaHilo tareaHilo;
std::thread hilo1(tareaHilo);
```

Evidentemente, además de la clase **thread**, disponemos de mecanismos para la sincronización de hilos: objetos de exclusión mutua (**mutex**), *locks* y variables de condición. A continuación mostramos algunos ejemplos:

```
std::mutex m;
CMiClase datos;

void fn()
{
    std::lock_guard<std::mutex> bloqueo(m);
    proceso(datos);
} // El desbloqueo se produce aquí
```

Aunque los *mutex* tengan métodos para bloquear y desbloquear, en la mayoría de escenarios la mejor manera de hacerlo es utilizando *locks*. El bloqueo más simple, como vemos en el ejemplo anterior, nos lo proporciona **lock_guard**. Si lo que queremos hacer es un bloqueo diferido, o un bloqueo sin o con un tiempo de espera y desbloquear antes de que el objeto sea destruido, podemos utilizar **unique_lock**:

```
std::timed_mutex m;
CMiClase datos;

void fn()
{
    std::unique_lock<std::timed_mutex>
        bloqueo(m, std::chrono::milliseconds(3)); // esperar 3 ms
    if (bloqueo) // si tenemos el bloqueo, acceder a los datos
        proceso(datos);
} // El desbloqueo se produce aquí
```

Estos es sólo una pequeña introducción a la programación con hilos. Evidentemente hay mucho más: mecanismos para esperar por eventos, almacenamiento local de hilos de ejecución, mecanismos para evitar el abrazo mortal, etc.

LA BIBLIOTECA ESTÁNDAR DE C++

La biblioteca estándar de C++ está definida en el espacio de nombres **std** y las declaraciones necesarias para su utilización son proporcionadas por un conjunto de ficheros de cabecera que se exponen a continuación. Con el fin de dar una idea general de la funcionalidad aportada por esta biblioteca, hemos clasificado estos ficheros, según su función, en los grupos siguientes:

- Entrada/Salida
- Cadenas
- Contenedores
- Iteradores
- Algoritmos
- Números
- Diagnósticos
- Utilidades generales
- Localización
- Soporte del lenguaje

La aportación que realizan los contenedores, iteradores y algoritmos a la biblioteca estándar a menudo se denomina *STL* (*Standard Template Library*, biblioteca estándar de plantillas).

A continuación mostramos un listado de los diferentes ficheros de cabecera de la biblioteca estándar, para hacernos una idea de lo que supone esta biblioteca. Un fichero de cabecera de la biblioteca estándar que comience por la letra *c* equivale a un fichero de cabecera de la biblioteca de C; esto es, un fichero *<f.h>* de la bi-

bliblioteca de C tiene su equivalente `<cf>` en la biblioteca estándar de C++ (generalmente, lo que sucede es que la implementación de `cf` incluye a `f.h`).

ENTRADA/SALIDA

<code><cstdio></code>	E/S de la biblioteca de C.
<code><cstdlib></code>	Funciones de clasificación de caracteres.
<code><cwchar></code>	E/S de caracteres extendidos.
<code><fstream></code>	Flujos para trabajar con ficheros en disco.
<code><iomanip></code>	Manipuladores.
<code><ios></code>	Tipos y funciones básicos de E/S.
<code><iosfwd></code>	Declaraciones adelantadas de utilidades de E/S.
<code><iostream></code>	Objetos y operaciones sobre flujos estándar de E/S.
<code><istream></code>	Objetos y operaciones sobre flujos de entrada.
<code><ostream></code>	Objetos y operaciones sobre flujos de salida.
<code><sstream></code>	Flujos para trabajar con cadenas de caracteres.
<code><streambuf></code>	Búferes de flujos.

CADENAS

<code><cctype></code>	Examinar y convertir caracteres.
<code><cstdlib></code>	Funciones de cadena estilo C.
<code><cstring></code>	Funciones de cadena estilo C.
<code><cwchar></code>	Funciones de cadena de caracteres extendidos estilo C.
<code><cwctype></code>	Clasificación de caracteres extendidos.
<code><string></code>	Clases para manipular cadenas de caracteres.

CONTENEDORES

<code><bitset></code>	Matriz de bits.
<code><deque></code>	Cola de dos extremos de elementos de tipo <i>T</i> .
<code><list></code>	Lista doblemente enlazada de elementos de tipo <i>T</i> .
<code><map></code>	Matriz asociativa de elementos de tipo <i>T</i> .
<code><queue></code>	Cola de elementos de tipo <i>T</i> .
<code><set></code>	Conjunto de elementos de tipo <i>T</i> (contenedor asociativo).
<code><stack></code>	Pila de elementos de tipo <i>T</i> .
<code><vector></code>	Matriz de elementos de tipo <i>T</i> .

ITERADORES

<code><iterator></code>	Soporte para iteradores.
-------------------------------	--------------------------

ALGORITMOS

<code><algorithm></code>	Algoritmos generales (buscar, ordenar, contar, etc.).
<code><cstdlib></code>	<i>bsearch</i> y <i>qsort</i> .

NÚMEROS

<code><cmath></code>	Funciones matemáticas.
<code><complex></code>	Operaciones con números complejos.
<code><cstdlib></code>	Números aleatorios estilo C.
<code><numeric></code>	Algoritmos numéricos generalizados.
<code><valarray></code>	Operaciones con matrices numéricas.

DIAGNÓSTICOS

<code><cassert></code>	Macro ASSERT.
<code><cerrno></code>	Tratamiento de errores estilo C.
<code><exception></code>	Clase base para todas las excepciones.
<code><stdexcept></code>	Clases estándar utilizadas para manipular excepciones.

UTILIDADES GENERALES

<code><ctime></code>	Fecha y hora estilo C.
<code><functional></code>	Objetos función.
<code><memory></code>	Funciones para manipular bloques de memoria.
<code><utility></code>	Manipular pares de objetos.

LOCALIZACIÓN

<code><locale></code>	Control estilo C de las diferencias culturales.
<code><locale></code>	Control de las diferencias culturales.

SOPORTE DEL LENGUAJE

<code><float></code>	Límites numéricos en coma flotante estilo C.
<code><limits></code>	Límites numéricos estilo C.
<code><setjmp></code>	Salvar y restaurar el estado de la pila.
<code><signal></code>	Establecimiento de manejadores para condiciones excepcionales (también conocidos como señales).
<code><stdarg></code>	Lista de parámetros de función de longitud variable.
<code><stddef></code>	Soporte de la biblioteca al lenguaje C.
<code><stdlib></code>	Definición de funciones, variables y tipos comunes.
<code><time></code>	Manipulación de la fecha y hora.
<code><exception></code>	Tratamiento de excepciones.

<code><limits></code>	Límites numéricos.
<code><new></code>	Gestión de memoria dinámica.
<code><typeinfo></code>	Identificación de tipos durante la ejecución.

LA BIBLIOTECA DE C

La biblioteca de C puede ser utilizada también desde un programa C++. Por ejemplo, con frecuencia algunos programadores prefieren utilizar las funciones de E/S de C, que se encuentran en *stdio.h* (*cstdio* en la biblioteca de C++ estándar), por ser más familiares. En este caso, con la llamada a **sync_with_stdio(false)** de la clase *ios_base* antes de la primera operación de E/S puede desactivar la sincronización de las funciones *iostream* con las funciones *cstdio*, que por omisión está activada. Esta función retorna el modo de sincronización (**true** o **false**) previo.

```
bool sync_with_stdio(bool sync = true);
```

Cuando la sincronización está desactivada (*sync* = **false**), las operaciones de E/S con **cin**, **cout**, **cerr** y **clog** se realizan utilizando un búfer de tipo **filebuf** y las operaciones con **stdin**, **stdout** y **stderr** se realizan utilizando un búfer de tipo **stdiobuf**; esto es, los flujos *iostream* y los flujos *cstdio* operan independiente, lo cual puede mejorar la ejecución pero sin garantizar la sincronización. En cambio, cuando hay sincronización (*sync* = **true**), todos los flujos utilizan el mismo búfer, que es **stdiobuf**. El siguiente ejemplo le permitirá comprobar la sincronización en operaciones de E/S:

```
// Comprobar si sync_with_stdio(true) trabaja
#include <cstdio>
#include <iostream>
using namespace std;

int main()
{
    /*
    1. ¿Qué se escribe en test.txt cuando se invoca a
       sync_with_stdio con el argumento true?
    2. ¿Y con el argumento false?
```

```
3. ¿Y cuando no se invoca a sync_with_stdio? (caso por omisión)
*/
ios_base::sync_with_stdio();

// Vincular stdout con el fichero test.txt
freopen ("test.txt", "w", stdout);

for (int i = 0; i < 2; i++)
{
    printf("1");
    cout << "2";
    putc('3', stdout);
    cout << '4';
    fputs("5", stdout);
    cout << 6;
    putchar('7');
    cout << 8 << '9';
    if (i)
        printf("0\n");
    else
        cout << "0" << endl;
}
}

/*
Resultados:
1. 1234567890
   1234567890

2. 1357246890
   13570
   24689

3. 1234567890
   1234567890
*/
```

A continuación se resumen las funciones más comunes de la biblioteca de C.

ENTRADA Y SALIDA

printf

La función **printf** escribe bytes (caracteres ASCII) de **stdout**.

```
#include <cstdio>
int printf(const char *formato[, argumento]...);
```

formato Especifica cómo va a ser la salida. Es una cadena de caracteres formada por caracteres ordinarios, secuencias de escape y especificaciones de formato. El formato se lee de izquierda a derecha.

```
unsigned int edad = 0;
float peso = 0;

// ...
printf("Tiene %u años y pesa %g kilos\n", edad, peso);
```

argumento Representa el valor o valores a escribir. Cada argumento debe tener su correspondiente especificación de formato y en el mismo orden.

```
printf("Tiene %u años y pesa %g kilos\n", edad, peso);
```

Una especificación de formato está compuesta por:

`%[flags][ancho][.precisión][{h|l|L}]tipo`

Una especificación de formato siempre comienza con %. El significado de cada uno de los elementos se indica a continuación:

flags	significado
–	Justifica el resultado a la izquierda, dentro del <i>ancho</i> especificado. Por defecto la justificación se hace a la derecha.
+	Antepone el signo + o – al valor de salida. Por defecto sólo se pone signo – a los valores negativos.
0	Rellena la salida con ceros no significativos hasta alcanzar el ancho mínimo especificado.
blanco	Antepone un espacio en blanco al valor de salida si es positivo. Si se utiliza junto con +, entonces se ignora.
#	Cuando se utiliza con la especificación de formato o , x o X , antepone al valor de salida 0 , 0x o 0X , respectivamente. Cuando se utiliza con la especificación de formato e , E o f , fuerza a que el valor de salida contenga un punto decimal en todos los casos. Cuando se utiliza con la especificación de formato g o G , fuerza a que el valor de salida contenga un punto decimal en todos los casos y evita que los ceros arrastrados sean truncados. Se ignora con c , d , i , u o s .

ancho Mínimo número de posiciones para la salida. Si el valor a escribir ocupa más posiciones de las especificadas, el ancho es incrementado en lo necesario.

precisión El significado depende del tipo de la salida.

tipo Es uno de los siguientes caracteres:

carácter	salida
d	(int) enteros con signo en base 10.
i	(int) enteros con signo en base 10.
u	(int) enteros sin signo en base 10.
o	(int) enteros sin signo en base 8.
x	(int) enteros sin signo en base 16 (01...abcdef).
X	(int) enteros sin signo en base 16 (01...ABCDEF).
f	(double) valor con signo de la forma: $[-]dddd.dddd$. El número de dígitos antes del punto decimal depende de la magnitud del número y el número de decimales de la precisión, la cual es 6 por defecto.
e	(double) valor con signo, de la forma $[-]d.dddde[\pm]ddd$.
E	(double) valor con signo, de la forma $[-]d.ddddE[\pm]ddd$.
g	(double) valor con signo, en formato f o e (el que sea más compacto para el valor y precisión dados).
G	(double) igual que g , excepto que G introduce el exponente E en vez de e .
c	(int) un solo carácter, correspondiente al byte menos significativo.
s	(<i>cadena de caracteres</i>) escribir una cadena de caracteres hasta el primer carácter nulo (<code>\0</code>).

Ejemplo:

```
#include <cstdio>
int main()
{
    int a = 12345;
    float b = 54.865F;
    printf("%d\n", a);           /* escribe 12345\n */
    printf("\n%10s\n%10s\n", "abc", "abcdef");
    printf("\n%-10s\n%-10s\n", "abc", "abcdef");
    printf("\n");               /* avanza a la siguiente línea */
    printf("%.2f\n", b);        /* escribe b con dos decimales */
}
```

La *precisión*, en función del tipo, tiene el siguiente significado:

d,i,u,o,x,X	Especifica el mínimo número de dígitos que se tienen que escribir. Si es necesario, se rellena con ceros a la izquierda. Si el valor excede de la precisión, no se trunca.
e,E,f	Especifica el número de dígitos que se tienen que escribir después del punto decimal. Por defecto es 6. El valor es redondeado.
g,G	Especifica el máximo número de dígitos significativos (por defecto 6) que se tienen que escribir.
c	La precisión no tiene efecto.
s	Especifica el máximo número de caracteres que se escribirán. Los caracteres que excedan este número, se ignoran.

h	Se utiliza como prefijo con los tipos d , i , o , x y X , para especificar que el argumento es short int , o con u para especificar un short unsigned int .
l	Se utiliza como prefijo con los tipos d , i , o , x y X , para especificar que el argumento es long int , o con u para especificar un long unsigned int . También se utiliza con los tipos e , E , f , g y G para especificar un double antes que un float .
L	Se utiliza como prefijo con los tipos e , E , f , g y G , para especificar long double . Este prefijo no es compatible con ANSI C.

scanf

La función **scanf** lee bytes (caracteres ASCII) de **stdin**.

```
#include <stdio>
int scanf(const char *formato[, argumento]...);
```

formato Interpreta cada dato de entrada. Está formado por caracteres que genéricamente se denominan espacios en blanco (' ', \t, \n), caracteres ordinarios y especificaciones de formato. El formato se lee de izquierda a derecha.

Cada argumento debe tener su correspondiente especificación de formato y en el mismo orden (vea también la función **printf**).

Si un carácter en **stdin** no es compatible con el tipo especificado por el formato, la entrada de datos se interrumpe.

argumento Es la variable pasada por referencia que se quiere leer.

Cuando se especifica más de un argumento, los valores tecleados en la entrada hay que separarlos por uno o más espacios en blanco (' ', \t, \n), o por el carácter que se especifique en el formato. Por ejemplo:

```
scanf("%d %f %c", &a, &b, &c);
```

Entrada de datos:

```
5 23.4 z
```

o también:

```
5
23.4
z
```

Una especificación de formato está compuesta por:

```
%[*][ancho][{h/l}]tipo
```

Una especificación de formato siempre comienza con %. El resto de los elementos que puede incluir se explican a continuación:

- *** Un *asterisco* a continuación del símbolo % suprime la asignación del siguiente dato en la entrada.
- ancho*** Máximo número de caracteres a leer de la entrada. Los caracteres en exceso no se tienen en cuenta.
- h*** Se utiliza como prefijo con los tipos **d**, **i**, **n**, **o** y **x** para especificar que el argumento es **short int**, o con **u** para especificar que es **short unsigned int**.
- l*** Se utiliza como prefijo con los tipos **d**, **i**, **n**, **o** y **x** para especificar que el argumento es **long int**, o con **u** para especificar que es **long unsigned int**. También se utiliza con los tipos **e**, **f** y **g** para especificar que el argumento es **double**.
- tipo*** El tipo determina cómo tiene que ser interpretado el dato de entrada: como un carácter, como una cadena de caracteres o como un número. El formato más simple contiene el símbolo % y el *tipo*. Por ejemplo, %i. Los tipos que se pueden utilizar son los siguientes:

El argumento es		
Carácter	un puntero a	Entrada esperada
d	int	enteros con signo en base 10.
o	int	enteros con signo en base 8.
x, X	int	enteros con signo en base 16.

i	int	enteros con signo en base 10, 16 u 8. Si el entero comienza con 0 , se toma el valor en octal y si empieza con 0x o 0X , el valor se toma en hexadecimal.
u	unsigned int	enteros sin signo en base 10.
f		
e, E		
g, G	float	valor con signo de la forma $[-]d.ddd[\{e E\} [\pm]ddd]$
c	char	un solo carácter.
s	char	cadena de caracteres.

getchar

Leer un carácter de la entrada estándar (**stdin**).

```
#include <stdio>
int getchar(void);
```

```
char car;
car = getchar();
```

putchar

Escribir un carácter en la salida estándar (**stdout**).

```
#include <stdio>
int putchar(int c);
```

```
putchar('\n');
putchar(car);
```

gets

Leer una cadena de caracteres de **stdin**.

```
#include <stdio>
char *gets(char *var);
```

```
char nombre[41];
gets(nombre);
printf("%s\n", nombre);
```

puts

Escribir una cadena de caracteres en **stdout**.

```
#include <cstdio>
int puts(const char *var);

char nombre[41];
gets(nombre);
puts(nombre);
```

CADENAS DE CARACTERES

strcat

Añade la *cadena2* a la *cadena1*. Devuelve un puntero a *cadena1*.

```
#include <cstring>
char *strcat(char *cadena1, const char *cadena2);
```

strcpy

Copia la *cadena2*, incluyendo el carácter de terminación nulo, en la *cadena1*. Devuelve un puntero a *cadena1*.

```
#include <cstring>
char *strcpy(char *cadena1, const char *cadena2);
```

```
char cadena[81];
strcpy(cadena, "Hola. ");
strcat(cadena, "Hasta luego.");
```

strchr

Devuelve un puntero a la primera ocurrencia de *c* en *cadena* o un valor **NULL** si el carácter no es encontrado. El carácter *c* puede ser el carácter nulo (`'\0'`).

```
#include <cstring>
char *strchr(const char *cadena, int c);
```

```
char *pdest;
pdest = strchr(cadena, 'a');
```

strrchr

Devuelve un puntero a la última ocurrencia de *c* en *cadena* o un valor **NULL** si el carácter no se encuentra. El carácter *c* puede ser un carácter nulo (`'\0'`).

```
#include <cstring>
char *strrchr(const char *cadena, int c);
```

strcmp

Compara la *cadena1* con la *cadena2* lexicográficamente y devuelve un valor:

<0 si la *cadena1* es menor que la *cadena2*,
=0 si la *cadena1* es igual a la *cadena2* y
>0 si la *cadena1* es mayor que la *cadena2*.

Diferencia las letras mayúsculas de las minúsculas.

```
#include <cstring>
int strcmp(const char *cadena1, const char *cadena2);

resu = strcmp(cadena1, cadena2);
```

strcspn

Da como resultado la posición (subíndice) del primer carácter de *cadena1*, que pertenece al conjunto de caracteres contenidos en *cadena2*.

```
#include <cstring>
size_t strcspn(const char *cadena1, const char *cadena2);

pos = strcspn(cadena, "abc");
```

strlen

Devuelve la longitud en bytes de *cadena*, no incluyendo el carácter de terminación nulo. El tipo **size_t** es sinónimo de **unsigned int**.

```
#include <cstring>
size_t strlen(char *cadena);

char cadena[80] = "Hola";
printf("El tamaño de cadena es %d\n", strlen(cadena));
```

strncat

Añade los primeros *n* caracteres de *cadena2* a la *cadena1*, termina la cadena resultante con el carácter nulo y devuelve un puntero a *cadena1*.

```
#include <cstring>
char *strncat(char *cadena1, const char *cadena2, size_t n);
```

strncpy

Copia *n* caracteres de la *cadena2* en la *cadena1* (sobrescribiendo los caracteres de *cadena1*) y devuelve un puntero a *cadena1*.

```
#include <cstring>
char *strncpy(char *cadena1, const char *cadena2, size_t n);
```

strncmp

Compara lexicográficamente los primeros *n* caracteres de *cadena1* y de *cadena2*, distinguiendo mayúsculas y minúsculas, y devuelve un valor:

<0 si la *cadena1* es menor que la *cadena2*,
=0 si la *cadena1* es igual a la *cadena2* y
>0 si la *cadena1* es mayor que la *cadena2*.

```
#include <cstring>
int strncmp(const char *cadena1, const char *cadena2, size_t n);
```

strspn

Da como resultado la posición (subíndice) del primer carácter de *cadena1*, que no pertenece al conjunto de caracteres contenidos en *cadena2*.

```
#include <cstring>
size_t strspn(const char *cadena1, const char *cadena2);
```

strstr

Devuelve un puntero a la primera ocurrencia de *cadena2* en *cadena1* o un valor **NULL** si la *cadena2* no se encuentra en la *cadena1*.

```
#include <cstring>
char *strstr(const char *cadena1, const char *cadena2);
```

strtok

Permite obtener de la *cadena1* los elementos en los que se divide según los delimitadores especificados en *cadena2*.

Para obtener el primer elemento, **strtok** debe tener *cadena1* como primer argumento y para obtener los siguientes elementos, debe tener **NULL**. Cada llama-

da a **strtok** devuelve un puntero al siguiente elemento o **NULL** si no hay más elementos.

```
#include <cstring>
char *strtok(char *cadena1, const char *cadena2);

#include <stdio>
#include <cstring>

int main(void)
{
    char cadena[] = "Esta cadena, está formada por varias palabras";
    char *elemento;
    elemento = strtok(cadena, " ,");
    while (elemento != NULL)
    {
        printf("%s\n", elemento);
        elemento = strtok(NULL, " ,");
    }
}
```

strlwr

Convierte las letras mayúsculas de *cadena* en minúsculas. El resultado es la propia cadena en minúsculas.

```
#include <cstring>
char *strlwr(char *cadena);
```

strupr

Convierte las letras minúsculas de *cadena* en mayúsculas. El resultado es la propia cadena en mayúsculas.

```
#include <cstring>
char *strupr(char *cadena);
```

CONVERSIÓN DE DATOS

atof

Convierte una cadena de caracteres a un valor de tipo **double**.

```
#include <cstdlib>
double atof(const char *cadena);
```

atoi

Convierte una cadena de caracteres a un valor de tipo **int**.

```
#include <cstdlib>
int atoi(const char *cadena);
```

atol

Convierte una cadena de caracteres a un valor de tipo **long**.

```
#include <cstdlib>
long atol(const char *cadena);
```

Cuando las funciones **atof**, **atoi** y **atol** toman de la variable *cadena* un carácter que no es reconocido como parte de un número, interrumpen la conversión.

sprintf

Convierte los valores de los argumentos especificados a una cadena de caracteres que almacena en *buffer*. Devuelve como resultado un entero correspondiente al número de caracteres almacenados en *buffer* sin contar el carácter nulo de terminación.

```
#include <stdio>
int sprintf(char *buffer, const char *formato [, argumento] ...);

#include <stdio>
int main(void)
{
    char buffer[200], s[] = "ordenador", c = '/';
    int i = 40, j;
    float f = 1.414214F;

    j = sprintf(buffer, "\tCadena: %s\n", s);
    j += sprintf(buffer + j, "\tCarácter: %c\n", c);
    j += sprintf(buffer + j, "\tEntero: %d\n", i);
    j += sprintf(buffer + j, "\tReal: %f\n", f);
    printf("Salida:\n%s\nNúmero de caracteres = %d\n", buffer, j);
}
```

toascii

Pone a 0 todos los bits de *c*, excepto los siete bits de menor orden. Dicho de otra forma, convierte *c* a un carácter ASCII.


```
#include <cctype>
int toascii(int c);
```

tolower

Convierte c a una letra minúscula, si procede.

```
#include <cstdlib>
int tolower(int c);
```

toupper

Convierte c a una letra mayúscula, si procede.

```
#include <cstdlib>
int toupper(int c);
```

FUNCIONES MATEMÁTICAS

acos

Da como resultado el arco, en el rango 0 a π , cuyo coseno es x . El valor de x debe estar entre -1 y 1 ; de lo contrario se obtiene un error (argumento fuera del dominio de la función).

```
#include <cmath>
double acos(double x);
```

asin

Da como resultado el arco, en el rango $-\pi/2$ a $\pi/2$, cuyo seno es x . El valor de x debe estar entre -1 y 1 ; si no, se obtiene un error (argumento fuera del dominio de la función).

```
#include <cmath>
double asin(double x);
```

atan

Da como resultado el arco, en el rango $-\pi/2$ a $\pi/2$, cuya tangente es x .

```
#include <cmath>
double atan(double x);
```

atan2

Da como resultado el arco, en el rango $-\pi$ a π , cuya tangente es y/x . Si ambos argumentos son 0, se obtiene un error (argumento fuera del dominio de la función).

```
#include <cmath>
double atan2(double y, double x);
```

cos

Da como resultado el coseno de x (x en radianes).

```
#include <cmath>
double cos(double x);
```

sin

Da como resultado el seno de x (x en radianes).

```
#include <cmath>
double sin(double x);
```

tan

Da como resultado la tangente de x (x en radianes).

```
#include <cmath>
double tan(double x);
```

cosh

Da como resultado el coseno hiperbólico de x (x en radianes).

```
#include <cmath>
double cosh(double x);
```

sinh

Da como resultado el seno hiperbólico de x (x en radianes).

```
#include <cmath>
double sinh(double x);
```

tanh

Da como resultado la tangente hiperbólica de x (x en radianes).

```
#include <cmath>
double tanh(double x);
```

exp

Da como resultado el valor de e^x ($e = 2.718282$).

```
#include <cmath>
double exp(double x);
```

log

Da como resultado el logaritmo natural de x .

```
#include <cmath>
double log(double x);
```

log10

Da como resultado el logaritmo en base 10 de x .

```
#include <cmath>
double log10(double x);
```

ceil

Da como resultado un valor **double**, que representa el entero más pequeño que es mayor o igual que x .

```
#include <cmath>
double ceil(double x);

double x = 2.8, y = -2.8;
printf("%g %g\n", ceil(x), ceil(y)); // resultado: 3 -2
```

fabs

Da como resultado el valor absoluto de x . El argumento x es un valor real en doble precisión. Igualmente, **abs** y **labs** dan el valor absoluto de un **int** y un **long**, respectivamente.

```
#include <cmath>
double fabs(double x);
```

floor

Da como resultado un valor **double**, que representa el entero más grande que es menor o igual que x .

```
#include <cmath>
double floor(double x);

double x = 2.8, y = -2.8;
printf("%g %g\n", floor(x), floor(y)); // resultado: 2 -3
```

pow

Da como resultado x^y . Si x es 0 e y negativo o si x e y son 0 o si x es negativo e y no es entero, se obtiene un error (argumento fuera del dominio de la función). Si x^y da un resultado superior al valor límite para el tipo **double**, el resultado es este valor límite (1.79769e+308).

```
#include <cmath>
double pow(double x, double y);

double x = 2.8, y = -2.8;
printf("%g\n", pow(x, y)); // resultado: 0.0559703
```

sqrt

Da como resultado la raíz cuadrada de x . Si x es negativo, ocurre un error (argumento fuera del dominio de la función).

```
#include <cmath>
double sqrt(double x);
```

rand

Da como resultado un número pseudoaleatorio entero, entre 0 y **RAND_MAX** (32767).

```
#include <cstdlib>
int rand(void);
```

srand

Fija el punto de comienzo para generar números pseudoaleatorios; en otras palabras, inicia el generador de números pseudoaleatorios en función del valor de su argumento. Cuando esta función no se utiliza, el valor del primer número pseudoaleatorio generado siempre es el mismo para cada ejecución (corresponde a un argumento de valor 1).

```
#include <cstdlib>
void srand(unsigned int arg);
```

FUNCIONES DE FECHA Y HORA

clock

Indica el tiempo empleado por el procesador en el proceso en curso.

```
#include <ctime>
clock_t clock(void);
```

El tiempo expresado en segundos se obtiene al dividir el valor devuelto por **clock** entre la constante `CLOCKS_PER_SEC`. Si no es posible obtener este tiempo, la función **clock** devuelve el valor `(clock_t)-1`. El tipo **clock_t** está declarado así:

```
typedef long clock_t;
```

time

Retorna el número de segundos transcurridos desde las 0 horas del 1 de enero de 1970.

```
#include <ctime>
time_t time(time_t *seg);
```

El tipo **time_t** está definido así:

```
typedef long time_t;
```

El argumento puede ser **NULL**. Según esto, las dos sentencias siguientes para obtener los segundos transcurridos son equivalentes:

```
time_t segundos;
time(&segundos);
segundos = time(NULL);
```

ctime

Convierte un tiempo almacenado como un valor de tipo **time_t**, en una cadena de caracteres de la forma:

```
Thu Jul 08 12:01:29 2010\n\0
```

```
#include <ctime>
char *ctime(const time_t *seg);
```

Esta función devuelve un puntero a la cadena de caracteres resultante o un puntero nulo si *seg* representa un dato anterior al 1 de enero de 1970. Por ejemplo, el siguiente programa presenta la fecha actual y, a continuación, genera cinco números pseudoaleatorios, uno cada segundo.

```
/****** Generar un número aleatorio cada segundo *****/
#include <cstdio>
#include <cstdlib>
#include <ctime>
int main()
{
    long x, tm;
    time_t segundos;
    time(&segundos);
    printf("\n%s\n", ctime(&segundos));
    srand((unsigned)time(NULL));

    for (x = 1; x <= 5; x++)
    {
        do // tiempo de espera igual a 1 segundo
            tm = clock();
        while (tm/CLOCKS_PER_SEC < x);
        // Se genera un número aleatorio cada segundo
        printf("Iteración %ld: %d\n", x, rand());
    }
}
```

localtime

Convierte el número de segundos transcurridos desde las 0 horas del 1 de enero de 1970, valor obtenido por la función **time**, a la fecha y hora correspondiente (corregida en función de la zona horaria en la que nos encontremos). El resultado es almacenado en una estructura de tipo **tm**, definida en *ctime*.

```
#include <ctime>
struct tm *localtime(const time_t *seg);
```

La función **localtime** devuelve un puntero a la estructura que contiene el resultado o un puntero nulo si el tiempo no puede ser interpretado. Los miembros de la estructura son los siguientes:

Campo	Valor almacenado
tm_sec	Segundos (0 - 59).
tm_min	Minutos (0 - 59).
tm_hour	Horas (0 - 23).
tm_mday	Día del mes (1 - 31).
tm_mon	Mes (0 - 11; enero = 0).
tm_year	Año (actual menos 1900).
tm_wday	Día de la semana (0 - 6; domingo = 0).
tm_yday	Día del año (0 - 365; 1 de enero = 0).

El siguiente ejemplo muestra cómo se utiliza esta función.

```
#include <cstdio>
#include <ctime>
int main()
{
    struct tm *fh;
    time_t segundos;

    time(&segundos);
    fh = localtime(&segundos);
    printf("%d horas, %d minutos\n", fh->tm_hour, fh->tm_min);
}
```

La función **localtime** utiliza una variable de tipo **static struct tm** para realizar la conversión y lo que devuelve es la dirección de esa variable.

MANIPULAR BLOQUES DE MEMORIA

memset

Permite iniciar un bloque de memoria.

```
#include <cstring>
void *memset(void *destino, int b, size_t nbytes);
```

El argumento *destino* es la dirección del bloque de memoria que se desea iniciar, *b* es el valor empleado para iniciar cada byte del bloque y *nbytes* es el número de bytes del bloque que se iniciarán. Por ejemplo, el siguiente código inicia a 0 la matriz *a*:

```
double a[10][10];  
// ...  
memset(a, 0, sizeof(a));
```

memcpy

Copia un bloque de memoria en otro.

```
#include <cstring>  
void *memcpy(void *destino, const void *origen, size_t nbytes);
```

El argumento *destino* es la dirección del bloque de memoria destino de los datos, *origen* es la dirección del bloque de memoria origen de los datos y *nbytes* es el número de bytes que se copiarán desde el origen al destino. Por ejemplo, el siguiente código copia la matriz *a* en *b*:

```
double a[10][10], b[10][10];  
// ...  
memcpy(b, a, sizeof(a));
```

memcmp

Compara byte a byte dos bloques de memoria.

```
#include <cstring>  
int memcmp(void *bm1, const void *bm2, size_t nbytes);
```

Los argumentos *bm1* y *bm2* son las direcciones de los bloques de memoria a comparar y *nbytes* es el número de bytes que se compararán. El resultado devuelto por la función es el mismo que se expuso para **strcmp**. Por ejemplo, el siguiente código compara la matriz *a* con la *b*:

```
double a[10][10], b[10][10];  
// ...  
if (memcmp(a, b, sizeof(a)) == 0)  
    printf("Las matrices a y b contienen los mismos datos\n");  
else  
    printf("Las matrices a y b no contienen los mismos datos\n");
```

ASIGNACIÓN DINÁMICA DE MEMORIA

malloc

Permite asignar un bloque de memoria de *nbytes* consecutivos en memoria para almacenar uno o más objetos de un tipo cualquiera. Esta función devuelve un puntero genérico (**void ***) que referencia el espacio asignado. Si no hay suficiente es-

pacio de memoria, la función **malloc** retorna un puntero nulo (valor **NULL**) y si el argumento *nbytes* es 0, asigna un bloque de tamaño 0 devolviendo un puntero válido.

```
#include <stdlib.h>
void *malloc(size_t nbytes);
```

free

Permite liberar un bloque de memoria asignado por las funciones **malloc**, **calloc** o **realloc** (estas dos últimas las veremos a continuación), pero no pone el puntero a **NULL**. Si el puntero que referencia el bloque de memoria que deseamos liberar es nulo, la función **free** no hace nada.

```
#include <stdlib.h>
void free(void *vpuntero);
```

realloc

Permite cambiar el tamaño de un bloque de memoria previamente asignado.

```
#include <stdlib.h>
void *realloc(void *pBlomem, size_t nBytes);
```

<i>pBlomem</i>	<i>nBytes</i>	<i>Acción</i>
NULL	0	Asigna 0 bytes (igual que malloc).
NULL	Distinto de 0	Asigna <i>nBytes</i> bytes (igual que malloc). Si no es posible, devuelve NULL .
Distinto de NULL	0	Devuelve NULL y libera el bloque original.
Distinto de NULL	Distinto de 0	Reasigna <i>nBytes</i> bytes. El contenido del espacio conservado no cambia. Si la reasignación no es posible, devuelve NULL y el bloque original no cambia.

FICHEROS

fopen

Permite crear un flujo desde un fichero, hacia un fichero o bien desde y hacia un fichero. En términos más simplificados, permite abrir un fichero para leer, para escribir o para leer y escribir.

```
#include <stdio.h>
FILE *fopen(const char *nomfi, const char *modo);
```

nomfi es el nombre del fichero y *modo* especifica cómo se va a abrir el fichero:

Modo	Descripción
"r"	Abrir un fichero para leer. Si el fichero no existe o no se encuentra, se obtiene un error.
"w"	Abrir un fichero para escribir. Si el fichero no existe, se crea; y si existe, su contenido se destruye para ser creado de nuevo.
"a"	Abrir un fichero para añadir información al final del mismo. Si el fichero no existe, se crea.
"r+"	Abrir un fichero para leer y escribir. El fichero debe existir.
"w+"	Abrir un fichero para escribir y leer. Si el fichero no existe, se crea; y si existe, su contenido se destruye para ser creado de nuevo.
"a+"	Abrir un fichero para leer y añadir. Si el fichero no existe, se crea.

```
FILE *pf;
pf = fopen("datos", "w"); // abrir el fichero datos
```

freopen

Desvincula el dispositivo o fichero actualmente asociado con el flujo referenciado por *pflujo* y reasigna *pflujo* al fichero identificado por *nomfi*.

```
#include <stdio>
FILE *freopen(const char *nomfi, const char *modo, FILE *pflujo);

pf = freopen("datos", "w", stdout);
```

fclose

Cierra el flujo referenciado por *pf*.

```
#include <stdio>
int fclose(FILE *pf);
```

ferror

Verifica si ocurrió un error en la última operación de E/S.

```
#include <stdio>
int ferror(FILE *pf);
```

clearerr

Pone a 0 los bits de error que estén a 1, incluido el bit de fin de fichero.

```
#include <stdio>
void clearerr(FILE *pf);

if (ferror(pf))
{
    printf("Error al escribir en el fichero\n");
    clearerr(pf);
}
```

feof

Devuelve un valor distinto de 0 cuando se intenta leer más allá de la marca *eof* (*end of file* - fin de fichero), no cuando se lee el último registro. En otro caso devuelve un 0.

```
#include <stdio>
int feof(FILE *pf);

while (!feof(pf)) // mientras no se llegue al final del fichero
{
    // Leer aquí el siguiente registro del fichero
}
fclose(pf);
```

ftell

Devuelve la posición actual en el fichero asociado con *pf* del puntero de L/E, o bien el valor $-1L$ si ocurre un error. Esta posición es relativa al principio del fichero.

```
#include <stdio>
long ftell(FILE *pf);
```

fseek

Mueve el puntero de L/E del fichero asociado con *pf* a una nueva localización desplazada *desp* bytes (un valor positivo avanza el puntero y un valor negativo lo retrocede) de la posición especificada por el argumento *pos*, la cual puede ser una de las siguientes:

SEEK_SET Hace referencia a la primera posición en el fichero.

SEEK_CUR Hace referencia a la posición actual del puntero de L/E.
SEEK_END Hace referencia a la última posición en el fichero.

```
#include <cstdio>
int fseek(FILE *pf, long desp, int pos);

// Calcular el nº total de registros un fichero
fseek(pf, 0L, SEEK_END);
totalreg = (int)ftell(pf)/sizeof(registro);
```

rewind

Mueve el puntero de L/E al principio del fichero asociado con *pf*.

```
#include <cstdio>
void rewind(FILE *pf);
```

fputc

Escribe un carácter *car* en la posición indicada por el puntero de lectura/escritura (L/E) del fichero o dispositivo asociado con *pf*.

```
#include <cstdio>
int fputc(int car, FILE *pf);
```

fgetc

Lee un carácter de la posición indicada por el puntero de L/E del fichero o dispositivo asociado con *pf* y avanza al siguiente carácter a leer. Devuelve el carácter leído o un **EOF**, si ocurre un error o se detecta el final del fichero.

```
#include <cstdio>
int fgetc(FILE *pf);
```

fputs

Permite copiar una cadena de caracteres en un fichero o dispositivo.

```
#include <cstdio>
int fputs(const char *cadena, FILE *pf);
```

fgets

Permite leer una cadena de caracteres de un fichero o dispositivo. Devuelve **NULL** si ocurre un error.

```
#include <stdio>
char *fgets(char *cadena, int n, FILE *pf);
```

fprintf

Permite escribir sus argumentos, con el formato especificado, en un fichero o dispositivo.

```
#include <stdio>
int fprintf(FILE *pf, const char *formato[, arg]...);
```

fscanf

Permite leer los argumentos especificados, con el formato especificado, desde un fichero o dispositivo. Devuelve un **EOF** si se detecta el final del fichero.

```
#include <stdio>
int fscanf(FILE *pf, const char *formato[, arg]...);
```

fwrite

Permite escribir *c* elementos de longitud *n* bytes almacenados en el *buffer* especificado, en el fichero asociado con *pf*.

```
#include <stdio>
size_t fwrite(const void *buffer, size_t n, size_t c, FILE *pf);
```

```
FILE *pf1 = NULL, *pf2 = NULL;
char car, cadena[36];
gets(cadena); car = getchar();
// ...
fwrite(&car, sizeof(char), 1, pf1);
fwrite(cadena, sizeof(cadena), 1, pf2);
```

fread

Permite leer *c* elementos de longitud *n* bytes del fichero asociado con *pf* y los almacena en el *buffer* especificado.

```
#include <stdio>
```

```
size_t fread(void *buffer, size_t n, size_t c, FILE *pf);
```

```
FILE *pf1 = NULL, *pf2 = NULL;  
char car, cadena[36];  
// ...  
fread(&car, sizeof(char), 1, pf);  
fread(cadena, sizeof(cadena), 1, pf);
```

fflush

Escribe en el fichero asociado con el flujo apuntado por *pf* el contenido del *buffer* definido para este flujo. En Windows, no en UNIX, si el fichero en lugar de estar abierto para escribir está abierto para leer, **fflush** borra el contenido del *buffer*.

```
#include <cstdio>  
int fflush(FILE *pf);
```

MISCELÁNEA **system**

Envía una orden al sistema operativo.

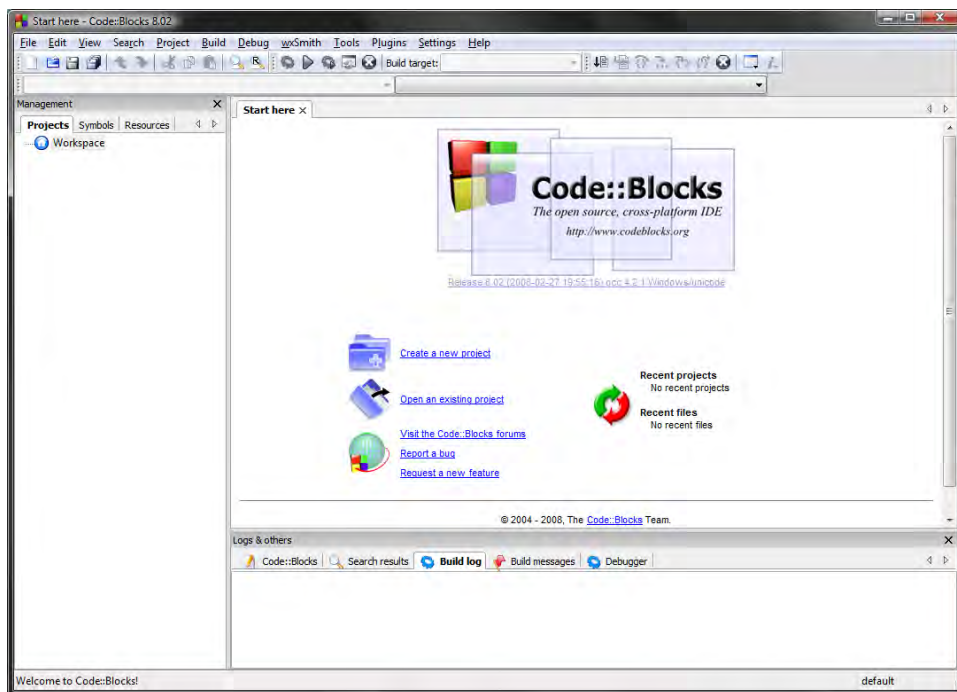
```
#include <cstdlib>  
int system(const char *cadena-de-caracteres);  
  
system("cls"); // limpiar la pantalla en Windows  
system("clear"); // limpiar la pantalla en UNIX
```

ENTORNOS DE DESARROLLO

Cuando se utiliza un entorno de desarrollo integrado (EDI), lo primero que hay que hacer una vez instalado es asegurarse de que las rutas donde se localizan las herramientas, las bibliotecas, la documentación y los ficheros fuente hayan sido establecidas; algunos EDI sólo requieren la ruta donde se instaló el compilador. Este proceso normalmente se ejecuta automáticamente durante el proceso de instalación de dicho entorno. Si no es así, el entorno proporcionará algún menú con las órdenes apropiadas para realizar dicho proceso. Por ejemplo, en el EDI que se presenta a continuación puede comprobar esto a través de las opciones del menú *Settings*.

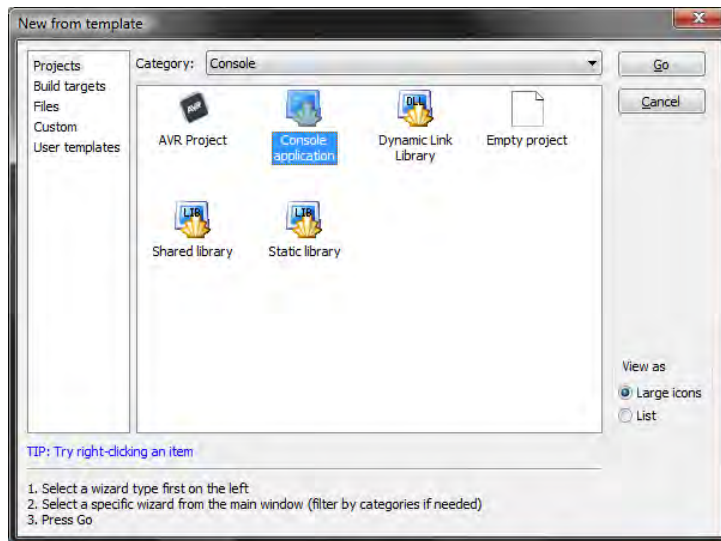
CodeBlocks

En la figura siguiente se puede observar el aspecto del entorno de desarrollo integrado *CodeBlocks* incluido en el CD que acompaña al libro.

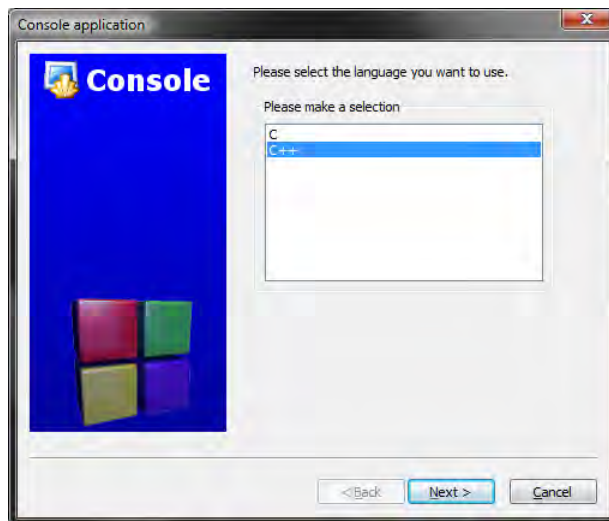


Para editar y ejecutar el programa *HolaMundo.cpp* visto en el capítulo 1, o cualquier otro programa, utilizando este entorno de desarrollo integrado, los pasos a seguir se indican a continuación:

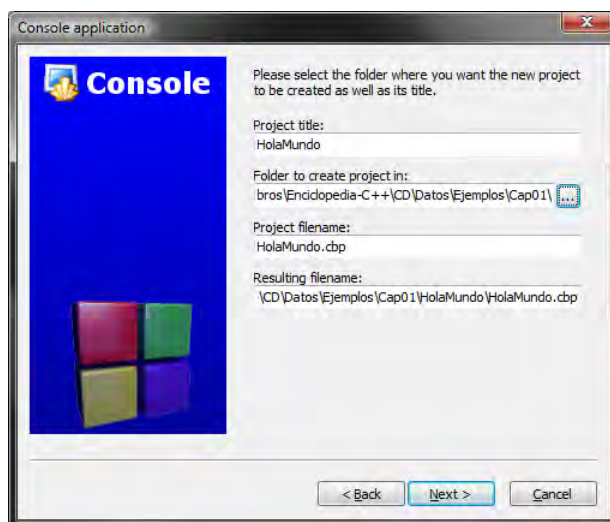
1. Suponiendo que ya está visualizado el entorno de desarrollo, creamos un nuevo proyecto C++ (*File, New, Project*). Se muestra la ventana siguiente:



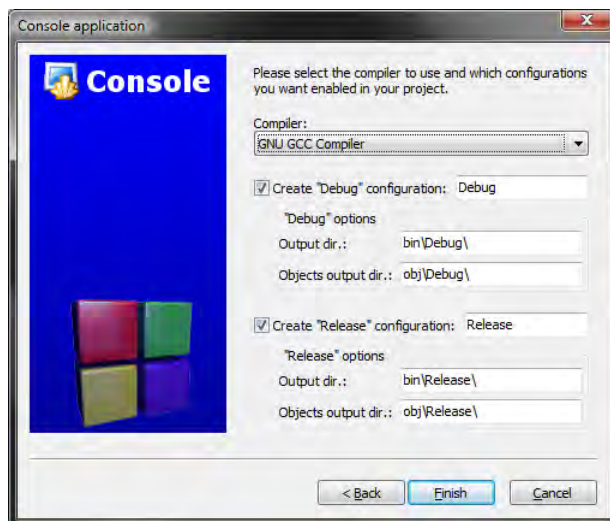
2. Elegimos la categoría consola (*Console*), la plantilla *Console application* y pulsamos el botón *Go*.



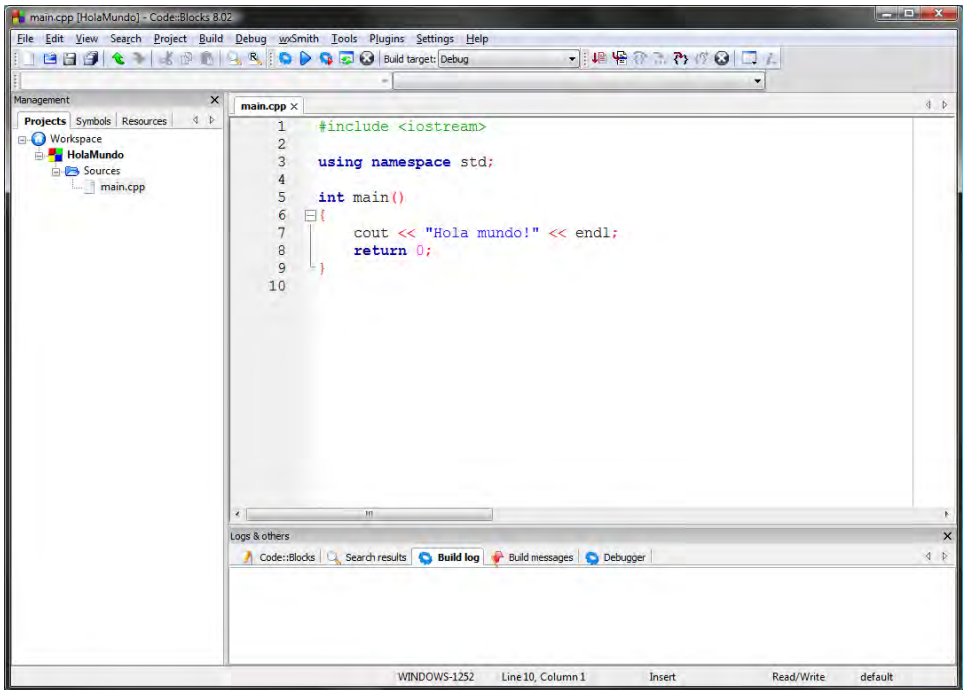
3. Seleccionamos el lenguaje C++ y hacemos clic en el botón *Next*.



4. Especificamos el nombre del proyecto, la carpeta donde será guardado y hacemos clic en *Next*.



5. Si los datos presentados en la ventana anterior son correctos, hacemos clic en el botón *Finish*. El proyecto está creado; contiene un fichero *main.cpp* que incluye la función **main** por donde se iniciará y finalizará la ejecución del programa.

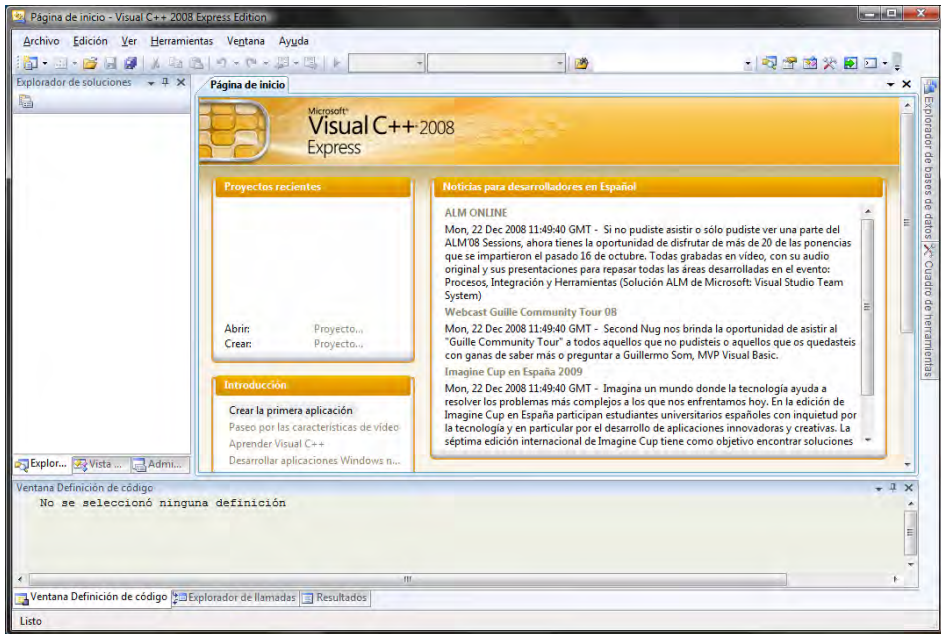


6. A continuación, según se puede observar en la figura anterior, editamos el código que compone el programa y lo guardamos.
7. Después, para compilar el programa, ejecutamos la orden *Build* del menú *Build* y, una vez compilado (sin errores), lo podemos ejecutar seleccionando la orden *Run* del mismo menú (si no pudiéramos ver la ventana con los resultados porque desaparece -no es el caso-, añadiríamos al final de la función **main**, antes de **return**, la sentencia “`system("pause");`” y al principio del fichero *.cpp* la directriz `#include <cstdlib>`, si fuera necesario).

En el caso de que la aplicación esté compuesta por varios ficheros fuente, simplemente tendremos que añadirlos al proyecto ejecutando la orden *File* del menú *File*.

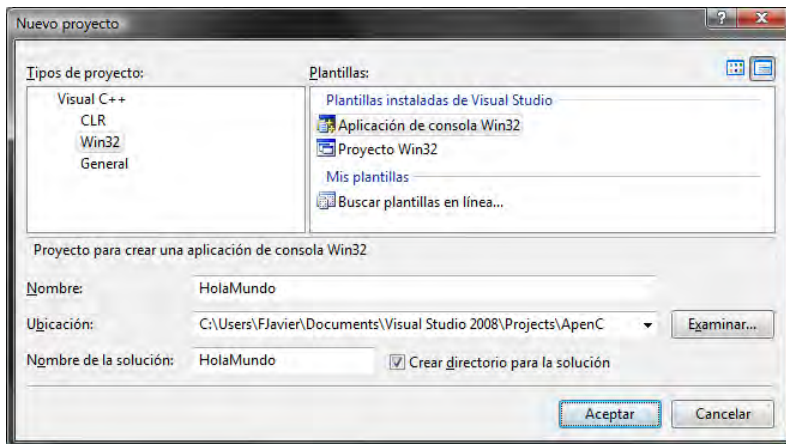
Microsoft Visual C++

En la figura siguiente se puede observar la página de inicio del entorno de desarrollo integrado *Microsoft Visual C++*.

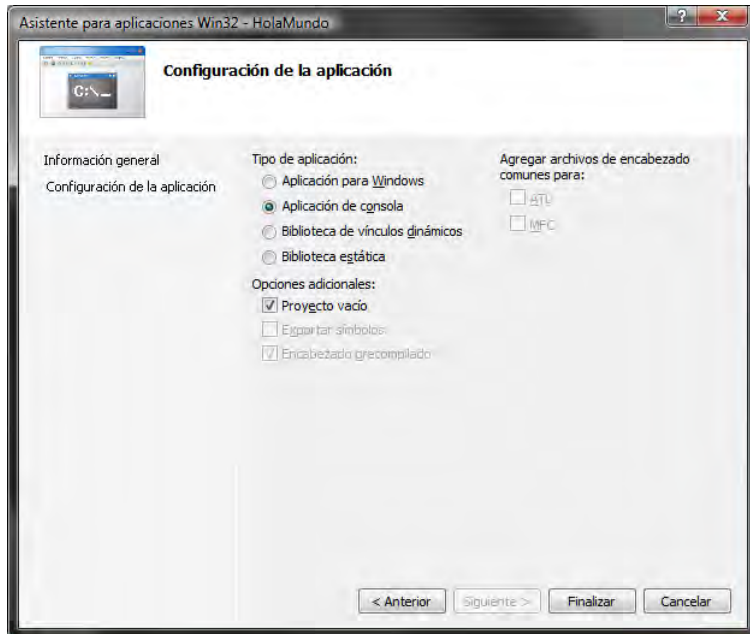


Para editar y ejecutar el programa *HolaMundo.cpp* expuesto en el capítulo 1 utilizando este entorno de desarrollo, los pasos a seguir son los siguientes:

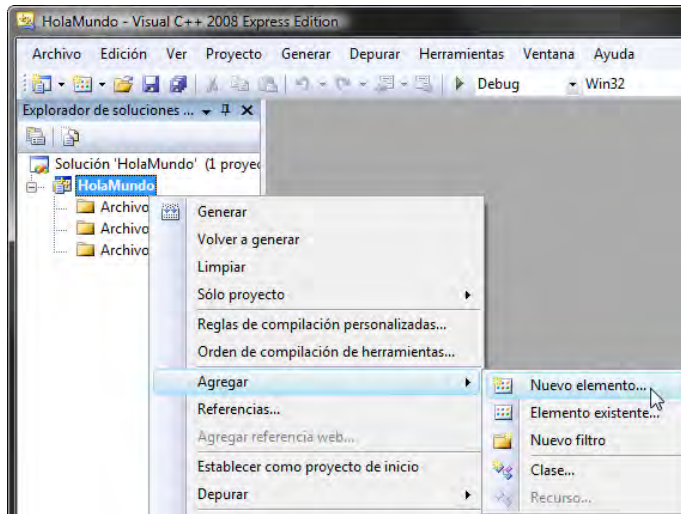
1. Partiendo de la página de inicio de *Visual C++*, hacemos clic en el botón *Crear proyecto* para crear un proyecto nuevo o bien ejecutamos la orden *Archivo > Nuevo > Proyecto*. Esta acción hará que se visualice una ventana que mostrará en su panel izquierdo los tipos de proyectos que se pueden crear, y en su panel derecho las plantillas que se pueden utilizar; la elección de una o de otra dependerá del tipo de aplicación que deseemos construir. La figura siguiente muestra esta ventana:



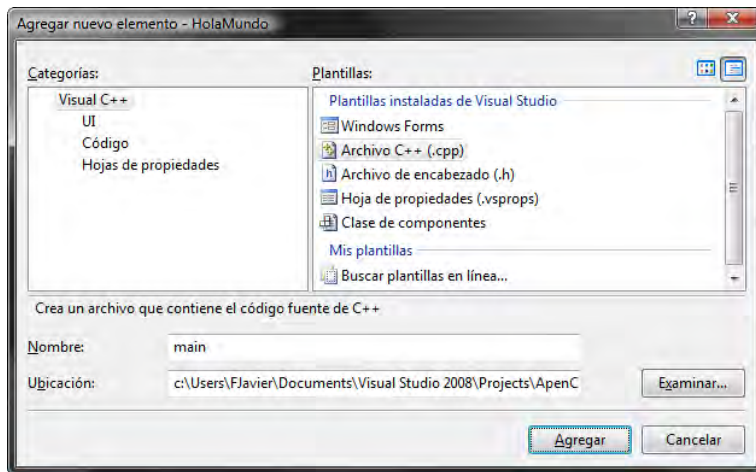
2. Para nuestro ejemplo, elegimos el tipo de proyecto *Visual C++ Win32* y la plantilla *Aplicación de consola Win32*. Después especificamos el nombre del proyecto y su ubicación; observe que el proyecto será creado en una carpeta con el mismo nombre. A continuación pulsamos el botón *Aceptar*. Esta acción visualizará la ventana mostrada en la figura siguiente, que permitirá establecer las opciones necesarias para generar una aplicación de consola partiendo de un proyecto vacío:



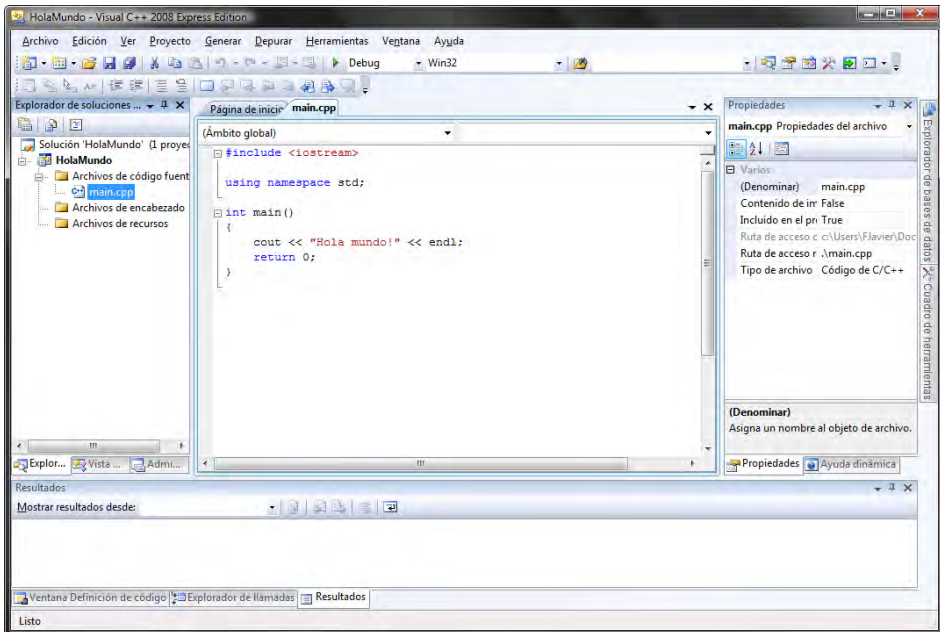
3. Una vez configurada la aplicación, pulsamos el botón *Finalizar*. El resultado será un proyecto vacío al que podremos añadir ficheros. Por ejemplo, para añadir el fichero *HolaMundo.cpp*, hacemos clic con el botón derecho del ratón sobre el nombre del proyecto y seleccionamos la orden *Agregar > Nuevo elemento...*



4. La acción ejecutada en el punto anterior muestra la ventana que se muestra a continuación, la cual nos permitirá elegir la plantilla para el fichero, en nuestro caso *Archivo C++ (.cpp)*, y especificar el nombre y la ubicación del mismo.



5. El siguiente paso es escribir el código que se almacenará en este fichero, según muestra la figura siguiente:



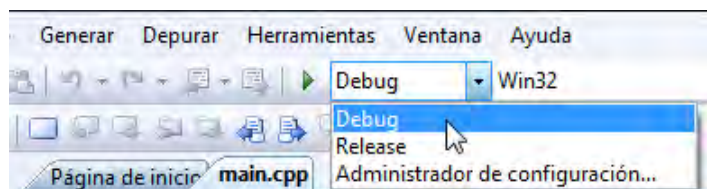
En esta figura observamos una ventana principal que contiene otras ventanas con varias páginas cada una de ellas. La que está en la parte central está mostrando la página de edición del fichero *main.cpp* que estamos editando y tiene oculta la página de inicio. La que está en la parte izquierda está mostrando el explorador de soluciones; éste lista el nombre de la solución (una solución puede contener uno o más proyectos), el nombre del proyecto o proyectos y el nombre de los ficheros que componen el proyecto; en nuestro caso sólo tenemos el fichero *main.cpp* donde escribiremos el código de las acciones que tiene que llevar a cabo nuestra aplicación; el explorador de soluciones oculta la vista de clases, cuya misión es mostrar el conjunto de clases que forman una aplicación orientada a objetos; haga clic en la pestaña *Vista de clases* para observar su contenido. La ventana que hay en la parte derecha muestra la página de propiedades y oculta la página correspondiente a la ayuda dinámica; haga clic en la pestaña *Ayuda dinámica* si quiere consultar la ayuda relacionada con la selección que haya realizado en la página de edición. Y la ventana que hay debajo de la página de edición puede mostrar varias páginas, por ejemplo, la de resultados de la compilación.

6. Una vez editado el programa, para compilarlo ejecutamos la orden *Generar ...* del menú *Generar* y para ejecutarlo, seleccionamos la orden *Iniciar sin depurar* del menú *Depurar* o bien pulsamos las teclas *Ctrl+F5*.

DEPURAR LA APLICACIÓN

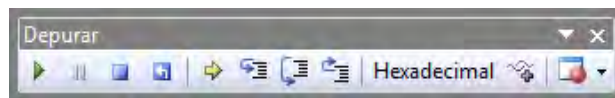
¿Por qué se depura una aplicación? Porque los resultados que estamos obteniendo con la misma no son correctos y no sabemos por qué. El proceso de depuración consiste en ejecutar la aplicación paso a paso, indistintamente por sentencias o por funciones, con el fin de observar el flujo seguido durante su ejecución, así como los resultados intermedios que se van sucediendo, con la finalidad de detectar las anomalías que producen un resultado final erróneo. Para llevarlo a cabo es preciso compilar la aplicación indicando que va a ser depurada; de esta forma, el compilador añadirá el código que permitirá este proceso.

Hay dos configuraciones bajo las que se puede compilar una aplicación: *Release* y *Debug*. La primera permite obtener un programa ejecutable optimizado en código y en velocidad, y la segunda, un programa ejecutable con código extra necesario para depurar la aplicación.



Por ejemplo, para depurar una aplicación utilizando el depurador del entorno de desarrollo de *Visual C++*, debe activar la configuración *Debug* antes de iniciar su compilación. Para ello, proceda como muestra la figura anterior.

Una vez construida la aplicación bajo la configuración *Debug* podrá, si lo necesita, depurar la misma. Para ello, ejecute la orden *Depurar > Paso por instrucciones* y utilice las órdenes del menú *Depurar* o los botones correspondientes de la barra de herramientas (para saber el significado de cada botón, ponga el puntero del ratón sobre cada uno de ellos).



De forma resumida, las órdenes disponibles para depurar una aplicación son las siguientes:

- *Continuar* o *F5*. Continúa la ejecución de la aplicación en modo depuración hasta encontrar un punto de parada o hasta el final si no hay puntos de parada.
- *Interrumpir todos*. El depurador detendrá la ejecución de todos los programas que se ejecutan bajo su control.

- *Detener depuración* o *Mayús+F5*. Detiene el proceso de depuración.
- *Reiniciar* o *Ctrl+Mayús+F5*. Reinicia la ejecución de la aplicación en modo depuración.
- *Mostrar la instrucción siguiente*. Muestra la siguiente instrucción a ejecutar.
- *Paso a paso por instrucciones* o *F11*. Ejecuta la aplicación paso a paso. Si la línea a ejecutar coincide con una llamada a una función definida por el usuario, dicha función también se ejecuta paso a paso.
- *Paso a paso por procedimientos* o *F10*. Ejecuta la aplicación paso a paso. Si la línea a ejecutar coincide con una llamada a una función definida por el usuario, dicha función no se ejecuta paso a paso, sino de una sola vez.
- *Paso a paso para salir* o *Mayús+F11*. Cuando una función definida por el usuario ha sido invocada para ejecutarse paso a paso, utilizando esta orden se puede finalizar su ejecución en un solo paso.
- *Insertar/Quitar punto de interrupción* o *F9*. Pone o quita un punto de parada en la línea sobre la que está el punto de inserción.
- *Ejecutar hasta el cursor* o *Ctrl+F10*. Ejecuta el código que hay entre la última línea ejecutada y la línea donde se encuentra el punto de inserción.
- *Inspección rápida* o *Ctrl+Alt+Q*. Visualiza el valor de la variable que está bajo el punto de inserción o el valor de la expresión seleccionada (sombreada).

Para ejecutar la aplicación en un solo paso, seleccione la orden *Iniciar sin depurar* (*Ctrl+F5*) del menú *Depurar*.

Con otro entorno integrado de desarrollo, por ejemplo *CodeBlocks*, los pasos a seguir para depurar una aplicación son similares.

MICROSOFT C++: INTERFAZ DE LÍNEA DE ÓRDENES

Los ficheros que componen una aplicación C++ pueden ser escritos utilizando cualquier editor de texto ASCII; por ejemplo, el *Bloc de notas*. Una vez editados y guardados todos los ficheros que componen la aplicación, el siguiente paso es compilarlos y enlazarlos para obtener el fichero ejecutable correspondiente a la misma. La orden para realizar estas operaciones utilizando la implementación *Microsoft C++* es la siguiente:

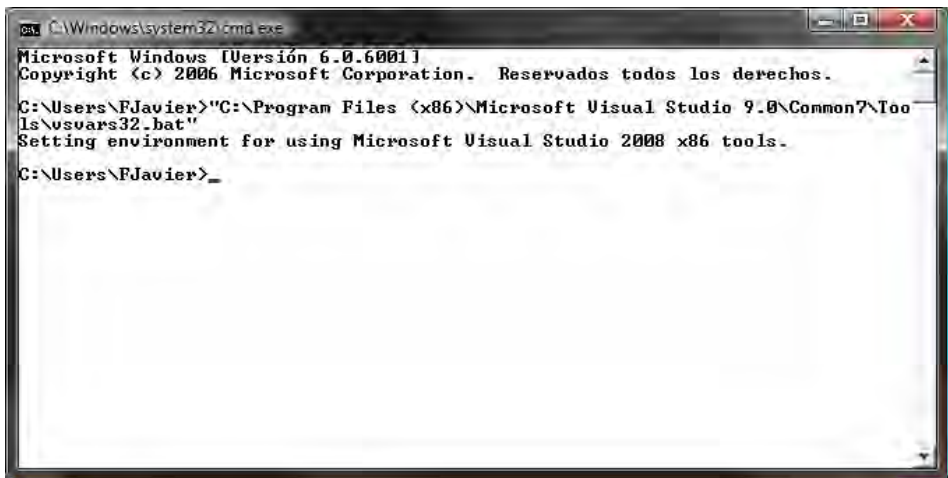
```
cl fichero01.cpp [fichero02 [fichero03] ...]
```

El nombre del fichero ejecutable resultante será el mismo que el nombre del primer fichero especificado, pero con extensión *.exe*.

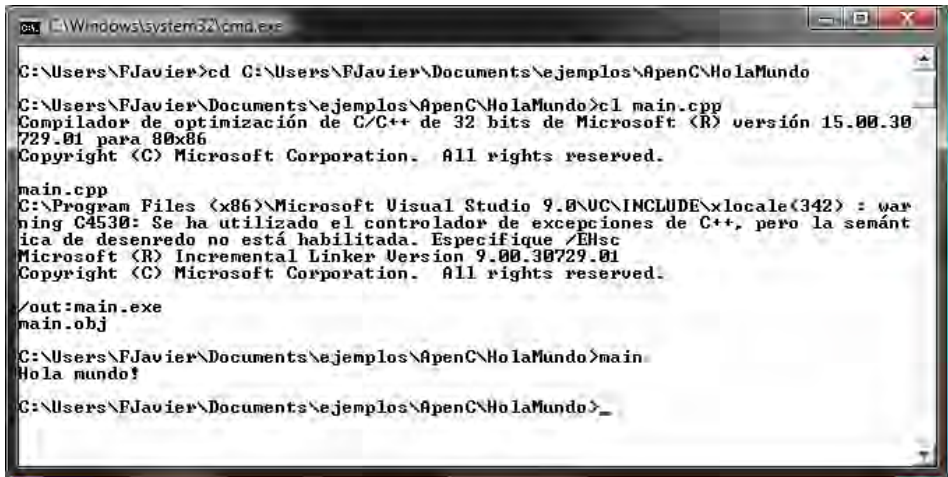
Previamente, para que el sistema operativo encuentre la utilidad *cl*, los ficheros de cabecera (directriz **include**) y las bibliotecas dinámicas y estáticas, cuando son invocados desde la línea de órdenes, hay que definir en el entorno de trabajo las siguientes variables:

```
set path=%path%;ruta de los ficheros .exe y .dll
set include=ruta de los ficheros .h
set lib=ruta de los ficheros .lib
```

La expresión *%path%* representa el valor actual de la variable de entorno *path*. Una ruta va separada de la anterior por un punto y coma. Estas variables también pueden ser establecidas ejecutando el fichero *vcvars32.bat* que aporta Visual C++. En la figura siguiente puede observarse un ejemplo:



Una vez establecidas estas variables, ya puede invocar al compilador C++ y al enlazador. En la figura siguiente se puede observar, como ejemplo, el proceso seguido para compilar *main.cpp*:



```

C:\Windows\system32\cmd.exe

C:\Users\FJavier>cd C:\Users\FJavier\Documents\ejemplos\ApnC\HolaMundo

C:\Users\FJavier\Documents\ejemplos\ApnC\HolaMundo>cl main.cpp
Compilador de optimización de C/C++ de 32 bits de Microsoft (R) versión 15.00.30729.01 para 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

main.cpp
C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\INCLUDE\xlocale(342) : warning C4530: Se ha utilizado el controlador de excepciones de C++, pero la semántica de desenredo no está habilitada. Especifique /EHsc
Microsoft (R) Incremental Linker Version 9.00.30729.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:main.exe
main.obj

C:\Users\FJavier\Documents\ejemplos\ApnC\HolaMundo>main
Hola mundo!

C:\Users\FJavier\Documents\ejemplos\ApnC\HolaMundo>

```

Observe que antes de invocar al compilador hemos cambiado al directorio de la aplicación (*cd*). Después invocamos al compilador C++ (*cl*). El resultado es *main.exe*. Para ejecutar este fichero, escriba *main* en la línea de órdenes y pulse *Entrar*.

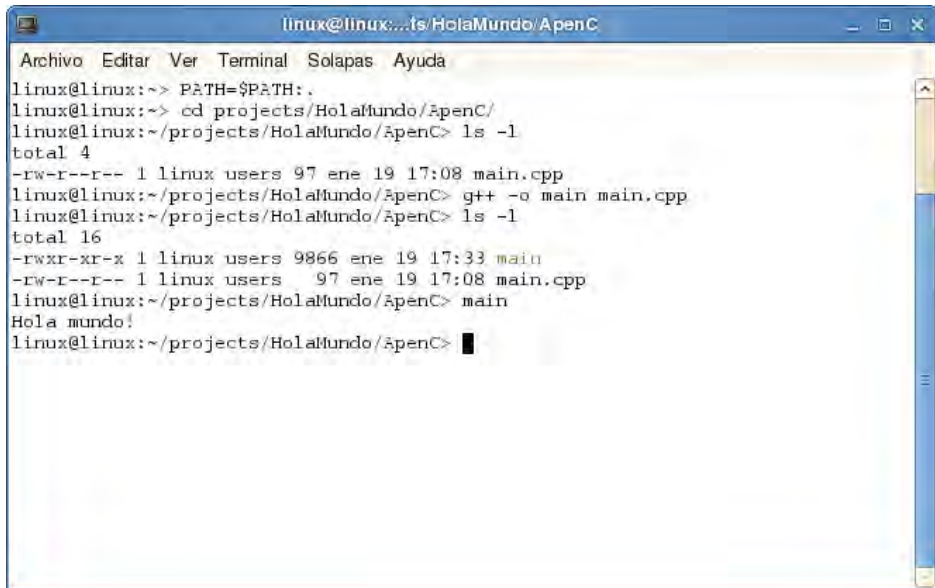
LINUX: INTERFAZ DE LÍNEA DE ÓRDENES

Los ficheros que componen una aplicación C++ realizada bajo GNU-Linux pueden ser escritos utilizando cualquier editor de texto ASCII proporcionado por éste. Una vez editados y guardados todos los ficheros que componen la aplicación, el siguiente paso es compilarlos y enlazarlos para obtener el fichero ejecutable correspondiente a la misma. La orden para realizar estas operaciones es la siguiente:

```
g++ fichero01.cpp [fichero02 [fichero03] ...] -o fichero_ejecutable
```

En el caso de Linux, las rutas de acceso para que el sistema operativo encuentre la utilidad *g++*, los ficheros de cabecera y las bibliotecas, cuando son invocados desde la línea de órdenes, ya están definidas en el entorno de trabajo.

En la figura siguiente se puede observar, como ejemplo, el proceso seguido para compilar *HolaMundo.cpp*:



```

linux@linux:~/projects/HolaMundo/ApenC
Archivo Editar Ver Terminal Solapas Ayuda
linux@linux:~> PATH=$PATH:.
linux@linux:~> cd projects/HolaMundo/ApenC/
linux@linux:~/projects/HolaMundo/ApenC> ls -l
total 4
-rw-r--r-- 1 linux users 97 ene 19 17:08 main.cpp
linux@linux:~/projects/HolaMundo/ApenC> g++ -o main main.cpp
linux@linux:~/projects/HolaMundo/ApenC> ls -l
total 16
-rwxr-xr-x 1 linux users 9866 ene 19 17:33 main
-rw-r--r-- 1 linux users 97 ene 19 17:08 main.cpp
linux@linux:~/projects/HolaMundo/ApenC> main
Hola mundo!
linux@linux:~/projects/HolaMundo/ApenC>

```

Observe que primero hemos cambiado al directorio de la aplicación (*cd*), después hemos visualizado el contenido de ese directorio (*ls -l*) y finalmente hemos invocado al compilador C++ (*g++*). El fichero ejecutable resultante es el especificado por la opción *-o*, en el ejemplo *main*, o *a.out* por omisión.

Para ejecutar la aplicación del ejemplo, escriba *main* en la línea de órdenes y pulse *Entrar*. Si al realizar esta operación se encuentra con que no puede hacerlo porque el sistema no encuentra el fichero especificado, tiene que añadir la ruta del directorio actual de trabajo a la variable de entorno *PATH*. Esto se hace así:

```
PATH=$PATH:.
```

La expresión *\$PATH* representa el valor actual de la variable de entorno *PATH*. Una ruta va separada de la anterior por dos puntos. El directorio actual está representado por el carácter punto.

El depurador gdb de GNU

Cuando se tiene la intención de depurar un programa C escrito bajo GNU, en el momento de compilarlo se debe especificar la opción *-g* o *-g3*. Esta opción indica al compilador que incluya información extra para el depurador en el fichero objeto. Por ejemplo:

```
g++ -g3 prog01.cpp -o prog01.exe
```

La orden anterior compila y enlaza el fichero fuente *prog01.cpp*. El resultado es un fichero ejecutable *prog01.exe* con información para el depurador.

Una vez compilado un programa con las opciones necesarias para depurarlo, invocaremos a *gdb* para proceder a su depuración. La sintaxis es la siguiente:

gdb fichero-ejecutable

El siguiente ejemplo invoca al depurador *gdb* de GNU-Linux, que carga el fichero ejecutable *prog01* en memoria para depurarlo.

```
gdb prog01.exe
```

Una vez que se ha invocado el depurador, desde la línea de órdenes se pueden ejecutar órdenes como las siguientes:

- *break [fichero:]función*. Establece un punto de parada en la función indicada del fichero especificado. Por ejemplo, la siguiente orden pone un punto de parada en la función *escribir*.

```
b escribir
```

- *break [fichero:]línea*. Establece un punto de parada en la línea indicada. Por ejemplo, la siguiente orden pone un punto de parada en la línea 10.

```
b 10
```

- *delete punto-de-parada*. Elimina el punto de parada especificado. Por ejemplo, la siguiente orden elimina el punto de parada 1 (primero).

```
d 1
```

- *run [argumentos]*. Inicia la ejecución de la aplicación que deseamos depurar. La ejecución se detiene al encontrar un punto de parada o al finalizar la aplicación. Por ejemplo:

```
run
```

- *print expresión*. Visualiza el valor de una variable o de una expresión. Por ejemplo, la siguiente orden visualiza el valor de la variable *total*.

```
p total
```

- *next*. Ejecuta la línea siguiente. Si la línea coincide con una llamada a una función definida por el usuario, no se entra a depurar la función. Por ejemplo:

n

- *continue*. Continúa con la ejecución de la aplicación. Por ejemplo:

c

- *step*. Ejecuta la línea siguiente. Si la línea coincide con una llamada a una función definida por el usuario, se entra a depurar la función. Por ejemplo:

s

- *list*. Visualiza el código fuente. Por ejemplo:

l

- *bt*. Visualiza el estado de la pila de llamadas en curso (las llamadas a funciones).
- *help [orden]*. Solicita ayuda sobre la orden especificada.
- *quit*. Finaliza el trabajo de depuración.

INSTALACIÓN DEL PAQUETE DE DESARROLLO

En el apéndice *Entornos de desarrollo* hemos visto cómo escribir y ejecutar una aplicación C++ desde dos entornos de desarrollo diferentes: *CodeBlocks*, que incluye un compilador C/C++ de *GCC (GNU Compiler Collection)*, y *Microsoft Visual Studio* (o bien *Microsoft Visual C++ Express*), que incluye el compilador Microsoft C/C++. También hemos visto que podemos hacerlo de dos formas diferentes: editando, compilando y depurando desde el entorno de desarrollo, o bien desde la línea de órdenes. Veamos a continuación cómo instalar estos compiladores en una plataforma Windows (Windows 2000/XP/Vista).

INSTALACIÓN DE MinGW

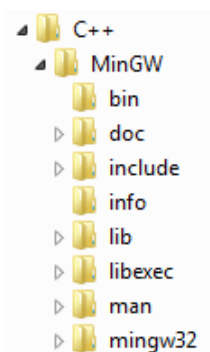
MinGW (Minimalist GNU for Win32) es un paquete que proporciona una versión nativa de Win32 de GCC (*gcc*, *g++*, *g77*, etc.), el depurador *gdb*, *make*, *win32api*, y otras utilidades. Se puede realizar una instalación personalizada instalando por una parte la implementación GCC, y por otra el entorno de desarrollo integrado (EDI) *CodeBlocks*, o bien se puede instalar una versión de *CodeBlocks* que ya incluye *MinGW*. En nuestro caso vamos a instalar la implementación *MinGW* y el entorno integrado *CodeBlocks* por separado. De esta forma podrá instalar otros EDI como *Eclipse* o *NetBeans* que necesitan de GCC.

Para realizar la instalación descargue el fichero *MinGW-x.x.x.exe*, o bien utilice la versión suministrada en el CD del libro y ejecútelo. Después, siguiendo los pasos especificados por el programa de instalación, seleccione *descargar e instalar*, acepte el acuerdo de licencia, elija la versión que quiere descargar (se recomienda descargar la *actual*), seleccione los componentes que desea instalar (al

menos, como muestra la figura siguiente, *MinGW* y *g++*), seleccione la carpeta donde lo quiere instalar y proceda a la descarga e instalación.



La figura siguiente muestra un ejemplo de instalación:



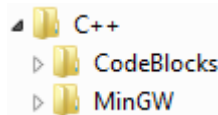
Observe si en la carpeta *bin* están las utilidades *gdb.exe* (depurador) y *make.exe* o *mingw32-make.exe* (para la construcción de proyectos). Si no están, descárguelos e instálelos.

Esta instalación le permitirá editar, compilar, ejecutar y depurar sus programas C++ desde una ventana de consola. Para ello, una vez abierta la ventana debe establecer la siguiente variable de entorno:

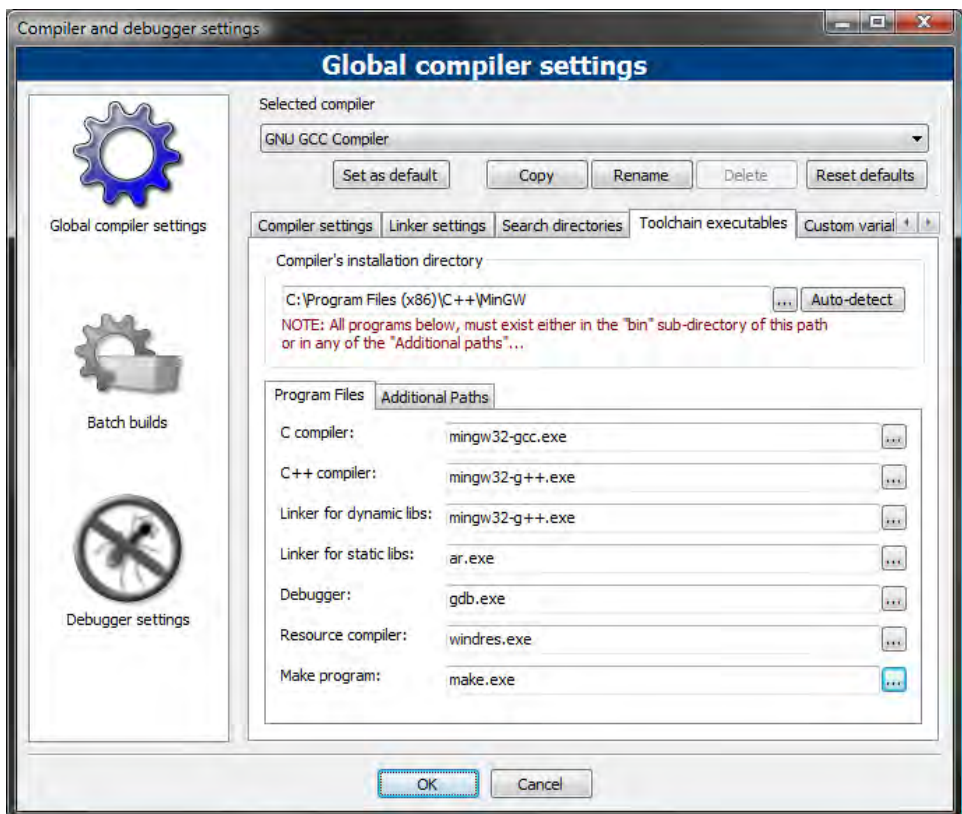
```
SET PATH=%PATH%;C:\...\MinGW\bin
```


Instalación de CodeBlocks

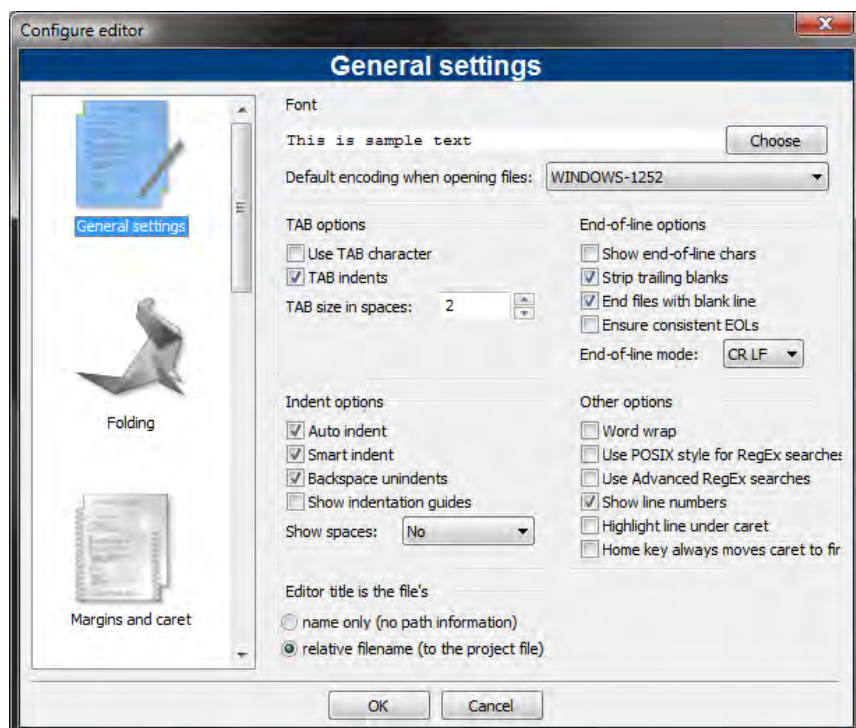
Una vez instalado el entorno de desarrollo de C++ puede instalar un entorno de desarrollo integrado (EDI) que integre el editor soportado por él y el compilador y el depurador anteriormente instalados. Para ello, ejecute el fichero *codeblocks-8.02-setup.exe* localizado en la carpeta *EDI\MinGW+CodeBlocks* del CD del libro, o bien descárguelo de Internet, e instálelo, según la figura anterior, en la carpeta C++. El resultado será similar al presentado por la figura siguiente:



A continuación abra el EDI, seleccione en la orden *Compiler and debugger...* del menú *Settings*, haga clic en la pestaña *Toolchain executables* de la ventana que se visualiza y verifique que la ruta de *MinGW* es la especificada y que las utilidades seleccionadas están en la carpeta *MinGW\bin*.



Finalmente, personalice la instalación a su medida a través de las órdenes *Editor...* y *Environment...* del menú *Settings*. Por ejemplo, active la casilla de verificación *Show line numbers* si quiere mostrar los números de la líneas del programa.



Estas dos instalaciones que acabamos de realizar pueden ser hechas de una sola vez ejecutando el fichero *codeblocks-8.02mingw-setup.exe* localizado en la carpeta *EDI\CodeBlocks* del CD del libro, o bien descargándolo de Internet, e instalándolo en la carpeta deseada. Esta forma de proceder es menos versátil que la anterior, ya que no permite actualizar los paquetes *MinGW* y *CodeBlocks* de forma independiente y tampoco deja *MinGW* a disposición de otros EDI.

INSTALACIÓN DE Microsoft C/C++

También, si lo prefiere puede utilizar el compilador C/C++ de Microsoft. Para ello, tiene que instalar el paquete *Microsoft Visual Studio* o *Microsoft Visual C++ Express Edition* (esta última versión puede descargarla de forma gratuita de Internet), ya que el kit de desarrollo de software (*.Net Framework SDK*) de Microsoft sólo incluye la colección de compiladores C# y Visual Basic.

CÓDIGOS DE CARACTERES

UTILIZACIÓN DE CARACTERES ANSI CON WINDOWS

Una tabla de códigos es un juego de caracteres donde cada uno tiene asignado un número utilizado para su representación interna. Visual Basic utiliza Unicode para almacenar y manipular cadenas, pero también puede manipular caracteres en otros códigos como ANSI o ASCII.

ANSI (*American National Standards Institute*) es el juego de caracteres estándar más utilizado por los equipos personales. Como el estándar ANSI sólo utiliza un byte para representar un carácter, está limitado a un máximo de 256 caracteres. Aunque es adecuado para el inglés, no acepta totalmente otros idiomas. Para escribir un carácter ANSI que no esté en el teclado:

1. Localice en la tabla que se muestra en la página siguiente el carácter ANSI que necesite y observe su código numérico.
2. Pulse la tecla *Bloq Núm* (Num Lock) para activar el teclado numérico.
3. Mantenga pulsada la tecla *Alt* y utilice el teclado numérico para pulsar el 0 y a continuación las teclas correspondientes al código del carácter.

Por ejemplo, para escribir el carácter \pm en el entorno Windows, mantenga pulsada la tecla *Alt* mientras escribe 0177 en el teclado numérico. Pruebe en la consola del sistema (línea de órdenes).

Los 128 primeros caracteres (códigos 0 a 127) son los mismos en las tablas de códigos ANSI, ASCII y Unicode.

JUEGO DE CARACTERES ANSI

DEC	CAR	DEC	CAR	DEC	CAR	DEC	CAR
33	!	89	Y	146	'	202	Ê
34	"	90	Z	147	``	203	Ë
35	#	91	[148	"	204	Ì
36	\$	92	\	149	o	205	Í
37	%	93]	150	-	206	Î
38	&	94		151	—	207	Ï
39	'	96	`	152	☒	208	Ð
40	(97	a	153	☒	209	Ñ
41)	98	b	154	☒	210	Ò
42	*	99	c	155	☒	211	Ó
43	+	100	d	156	☒	212	Ô
44	,	101	e	157	☒	213	Õ
45	-	102	f	157	☒	214	Ö
46	.	103	g	159	☒	215	×
47	/	104	h	160		216	Ø
48	0	105	i	161	;	217	Ù
49	1	106	j	162	;	218	Ú
50	2	107	k	163	£	219	Û
51	3	108	l	164	¤	220	Ü
52	4	109	m	165	¥	221	Ý
53	5	110	n	166		222	Þ
54	6	111	o	167	§	223	ß
55	7	112	p	168	"	224	à
56	8	113	q	169	•	225	á
57	9	114	r	170	•	226	â
58	:	115	s	171	“	227	ã
59	;	116	t	172	”	228	ä
60	<	117	u	173	-	229	å
61	=	118	v	174	•	230	æ
62	>	119	w	175	-	231	ç
63	?	120	x	176	•	232	è
64	@	121	y	177	±	233	é
65	A	122	z	178	²	234	ê
66	B	123	{	179	³	235	ë
67	C	124		180	´	236	ì
68	D	125	}	181	µ	237	í
69	E	126	~	182	¶	238	î
70	F	127	☒	183	·	239	ï
71	G	128	☒	184	·	240	ð
72	H	129	☒	185	·	241	ñ
73	I	130	☒	186	°	242	ò
74	J	131	☒	187	»	243	ó
75	K	132	☒	188	¼	244	ô
76	L	133	☒	189	½	245	õ
77	M	134	☒	190	¾	246	ö
78	N	135	☒	191	¿	247	÷
79	O	136	☒	192	À	248	ø
80	P	137	☒	193	Á	249	ù
81	Q	138	☒	194	Â	250	ú
82	R	139	☒	195	Ã	251	û
83	S	140	☒	196	Ä	252	ü
84	T	141	☒	197	Å	253	ý
85	U	142	☒	198	Æ	254	þ
86	V	143	☒	199	Ç	255	ÿ
87	W	144	☒	200	È		
88	X	145	·	201	É		

UTILIZACIÓN DE CARACTERES ASCII

En MS-DOS y fuera del entorno Windows se utiliza el juego de caracteres ASCII. Para escribir un carácter ASCII que no esté en el teclado:

1. Busque el carácter en la tabla de códigos que coincida con la tabla activa. Utilice la orden **chcp** para saber qué tabla de códigos está activa.
2. Mantenga pulsada la tecla *Alt* y utilice el teclado numérico para pulsar las teclas correspondientes al número del carácter que desee.

Por ejemplo, si está utilizando la tabla de códigos 850, para escribir el carácter π mantenga pulsada la tecla *Alt* mientras escribe 227 en el teclado numérico.

JUEGO DE CARACTERES ASCII

VALOR DECIMAL	VALOR HEXA-DECIMAL	CONTROL CARACT.	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.
000	00	NUL		043	2B	+	086	56	V	129	81	ü	172	AC	¼	215	D7	#
001	01	SOH	☺	044	2C	,	087	57	W	130	82	é	173	AD	í	216	D8	≠
002	02	STX	☹	045	2D	-	088	58	X	131	83	ð	174	AE	«	217	D9	┘
003	03	ETX	♥	046	2E	.	089	59	Y	132	84	å	175	AF	»	218	DA	┐
004	04	EOT	♦	047	2F	/	090	5A	Z	133	85	ä	176	B0	▒	219	DB	■
005	05	ENQ	♣	048	30	0	091	5B	[134	86	å	177	B1	▒	220	DC	▀
006	06	ACK	♠	049	31	1	092	5C	\	135	87	æ	178	B2	▒	221	DD	▄
007	07	BEL	•	050	32	2	093	5D	}	136	88	ø	179	B3		222	DE	▄
008	08	BS	◼	051	33	3	094	5E	^	137	89	ø	180	B4	┘	223	DF	▀
009	09	HT	○	052	34	4	095	5F	_	138	8A	è	181	B5	≡	224	E0	α
010	0A	LF	◐	053	35	5	096	60	`	139	8B	ï	182	B6	≡	225	E1	β
011	0B	VT	♂	054	36	6	097	61	a	140	8C	î	183	B7	≡	226	E2	γ
012	0C	FF	♀	055	37	7	098	62	b	141	8D	ï	184	B8	≡	227	E3	π
013	0D	CR	♪	056	38	8	099	63	c	142	8E	Ë	185	B9	≡	228	E4	Σ
014	0E	SO	♫	057	39	9	100	64	d	143	8F	Ë	186	BA	≡	229	E5	σ
015	0F	SI	☼	058	3A	:	101	65	e	144	90	É	187	BB	≡	230	E6	μ
016	10	DLE	▶	059	3B	;	102	66	f	145	91	æ	188	BC	≡	231	E7	τ
017	11	DC1	◀	060	3C	<	103	67	g	146	92	Æ	189	BD	≡	232	E8	φ
018	12	DC2	↑	061	3D	=	104	68	h	147	93	ó	190	BE	≡	233	E9	⊖
019	13	DC3	!!	062	3E	>	105	69	i	148	94	ö	191	BF	≡	234	EA	⊗
020	14	DC4	¶	063	3F	?	106	6A	j	149	95	ö	192	C0	≡	235	EB	δ
021	15	NAK	§	064	40	@	107	6B	k	150	96	ü	193	C1	≡	236	EC	∞
022	16	SYN	—	065	41	A	108	6C	l	151	97	ü	194	C2	≡	237	ED	∅
023	17	ETB	↓	066	42	B	109	6D	m	152	98	ÿ	195	C3	≡	238	EE	€
024	18	CAN	↑	067	43	C	110	6E	n	153	99	ÿ	196	C4	≡	239	EF	∩
025	19	EM	↓	068	44	D	111	6F	o	154	9A	Û	197	C5	+	240	F0	≡
026	1A	SUB	—	069	45	E	112	70	p	155	9B	€	198	C6	≡	241	F1	±
027	1B	ESC	—	070	46	F	113	71	q	156	9C	£	199	C7	≡	242	F2	≥
028	1C	FS	└	071	47	G	114	72	r	157	9D	¥	200	C8	≡	243	F3	≤
029	1D	GS	↔	072	48	H	115	73	s	158	9E	Pt	201	C9	≡	244	F4	↑
030	1E	RS	▲	073	49	I	116	74	t	159	9F	f	202	CA	≡	245	F5	↓
031	1F	US	▼	074	4A	J	117	75	u	160	A0	á	203	CB	≡	246	F6	+
032	20	SP	Space	075	4B	K	118	76	v	161	A1	í	204	CC	≡	247	F7	≈
033	21		!	076	4C	L	119	77	w	162	A2	ó	205	CD	≡	248	F8	°
034	22		"	077	4D	M	120	78	x	163	A3	û	206	CE	≡	249	F9	•
035	23		#	078	4E	N	121	79	y	164	A4	ñ	207	CF	≡	250	FA	·
036	24		\$	079	4F	O	122	7A	z	165	A5	Ñ	208	D0	≡	251	FB	√
037	25		%	080	50	P	123	7B	{	166	A6	°	209	D1	≡	252	FC	∩
038	26		&	081	51	Q	124	7C		167	A7	°	210	D2	≡	253	FD	'
039	27		'	082	52	R	125	7D	}	168	A8	è	211	D3	≡	254	FE	•
040	28		(083	53	S	126	7E	~	169	A9	—	212	D4	≡	255	FF	
041	29)	084	54	T	127	7F	⌢	170	AA	—	213	D5	≡			
042	2A		*	085	55	U	128	80	Ç	171	AB	½	214	D6	≡			

JUEGO DE CARACTERES UNICODE

UNICODE es un juego de caracteres en el que se emplean 2 bytes (16 bits) para representar cada carácter. Esto permite la representación de cualquier carácter en cualquier lenguaje escrito en el mundo, incluyendo los símbolos del chino, japonés o coreano.

Códigos Unicode de los dígitos utilizados en español:

\u0030 - \u0039 0-9 ISO-LATIN-1

Códigos Unicode de las letras y otros caracteres utilizados en español:

\u0024	\$ signo dólar
\u0041 - \u005a	A-Z
\u005f	_
\u0061 - \u007a	a-z
\u00c0 - \u00d6	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö
\u00d8 - \u00f6	Ø Ù Ú Û Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö
\u00f8 - \u00ff	ø ù ú û ü ý þ ÿ

Dos caracteres son idénticos sólo si tienen el mismo código Unicode.

