



UNIVERSIDAD DE MÁLAGA
Dpt. Lenguajes y CC. Computación
E.T.S.I. Informática
Ingeniería Informática

Fundamentos de Programación
con
el Lenguaje de Programación
C++

Vicente Benjumea

Manuel Roldán

6 de julio de 2011



Esta obra está bajo una licencia **Reconocimiento-NoComercial-CompartirIgual 3.0 Unported** de Creative Commons: No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original. Para ver una copia de esta licencia, visite http://creativecommons.org/licenses/by-nc-sa/3.0/deed.es_ES o envíe una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

- Usted es libre de:
 - Copiar, distribuir y comunicar públicamente la obra.
 - Hacer obras derivadas.
- Bajo las siguientes condiciones:
 - Reconocimiento (Attribution) – Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).
 - No comercial (Non commercial) – No puede utilizar esta obra para fines comerciales.
 - Compartir bajo la misma licencia (Share alike) – Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.
- Entendiendo que:
 - Renuncia – Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
 - Dominio Público – Cuando la obra o alguno de sus elementos se halle en el dominio público según la ley vigente aplicable, esta situación no quedará afectada por la licencia.
 - Otros derechos – Los derechos siguientes no quedan afectados por la licencia de ninguna manera:
 - Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.
 - Los derechos morales del autor
 - Derechos que pueden ostentar otras personas sobre la propia obra o su uso, como por ejemplo derechos de imagen o de privacidad.
 - Aviso – Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.

Índice general

Prólogo	9
I Programación Básica	11
1. Un Programa C++	13
2. Tipos Simples	17
2.1. Declaración Vs. Definición	17
2.2. Tipos Simples Predefinidos	17
2.3. Tipos Simples Enumerados	19
2.4. Constantes y Variables	20
2.5. Operadores	21
2.6. Conversiones Automáticas (Implícitas) de Tipos	23
2.7. Conversiones Explícitas de Tipos	23
2.8. Tabla ASCII	25
2.9. Algunas Consideraciones Respecto a Operaciones con Números Reales	25
3. Entrada y Salida de Datos Básica	27
3.1. Salida de Datos	27
3.2. Entrada de Datos	29
3.3. El “Buffer” de Entrada y el “Buffer” de Salida	30
3.4. Otras Operaciones de Entrada y Salida	30
3.5. Control del Estado del Flujo de Datos	30
4. Estructuras de Control	33
4.1. Sentencia, Secuencia y Bloque	33
4.2. Declaraciones Globales y Locales	33
4.3. Sentencias de Asignación	34
4.4. Sentencias de Selección	35
4.5. Sentencias de Iteración. Bucles	37
4.6. Programación Estructurada	39
4.7. Ejemplos	39
5. Subprogramas. Funciones y Procedimientos	41
5.1. Funciones y Procedimientos	41
5.2. Definición de Subprogramas	42
5.3. Ejecución de Subprogramas	43
5.4. Paso de Parámetros. Parámetros por Valor y por Referencia	43
5.5. Criterios de Modularización	45
5.6. Subprogramas “en Línea”	46
5.7. Declaración de Subprogramas. Prototipos	46
5.8. Sobrecarga de Subprogramas y Operadores	46
5.9. Parámetros por Defecto	47

5.10. Subprogramas y Flujos de Entrada y Salida	48
5.11. Pre-Condiciones y Post-Condiciones	48
5.12. Ejemplos	49
6. Tipos Compuestos	51
6.1. Paso de Parámetros de Tipos Compuestos	51
6.2. Cadenas de Caracteres en C++: el Tipo String	52
6.3. Registros o Estructuras	59
6.4. Agregados: el Tipo Array	62
6.5. Resolución de Problemas Utilizando Tipos Compuestos	71
7. Búsqueda y Ordenación	77
7.1. Búsqueda Lineal (Secuencial)	77
7.2. Búsqueda Binaria	78
7.3. Ordenación por Intercambio (Burbuja)	79
7.4. Ordenación por Selección	80
7.5. Ordenación por Inserción	80
7.6. Ordenación por Inserción Binaria	81
7.7. Aplicación de los Algoritmos de Búsqueda y Ordenación	82
8. Otros Tipos Compuestos en C	87
8.1. Agregados o Arrays Predefinidos	87
8.2. Cadenas de Caracteres al Estilo-C	98
8.3. Uniones	104
8.4. Campos de Bits	105
8.5. Resolución de Problemas Utilizando Tipos Compuestos	105
9. Algunas Bibliotecas Útiles	111
II Programación Intermedia	113
10. Almacenamiento en Memoria Secundaria: Ficheros	115
10.1. Flujos de Entrada y Salida Asociados a Ficheros	116
10.2. Entrada de Datos desde Ficheros de Texto	117
10.3. Salida de Datos a Ficheros de Texto	119
10.4. Ejemplos	121
10.5. Otros Tipos de Flujos de Ficheros	126
10.5.1. Ficheros Binarios	126
10.5.2. Acceso Directo en Ficheros	129
10.5.3. Flujos de Entrada y Salida	130
10.6. Flujos de Entrada y Salida Vinculados a Cadenas de Caracteres	130
11. Módulos y Bibliotecas	133
11.1. Interfaz e Implementación del Módulo	133
11.2. Compilación Separada y Enlazado	135
11.3. Espacios de Nombre	136
11.4. Herramientas de Ayuda a la Gestión de la Compilación Separada	141
12. Manejo de Errores. Excepciones	145
12.1. Errores de Programación y Asertos	145
12.2. Situaciones Anómalas Excepcionales	146
12.3. Gestión de Errores Mediante Excepciones	147
12.4. Excepciones Estándares	152

13. Tipos Abstractos de Datos	155
13.1. Tipos Abstractos de Datos en C++: Clases	156
13.1.1. Definición e Implementación de Clases “en Línea”	156
13.1.2. Definición de Clases e Implementación Separada	159
13.2. Métodos Definidos Automáticamente por el Compilador	172
13.3. Requisitos de las Clases Respecto a las Excepciones	173
13.4. Más sobre Métodos y Atributos	173
13.5. Sobrecarga de Operadores	177
14. Introducción a la Programación Genérica. Plantillas	179
14.1. Subprogramas Genéricos	179
14.2. Tipos Abstractos de Datos Genéricos	181
14.3. Parámetros Genéricos por Defecto	184
14.4. Definición de Tipos dentro de la Definición de Tipos Genéricos	185
14.5. Separación de Definición e Implementación	186
15. Memoria Dinámica. Punteros	191
15.1. Punteros	192
15.2. Gestión de Memoria Dinámica	193
15.3. Operaciones con Variables de Tipo Puntero	194
15.4. Paso de Parámetros de Variables de Tipo Puntero	196
15.5. Abstracción en la Gestión de Memoria Dinámica	197
15.6. Estructuras Enlazadas	198
15.7. Operaciones con Listas Enlazadas	200
15.8. Gestión de Memoria Dinámica en Presencia de Excepciones	207
15.9. Comprobación de Gestión de Memoria Dinámica	217
15.10. Operador de Dirección	218
16. Introducción a los Contenedores de la Biblioteca Estándar (STL)	221
16.1. Vector	221
16.2. Stack	225
16.3. Queue	227
16.4. Resolución de Problemas Utilizando Contenedores	229
III Programación Avanzada	233
17. Programación Orientada a Objetos	237
18. Memoria Dinámica Avanzada	241
18.1. Memoria Dinámica de Agregados	241
18.2. Punteros a Subprogramas	242
18.3. Punteros Inteligentes	242
19. Tipos Abstractos de Datos Avanzados	245
19.1. Punteros a Miembros	245
19.2. Ocultar la Implementación	245
19.3. Control de Elementos de un Contenedor	250
20. Programación Genérica Avanzada	255
20.1. Parámetros Genéricos por Defecto	255
20.2. Tipos dentro de Clases Genéricas	255
20.3. Métodos de Clase Genéricos	255
20.4. Amigos Genéricos	256
20.5. Restricciones en Programación Genérica	257

20.6. Especializaciones	258
20.7. Meta-programación	259
20.8. SFINAE	260
21.Buffer y Flujos de Entrada y Salida	263
21.1. Operaciones de Salida	263
21.2. Operaciones de Entrada	263
21.3. Buffer	265
21.4. Redirección Transparente de la Salida Estándar a un String	267
21.5. Ficheros	268
21.6. Ficheros de Entrada	268
21.7. Ficheros de Salida	269
21.8. Ejemplo de Ficheros	269
21.9. Ficheros de Entrada y Salida	270
21.10Flujo de Entrada desde una Cadena	270
21.11Flujo de Salida a una Cadena	270
21.12Jerarquía de Clases de Flujo Estándar	271
22.Técnicas de Programación Usuales en C++	273
22.1. Adquisición de Recursos es Inicialización (RAII)	273
22.1.1. auto_ptr (unique_ptr)	273
22.1.2. RAII Simple de Memoria Dinámica	274
22.1.3. RAII Simple Genérico	275
22.1.4. RAII Genérico	276
22.1.5. RAII Genérico	276
23.Gestión Dinámica de Memoria	279
23.1. Gestión de Memoria Dinámica	279
23.2. Gestión de Memoria Dinámica sin Inicializar	282
23.3. RAII: auto_ptr	282
23.4. Comprobación de Gestión de Memoria Dinámica	283
24.Biblioteca Estándar de C++. STL	285
24.1. Características Comunes	285
24.1.1. Ficheros	285
24.1.2. Contenedores	286
24.1.3. Tipos Definidos	286
24.1.4. Iteradores	286
24.1.5. Acceso	287
24.1.6. Operaciones de Pila y Cola	287
24.1.7. Operaciones de Lista	287
24.1.8. Operaciones	287
24.1.9. Constructores	287
24.1.10.Asignación	288
24.1.11.Operaciones Asociativas	288
24.1.12.Resumen	288
24.1.13.Operaciones sobre Iteradores	288
24.2. Contenedores	288
24.3. Vector	289
24.4. List	290
24.5. Deque	293
24.6. Stack	294
24.7. Queue	295
24.8. Priority-Queue	296
24.9. Map	296

24.10Multimap	297
24.11Set	298
24.12Multiset	299
24.13Bitset	299
24.14Iteradores	300
24.15Directos	300
24.16Inversos	301
24.17Inserters	302
24.18Stream Iterators	302
24.19Operaciones sobre Iteradores	306
24.20Objetos Función y Predicados	306
24.21Algoritmos	309
24.22Garantías (Excepciones) de Operaciones sobre Contenedores	311
24.23Numéricos	312
24.24Límites	312
24.25Run Time Type Information (RTTI)	313
A. Precedencia de Operadores en C	315
B. Precedencia de Operadores en C++	317
C. Biblioteca Básica ANSI-C (+ conio)	321
C.1. cassert	321
C.2. ctype	321
C.3. cmath	321
C.4. cstdlib	322
C.5. climits	323
C.6. cfloat	323
C.7. ctime	323
C.8. cstring	323
C.9. cstdio	324
C.10.cstdarg	325
C.11.conio.h	326
D. El Preprocesador	329
E. Errores Más Comunes	331
F. Características no Contempladas	333
G. Bibliografía	335
Índice	335

Prólogo

Este manual pretende ser una guía de referencia para la utilización del lenguaje de programación C++ en el desarrollo de programas. Está orientada a alumnos de primer curso de programación de Ingeniería Informática. No obstante, hay algunos capítulos y secciones que presentan conceptos avanzados y requieren mayores conocimientos por parte del lector, y por lo tanto exceden de los conocimientos básicos requeridos para un alumno de primer curso de programación. Estos capítulos y secciones que presentan o requieren *conceptos avanzados* se encuentran marcados con un símbolo \mathbb{A} , y no es necesario que sean leídos por el lector, a los cuales podrá acceder en cualquier otro momento posterior. Otros capítulos y secciones muestran estructuras y técnicas de programación *obsoletas* y no recomendadas en C++, sin embargo son mostradas en este manual por completitud. Estos capítulos y secciones están marcados con el símbolo $\boxed{\text{OBS}}$.

Este manual se concibe como *material de apoyo a la docencia*, y requiere de las explicaciones impartidas en clase por el profesor para su aprendizaje. Así mismo, este manual no pretende “enseñar a programar”, supone que el lector posee los fundamentos necesarios relativos a la programación, y simplemente muestra como aplicarlos utilizando el *Lenguaje de Programación C++*.

El lenguaje de programación C++ es un lenguaje muy flexible y versátil, y debido a ello, si se utiliza sin rigor puede dar lugar a construcciones y estructuras de programación complejas, difíciles de comprender y propensas a errores. Debido a ello, restringiremos tanto las estructuras a utilizar como la forma de utilizarlas.

No pretende ser una guía extensa del lenguaje de programación C++. Además, dada la amplitud del lenguaje, hay características del mismo que no han sido contempladas por exceder lo que entendemos que es un curso de programación elemental.

Este manual ha sido elaborado en el Dpto. de Lenguajes y Ciencias de la Computación de la Universidad de Málaga.

ES UNA VERSIÓN PRELIMINAR, INCOMPLETA Y SE ENCUENTRA ACTUALMENTE BAJO DESARROLLO. Se difunde en la creencia de que puede ser útil, aún siendo una versión preliminar.

La última versión de este documento puede ser descargada desde la siguiente página web:

<http://www.lcc.uma.es/%7Evicente/docencia/index.html>

o directamente desde la siguiente dirección:

http://www.lcc.uma.es/%7Evicente/docencia/cpp/manual_referencia_cxx.pdf

Parte I

Programación Básica

Capítulo 1

Un Programa C++

En principio, un *programa C++* se almacena en un fichero cuya extensión será una de las siguientes: “.cpp”, “.cxx”, “.cc”, etc. Más adelante consideraremos programas complejos cuyo código se encuentra distribuido entre varios ficheros (véase 11).

Dentro de este fichero, normalmente, aparecerán al principio unas líneas para incluir las definiciones de los módulos de biblioteca que utilice nuestro programa. Posteriormente, se realizarán declaraciones y definiciones de tipos, de constantes (véase 2) y de subprogramas (véase 5) cuyo ámbito de visibilidad será global a todo el fichero (desde el punto donde ha sido declarado hasta el final del fichero).

De entre las definiciones de subprogramas, debe definirse una función principal, denominada *main*, que indica donde comienza la ejecución del programa. Al finalizar, dicha función devolverá un número entero que indica al Sistema Operativo el estado de terminación tras la ejecución del programa (un número 0 indica terminación normal). En caso de no aparecer explícitamente el valor de retorno de *main*, el sistema recibirá por defecto un valor indicando terminación normal.

Ejemplo de un programa que convierte una cantidad determinada de *euros* a su valor en *pesetas*.

```
//- fichero: euros.cpp -----
#include <iostream>
using namespace std;
const double EUR_PTS = 166.386;
int main()
{
    cout << "Introduce la cantidad (en euros): ";
    double euros;
    cin >> euros;
    double pesetas = euros * EUR_PTS;
    cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl;
    // return 0;
}
//- fin: euros.cpp -----
```

Se deberá compilar el programa (código fuente) contenido en el fichero **euros.cpp** para traducirlo a un programa ejecutable mediante un compilador. En caso de utilizar el compilador *GNU GCC*, la compilación se realizará de la siguiente forma:

```
g++ -ansi -Wall -Werror -o euros euros.cpp
```

cuya ejecución podrá ser como se indica a continuación, donde el texto enmarcado corresponde a una entrada de datos del usuario:

```
Introduce la cantidad (en euros): 3.5 ENTER
3.5 Euros equivalen a 582.351 Pts
```

En algunos entornos de programación, por ejemplo Dev-C++ en Windows, puede ser necesario pausar el programa antes de su terminación, para evitar que desaparezca la ventana de ejecución. En este caso el programa anterior quedaría:

```
//- fichero: euros.cpp -----
#include <iostream>
#include <cstdlib>
using namespace std;
const double EUR_PTS = 166.386;
int main()
{
    cout << "Introduce la cantidad (en euros): ";
    double euros;
    cin >> euros;
    double pesetas = euros * EUR_PTS;
    cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl;
    system("pause"); // llamada para que el S.O. Windows pause el programa
    // return 0;
}
//- fin: euros.cpp -----
```

Ejemplo de un programa que imprime los números menores que uno dado por teclado.

```
//- fichero: numeros.cpp -----
#include <iostream> // biblioteca de entrada/salida
using namespace std; // utilización del espacio de nombres de la biblioteca
// -----
// Imprime los números menores a 'n'
// -----
void imprimir_numeros(int n)
{
    for (int i = 0; i < n; ++i) {
        cout << i << " "; // escribe el valor de 'i'
    }
    cout << endl; // escribe 'fin de línea'
}
// -----
// Imprime los números menores a 'n'
// -----
int main()
{
    int maximo;
    cout << "Introduce un número: ";
    cin >> maximo;
    imprimir_numeros(maximo);
    // return 0;
}
//- fin: numeros.cpp -----
```

En un programa C++ podemos distinguir los siguientes elementos básicos, considerando que las letras minúsculas se consideran diferentes de las letras mayúsculas:

■ Palabras reservadas

Son un conjunto de palabras que tienen un significado predeterminado para el compilador, y sólo pueden ser utilizadas con dicho sentido. Por ejemplo: `using`, `namespace`, `const`, `double`, `int`, `char`, `bool`, `void`, `for`, `while`, `do`, `if`, `switch`, `case`, `default`, `return`, `typedef`, `enum`, `struct`, etc.

■ *Identificadores*

Son nombres elegidos por el programador para representar entidades (tipos, constantes, variables, funciones, etc) en el programa.

Se construyen mediante una secuencia de letras y dígitos, siendo el primer carácter una letra. El carácter '_' se considera como una letra, sin embargo, los nombres que comienzan con dicho carácter se reservan para situaciones especiales, por lo que no deberían utilizarse en programas.

En este manual, seguiremos la siguiente convención para los identificadores:

Constantes Simbólicas: Sólo se utilizarán letras mayúsculas, dígitos y el carácter '_'. Ejemplo: EUR_PTS

Tipos: Comenzarán por una letra mayúscula seguida por letras mayúsculas, minúsculas, dígitos o '_'. Deberá contener al menos una letra minúscula. Ejemplo: **Persona**

Variables: Sólo se utilizarán letras minúsculas, dígitos y el carácter '_'. Ejemplo: euros, pesetas, n, i1, etc.

Funciones: Sólo se utilizarán letras minúsculas, dígitos y el carácter '_'. Ejemplo: imprimir_numeros

Campos de Registros: Sólo se utilizarán letras minúsculas, dígitos y el carácter '_'. Ejemplo: nombre

■ *Constantes literales*

Son valores que aparecen explícitamente en el programa, y podrán ser lógicos, numéricos, caracteres y cadenas. Ejemplo: true, false, 0, 25, 166.386, " Pts", ' ', etc.

■ *Operadores*

Símbolos con significado propio según el contexto en el que se utilicen. Ejemplo: = < > * / % + - < > <= >= == != ++ -- . , etc.

■ *Delimitadores*

Símbolos que indican comienzo o fin de una entidad. Ejemplo: () { } ; , < >

■ *Comentarios y espacios en blanco*

Los espacios en blanco, tabuladores, nueva línea, retorno de carro, avance de página y los comentarios son ignorados por el compilador, excepto en el sentido en que separan elementos.

Los comentarios en un programa es texto que el programador escribe para facilitar la comprensión, o remarcar algún hecho importante a un lector humano, y son, por lo tanto, ignorados por el compilador.

Los comentarios en C++ se expresan de dos formas diferentes:

- Comentarios hasta fin de línea: los símbolos // marcan el comienzo del comentario, que se extiende hasta el final de la línea.

```
// acumula el siguiente número
suma = suma + n; // acumula el valor de 'n'
```

- Comentarios enmarcados: los símbolos /* marcan el comienzo del comentario, que se extiende hasta los símbolos del fin del comentario */

```
/*
 * acumula el siguiente número
 */
suma = suma + n; /* acumula el valor de 'n' */
```


Capítulo 2

Tipos Simples

El *tipo* define las características que tiene una determinada entidad, de tal forma que toda entidad manipulada por un programa lleva asociado un determinado tipo. Las características que el tipo define son:

- El rango de posibles valores que la entidad puede tomar.
- El conjunto de operaciones/manipulaciones aplicables a la entidad.
- El espacio de almacenamiento necesario para almacenar dichos valores.
- La interpretación del valor almacenado.

Los tipos se pueden clasificar en tipos simples y tipos compuestos. Los *tipos simples* se caracterizan porque sus valores son indivisibles, es decir, no se puede acceder o modificar parte de ellos (aunque ésto se pueda realizar indirectamente mediante operaciones de bits) y los *tipos compuestos* se caracterizan por estar formados como un agregado o composición de otros tipos, ya sean simples o compuestos.

2.1. Declaración Vs. Definición

Con objeto de clarificar la terminología, en C++ una *declaración* “presenta” un identificador para el cual la entidad a la que hace referencia deberá ser definida posteriormente.

Una *definición* “establece las características” de una determinada entidad para el identificador al cual se refiere. Toda definición es a su vez también una declaración.

Es obligatorio que por cada entidad, sólo exista **una única definición** en la unidad de compilación, aunque pueden existir varias declaraciones. Así mismo, también es obligatorio la declaración de las entidades que se manipulen en el programa, especificando su tipo, identificador, valores, etc. antes de que sean utilizados.

2.2. Tipos Simples Predefinidos

Los *tipos simples predefinidos* en el lenguaje de programación C++ son:

```
bool char int float double
```

El tipo `bool` se utiliza para representar valores lógicos o booleanos, es decir, los valores “Verdadero” o “Falso” o las constantes lógicas `true` y `false`. Suele almacenarse en el tamaño de palabra más pequeño posible direccionable (normalmente 1 byte).

El tipo `char` se utiliza para representar los caracteres, es decir, símbolos alfanuméricos (dígitos y letras mayúsculas y minúsculas), de puntuación, espacios, control, etc. Normalmente utiliza un espacio de almacenamiento de 1 byte (8 bits) y puede representar 256 valores diferentes.

El tipo `int` se utiliza para representar los números Enteros. Su representación suele coincidir con la definida por el tamaño de palabra del procesador sobre el que va a ser ejecutado, hoy día puede ser de 4 bytes (32 bits), aunque actualmente en los ordenadores más modernos, puede ser de 8 bytes (64 bits).

Puede ser modificado para representar un rango de valores menor mediante el modificador `short` (normalmente 2 bytes [16 bits]) o para representar un rango de valores mayor mediante el modificador `long` (normalmente 4 bytes [32 bits] u 8 bytes [64 bits]) y `long long` (normalmente 8 bytes [64 bits]).

También puede ser modificado para representar solamente números Naturales (enteros positivos) utilizando el modificador `unsigned`.

Tanto el tipo `float` como el `double` se utilizan para representar números reales en formato de punto flotante diferenciándose en el rango de valores que representan, utilizándose el tipo `double` (normalmente 8 bytes [64 bits]) para representar números de punto flotante en “doble precisión” y el tipo `float` (normalmente 4 bytes [32 bits]) para representar la “simple precisión”. El tipo `double` también puede ser modificado con `long` para representar “cuádruple precisión” (normalmente 12 bytes [96 bits]).

Todos los tipos simples tienen la propiedad de ser indivisibles y además mantener una relación de orden entre sus elementos (se les pueden aplicar los operadores relacionales). Se les conoce también como tipos **Escalares**. Todos ellos, salvo los de punto flotante (`float` y `double`), tienen también la propiedad de que cada posible valor tiene un único antecesor y un único sucesor. A éstos se les conoce como tipos **Ordinales** (en terminología C++, también se les conoce como tipos integrales, o enteros).

Valores Límites de los Tipos Predefinidos

Tanto la biblioteca estándar `climits` como `cfloat` definen constantes, accesibles por los programas, que proporcionan los valores límites que pueden contener las entidades (constantes y variables) de tipos simples. Para ello, si un programa necesita acceder a algún valor definido, deberá incluir la biblioteca correspondiente, y utilizar las constantes adecuadas.

Biblioteca `climits`

```
#include <climits>
```

		char	short	int	long	long long
<i>unsigned</i>	máximo	UCHAR_MAX	USHRT_MAX	UINT_MAX	ULONG_MAX	ULONG_LONG_MAX
<i>signed</i>	máximo	SCHAR_MAX	SHRT_MAX	INT_MAX	LONG_MAX	LONG_LONG_MAX
	mínimo	SCHAR_MIN	SHRT_MIN	INT_MIN	LONG_MIN	LONG_LONG_MIN
	máximo	CHAR_MAX				
	mínimo	CHAR_MIN				
	<i>n bits</i>	CHAR_BIT				

Biblioteca `cfloat`

```
#include <cfloat>
```

FLT_EPSILON	Menor número <code>float</code> x tal que $1,0 + x \neq 1,0$
FLT_MAX	Máximo número <code>float</code> de punto flotante
FLT_MIN	Mínimo número <code>float</code> normalizado de punto flotante
DBL_EPSILON	Menor número <code>double</code> x tal que $1,0 + x \neq 1,0$
DBL_MAX	Máximo número <code>double</code> de punto flotante
DBL_MIN	Mínimo número <code>double</code> normalizado de punto flotante
LDBL_EPSILON	Menor número <code>long double</code> x tal que $1,0 + x \neq 1,0$
LDBL_MAX	Máximo número <code>long double</code> de punto flotante
LDBL_MIN	Mínimo número <code>long double</code> normalizado de punto flotante

Para ver el tamaño (en bytes) que ocupa un determinado tipo/entidad en memoria, podemos aplicarle el siguiente operador:

```
■ unsigned sz = sizeof(tipo);
■ unsigned sz = sizeof(variable);
```

Por definición, `sizeof(char)` es 1. El número de bits que ocupa un determinado tipo se puede calcular de la siguiente forma:

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
    unsigned nbytes = sizeof(int);
    unsigned nbits = sizeof(int) / sizeof(char) * CHAR_BIT ;
    cout << "int: "<<nbytes<<" "<<nbits<<" "<<INT_MIN<<" "<<INT_MAX<<endl;
}
```

Veamos un cuadro resumen con los tipos predefinidos, su espacio de almacenamiento y el rango de valores para una máquina de 32 bits, donde para una representación de n bits, en caso de ser un tipo entero con signo, sus valores mínimo y máximo vienen especificados por el rango $[(-2^{n-1}) \dots (+2^{n-1} - 1)]$, y en caso de ser un tipo entero sin signo, sus valores mínimo y máximo vienen especificados por el rango $[0 \dots (+2^n - 1)]$:

Tipo	Bytes	Bits	Min.Valor	Max.Valor
bool	1	8	false	true
char	1	8	-128	127
short	2	16	-32768	32767
int	4	32	-2147483648	2147483647
long	4	32	-2147483648	2147483647
long long	8	64	-9223372036854775808	9223372036854775807
unsigned char	1	8	0	255
unsigned short	2	16	0	65535
unsigned	4	32	0	4294967295
unsigned long	4	32	0	4294967295
unsigned long long	8	64	0	18446744073709551615
float	4	32	1.17549435e-38	3.40282347e+38
double	8	64	2.2250738585072014e-308	1.7976931348623157e+308
long double	12	96	3.36210314311209350626e-4932	1.18973149535723176502e+4932

2.3. Tipos Simples Enumerados

Además de los tipos simples predefinidos, el programador puede definir nuevos tipos simples que expresen mejor las características de las entidades manipuladas por el programa. Así, dicho tipo se definirá en base a una enumeración de los posibles valores que pueda tomar la entidad asociada. A dicho tipo se le denomina *tipo enumerado*, es un tipo simple ordinal, y se define de la siguiente forma:

```
enum Color {
    ROJO,
    AZUL,
    AMARILLO
};
```

De esta forma definimos el tipo `Color`, que definirá una entidad (constante o variable) que podrá tomar cualquiera de los diferentes valores enumerados. Los tipos enumerados, al ser tipos definidos por el programador, no tiene entrada ni salida predefinida por el lenguaje, sino que deberá ser el programador el que especifique (programe) como se realizará la entrada y salida de datos en caso de ser necesaria. Otro ejemplo de enumeración:

```
enum Meses {
    Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio,
    Agosto, Septiembre, Octubre, Noviembre, Diciembre
};
```

Es posible asignar valores concretos a las enumeraciones. Aquellos valores que no se especifiquen serán consecutivos a los anteriores. Si no se especifica ningún valor, al primer valor se le asigna el cero.

```
enum Color {
    ROJO = 1,
    AZUL,
    AMARILLO = 4
};

cout << ROJO << " " << AZUL << " " << AMARILLO << endl; // 1 2 4
```

2.4. Constantes y Variables

Podemos dividir las entidades que nuestro programa manipula en dos clases fundamentales: aquellos cuyo valor no varía durante la ejecución del programa (*constantes*) y aquellos otros cuyo valor puede ir cambiando durante la ejecución del programa (*variables*).

Constantes

Las constantes pueden aparecer a su vez como *constantes literales*, son aquellas cuyo valor aparece directamente en el programa, y como *constantes simbólicas*, aquellas cuyo valor se asocia a un identificador, a través del cual se representa.

Ejemplos de constantes literales:

- Constantes lógicas (bool):

```
false, true
```

- Constantes carácter (**char**), el símbolo constante aparece entre comillas simples:

```
'a', 'b', ..., 'z',
'A', 'B', ..., 'Z',
'0', '1', ..., '9',
' ', '.', ',', ':', ';', ...
```

Así mismo, ciertos caracteres constantes tienen un significado especial (caracteres de escape):

- '\n': fin de línea (newline)
 - '\r': retorno de carro (carriage-return)
 - '\b': retroceso (backspace)
 - '\t': tabulador horizontal
 - '\v': tabulador vertical
 - '\f': avance de página (form-feed)
 - '\a': sonido (audible-bell)
 - '\0': fin de cadena
 - '\137', '\x5F': carácter correspondiente al valor especificado en notación octal y hexadecimal respectivamente
- Constantes cadenas de caracteres literales, la secuencia de caracteres aparece entre comillas dobles (puede contener caracteres de escape):

```
"Hola Pepe"
"Hola\nJuan\n"
"Hola " "María"
```

- Constantes enteras, pueden ser expresadas en decimal (base 10), hexadecimal (base 16) y octal (base 8). El sufijo L se utiliza para especificar `long`, el sufijo LL se utiliza para especificar `long long`, el sufijo U se utiliza para especificar `unsigned`, el sufijo UL especifica `unsigned long`, y el sufijo ULL especifica `unsigned long long`:

```
123, -1520, 2345U, 30000L, 50000UL, 0x10B3FC23 (hexadecimal), 0751 (octal)
```

- Constantes reales, números en punto flotante. El sufijo F especifica `float`, y el sufijo L especifica `long double`:

```
3.1415, -1e12, 5.3456e-5, 2.54e-1F, 3.25e200L
```

Constantes Simbólicas

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

```
const bool OK = true;
const char SONIDO = '\a';
const short ELEMENTO = 1000;
const int MAXIMO = 5000;
const long ULTIMO = 100000L;
const long long TAMANO = 1000000LL;
const unsigned short VALOR = 100U;
const unsigned FILAS = 200U;
const unsigned long COLUMNAS = 200UL;
const unsigned long long NELMS = 2000ULL;
const float N_E = 2.7182F;
const double LOG10E = log(N_E);
const long double N_PI = 3.141592L;
const Color COLOR_DEFECTO = ROJO;
```

Variables

Las *variables* se definen, dentro de un bloque de sentencias (véase 4.1), especificando su tipo y el identificador con el que nos referiremos a ella, y serán visibles desde el punto de declaración hasta el final del cuerpo (bloque) donde han sido declaradas. Se les podrá asignar un valor inicial en la definición (mediante el símbolo =), si no se les asigna ningún valor inicial, entonces tendrán un valor inespecificado. Su valor podrá cambiar mediante la sentencia de asignación (véase 4.3) o mediante una sentencia de entrada de datos (véase 3.2).

```
{
    char letra; // valor inicial inespecificado
    int contador = 0;
    double total = 5.0;
    ...
}
```

2.5. Operadores

Los siguientes *operadores* se pueden aplicar a los datos (de mayor a menor orden de precedencia):

Operador	Tipo de Operador	Asociatividad
! ~ -	Unarios	Dch. a Izq.
* / %	Binarios	Izq. a Dch.
+ -	Binarios	Izq. a Dch.
<< >>	Binarios	Izq. a Dch.
< <= > >=	Binarios	Izq. a Dch.
== !=	Binarios	Izq. a Dch.
&	Binario	Izq. a Dch.
^	Binario	Izq. a Dch.
	Binario	Izq. a Dch.
&&	Binario	Izq. a Dch.
	Binario	Izq. a Dch.
?:	Ternario	Dch. a Izq.

Significado de los operadores:

- Aritméticos. El resultado es del mismo tipo que los operandos (véase 2.6):

- valor	Menos unario
valor * valor	Producto (multiplicación)
valor / valor	División (entera o real según el tipo de operandos)
valor % valor	Módulo (resto de la división) (sólo tipos enteros)
valor + valor	Suma
valor - valor	Resta

- Relacionales/Comparaciones. El resultado es de tipo `bool`

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

- Operadores de Bits, sólo aplicable a operandos de tipos enteros. El resultado es del mismo tipo que los operandos (véase 2.6):

~ valor	Negación de bits (complemento)
valor << despl	Desplazamiento de bits a la izq.
valor >> despl	Desplazamiento de bits a la dch.
valor & valor	AND de bits
valor ^ valor	XOR de bits
valor valor	OR de bits

- Lógicos, sólo aplicable operandos de tipo booleano. Tanto el operador `&&` como el operador `||` se evalúan en cortocircuito. El resultado es de tipo `bool`:

! valor	Negación lógica	(Si <code>valor</code> es <code>true</code> entonces <code>false</code> , en otro caso <code>true</code>)
valor1 && valor2	AND lógico	(Si <code>valor1</code> es <code>false</code> entonces <code>false</code> , en otro caso <code>valor2</code>)
valor1 valor2	OR lógico	(Si <code>valor1</code> es <code>true</code> entonces <code>true</code> , en otro caso <code>valor2</code>)

x	! x
F	T
T	F

x	y	x && y	x y
F	F	F	F
F	T	F	T
T	F	F	T
T	T	T	T

- Condicional. El resultado es del mismo tipo que los operandos:

`cond ? valor1 : valor2` Si `cond` es `true` entonces `valor1`, en otro caso `valor2`

2.6. Conversiones Automáticas (Implícitas) de Tipos

Es posible que nos interese realizar operaciones en las que se mezclen datos de tipos diferentes. El lenguaje de programación C++ realiza *conversiones de tipo automáticas* (“castings”), de tal forma que el resultado de la operación sea del tipo más amplio de los implicados en ella. *Siempre que sea posible*, los valores se convierten de tal forma que no se pierda información.

Promociones Enteras

Son conversiones *implícitas* que preservan valores. Antes de realizar una operación aritmética, se utiliza *promoción a entero* para crear `int` a partir de otros tipos integrales mas cortos. Para los siguientes tipos origen: `bool`, `char`, `signed char`, `unsigned char`, `short`, `unsigned short`

- Si `int` puede representar todos los valores posibles del tipo origen, entonces sus valores promocionan a `int`.
- En otro caso, promocionan a `unsigned int`

Conversiones Enteras

Si el tipo destino es `unsigned`, el valor resultante es el menor `unsigned` congruente con el valor origen módulo 2^n , siendo n el número de bits utilizados en la representación del tipo destino (en representación de complemento a dos, simplemente tiene tantos bits del origen como quepan en el destino, descartando los de mayor orden).

Si el tipo destino es `signed`, el valor no cambia si se puede representar en el tipo destino, si no, viene definido por la implementación.

Conversiones Aritméticas Implícitas Habituales

Se realizan sobre los operandos de un operador binario para convertirlos a un tipo común que será el tipo del resultado:

1. Si algún operando es de tipo de punto flotante (real):
 - a) Si algún operando es de tipo `long double`, el otro se convierte a `long double`.
 - b) En otro caso, si algún operando es `double` el otro se convierte a `double`.
 - c) En otro caso, si algún operando es `float` el otro se convierte a `float`.
2. En otro caso, se realizan promociones enteras (véase sec. 2.6) sobre ambos operandos:
 - a) Si algún operando es de tipo `unsigned long`, el otro se convierte a `unsigned long`.
 - b) En otro caso, si algún operando es `long int` y el otro es `unsigned int`, si un `long int` puede representar todos los valores de un `unsigned int`, el `unsigned int` se convierte a `long int`; en caso contrario, ambos se convierten a `unsigned long int`.
 - c) En otro caso, si algún operando es `long` el otro se convierte a `long`.
 - d) En otro caso, si algún operando es `unsigned` el otro se convierte a `unsigned`.
 - e) En otro caso, ambos operandos son `int`

2.7. Conversiones Explícitas de Tipos

También es posible realizar *conversiones de tipo explícitas*. Para ello, se escribe el tipo al que queremos convertir y entre paréntesis la expresión cuyo valor queremos convertir. Por ejemplo:

<code>char x = char(65);</code>	produce el carácter 'A'
<code>int x = int('a');</code>	convierte el carácter 'a' a su valor entero (97)
<code>int x = int(ROJO);</code>	produce el entero 0
<code>int x = int(AMARILLO);</code>	produce el entero 2
<code>int x = int(3.7);</code>	produce el entero 3
<code>double x = double(2);</code>	produce el real (doble precisión) 2.0
<code>Color x = Color(1);</code>	produce el Color AZUL
<code>Color x = Color(c+1);</code>	si c es de tipo Color, produce el siguiente valor de la enumeración

El tipo enumerado se convierte automáticamente a entero, aunque la conversión inversa no se realiza de forma automática. Así, para incrementar una variable de tipo color se realizará de la siguiente forma:

```
enum Color {
    ROJO, AZUL, AMARILLO
};
int main()
{
    Color c = ROJO;
    c = Color(c + 1);
    // ahora c tiene el valor AZUL
}
```

Ⓐ Otras Conversiones Explícitas de Tipos

Otra posibilidad de realizar conversiones explícitas es mediante los siguientes operadores:

- El operador `static_cast<tipo>(valor)` realiza conversiones entre tipos relacionados como por ejemplo para conversiones entre tipo *enumerado* y tipo *integral*, o entre tipo de *coma flotante* y tipo *integral*, o para convertir de tipo *puntero a Derivado* a tipo *puntero a Base* en la misma jerarquía de clases. Para un tipo primitivo, la conversión de tipos expresada de la siguiente forma: `Tipo(valor)` es equivalente a `static_cast<Tipo>(valor)`. Ejemplo:

```
int* p_int = static_cast<int*>(p_void);
```

- El operador `reinterpret_cast<tipo>(valor)` trata las conversiones entre tipos no relacionados como un puntero y un entero, o un puntero y otro puntero no relacionado. Generalmente produce un valor del nuevo tipo que tiene el mismo patrón de bits que su parámetro. Suelen corresponder a zonas de código no portable, y posiblemente problemática. Ejemplo:

```
disp* ptr = reinterpret_cast<disp*>(0xff00);
```

- El operador `dynamic_cast<tipo*>(ptr)` comprueba en tiempo de ejecución que `ptr` puede ser convertido al tipo destino (utiliza información de tipos en tiempo de ejecución RTTI). Si no puede ser convertido devuelve 0. Nota: `ptr` debe ser un puntero a un tipo polimórfico, y se suele utilizar para convertir de tipo *puntero a Base* a tipo *puntero a Derivado*. Ejemplo:

```
ave* ptr = dynamic_cast<ave*>(p_animal);
```

- El operador `dynamic_cast<tipo&>(ref)` comprueba en tiempo de ejecución que `ref` puede ser convertido al tipo destino (utiliza información de tipos en tiempo de ejecución RTTI). Si no puede ser convertido lanza la excepción `bad_cast`. Nota: `ref` debe ser una referencia a un tipo polimórfico, y se suele utilizar para convertir de tipo *Base* a tipo *Derivado*. Ejemplo:

```
ave& ref = dynamic_cast<ave&>(r_animal);
```

- El operador `const_cast<tipo*>(ptr_const_tipo)` elimina la restricción constante de valor. No se garantiza que funcione cuando se aplica a una entidad declarada originalmente como `const`. Ejemplo:

```
char* ptr = const_cast<char*>(ptr_const_char)
```


Esta distinción permite al compilador aplicar una verificación de tipos mínima para `static_cast` y haga más fácil al programador la búsqueda de conversiones peligrosas representadas por `reinterpret_cast`. Algunos `static_cast` son portables, pero pocos `reinterpret_cast` lo son.

2.8. Tabla ASCII

La tabla ASCII es comúnmente utilizada como base para la representación de los caracteres, donde los números del 0 al 31 se utilizan para representar caracteres de control, y los números del 128 al 255 se utilizan para representar caracteres extendidos.

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

2.9. Algunas Consideraciones Respecto a Operaciones con Números Reales \mathbb{A}

La representación de los números reales es limitada en cuanto al rango de valores y a la precisión de los mismos, ya que el número de dígitos decimales que puede almacenar es finito. Además, la representación de números reales en base 2 hace que la representación de determinados números en base 10 sea **inexacta**, por lo tanto la realización de operaciones aritméticas en punto flotante puede dar lugar a pérdidas de precisión que den lugar a resultados inesperados, distintas operaciones que matemáticamente son equivalentes pueden ser computacionalmente diferentes. Así, la comparación directa de igualdad o desigualdad de números reales debe ser **evitada**. Por ejemplo, el siguiente programa:

```
#include <iostream>
#include <string>
using namespace std;
inline double abs(double x)
```

```

{
    return x < 0 ? -x : x;
}
inline bool iguales(double x, double y)
{
    const double ERROR_PREC = 1e-8;
    return abs(x - y) < ERROR_PREC;
}
void check_iguales(double x, double y, const string& op1, const string& op2)
{
    //-----
    if (x == y) {
        cout << "   Correcto   ("<<op1<<" == "<<op2<<"): ("<<x<<" == "<<y<<)"
        <<endl;
    } else {
        cout << "   Distintos ("<<op1<<" != "<<op2<<"): ("<<x<<" != "<<y
        <<") Error. Diferencia: "<< abs(x-y) <<endl;
    }
    //-----
    if (iguales(x, y)) {
        cout << "   Función: Iguales("<<op1<<", "<<op2<<") Correcta"<<endl;
    } else {
        cout << "   Función: Iguales("<<op1<<", "<<op2<<") Error !!"<<endl;
    }
    //-----
}
int main()
{
    //-----
    // Error en conversión float a double
    cout << "* Check Conversión float a double" << endl;
    check_iguales(0.1F, 0.1, "0.1F", "0.1");
    //-----
    // Error en pérdida de precisión
    cout << "* Check Precisión" << endl;
    check_iguales(0.01, (0.1*0.1), "0.01", "(0.1*0.1)");
    //-----
    // Error de apreciación
    cout << "* Check Apreciación" << endl;
    check_iguales(((10e30+ -10e30)+1.0), (10e30+(-10e30+1.0)),
        "((10e30+ -10e30)+1.0)", "(10e30+(-10e30+1.0))");
    //-----
}

```

produce los siguientes resultados en una máquina de 32 bits:

```

* Check Conversión float a double
Distintos (0.1F != 0.1): (0.1 != 0.1) Error. Diferencia: 1.49012e-09
Función: Iguales(0.1F, 0.1) Correcta
* Check Precisión
Distintos (0.01 != (0.1*0.1)): (0.01 != 0.01) Error. Diferencia: 1.73472e-18
Función: Iguales(0.01, (0.1*0.1)) Correcta
* Check Apreciación
Distintos (((10e30+ -10e30)+1.0) != (10e30+(-10e30+1.0))): (1 != 0) Error. Diferencia: 1
Función: Iguales(((10e30+ -10e30)+1.0), (10e30+(-10e30+1.0))) Error !!

```

Capítulo 3

Entrada y Salida de Datos Básica

Para poder realizar entrada y salida de datos básica es necesario incluir la biblioteca `iostream` que contiene las declaraciones de tipos y operaciones que la realizan. Todas las definiciones y declaraciones de la biblioteca estándar se encuentran bajo el espacio de nombres `std` (ver capítulo 11), por lo que para utilizarlos adecuadamente habrá que utilizar la *directiva* `using` al comienzo del programa.

```
#include <iostream> // inclusión de la biblioteca de entrada/salida
using namespace std; // utilización del espacio de nombres de la biblioteca
const double EUR_PTS = 166.386;
int main()
{
    cout << "Introduce la cantidad (en euros): ";
    double euros;
    cin >> euros;
    double pesetas = euros * EUR_PTS;
    cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl;
}
```

3.1. Salida de Datos

La *salida* de datos se realiza a través de los *flujos de salida*, así el flujo de salida asociado a la *salida estándar* (usualmente la pantalla o terminal de la consola) se denomina `cout`. De esta forma, la salida de datos a pantalla se realiza utilizando el operador `<<` sobre el flujo `cout` especificando el dato cuyo valor se mostrará. Por ejemplo:

```
cout << "Introduce la cantidad (en euros): ";
```

escribirá en la salida estándar el mensaje correspondiente a la cadena de caracteres especificada. El siguiente ejemplo escribe en la salida estándar el valor de las variables `euros` y `pesetas`, así como un mensaje para interpretarlos adecuadamente. El símbolo `endl` indica que la sentencia deberá escribir un *fin de línea*.

```
cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl;
```

También existe otro flujo de salida asociado a la *salida estándar de errores* denominado `cerr` (usualmente asociado también a la pantalla o terminal de la consola) que se utiliza para notificar mensajes de error.

```
cerr << "Error. División por cero." << numero << endl;
```

También es posible que se fuerce la salida al dispositivo mediante el manipulador `flush`, de tal forma que toda la información contenida en el buffer de salida (véase 3.3) se vuelque inmediatamente al dispositivo asociado:

```
cout << euros << " Euros equivalen a " << pesetas << " Pts" << flush;
```

Salida de Datos Formateada

Es posible especificar el formato bajo el que se realizará la salida de datos. Para ello se debe incluir la biblioteca estándar `iomanip`. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    bool x = true;
    cout << boolalpha << x; // escribe los booleanos como 'false' o 'true'

    cout << dec << 27 ; // escribe 27 (decimal)
    cout << hex << 27 ; // escribe 1b (hexadecimal)
    cout << oct << 27 ; // escribe 33 (octal)

    cout << setprecision(2) << 4.567 ; // escribe 4.6
    cout << setw(5) << 234 ; // escribe " 234"
    cout << setfill('#') << setw(5) << 234 ; // escribe "##234"
}
```

Ⓐ Salida Formateada Avanzada

Además, es posible activar o desactivar las siguientes marcas (*flags*) para modificar el comportamiento de la salida de datos:

<code>ios::fmtflags flags =</code>	<code>cout.flags();</code>	obtiene los flags actuales
	<code>cout.flags(ios::fmtflags flags);</code>	pone los flags especificados
	<code>cout.unsetf(flag);</code>	desactiva el flag especificado
	<code>cout.setf(flag);</code>	activa el flag especificado
	<code>cout.setf(flag, mask);</code>	borra mask y activa el flag especificado

Las máscaras y flags pueden ser:

Máscara	Flags	Significado
	<code>ios::skipws</code>	saltar los espacios en blanco (en flujo de entrada)
	<code>ios::showbase</code>	mostrar prefijo que indique la base (0 0x)
	<code>ios::showpoint</code>	mostrar siempre el punto decimal
	<code>ios::showpos</code>	mostrar el signo positivo
	<code>ios::uppercase</code>	usar mayúsculas en representación numérica
	<code>ios::unitbuf</code>	forzar la salida después de cada operación
<code>ios::floatfield</code>		máscara para notación científica
	<code>ios::fixed</code>	no usar notación científica (exponencial)
	<code>ios::scientific</code>	usar notación científica (exponencial)
<code>ios::adjustfield</code>		máscara de justificación
	<code>ios::left</code>	justificación izquierda
	<code>ios::right</code>	justificación derecha
	<code>ios::internal</code>	justificación interna
<code>ios::basefield</code>		máscara para indicar la base
	<code>ios::dec</code>	entrada/salida en base decimal
	<code>ios::oct</code>	entrada/salida en base octal
	<code>ios::hex</code>	entrada/salida en base hexadecimal

La precisión en el caso de formato por defecto de punto flotante, indica el máximo número de dígitos. En caso de formato `fixed` o `scientific` indica el número de dígitos después del punto decimal.

```

cout.setf(0, ios::floatfield);
cout << setprecision(2) << 4.567;           // imprime 4.6

cout.setf(ios::fixed, ios::floatfield);
cout << setprecision(2) << 4.567;           // imprime 4.57

cout.setf(ios::scientific, ios::floatfield);
cout << setprecision(2) << 4.567;           // imprime 4.57e+00

```

3.2. Entrada de Datos

La *entrada* de datos se realiza a través de los *flujos de entrada*, así el flujo de entrada asociado a la *entrada estándar* (usualmente el teclado) se denomina `cin`. De esta forma, la entrada de datos desde el teclado se realiza mediante el operador `>>` sobre el flujo `cin` especificando la variable donde almacenar el valor de entrada leído desde el teclado:

```
cin >> euros;
```

incluso es posible leer varios valores consecutivamente en la misma sentencia de entrada:

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

Dicho operador de entrada se comporta de la siguiente forma: elimina los espacios en blanco que hubiera al principio de la entrada de datos, y lee dicha entrada hasta que encuentre algún carácter no válido para dicha entrada, que no será leído y permanecerá disponible en el buffer de entrada (véase 3.3) hasta la próxima operación de entrada. En caso de que durante la entrada surja alguna situación de error, dicha entrada se detiene y el flujo de entrada se pondrá en un estado erróneo (véase 3.5). Se consideran espacios en blanco los siguientes caracteres: espacio en blanco, tabuladores, retorno de carro y nueva línea (' ', '\t', '\v', '\f', '\r', '\n').

Ⓐ Entrada de Datos Avanzada

También es posible leer un carácter, desde el flujo de entrada, sin eliminar los espacios iniciales:

```

{
    char c;
    cin.get(c) ; // lee un carácter sin eliminar espacios en blanco iniciales
    ...
}

```

En caso de querer eliminar los espacios iniciales explícitamente:

```

{
    char c;
    cin >> ws ; // elimina los espacios en blanco iniciales
    cin.get(c) ; // lee sin eliminar espacios en blanco iniciales
    ...
}

```

Es posible también eliminar un número determinado de caracteres del flujo de entrada, o hasta un determinado carácter:

```

{
    cin.ignore() ;           // elimina el próximo carácter
    cin.ignore(5) ;          // elimina los 5 próximos caracteres
    cin.ignore(1000, '\n') ; // elimina 1000 caracteres o hasta nueva-línea
    cin.ignore(numeric_limits<streamsize>::max(), '\n') ; // elimina hasta nueva-línea
}

```

Para utilizar `numeric_limits` es necesario incluir al principio del programa la biblioteca `limits`.

La entrada y salida de cadenas de caracteres se puede ver en los capítulos correspondientes (cap. 6.2 y cap. 8.2).

3.3. El “Buffer” de Entrada y el “Buffer” de Salida

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

Así, cuando se pulsa alguna tecla, el Sistema Operativo almacena en secuencia las teclas pulsadas en una zona de memoria intermedia: *el “buffer” de entrada*. Cuando un programa realiza una operación de entrada de datos (`cin >> valor`), accede al “buffer” de entrada y obtiene los valores allí almacenados si los hubiera, o esperará hasta que los haya (se pulsen una serie de teclas seguidas por la tecla “enter”). Una vez obtenidos las teclas pulsadas (caracteres), se convertirán a un valor del tipo especificado por la operación de entrada, y dicho valor se asignará a la variable especificada.

De igual forma, cuando se va a mostrar alguna información de salida dichos datos no van directamente a la pantalla, sino que se convierten a un formato adecuado para ser impresos (caracteres) y se almacenan en una zona de memoria intermedia denominada *“buffer” de salida*, desde donde el Sistema Operativo tomará la información para ser mostrada por pantalla.

```
cout << "Valor: " << val << endl;
```

3.4. Otras Operaciones de Entrada y Salida

Además de con el operador `>>`, también es posible enviar un carácter a un determinado flujo de salida mediante la siguiente operación:

```
cout.put('#');
```

Dado un flujo de entrada, es posible conocer el próximo carácter (sin eliminarlo del buffer de entrada) que se leerá en la siguiente operación de entrada:

```
char c = char(cin.peek());
```

Así mismo, también es posible devolver al flujo (buffer) de entrada el último carácter previamente leído.

```
cin.unget();
```

o incluso devolver al flujo (buffer) de entrada otro carácter diferente al previamente leído:

```
cin.putback(c);
```

3.5. Control del Estado del Flujo de Datos

Después de realizar una operación de entrada o salida con un determinado flujo, es posible que dicha operación no se hubiese realizado satisfactoriamente. En dicho caso el flujo se pondrá en un estado erróneo, y cualquier operación de entrada o salida posterior sobre dicho flujo también fallará. El lenguaje de programación C++ proporciona una serie de métodos para comprobar cual es el estado en que se encuentra un determinado flujo y también para restaurarlo a un estado adecuado en caso de ser necesario. La situación más usual es comprobar si la última operación sobre el flujo se realizó adecuadamente:

```
cin >> valor;
if (cin.fail()) {
    // ocurrió un error y próxima operación fallará
    // (cin.rdstate() & (ios::badbit | ios::failbit)) != 0
} else {
    // ...
}
```

En caso de haber fallado, para comprobar si se ha alcanzado el final de la entrada de datos (final de fichero), o el flujo se encuentra en mal estado:

```
cin >> valor;
if (cin.fail()) {
    // ocurrió un error y próxima operación fallará
    // (cin.rdstate() & (ios::badbit | ios::failbit))!=0
    if (cin.bad()) {
        // flujo en mal estado
        //(cin.rdstate() & ios::badbit) != 0
    } else if (cin.eof()) {
        // fin de fichero
        // (cin.rdstate() & ios::eofbit) != 0
    } else {
        // ocurrió un error de formato de entrada
        //(cin.rdstate() & ios::failbit) != 0
    }
} else {
    // lectura de datos correcta
    // ...
}
```

También se puede preguntar si un determinado flujo se encuentra en buen estado:

```
cout << valor;
if (cout.good()) {
    // ...
}
```

Como se ha visto en el ejemplo, la operación `rdstate()` sobre el flujo de datos proporciona información sobre el motivo del error, comprobándose con respecto a las siguientes marcas (*flags*):

- **goodbit**: Indica que todo es correcto (`goodbit == iostate(0)`).
- **eofbit**: Indica que una operación de entrada alcanzó el final de una secuencia de entrada.
- **failbit**: Indica que una operación de entrada falló al leer los caracteres esperados, o que una operación de salida falló al generar los caracteres deseados.
- **badbit**: Indica pérdida de integridad en una secuencia de entrada o salida (tal como un error irreparable de un fichero).

Es posible recuperar el estado erróneo de un determinado flujo mediante la operación `clear()` o `setstate()` sobre el flujo de datos.

```
FLAGS: [ios::goodbit | ios::failbit | ios::eofbit | ios::badbit]

cin.clear();          // pone el estado a ios::goodbit
cin.clear(flags);     // pone el estado al especificado por los flags

ios::iostate e = cin.rdstate(); // obtiene el estado del flujo
cin.setstate(flags);      // equiv. a cin.clear(cin.rdstate() | flags);
```

Por ejemplo, para leer un número y restaurar el estado en caso de lectura errónea:

```
void leer(int& numero, bool& ok)
{
    cin >> numero;
    if (cin.fail()) {
        ok = false;
        if (! cin.eof() && ! cin.bad()) {
            cin.clear();
            cin.ignore(10000, '\n');
        }
    }
}
```

```
    } else {  
        ok = true;  
    }  
}
```

También es posible iterar hasta que se produzca una entrada correcta:

```
void leer(int& numero)  
{  
    cin >> numero;  
    while (cin.fail() && ! cin.eof() && ! cin.bad()) {  
        cout << "Error. Inténtelo de nuevo" << endl;  
        cin.clear();  
        cin.ignore(10000, '\n');  
        cin >> numero;  
    }  
}
```

Es posible configurar el sistema de entrada y salida de C++ para hacer que lance una *excepción* (véase 12) del tipo `ios::failure` cuando ocurra algún error:

```
ios::iostate old_state = cin.exceptions();  
cin.exceptions(ios::badbit | ios::failbit );  
cout.exceptions(ios::badbit | ios::failbit );
```


Capítulo 4

Estructuras de Control

Las estructuras de control en el lenguaje de programación C++ son muy flexibles, sin embargo, la excesiva flexibilidad puede dar lugar a estructuras complejas. Por ello sólo veremos algunas de ellas y utilizadas en contextos y situaciones restringidas.

4.1. Sentencia, Secuencia y Bloque

En C++ la unidad básica de acción es la sentencia, y expresamos la composición de sentencias como una *secuencia de sentencias* terminadas cada una de ellas por el carácter “punto y coma” (;), de tal forma que su flujo de *ejecución es secuencial*, es decir, se ejecuta una sentencia, y cuando ésta termina, entonces se ejecuta la siguiente sentencia, y así sucesivamente.

Un *bloque* es una unidad de ejecución mayor que la sentencia, y permite agrupar una secuencia de sentencias como una unidad. Para ello enmarcamos la secuencia de sentencias entre dos llaves para formar un bloque. Es posible el anidamiento de bloques.

```
int main()
{
    <sentencia_1>;
    <sentencia_2>;
    {
        <sentencia_3>;
        <sentencia_4>;
        . . .
    }
    <sentencia_n>;
}
```

4.2. Declaraciones Globales y Locales

Como ya se vio en el capítulo anterior, es obligatoria la declaración de las entidades manipuladas en el programa. Distinguiremos dos clases de declaraciones: globales y locales.

Entidades globales son aquellas que han sido definidas fuera de cualquier bloque. Su *ámbito de visibilidad* comprende desde el punto en el que se definen hasta el final del fichero. Respecto a su tiempo de vida, se crean al principio de la ejecución del programa y se destruyen al finalizar éste. Normalmente serán constantes simbólicas, definiciones de tipos, declaración de prototipos de subprogramas y definiciones de subprogramas.

Entidades locales son aquellas que se definen dentro de un bloque. Su *ámbito de visibilidad* comprende desde el punto en el que se definen hasta el final de dicho bloque. Respecto a su tiempo de vida, se crean en el punto donde se realiza la definición, y se destruyen al finalizar el bloque. Normalmente serán constantes simbólicas y variables locales.

```

#include <iostream>
using namespace std;
const double EUR_PTS = 166.386;      // Declaración de constante GLOBAL
int main()
{
    cout << "Introduce la cantidad (en euros): ";
    double euros;                     // Declaración de variable LOCAL
    cin >> euros;
    double pesetas = euros * EUR_PTS; // Declaración de variable LOCAL
    cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl;
}

```

Respecto al ámbito de visibilidad de una entidad, en caso de declaraciones de diferentes entidades con el mismo identificador en diferentes niveles de anidamiento, la entidad visible será aquella que se encuentre declarada en el bloque de nivel de anidamiento más interno. Es decir, cuando se solapa el ámbito de visibilidad de dos entidades con el mismo identificador, en dicha zona de solapamiento será visible el identificador declarado/definido en el bloque más interno. Sin embargo, no es una buena práctica de programación ocultar identificadores al redefinirlos en niveles de anidamiento mas internos, ya que conduce a programas difíciles de leer y propensos a errores.

```

int main()
{
    int x = 3;
    int z = x * 2;      // x es vble de tipo int con valor 3
    {
        double x = 5.0;
        double n = x * 2; // x es vble de tipo double con valor 5.0
    }
    int y = x + 4;      // x es vble de tipo int con valor 3
}

```

4.3. Sentencias de Asignación

La sentencia de asignación permite asignar a una variable el resultado de evaluar una expresión aritmética expresada en notación infija, de tal forma que primero se evalúa la expresión, considerando las reglas de precedencia y asociatividad de los operadores (véase 2.5), y a continuación el valor resultante se asigna a la variable, que perderá el valor anterior que tuviese. Por ejemplo:

```

const int MAXIMO = 15;
int main()
{
    int cnt;           // valor inicial inespecificado
    cnt = 30 * MAXIMO + 1; // asigna a cnt el valor 451
    cnt = cnt + 10;     // cnt pasa ahora a contener el valor 461
}

```

Además, se definen las siguientes sentencias de incremento/decremento:

Sentencia	Equivalencia
<code>++variable;</code>	<code>variable = variable + 1;</code>
<code>--variable;</code>	<code>variable = variable - 1;</code>
<code>variable++;</code>	<code>variable = variable + 1;</code>
<code>variable--;</code>	<code>variable = variable - 1;</code>
<code>variable += expresion;</code>	<code>variable = variable + (expresion);</code>
<code>variable -= expresion;</code>	<code>variable = variable - (expresion);</code>
<code>variable *= expresion;</code>	<code>variable = variable * (expresion);</code>
<code>variable /= expresion;</code>	<code>variable = variable / (expresion);</code>
<code>variable %= expresion;</code>	<code>variable = variable % (expresion);</code>
<code>variable &= expresion;</code>	<code>variable = variable & (expresion);</code>
<code>variable ^= expresion;</code>	<code>variable = variable ^ (expresion);</code>
<code>variable = expresion;</code>	<code>variable = variable (expresion);</code>
<code>variable <<= expresion;</code>	<code>variable = variable << (expresion);</code>
<code>variable >>= expresion;</code>	<code>variable = variable >> (expresion);</code>

Nota: las sentencias de asignación vistas anteriormente se pueden utilizar en otras formas muy diversas, pero nosotros restringiremos su utilización a la expresada anteriormente, debido a que otras utilizaciones pueden dificultar la legibilidad y aumentar las posibilidades de cometer errores de programación.

4.4. Sentencias de Selección

Las sentencias de selección alteran el flujo secuencial de ejecución de un programa, de tal forma que permiten seleccionar flujos de ejecución alternativos y excluyentes dependiendo de expresiones lógicas. La más simple de todas es la sentencia de selección condicional `if` cuya sintaxis es la siguiente:

```
int main()
{
    if ( <expresión_lógica> ) {
        <secuencia_de_sentencias>;
    }
}
```

y cuya semántica consiste en evaluar la expresión lógica, y si su resultado es Verdadero (`true`) entonces se ejecuta la secuencia de sentencias entre las llaves. Ejemplo de programa que imprime el valor mayor de tres números:

```
#include <iostream>
using namespace std;
int main ()
{
    unsigned a, b, c;
    cin >> a >> b >> c;
    unsigned mayor = a;
    if (b > mayor) {
        mayor = b;
    }
    if (c > mayor) {
        mayor = c;
    }
    cout << mayor << endl;
}
```

Otra posibilidad es la sentencia de selección condicional compuesta, que tiene la siguiente sintaxis:

```

int main()
{
    if ( <expresión_lógica> ) {
        <secuencia_de_sentencias_v>;
    } else {
        <secuencia_de_sentencias_f>;
    }
}

```

y cuya semántica consiste en evaluar la expresión lógica, y si su resultado es Verdadero (**true**) entonces se ejecuta la *<secuencia_de_sentencias_v>*. Sin embargo, si el resultado de evaluar la expresión lógica es Falso (**false**) entonces se ejecuta la *<secuencia_de_sentencias_f>*.

La sentencia de selección condicional se puede encadenar de la siguiente forma con el flujo de control esperado:

```

#include <iostream>
using namespace std;
int main ()
{
    double nota;
    cin >> nota;
    if ( ! ((nota >= 0.0) && (nota <= 10.0))) {
        cout << "Error: 0 <= n <= 10" << endl;
    } else if (nota >= 9.5) {
        cout << "Matrícula de Honor" << endl;
    } else if (nota >= 9.0) {
        cout << "Sobresaliente" << endl;
    } else if (nota >= 7.0) {
        cout << "Notable" << endl;
    } else if (nota >= 5.0) {
        cout << "Aprobado" << endl;
    } else {
        cout << "Suspenso" << endl;
    }
}

```

La sentencia **switch** es otro tipo de sentencia de selección en la cual la secuencia de sentencias alternativas a ejecutar no se decide en base a expresiones lógicas, sino en función del valor que tome una determinada expresión de tipo **ordinal** (véase 2.2), es decir, una relación de igualdad entre el valor de una expresión y unos determinados valores constantes de tipo ordinal especificados. Su sintaxis es la siguiente:

```

int main()
{
    switch ( <expresión> ) {
        case <valor_1>:
            <secuencia_de_sentencias_1>;
            break;
        case <valor_2>:
        case <valor_3>:
            <secuencia_de_sentencias_2>;
            break;
        case <valor_4>:
            <secuencia_de_sentencias_3>;
            break;
        . . .
        default:
            <secuencia_de_sentencias_d>;
            break;
    }
}

```

```
}
```

en la cual se evalúa la expresión, y si su valor coincide con *<valor_1>* entonces se ejecuta la *<secuencia_de_sentencias_1>*. Si su valor coincide con *<valor_2>* o con *<valor_3>* se ejecuta la *<secuencia_de_sentencias_2>* y así sucesivamente. Si el valor de la expresión no coincide con ningún valor especificado, se ejecuta la secuencia de sentencias correspondiente a la etiqueta *default* (si es que existe). Nótese que la sentencia **break**; termina la secuencia de sentencias a ejecutar para cada caso. Ejemplo:

```
#include <iostream>
using namespace std;
enum Dia_Semana {
    lunes, martes, miercoles, jueves, viernes, sabado, domingo
};
int main ()
{
    Dia_Semana dia;
    ...
    // 'dia' tiene algún valor válido
    switch (dia) {
    case lunes:
        cout << "Lunes" << endl;
        break;
    case martes:
        cout << "Martes" << endl;
        break;
    case miercoles:
        cout << "Miércoles" << endl;
        break;
    case jueves:
        cout << "Jueves" << endl;
        break;
    case viernes:
        cout << "Viernes" << endl;
        break;
    case sabado:
        cout << "Sábado" << endl;
        break;
    case domingo:
        cout << "Domingo" << endl;
        break;
    default:
        cout << "Error" << endl;
        break;
    }
}
```

4.5. Sentencias de Iteración. Bucles

Vamos a utilizar tres tipos de sentencias diferentes para expresar la repetición de la ejecución de un grupo de sentencias: **while**, **for** y **do-while**.

La sentencia **while** es la más utilizada y su sintaxis es la siguiente:

```
int main()
{
    while ( <expresión_lógica> ) {
        <secuencia_de_sentencias>;
    }
}
```

en la cual primero se evalúa la expresión lógica, y si es cierta, se ejecuta la secuencia de sentencias entre llaves completamente. Posteriormente se vuelve a evaluar la expresión lógica y si vuelve a ser cierta se vuelve a ejecutar la secuencia de sentencias entre llaves. Este ciclo iterativo consistente en evaluar la condición y ejecutar las sentencias se realizará *MIENTRAS* que la condición se evalúe a Verdadera y finalizará cuando la condición se evalúe a Falsa. Ejemplo:

```
#include <iostream>
using namespace std;
int main ()
{
    unsigned num, divisor;
    cin >> num;
    if (num <= 1) {
        divisor = 1;
    } else {
        divisor = 2;
        while ((num % divisor) != 0) {
            ++divisor;
        }
    }
    cout << "El primer divisor de " << num << " es " << divisor << endl;
}
```

La sentencia `for` es semejante a la estructura `FOR` de Pascal o Modula-2, aunque en C++ toma una dimensión más amplia y flexible. En realidad se trata de la misma construcción `while` vista anteriormente pero con una sintaxis diferente para hacer más explícito los casos en los que la iteración está controlada por los valores que toma una determinada variable de control, de tal forma que existe una clara inicialización y un claro incremento de la variable de control hasta llegar al caso final. La sintaxis es la siguiente:

```
int main()
{
    for ( <inicialización> ; <expresión_lógica> ; <incremento> ) {
        <secuencia_de_sentencias>;
    }
}
```

y es equivalente a:

```
int main()
{
    <inicialización>;
    while ( <expresión_lógica> ) {
        <secuencia_de_sentencias>;
        <incremento>;
    }
}
```

Nota: es posible declarar e inicializar la variable de control del bucle en el lugar de la inicialización. En este caso especial, el ámbito de visibilidad de la variable de control del bucle es solamente hasta el final del bloque de la estructura `for`.

```
#include <iostream>
using namespace std;
int main ()
{
    unsigned n;
    cin >> n;
    for (unsigned i = 0; i < n; ++i) {
        cout << i << " ";
    }
}
```

```

    }
    // i ya no es visible aquí
    cout << endl;
}

```

La sentencia **do-while** presenta la siguiente estructura

```

int main()
{
    do {
        <secuencia_de_sentencias>;
    } while ( <expresión_lógica> );
}

```

también expresa la iteración en la ejecución de la secuencia de sentencias, pero a diferencia de la primera estructura iterativa ya vista, donde primero se evalúa la expresión lógica y después, en caso de ser cierta, se ejecuta la secuencia de sentencias, en esta estructura se ejecuta primero la secuencia de sentencias y posteriormente se evalúa la expresión lógica, y si ésta es cierta se repite el proceso.

```

#include <iostream>
using namespace std;
int main ()
{
    unsigned num;
    do {
        cin >> num;
    } while ((num % 2) != 0);
    cout << "El número par es " << num << endl;
}

```

4.6. Programación Estructurada

Un programa sigue una metodología de programación estructurada si todas las estructuras de control que se utilizan (secuencia, selección, iteración y modularización) tienen un único punto de entrada y un único punto de salida. Esta característica hace posible que se pueda aplicar la abstracción para su diseño y desarrollo. La abstracción se basa en la identificación de los elementos a un determinado nivel, ignorando los detalles especificados en niveles inferiores. Un algoritmo que use tan sólo las estructuras de control tratadas en este tema, se denomina estructurado.

Bohm y Jacopini demostraron que todo problema computable puede resolverse usando únicamente estas estructuras de control. Ésta es la base de la programación estructurada. La abstracción algorítmica va un paso más allá y considera que cada nivel de refinamiento corresponde con un subprograma independiente

4.7. Ejemplos

Ejemplo 1

Programa que multiplica dos números mediante sumas acumulativas:

```

#include <iostream>
using namespace std;
int main ()
{
    cout << "Introduzca dos números: ";
    unsigned m, n;
    cin >> m >> n;
    // Sumar: m+m+m+...+m (n veces)
}

```

```

    unsigned total = 0;
    for (unsigned i = 0; i < n; ++i) {
        // Proceso iterativo: acumular el valor de 'm' al total
        total = total + m;
    }
    cout << total << endl;
}

```

Ejemplo 2

Programa que suma los números naturales menor a uno dado:

```

#include <iostream>
using namespace std;
int main ()
{
    cout << "Introduzca un número: ";
    unsigned n;
    cin >> n;
    // Sumar: 1 2 3 4 5 6 7 ... n
    unsigned suma = 0;
    for (unsigned i = 1; i <= n; ++i) {
        // Proceso iterativo: acumular el valor de 'i' al total
        suma = suma + i;
    }
    cout << suma << endl;
}

```

Ejemplo 3

Programa que divide dos números mediante restas sucesivas:

```

#include <iostream>
using namespace std;
int main ()
{
    cout << "Introduzca dos números: ";
    unsigned dividendo, divisor;
    cin >> dividendo >> divisor;
    if (divisor == 0) {
        cout << "El divisor no puede ser cero" << endl;
    } else {
        unsigned resto = dividendo;
        unsigned cociente = 0;
        while (resto >= divisor) {
            resto -= divisor;
            ++cociente;
        }
        cout << cociente << " " << resto << endl;
    }
}

```


Capítulo 5

Subprogramas. Funciones y Procedimientos

La abstracción es una herramienta mental que nos permite analizar, comprender y construir sistemas complejos. Así, identificamos y denominamos conceptos abstractos y aplicamos refinamientos sucesivos hasta comprender y construir el sistema completo.

Los subprogramas constituyen una herramienta del lenguaje de programación que permite al programador aplicar explícitamente la abstracción al diseño y construcción del software, proporcionando abstracción algorítmica (procedimental) a la construcción de programas.

Los subprogramas pueden ser vistos como un *mini* programa encargado de resolver algorítmicamente un subproblema que se encuentra englobado dentro de otro mayor. En ocasiones también pueden ser vistos como una ampliación/elevación del conjunto de operaciones básicas (acciones primitivas) del lenguaje de programación, proporcionándole nuevos mecanismos para resolver nuevos problemas.

5.1. Funciones y Procedimientos

Los subprogramas codifican la solución algorítmica a un determinado problema, de tal forma que cuando es necesario resolver dicho problema en un determinado momento de la computación, se invocará a una instancia del subprograma (mediante una llamada al subprograma) para que resuelva el problema para unos determinados parámetros. Así, en la invocación al subprograma, se transfiere la información que necesita para resolver el problema, entonces se ejecuta el código del subprograma que resolverá el problema y produce unos resultados que devuelve al lugar donde ha sido requerido. Podemos distinguir dos tipos de subprogramas:

- **Procedimientos**: encargados de resolver un problema computacional general. En el siguiente ejemplo el procedimiento `ordenar` ordena los valores de las variables pasadas como parámetros (`x` e `y`), de tal forma que cuando termine el procedimiento, las variables `x` e `y` tendrán el menor y el mayor valor respectivamente de los valores originales:

```
int main()
{
    int x = 8;
    int y = 4;
    ordenar(x, y);
    cout << x << " " << y << endl;
}
```

- **Funciones**: encargadas de realizar un cálculo computacional y generar un único resultado, normalmente calculado en función de los datos recibidos. En el siguiente ejemplo, la función `calcular_menor` calcula (y devuelve) el menor valor de los dos valores recibidos como parámetros:

```

int main()
{
    int x = 8;
    int y = 4;
    int z = calcular_menor(x, y);
    cout << "Menor: " << z << endl;
}

```

La llamada (invocación) a un subprograma se realiza mediante el nombre seguido por los parámetros actuales entre paréntesis, considerando que:

- La llamada a un procedimiento constituye por sí sola una sentencia independiente que puede ser utilizada como tal en el cuerpo de otros subprogramas (y del programa principal).
- La llamada a una función **no** constituye por sí sola una sentencia, por lo que debe aparecer dentro de alguna sentencia que utilice el valor resultado de la función.

5.2. Definición de Subprogramas

Los subprogramas codifican la solución algorítmica parametrizada a un determinado problema, es decir, especifica la secuencia de acciones a ejecutar para resolver un determinado problema dependiendo de unos determinados parámetros formales. Donde sea necesaria la resolución de dicho problema, se invocará a una instancia del subprograma para unos determinados parámetros actuales.

Considerando la norma de C++ de que antes de utilizar una determinada entidad en un programa, esta entidad deberá estar previamente declarada, normalmente deberemos definir los subprogramas en una posición previa a donde sean utilizados. Esta disposición de los subprogramas puede ser alterada como se indica en la sección 5.7. La definición de los subprogramas presentados anteriormente podría ser como se indica a continuación:

```

#include <iostream>
using namespace std;
int calcular_menor(int a, int b)
{
    int menor;
    if (a < b) {
        menor = a;
    } else {
        menor = b;
    }
    return menor;
}
void ordenar(int& a, int& b)
{
    if (a > b) {
        int aux = a;
        a = b;
        b = aux;
    }
}
int main()
{
    int x = 8;
    int y = 4;
    int z = calcular_menor(x, y);
    cout << "Menor: " << z << endl;
    ordenar(x, y);
    cout << x << " " << y << endl;
}

```

La definición de un subprograma comienza con un encabezamiento en la que se especifica en primer lugar el tipo del valor devuelto por éste si es una función, o `void` en caso de ser un procedimiento. A continuación vendrá el nombre del subprograma seguido por la declaración de sus parámetros formales entre paréntesis.

Los parámetros formales especifican como se realiza la transferencia de información entre el(sub)programa llamante y el subprograma llamado (véase 5.4). Normalmente la solución de un subproblema dependerá del valor de algunos datos, modificará el valor de otros datos, y posiblemente generará nuevos valores. Todo este flujo de información se realiza a través de los parámetros del subprograma.

El cuerpo del subprograma especifica la secuencia de acciones a ejecutar necesarias para resolver el subproblema especificado, y podrá definir tantas variables locales como necesite para desempeñar su misión. En el caso de una función, el valor que devuelve (el valor que toma tras la llamada) vendrá dado por el resultado de evaluar la expresión de la sentencia `return`. Aunque C++ es más flexible, nosotros sólo permitiremos una única utilización de la sentencia `return` y deberá ser al final del cuerpo de la función. Así mismo, un procedimiento no tendrá ninguna sentencia `return` en su cuerpo.

5.3. Ejecución de Subprogramas

Cuando se produce una llamada (invocación) a un subprograma:

1. Se establecen las vías de comunicación entre los algoritmos llamante y llamado por medio de los parámetros.
2. Posteriormente el flujo de ejecución pasa a ejecutar la primera sentencia del cuerpo del subprograma llamado, ejecutándose éste.
3. Cuando sea necesario, se crean las variables locales especificadas en el cuerpo de la definición del subprograma.
4. Cuando finaliza la ejecución del subprograma, las variables locales y parámetros previamente creados se destruyen, el flujo de ejecución retorna al (sub)programa llamante, y continúa la ejecución por la sentencia siguiente a la llamada realizada.

5.4. Paso de Parámetros. Parámetros por Valor y por Referencia

Todo el intercambio y transferencia de información entre el programa llamante y el subprograma llamado se debe realizar a través de los parámetros. Los *parámetros formales* son los que aparecen en la definición del subprograma, mientras que los *parámetros actuales* son los que aparecen en la llamada (invocación) al subprograma.

- Denominamos *parámetros de entrada* a aquellos parámetros que se utilizan para recibir la información necesaria para realizar una computación. Por ejemplo los parámetros `a` y `b` de la función `calcular_menor` anterior.
- Los parámetros de entrada se definen mediante *paso por valor* (cuando son de tipos simples), que significa que los parámetros formales son variables independientes que toman sus valores iniciales como *copias* de los valores de los parámetros actuales de la llamada en el momento de la invocación al subprograma. Se declaran especificando el tipo y el identificador asociado.

```
int calcular_menor(int a, int b)
{
    return (a < b) ? a : b ;
}
```

- Cuando los parámetros de entrada son de tipos compuestos (véase 6), entonces se definen mediante *paso por referencia constante* que será explicado más adelante.
- Denominamos *parámetros de salida* a aquellos parámetros que se utilizan para transferir al programa llamante información producida como parte de la computación/solución realizada por el subprograma.

- Los parámetros de salida se definen mediante *paso por referencia* que significa que el parámetro formal es una referencia a la variable que se haya especificado como parámetro actual de la llamada en el momento de la invocación al subprograma. Es decir, cualquier acción dentro del subprograma que se haga sobre el parámetro formal es equivalente a que se realice sobre la variable referenciada que aparece como parámetro actual en la llamada al subprograma. Se declaran especificando el tipo, el símbolo “ampersand” (&) y el identificador asociado.

En el siguiente ejemplo, el procedimiento `dividir` recibe dos valores sobre los cuales realizará la operación de división (`dividendo` y `divisor` son parámetros de entrada y son pasados por valor), y devuelve dos valores como resultado de la división (`cociente` y `resto` son parámetros de salida y son pasados por referencia):

```
void dividir(int dividendo, int divisor, int& coc, int& resto)
{
    coc = dividendo / divisor;
    resto = dividendo % divisor;
}
int main()
{
    int cociente;
    int resto;

    dividir(7, 3, cociente, resto);
    // ahora 'cociente' valdrá 2 y 'resto' valdrá 1
}
```

- Denominamos *parámetros de entrada/salida* a aquellos parámetros que se utilizan para recibir información necesaria para realizar la computación, y que tras ser modificada se transfiere al lugar de llamada como parte de la información producida resultado de la computación del subprograma. Por ejemplo los parámetros `a` y `b` del procedimiento `ordenar` anterior.

Los parámetros de entrada/salida se definen mediante *paso por referencia* y se declaran como se especificó anteriormente.

```
void ordenar(int& a, int& b)
{
    if (a > b) {
        int aux = a;
        a = b;
        b = aux;
    }
}
```

- En el caso de tipos compuestos, es conveniente definir los parámetros de entrada mediante *paso por referencia constante* de tal forma que el parámetro formal será una referencia al parámetro actual especificado en la llamada, tomando así su valor, pero no podrá modificarlo al ser una referencia constante, evitando así la semántica de salida asociada al paso por referencia.

El paso por referencia constante suele utilizarse para el paso de parámetros de entrada con tipos compuestos, ya que evita la duplicación de memoria y la copia del valor, que en el caso de tipos compuestos suele ser costosa.

Para ello, los parámetros se declaran como se especificó anteriormente para el paso por referencia, pero anteponiendo la palabra reservada `const`.

```
void escribir(const Persona& a)
{
    cout << a.nombre << " " << a.telefono << endl;
}
```

	Tipos	
	Simples	Compuestos
(↓) Entrada	P.Valor (<code>unsigned x</code>)	P.Ref.Cte (<code>const Persona& x</code>)
(↑) Salida, (↕) E/S	P.Ref (<code>unsigned& x</code>)	P.Ref (<code>Persona& x</code>)

En la llamada a un subprograma, se deben cumplir los siguientes requisitos:

- El número de parámetros actuales debe coincidir con el número de parámetros formales.
- La correspondencia entre parámetros actuales y formales es posicional.
- El tipo del parámetro actual debe coincidir con el tipo del correspondiente parámetro formal.
- Un parámetro formal de salida o entrada/salida (paso por referencia) requiere que el parámetro actual sea una variable.
- Un parámetro formal de entrada (paso por valor o paso por referencia constante) requiere que el parámetro actual sea una variable, constante o expresión.

	Tipos Simples		Tipos Compuestos	
	(↓) Ent P.Valor (<code>unsigned x</code>)	(↑) Sal (↕) E/S P.Referencia (<code>unsigned& x</code>)	(↓) Ent P.Ref.Constante (<code>const Persona& x</code>)	(↑) Sal (↕) E/S P.Referencia (<code>Persona& x</code>)
Parámetro Formal				
Parámetro Actual	Constante Variable Expresión	Variable	Constante Variable Expresión	Variable

5.5. Criterios de Modularización

No existen métodos objetivos para determinar como descomponer la solución de un problema en subprogramas, es una labor subjetiva. No obstante, se siguen algunos criterios que pueden guiarnos para descomponer un problema y modularizar adecuadamente. El diseñador de software debe buscar un **bajo acoplamiento** entre los subprogramas y una **alta cohesión** dentro de cada uno.

- Acoplamiento
 - Un objetivo en el diseño descendente es crear subprogramas aislados e independientes.
 - Debe haber alguna conexión entre los subprogramas para formar un sistema coherente.
 - Dicha conexión se conoce como acoplamiento.
 - Maximizar la independencia entre subprogramas será minimizar el acoplamiento.
- Cohesión
 - Hace referencia al grado de relación entre las diferentes partes internas dentro de un mismo subprograma.
 - Si la cohesión es muy débil, la diversidad entre las distintas tareas realizadas dentro del subprograma es tal que posteriores modificaciones podrán resultar complicadas.
 - Se busca maximizar la cohesión dentro de cada subprograma

Si no es posible analizar y comprender un subprograma de forma aislada e independiente del resto, entonces podemos deducir que la división modular no es la más adecuada.

5.6. Subprogramas “en Línea”

La llamada a un subprograma conlleva un pequeño coste debido al control y gestión de la misma que ocasiona cierta pérdida de tiempo de ejecución.

Hay situaciones en las que el subprograma es muy pequeño y nos interesa eliminar el coste asociado a su invocación. En ese caso podemos especificar que el subprograma se traduzca como *código en línea* en vez de como una llamada a un subprograma. Para ello se especificará la palabra reservada `inline` justo antes del tipo. De esta forma, se mantiene los beneficios proporcionados por la abstracción, pero se eliminan los costes asociados a la invocación.

```
inline int calcular_menor(int a, int b)
{
    return (a < b) ? a : b ;
}
```

Este mecanismo sólo es adecuado cuando el cuerpo del subprograma es muy pequeño, de tal forma que el coste asociado a la invocación dominaría respecto a la ejecución del cuerpo del mismo.

5.7. Declaración de Subprogramas. Prototipos

Los subprogramas, al igual que los tipos, constantes y variables, deben ser declarados antes de ser utilizados. Dicha declaración se puede realizar de dos formas: una de ellas consiste simplemente en definir el subprograma antes de utilizarlo. La otra posibilidad consiste en declarar el *prototipo* del subprograma antes de su utilización, y definirlo posteriormente. El ámbito de visibilidad del subprograma será global al fichero, es decir, desde el lugar donde ha sido declarado hasta el final del fichero. Para declarar un subprograma habrá que especificar el tipo del valor devuelto (o void si es un procedimiento) seguido por el nombre y la declaración de los parámetros formales igual que en la definición del subprograma, pero sin definir el cuerpo del mismo. En lugar de ello se terminará la declaración con el carácter “punto y coma” (;).

```
int calcular_menor(int a, int b); // prototipo de 'calcular_menor'
int main()
{
    int x = 8;
    int y = 4;
    int z = calcular_menor(x, y);
}
int calcular_menor(int a, int b) // definición de 'calcular_menor'
{
    return (a < b) ? a : b ;
}
```

5.8. Sobrecarga de Subprogramas y Operadores

Se denomina sobrecarga cuando distintos subprogramas se denominan con el mismo identificador u operador, pero se aplican a parámetros distintos. En el lenguaje de programación C++ es posible sobrecargar tanto subprogramas como operadores siempre y cuando tengan parámetros diferentes, y el compilador pueda discriminar entre ellos por la especificación de la llamada.

```
void imprimir(int x)
{
    cout << "entero: " << x << endl;
}
void imprimir(double x)
{
    cout << "real: " << x << endl;
}
```

```

inline double media(int x, int y, int z)
{
    return double(x + y + z) / 3.0;
}
inline double media(int x, int y)
{
    return double(x + y) / 2.0;
}

enum Color {
    rojo, amarillo, azul
};
inline void operator ++(Color& x)
{
    x = Color(x + 1);
}

```

5.9. Parámetros por Defecto

Es posible en C++ definir valores por defecto para los parámetros de un determinado subprograma. Para ello es necesario especificar los valores por defecto que tomarán los parámetros en caso de que no sean proporcionados por la invocación al mismo, desde un determinado parámetro hasta el final de los mismos. Téngase presente que no es posible *intercalar* parámetros con o sin valores por defecto. Por ejemplo:

```

#include <iostream>
using namespace std;
void escribir(int uno, int dos = 2, int tres = 3)
{
    cout << uno << " " << dos << " " << tres << endl;
}
int main()
{
    escribir(5, 6, 7);      // escribe: 5 6 7
    escribir(5, 6);        // escribe: 5 6 3
    escribir(5);            // escribe: 5 2 3
    // escribir();          // Error de compilación
    // escribir(5, 6, 7, 8); // Error de compilación
}

```

Nótese que la invocación al subprograma `escribir()` sin parámetros actuales es un error de compilación, ya que en su definición requiere que la invocación tenga al menos un parámetro actual, y como máximo tres parámetros actuales.

En el caso de declarar el prototipo de un subprograma, los valores de los parámetros por defecto se especificarán en la propia declaración del prototipo, y no en la definición del subprograma, ya que la aplicación de los parámetros por defecto se realiza a la invocación del subprograma, no a su definición.

```

#include <iostream>
using namespace std;
void escribir(int uno, int dos = 2, int tres = 3); // prototipo
int main()
{
    escribir(5, 6, 7);      // escribe: 5 6 7
    escribir(5, 6);        // escribe: 5 6 3
    escribir(5);            // escribe: 5 2 3
    // escribir();          // Error de compilación
}

```

```
// escribir(5, 6, 7, 8); // Error de compilación
}
void escribir(int uno, int dos, int tres)
{
    cout << uno << " " << dos << " " << tres << endl;
}
```

5.10. Subprogramas y Flujos de Entrada y Salida

Los flujos de entrada, de salida y de error (`cin`, `cout` y `cerr` respectivamente) pueden ser pasados como parámetros actuales (por referencia no constante) a subprogramas como se indica en el siguiente ejemplo:

```
#include <iostream>
using namespace std;
void leer_int(istream& entrada, int& dato)
{
    entrada >> dato;
}
void escribir_int(ostream& salida, int dato)
{
    salida << "Valor: " << dato << endl;
}
int main()
{
    int x;
    leer_int(cin, x);
    escribir_int(cout, x);
    escribir_int(cerr, x);
}
```

5.11. Pre-Condiciones y Post-Condiciones

- Pre-condición es un enunciado que debe ser cierto antes de la llamada a un subprograma. Especifica las condiciones bajo las cuales se ejecutará dicho subprograma.
- Post-condición es un enunciado que debe ser cierto tras la ejecución de un subprograma. Especifica el comportamiento de dicho subprograma.
- Codificar las pre/post-condiciones mediante asertos proporciona una valiosa documentación, y tiene varias ventajas:
 - Hace al programador explícitamente consciente de los prerequisites y del objetivo del subprograma.
 - Durante la depuración, las pre-condiciones comprueban que la llamada al subprograma se realiza bajo condiciones validas.
 - Durante la depuración, las post-condiciones comprueban que el comportamiento del subprograma es adecuado.
 - Sin embargo, a veces no es posible codificarlas fácilmente.

En C++, las pre-condiciones y post-condiciones se pueden especificar mediante asertos, para los cuales es necesario incluir la biblioteca `cassert`. Por ejemplo:

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



```
//-----
bool es_par(int num)
{
    return num % 2 == 0;
}
//-----
void dividir_par(int dividendo, int divisor, int& cociente, int& resto)
{
    assert(es_par(dividendo) && (divisor != 0)); // PRE-CONDICIÓN
    cociente = dividendo / divisor;
    resto = dividendo % divisor;
    assert(dividendo == (divisor * cociente + resto)); // POST-CONDICIÓN
}
```

Nota: en *GNU GCC* es posible desactivar la comprobación de asertos mediante la siguiente directiva de compilación:

```
g++ -DNDEBUG -ansi -Wall -Werror -o programa programa.cpp
```

5.12. Ejemplos

Ejemplo 1

Ejemplo de un programa que imprime los números primos existentes entre dos valores leídos por teclado:

```
//- fichero: primos.cpp -----
#include <iostream>
using namespace std;
void ordenar(int& menor, int& mayor)
{
    if (mayor < menor) {
        int aux = menor;
        menor = mayor;
        mayor = aux;
    }
}
inline bool es_divisible(int x, int y)
{
    return ( x % y == 0 );
}
bool es_primo(int x)
{
    int i;
    for (i = 2; ((i <= x/2) && ( ! es_divisible(x, i))); ++i) {
        // vacío
    }
    return (i == x/2+1);
}
void primos(int min, int max)
{
    cout << "Números primos entre " << min << " y " << max << endl;
    for (int i = min; i <= max; ++i) {
        if (es_primo(i)) {
            cout << i << " ";
        }
    }
    cout << endl;
}
```

```

int main()
{
    int min, max;
    cout << "Introduzca el rango de valores " ;
    cin >> min >> max ;
    ordenar(min, max);
    primos(min, max);
}
// - fin: primos.cpp -----

```

Ejemplo 2

Ejemplo de un programa que convierte grados sexagesimales a radianes:

```

// - fichero: gradrad.cpp -----
#include <iostream>
#include <string>
using namespace std;
// -- Constantes -----
const double PI = 3.1416;
const int PI_GRAD = 180;
const int MIN_GRAD = 60;
const int SEG_MIN = 60;
const int SEG_GRAD = SEG_MIN * MIN_GRAD;
// -- Subalgoritmos ----
void leer_grados (int& grad, int& min, int& seg)
{
    cout << "Grados, minutos y segundos ";
    cin >> grad >> min >> seg;
}
//-----
void escribir_radianes (double rad)
{
    cout << "Radianes: " << rad << endl;
}
//-----
double calc_rad (double grad_tot)
{
    return (grad_tot * PI) / double(PI_GRAD);
}
//-----
double calc_grad_tot (int grad, int min, int seg)
{
    return double(grad) + (double(min) / double(MIN_GRAD)) + (double(seg) / double(SEG_GRAD));
}
//-----
double transf_gr_rad (int grad, int min, int seg)
{
    double gr_tot = calc_grad_tot(grad, min, seg);
    return calc_rad(gr_tot);
}
// -- Principal -----
int main ()
{
    int grad, min, seg;
    leer_grados(grad, min, seg);
    double rad = transf_gr_rad(grad, min, seg);
    escribir_radianes(rad);
}

```

Capítulo 6

Tipos Compuestos

Los *tipos compuestos* surgen de la composición y/o agregación de otros tipos para formar nuevos tipos de mayor entidad. Existen dos formas fundamentales para crear tipos de mayor entidad: la composición de elementos, que denominaremos “Registros” o “Estructuras” y la agregación de elementos del mismo tipo, y se conocen como “Agregados”, “Arreglos” o mediante su nombre en inglés “Arrays”. Además de los tipos compuestos definidos por el programador mencionados anteriormente, los lenguajes de programación suelen proporcionar algún tipo adicional para representar las “cadenas de caracteres”.

6.1. Paso de Parámetros de Tipos Compuestos

Los lenguajes de programación normalmente utilizan el *paso por valor* y el *paso por referencia* para implementar la transferencia de información entre subprogramas descrita en el interfaz. Para la transferencia de información de *entrada*, el paso por valor supone *duplicar y copiar el valor* del parámetro actual en el formal. En el caso de tipos *simples*, el paso por valor es adecuado para la transferencia de información de *entrada*, sin embargo, si el tipo de dicho parámetro es *compuesto*, es posible que dicha copia implique una *alta sobrecarga*, tanto en espacio de memoria como en tiempo de ejecución. El lenguaje de programación *C++* permite realizar de forma eficiente la transferencia de información de **entrada** para tipos compuestos mediante el *paso por referencia constante*.

```
void imprimir(const Fecha& fech)
{
    cout << fech.dia << (int(fech.mes)+1) << fech.anyo << endl;
}
```

	Tipos	
	Simples	Compuestos
(↓) Entrada	P.Valor (unsigned x)	P.Ref.Cte (const Persona& p)
(↑) Salida, (↓) E/S	P.Ref (unsigned& x)	P.Ref (Persona& p)

Funciones que Retornan Tipos Compuestos

Por la misma razón y como *norma general*, salvo excepciones, tampoco es adecuado que una *función* retorne un valor de tipo compuesto, debido a la sobrecarga que generalmente ésto conlleva. En estos casos, suele ser más adecuado que el subprograma devuelva el valor de tipo compuesto como un parámetro de salida mediante el paso por referencia.

6.2. Cadenas de Caracteres en C++: el Tipo String

Las cadenas de caracteres representan una sucesión o secuencia de caracteres. Es un tipo de datos muy versátil, y es útil para representar información muy diversa:

- Información textual (caracteres)
- Entrada de datos y salida de resultados en forma de secuencia de caracteres.
- Información abstracta por medio de una secuencia de caracteres

Es posible utilizar el tipo `string` de la biblioteca estándar para representar cadenas de caracteres de *longitud finita* limitada por la implementación. Para ello, se debe incluir la biblioteca estándar `<string>`, así como utilizar el espacio de nombres de `std`. La definición de cadenas de caracteres de este tipo permite definir cadenas de caracteres mas robustas y con mejores características que las cadenas de caracteres predefinidas al estilo-C explicadas en el capítulo 8.

Es posible definir tanto constantes simbólicas como variables y parámetros de tipo `string`. Una cadena de caracteres literal se representa mediante una sucesión de caracteres entre comillas dobles. También es posible la asignación de cadenas de caracteres:

```
#include <iostream>
#include <string>
using namespace std;
const string AUTOR = "José Luis";
int main()
{
    string nombre = "Pepe";
    // ...
    nombre = AUTOR;
}
```

AUTOR:

J	o	s	e		L	u	i	s
0	1	2	3	4	5	6	7	8

nombre:

P	e	p	e
0	1	2	3

nombre:

J	o	s	e		L	u	i	s
0	1	2	3	4	5	6	7	8

Si no se le asigna un valor inicial a una variable de tipo `string`, entonces la variable tendrá como valor por defecto la cadena vacía (`""`).

Entrada y Salida de Cadenas de Caracteres

El operador `<<` aplicado a un flujo de salida (`cout` para el flujo de salida estándar, usualmente el terminal) permite mostrar el contenido de las cadenas de caracteres, tanto constantes como variables. Por ejemplo:

```
#include <iostream>
#include <string>
using namespace std;
const string AUTOR = "José Luis";
int main()
{
    string nombre = "Pepe";
    cout << "Nombre: " << nombre << " " << AUTOR << endl;
}
```

El operador `>>` aplicado a un flujo de entrada (`cin` para el flujo de entrada estándar, usualmente el teclado) permite leer secuencias de caracteres y almacenarlas en variables de tipo `string`. Por ejemplo:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    cout << "Introduzca el nombre: ";
    string nombre;
```

```

    cin >> nombre;
    cout << "Nombre: " << nombre << endl;
}

```

Este operador de entrada (>>) se comporta (como se especificó en el capítulo 3.2 dedicado a la Entrada y Salida básica) de la siguiente forma: elimina los espacios en blanco que hubiera al principio de la entrada de datos, y lee dicha entrada hasta que encuentre algún carácter de espacio en blanco, que no será leído y permanecerá en el buffer de entrada (véase 3.3) hasta la próxima operación de entrada. En caso de que durante la entrada surja alguna situación de error, dicha entrada se detiene y el flujo de entrada se pondrá en un estado erróneo. Se consideran espacios en blanco los siguientes caracteres: espacio en blanco, tabuladores, retorno de carro y nueva línea (' ', '\t', '\v', '\f', '\r', '\n').

También es posible leer una línea completa, hasta leer el fin de línea, desde el flujo de entrada, sin eliminar los espacios iniciales:

```

#include <iostream>
#include <string>
using namespace std;
int main()
{
    cout << "Introduzca el nombre: ";
    string nombre;
    getline(cin, nombre);
    cout << "Nombre: " << nombre << endl;
}

```

También es posible leer una línea completa, hasta leer un delimitador especificado, desde el flujo de entrada, sin eliminar los espacios iniciales:

```

#include <iostream>
#include <string>
using namespace std;
const char DELIMITADOR = '.';
int main()
{
    cout << "Introduzca el nombre: ";
    string nombre;
    getline(cin, nombre, DELIMITADOR);
    cout << "Nombre: " << nombre << endl;
}

```

Nótese que realizar una operación `getline` después de una operación con >> puede tener complicaciones, ya que >> dejara los espacios en blanco (y fin de línea) en el buffer, que serán leídos por `getline`. Por ejemplo:

```

#include <iostream>
#include <string>
using namespace std;
int main()
{
    cout << "Introduzca número: ";
    int n;
    cin >> n;
    cout << "Introduzca el nombre: ";
    string nombre;
    getline(cin, nombre);
    cout << "Número: " << n << " Nombre: " << nombre << endl;
}

```

Para evitar este problema, el siguiente subprograma leerá una cadena que sea distinta de la vacía:

```

#include <iostream>
#include <string>
using namespace std;
inline void leer_linea_no_vacia(istream& ent, string& linea)
{
    ent >> ws;          // salta los espacios en blanco y fin de línea
    getline(ent, linea); // leerá la primera línea no vacía
}
int main()
{
    cout << "Introduzca número: ";
    int n;
    cin >> n;
    cout << "Introduzca el nombre (NO puede ser vacío): ";
    string nombre;
    leer_linea_no_vacia(cin, nombre);
    cout << "Número: " << n << " Nombre: " << nombre << endl;
}

```

Por el contrario, en caso de que la cadena vacía sea una entrada válida, será necesario eliminar el resto de caracteres (incluyendo los espacios en blanco y fin de línea) del buffer de entrada, después de leer un dato con `>>`, de tal forma que el buffer esté limpio antes de realizar la entrada de la cadena de caracteres con `getline`. Por ejemplo, el subprograma `leer_int` elimina los caracteres del buffer después de leer un dato de tipo `int`:

```

#include <iostream>
#include <string>
#include <limits>
using namespace std;
inline void leer_int(istream& ent, int& dato)
{
    ent >> dato;          // lee el dato
    ent.ignore(numeric_limits<streamsize>::max(), '\n'); // elimina los caracteres del buffer
    // ent.ignore(10000, '\n'); // otra posibilidad de eliminar los caracteres del buffer
}
int main()
{
    cout << "Introduzca número: ";
    int n;
    leer_int(cin, n);
    cout << "Introduzca el nombre (puede ser vacío): ";
    string nombre;
    getline(cin, nombre);
    cout << "Número: " << n << " Nombre: " << nombre << endl;
}

```

Téngase en cuenta que para utilizar `numeric_limits<...>`, es necesario incluir la biblioteca estándar `<limits>`.

Operaciones con Cadenas de Caracteres

- Las cadenas de caracteres se pueden asignar a variables de dicho tipo. Por ejemplo:

```

#include <iostream>
#include <string>
using namespace std;
const string AUTOR = "José Luis";
int main()
{
    string nombre = "Pepe";
}

```

```
// ...
nombre = AUTOR;
}
```

- Es posible realizar la comparación lexicográfica¹ entre cadenas de caracteres del tipo `string` mediante los operadores relacionales (`==`, `!=`, `>`, `>=`, `<`, `<=`). Por ejemplo:

- `if (nombre >= AUTOR) { /*...*/ }`

- Es posible la concatenación de cadenas y caracteres mediante los operadores de concatenación (`+`, `+=`):

```
#include <iostream>
#include <string>
using namespace std;
const string AUTOR = "José Luis";
int main ()
{
    string nombre = AUTOR + "López";
    nombre += "Vázquez";
    nombre += 'z';
    nombre = AUTOR + 's';
}
```

- Comprobar si la cadena de caracteres está vacía:

- `if (nombre.empty()) { /*...*/ }`

- Para acceder al número de caracteres que componen la cadena:

- `unsigned ncar = nombre.size();`
- `unsigned ncar = nombre.length();`

- Para acceder al *i*-ésimo carácter de la cadena:

- `char c = nombre[i];` donde $i \in [0..nombre.size()-1]$
- `nombre[i] = 'z';` donde $i \in [0..nombre.size()-1]$

- Para acceder al *i*-ésimo carácter de la cadena, comprobando que el valor del índice (*i*) es adecuado, de tal forma que si el índice (*i*) se encuentra fuera de rango, entonces lanza la excepción `out_of_range` (véase 12):

- `char c = nombre.at(i);` donde $i \in [0..nombre.size()-1]$.
- `nombre.at(i) = 'z';` donde $i \in [0..nombre.size()-1]$.

- Obtener una *nueva* subcadena a partir del índice *i*, con un tamaño especificado por *sz*. Si no se especifica el tamaño, o (`sz > nombre.size()-i`), entonces se toma la subcadena desde el índice hasta el final. Si el índice (*i*) se encuentra fuera de rango, entonces lanza la excepción `out_of_range` (véase 12):

- `string sb = nombre.substr(i);` donde $i \in [0..nombre.size()]$
- `string sb = nombre.substr(i, sz);` donde $i \in [0..nombre.size()]$
- Nótese que **no** es válida la asignación a una subcadena: `nombre.substr(i, sz) = "...";`

¹Comparación lexicográfica se basa en la ordenación alfabética, y es comúnmente utilizada en los diccionarios.

Ejemplos

Ejemplo 1

Programa que convierte una cadena de caracteres a mayúsculas:

```
#include <iostream>
#include <string>
using namespace std;
// -- Subalgoritmos ----
void mayuscula (char& letra)
{
    if ((letra >= 'a') && (letra <= 'z')) {
        letra = char(letra - 'a' + 'A');
    }
}
void mayusculas (string& palabra)
{
    for (unsigned i = 0; i < palabra.size(); ++i) {
        mayuscula(palabra[i]);
    }
}
// -- Principal -----
int main ()
{
    string palabra;
    cin >> palabra;
    mayusculas(palabra);
    cout << palabra << endl;
}
```

Ejemplo 2

Programa que lee una palabra (formada por letras minúsculas), y escribe su plural según las siguientes reglas:

- Si acaba en vocal se le añade la letra 's'.
- Si acaba en consonante se le añaden las letras 'es'. Si la consonante es la letra 'z', se sustituye por la letra 'c'.
- Suponemos que la palabra introducida es correcta y está formada por letras minúsculas.

```
#include <iostream>
#include <string>
using namespace std;
// -- Subalgoritmos ----
bool es_vocal (char c)
{
    return (c == 'a') || (c == 'e') || (c == 'i') || (c == 'o') || (c == 'u');
}
void plural_1 (string& palabra)
{
    if (palabra.size() > 0) {
        if (es_vocal(palabra[palabra.size() - 1])) {
            palabra += 's';
        } else {
            if (palabra[palabra.size() - 1] == 'z') {
                palabra[palabra.size() - 1] = 'c';
            }
            palabra += "es";
        }
    }
}
```



```

    }
}
void plural_2 (string& palabra)
{
    if (palabra.size() > 0) {
        if (es_vocal(palabra[palabra.size() - 1])) {
            palabra += 's';
        } else if (palabra[palabra.size() - 1] == 'z') {
            palabra = palabra.substr(0, palabra.size() - 1) + "ces";
        } else {
            palabra += "es";
        }
    }
}
// -- Principal -----
int main ()
{
    string palabra;
    cin >> palabra;
    plural_1(palabra);
    cout << palabra << endl;
}

```

Ejemplo 3

Diseñe una función que devuelva verdadero si la palabra recibida como parámetro es “palíndromo” y falso en caso contrario.

```

bool es_palindromo (const string& palabra)
{
    bool ok = false;
    if (palabra.size() > 0) {
        unsigned i = 0;
        unsigned j = palabra.size() - 1;
        while ((i < j) && (palabra[i] == palabra[j])) {
            ++i;
            --j;
        }
        ok = i >= j;
    }
    return ok;
}

```

Ejemplo 4

Diseñe un subprograma que reemplace una parte de la cadena, especificada por un índice y una longitud, por otra cadena.

```

void reemplazar (string& str, unsigned i, unsigned sz, const string& nueva)
{
    if (i + sz < str.size()) {
        str = str.substr(0, i) + nueva + str.substr(i + sz, str.size() - (i + sz));
    } else if (i <= str.size()) {
        str = str.substr(0, i) + nueva;
    }
}

```

- Este subprograma es equivalente a la operación `str.replace(i, sz, nueva)`
- `str.replace(i, 0, nueva)` es equivalente a `str.insert(i, nueva)`
- `str.replace(i, sz, "")` es equivalente a `str.erase(i, sz)`.

Ⓐ Otras Operaciones con Cadenas de Caracteres

En las siguientes operaciones, `s1` y `s2` representan cadenas de caracteres de tipo `string`. `i1` e `i2` representan índices dentro de dichas cadenas, así como `sz1` y `sz2` representan tamaños (número de caracteres) dentro de dichas cadenas.

```
string s1;
string s2 = "pepeluis";
string s3 = s2;

string s4(s2, i2);           // construcción
string s5(s2, i2, sz2);
string s6("juanluis", sz2);
string s8(sz2, 'a');

s1.clear();                  // limpieza. equivalente a: s1 = "";

s1.assign(s2);               // asignar
s1.assign(s2, i2);
s1.assign(s2, i2, sz2);
s1.assign("pepe");
s1.assign("pepe", sz2);
s1.assign(sz2, 'a');

s1.append(s2);               // añadir
s1.append(s2, i2);
s1.append(s2, i2, sz2);
s1.append("pepe");
s1.append("pepe", sz2);
s1.append(sz2, 'a');

s1.insert(i1, s2);           // insertar
s1.insert(i1, s2, i2);
s1.insert(i1, s2, i2, sz2);
s1.insert(i1, "pepe");
s1.insert(i1, "pepe", sz2);
s1.insert(i1, sz2, 'a');

s1.erase(i1);                // eliminar
s1.erase(i1, sz1);

s1.replace(i1, sz1, s2);      // reemplazar
s1.replace(i1, sz1, s2, i2);
s1.replace(i1, sz1, s2, i2, sz2);
s1.replace(i1, sz1, "pepe");
s1.replace(i1, sz1, "pepe", sz2);
s1.replace(i1, sz1, sz2, 'a');

s1.resize(sz);                // cambiar el tamaño
s1.resize(sz, 'a');

int c = s1.compare(s2);       // comparar. devuelve < == > que 0
int c = s1.compare(i1, sz1, s2);
int c = s1.compare(i1, sz1, s2, i2);
int c = s1.compare(i1, sz1, s2, i2, sz2);
int c = s1.compare("pepe");
int c = s1.compare(i1, sz1, "pepe");
int c = s1.compare(i1, sz1, "pepe", sz2);

s1.reserve(tamano);           // reserva de la memoria interna
```

```
string::size_type mx = s1.capacity(); // capacidad de la reserva
```

Obtener la secuencia de caracteres como un cadena de caracteres **al estilo-C** (terminado en `'\0'`), para compatibilidad con subprogramas y bibliotecas que necesitan las cadenas de caracteres en dicho estilo (aquellas con prototipos `const char* xx` y `const char xx[]`):

```
const char* cad_c = s1.c_str();

char dest_c[MAX];
s1.copy(dest_c, sz1, i1 = 0); // sz1 <= MAX ; no pone '\0'
```

En las siguientes búsquedas, el patrón a buscar puede ser un string, cadena de caracteres constante o un carácter. Si la posición indicada está fuera de los límites del string, entonces se lanza una excepción `out_of_range`.

```
unsigned i1 = s1.find('a', i1=0); // buscar el patrón exacto
unsigned i1 = s1.find(s2, i1=0);
unsigned i1 = s1.find("cd", i1=0);
unsigned i1 = s1.find("cd", i1, sz2);

unsigned i1 = s1.rfind('a', i1=npos); // buscar el patrón exacto
unsigned i1 = s1.rfind(s2, i1=npos);
unsigned i1 = s1.rfind("cd", i1=npos);
unsigned i1 = s1.rfind("cd", i1, sz2);

unsigned i1 = s1.find_first_of('a', i1=0); // buscar cualquier carácter del patrón
unsigned i1 = s1.find_first_of(s2, i1=0);
unsigned i1 = s1.find_first_of("cd", i1=0);
unsigned i1 = s1.find_first_of("cd", i1, sz2);

unsigned i1 = s1.find_first_not_of('a', i1=0);
unsigned i1 = s1.find_first_not_of(s2, i1=0);
unsigned i1 = s1.find_first_not_of("cd", i1=0);
unsigned i1 = s1.find_first_not_of("cd", i1, sz2);

unsigned i1 = s1.find_last_of('a', i1=npos);
unsigned i1 = s1.find_last_of(s2, i1=npos);
unsigned i1 = s1.find_last_of("cd", i1=npos);
unsigned i1 = s1.find_last_of("cd", i1, sz2);

unsigned i1 = s1.find_last_not_of('a', i1=npos);
unsigned i1 = s1.find_last_not_of(s2, i1=npos);
unsigned i1 = s1.find_last_not_of("cd", i1=npos);
unsigned i1 = s1.find_last_not_of("cd", i1, sz2);

string::npos -> valor devuelto si no encontrado
```

6.3. Registros o Estructuras

El tipo *registro* se utiliza para la definición de un nuevo tipo mediante la composición de un número determinado de elementos que pueden ser de distintos tipos (simples y compuestos).

Un tipo registro se especifica enumerando los elementos (campos) que lo componen, indicando su tipo y su identificador con el que referenciarlo. Una vez definido el tipo, podremos utilizar la entidad (constante o variable) de dicho tipo como un todo o acceder a los diferentes elementos que lo componen. Por ejemplo, dado el tipo *Meses* definido en el Capítulo 2, podemos definir un nuevo tipo que represente el concepto de *Fecha* como composición de *día*, *mes* y *año*.

```
struct Fecha {
    unsigned dia;
```

```

    Meses mes;
    unsigned anyo;
};

```

y posteriormente utilizarlo para definir constantes:

```
const Fecha f_nac = { 20 , Febrero, 2001} ;
```

o utilizarlo para definir variables:

```
Fecha f_nac;
```

Los valores del tipo `Fecha` se componen de tres elementos concretos (el día de tipo `unsigned`, el mes de tipo `Meses` y el año de tipo `unsigned`). Los identificadores `dia`, `mes` y `anyo` representan los nombres de sus elementos componentes, denominados **campos**, y su ámbito de visibilidad se restringe a la propia definición del registro. Los campos de un registro pueden ser de cualquier tipo de datos, simple o estructurado. Por ejemplo:

```

// -- Tipos -----
struct Empleado {
    unsigned codigo;
    unsigned sueldo;
    Fecha fecha_ingreso;
};
enum Palo {
    Oros, Copas, Espadas, Bastos
};
struct Carta {
    Palo palo;
    unsigned valor;
};
struct Tiempo {
    unsigned hor;
    unsigned min;
    unsigned seg;
};
// -- Principal -----
int main ()
{
    Empleado e;
    Carta c;
    Tiempo t1, t2;
    // ...
}

```

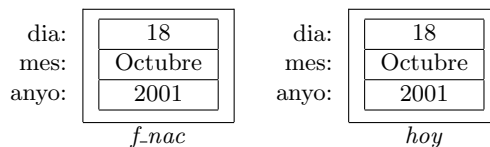
Una vez declarada una entidad (constante o variable) de tipo registro, por ejemplo la variable `f_nac`, podemos referirnos a ella en su globalidad (realizando asignaciones y pasos de parámetros) o acceder a sus componentes (campos) especificándolos tras el operador punto (`.`), donde un determinado componente podrá utilizarse en cualquier lugar en que resulten válidas las variables de su mismo tipo.

```

int main ()
{
    Fecha f_nac, hoy;
    f_nac.dia = 18;
    f_nac.mes = Octubre;
    f_nac.anyo = 2001;

    hoy = f_nac;
}

```



Ejemplo

```

#include <iostream>
#include <string>
using namespace std;
// -- Constantes -----
const unsigned SEGMIN = 60;
const unsigned MINHOR = 60;
const unsigned MAXHOR = 24;
const unsigned SEGHOR = SEGMIN * MINHOR;
// -- Tipos -----
struct Tiempo {
    unsigned horas;
    unsigned minutos;
    unsigned segundos;
};
// -- Subalgoritmos ----
unsigned leer_rango (unsigned inf, unsigned sup)
{
    unsigned num;
    do {
        cin >> num;
    } while ( ! ((num >= inf) && (num < sup)));
    return num;
}
void leer_tiempo (Tiempo& t)
{
    t.horas = leer_rango(0, MAXHOR);
    t.minutos = leer_rango(0, MINHOR);
    t.segundos = leer_rango(0, SEGMIN);
}
void escribir_tiempo (const Tiempo& t)
{
    cout << t.horas << ":" << t.minutos << ":" << t.segundos;
}
unsigned tiempo_a_seg (const Tiempo& t)
{
    return (t.horas * SEGHOR) + (t.minutos * SEGMIN) + (t.segundos);
}
void seg_a_tiempo (unsigned sg, Tiempo& t)
{
    t.horas = sg / SEGHOR;
    t.minutos = (sg % SEGHOR) / SEGMIN;
    t.segundos = (sg % SEGHOR) % SEGMIN;
}
void diferencia (const Tiempo& t1, const Tiempo& t2, Tiempo& dif)
{
    seg_a_tiempo(tiempo_a_seg(t2) - tiempo_a_seg(t1), dif);
}
// -- Principal -----
int main ()
{
    Tiempo t1, t2, dif;
    leer_tiempo(t1);
    leer_tiempo(t2);
    diferencia(t1, t2, dif);
    escribir_tiempo(dif);
    cout << endl;
}

```

Ⓐ Sobrecarga de Operadores para Estructuras

En caso de ser necesario, es posible definir los operadores relacionales para registros:

```
inline bool operator == (const Tiempo& t1, const Tiempo& t2)
{
    return (t1.hor == t2.hor) && (t1.min == t2.min) && (t1.sec == t2.sec);
}
inline bool operator != (const Tiempo& t1, const Tiempo& t2)
{
    return ! (t1 == t2);
}
inline bool operator < (const Tiempo& t1, const Tiempo& t2)
{
    return ((t1.hor < t2.hor)
        || ((t1.hor == t2.hor) && ((t1.min < t2.min)
            || ((t1.min == t2.min) && (t1.sec < t2.sec)))));
}
inline bool operator > (const Tiempo& t1, const Tiempo& t2)
{
    return (t2 < t1);
}
inline bool operator <= (const Tiempo& t1, const Tiempo& t2)
{
    return ! (t2 < t1);
}
inline bool operator >= (const Tiempo& t1, const Tiempo& t2)
{
    return ! (t1 < t2);
}
```

Así como también es posible definir los operadores de entrada y salida para registros:

```
inline ostream& operator << (ostream& out, const Tiempo& t)
{
    return out << t.hor << " " << t.min << " " << t.sec ;
}
inline istream& operator >> (istream& in, Tiempo& t)
{
    return in >> t.hor >> t.min >> t.sec ;
}
int main()
{
    Tiempo t1, t2;
    cin >> t1 >> t2;
    if (t1 < t2) {
        cout << t1 << endl;
    } else {
        cout << t2 << endl;
    }
}
```

Nótese como en la definición de los operadores de entrada y salida para registros, estos mismos retornan un valor de *tipo referencia*, el cual es un concepto complejo que será explicado más adelante (véase 13).

6.4. Agregados: el Tipo Array

El tipo *array* se utiliza para la definición de un nuevo tipo mediante la **agregación** de entidades menores del *mismo tipo*, es decir, se define como una colección de un número determinado (definido

en tiempo de compilación) de elementos de un mismo tipo de datos, de tal forma que se puede acceder a cada elemento individual de la colección de forma *parametrizada* mediante índices.

Los arrays son útiles en todas aquellas circunstancias en que necesitamos tener almacenados una colección de valores (un número fijo predeterminado en tiempo de compilación) a los cuales pretendemos acceder de forma parametrizada, normalmente para aplicar un proceso iterativo.

Es posible utilizar el tipo `array` de la biblioteca TR1 para definir agregados. Para ello, se debe incluir la biblioteca `<tr1/array>`, así como utilizar el espacio de nombres de `std::tr1`. La definición de agregados de este tipo permite definir agregados mas robustos y con mejores características que los agregados predefinidos explicados en el capítulo 8.

Un tipo *agregado* se especifica declarando el tipo base de los elementos que lo componen y el número de elementos (constante especificada en tiempo de compilación) de que consta dicha agregación. Así, por ejemplo, podemos definir un nuevo tipo `Vector` como un agregado de 5 elementos, cada uno del tipo `int`, y definir variables y constantes de dicho tipo (nótese que los elementos constantes del tipo array se especifican entre *llaves dobles*):

```
#include <tr1/array>
using namespace std::tr1;
// -- Constantes -----
const unsigned NELMS = 5;
// -- Tipos -----
typedef array<int, NELMS> Vector;
// -- Constantes -----
const Vector PRIMOS = {{ 2, 3, 5, 7, 11 }};
```

2	3	5	7	11
0	1	2	3	4

```
// -- Principal -----
int main ()
{
    Vector v;          v: 

|   |   |   |   |   |
|---|---|---|---|---|
| ? | ? | ? | ? | ? |
| 0 | 1 | 2 | 3 | 4 |


}
```

El tipo base (de los elementos) del array puede ser de tipo simple o compuesto, así, por ejemplo, podemos definir un nuevo tipo `Citas` como un agregado de 4 elementos, cada uno del tipo `Fecha`, y definir variables y constantes de dicho tipo:

```
#include <tr1/array>
using namespace std::tr1;
struct Fecha {
    unsigned dia;
    unsigned mes;
    unsigned anyo;
};
const int N_CITAS = 4;
typedef array<Fecha, N_CITAS> Citas;
const Citas CUMPLEANYOS = {{
    { 1, 1, 2001 },
    { 2, 2, 2002 },
    { 3, 3, 2003 },
    { 4, 4, 2004 }
}};
```

CUMPLEANYOS:			
1	2	3	4
1	2	3	4
2001	2002	2003	2004
0	1	2	3

```
int main()
{
    Citas cit;
    // ...
    cit = CUMPLEANYOS;
}
```

Un agregado de tipo `array<...>` acepta la asignación (=) entre variables de dicho tipo. También se le pueden aplicar los operadores relacionales (==, !=, >, >=, <, <=) a entidades del mismo tipo `array<...>` siempre y cuando a los elementos del agregado (del tipo base del array) se le puedan aplicar dichos operadores.

Para conocer el número de elementos que componen un determinado agregado, la operación `cit.size()` proporciona dicho valor, que en este ejemplo es 4.

Para acceder a un elemento concreto del agregado, especificaremos entre corchetes (`[` y `]`) el índice de la posición que ocupa el mismo, teniendo en cuenta que el primer elemento ocupa la posición 0 (cero) y el último elemento ocupa la posición `a.size()-1`. Por ejemplo `cit[0]` y `cit[cit.size()-1]` aluden al primer y último elemento del agregado respectivamente. Un determinado elemento puede utilizarse en cualquier lugar donde sea válido una variable de su mismo tipo base.

El lenguaje de programación C++ no comprueba que los accesos a los elementos de un agregado sean correctos y se encuentren dentro de los límites válidos del array, por lo que será responsabilidad del programador comprobar que así sea.

También es posible acceder a un determinado elemento mediante la operación `at(i)`, de tal forma que si el valor del índice `i` está fuera del rango válido, entonces se lanzará una excepción `out_of_range`. Se puede tanto utilizar como modificar el valor de este elemento.

```
#include <tr1/array>
using namespace std::tr1;
struct Fecha {
    unsigned dia;
    unsigned mes;
    unsigned anyo;
};
const int N_CITAS = 4;
typedef array<Fecha, N_CITAS> Citas;
int main()
{
    Citas cit;
    cit[0].dia = 18;
    cit[0].mes = 10;
    cit[0].anyo = 2001;
    for (int i = 0; i < cit.size(); ++i) {
        cit[i].dia = 1;
        cit[i].mes = 1;
        cit[i].anyo = 2002;
    }
    cit[N_CITAS] = { 1, 1, 2002 }; // ERROR. Acceso fuera de los límites
    cit.at(N_CITAS) = { 1, 1, 2002 }; // ERROR. Lanza excepción out_of_range
    // ...
}
```

Ejemplo 1

```
#include <iostream>
#include <tr1/array>
using namespace std;
using namespace std::tr1;
// -- Constantes -----
const unsigned NELMS = 5;
// -- Tipos -----
typedef array<int, NELMS> Vector;
// -- Subalgoritmos ----
void leer (Vector& v)
{
    for (unsigned i = 0; i < v.size(); ++i) {
        cin >> v[i];
    }
}
unsigned sumar (const Vector& v)
```



```

{
    int suma = 0;
    for (unsigned i = 0; i < v.size(); ++i) {
        suma += v[i];
    }
    return suma;
}
// -- Principal -----
int main ()
{
    Vector v1, v2;
    leer(v1);
    leer(v2);
    if (sumar(v1) == sumar(v2)) {
        cout << "Misma suma" << endl;
    }
    if (v1 < v2) {
        cout << "Vector Menor" << endl;
    }
    v1 = v2;    // Asignación
    if (v1 == v2) {
        cout << "Vectores Iguales" << endl;
    }
}

```

Ejemplo 2

Programa que lee las ventas de cada “agente” e imprime su sueldo que se calcula como una cantidad fija (1000 €) más un incentivo que será un 10% de las ventas que ha realizado. Dicho incentivo sólo será aplicable a aquellos agentes cuyas ventas superen los 2/3 de la media de ventas del total de los agentes.

```

#include <iostream>
#include <tr1/array>
using namespace std;
using namespace std::tr1;
// -- Constantes -----
const unsigned NAGENTES = 20;
const double SUELDO_FIJO = 1000.0;
const double INCENTIVO = 10.0;
const double PROMEDIO = 2.0 / 3.0;
// -- Tipos -----
typedef array<double, NAGENTES> Ventas;
// -- Subalgoritmos ----
double calc_media (const Ventas& v)
{
    double suma = 0.0;
    for (unsigned i = 0; i < v.size(); ++i) {
        suma += v[i];
    }
    return suma / double(v.size());
}
inline double porcentaje (double p, double valor)
{
    return (p * valor) / 100.0;
}
void leer_ventas (Ventas& v)
{
    for (unsigned i = 0; i < v.size(); ++i) {

```

```

        cout << "Introduzca ventas del Agente " << i << ": ";
        cin >> v[i];
    }
}

void imprimir_sueldos (const Ventas& v)
{
    double umbral = PROMEDIO * calc_media(ventas);
    for (unsigned i = 0; i < v.size(); ++i) {
        double sueldo = SUELDO_FIJO;
        if (v[i] >= umbral) {
            sueldo += porcentaje(INCENTIVO, v[i]);
        }
        cout << "Agente: " << i << " Sueldo: " << sueldo << endl;
    }
}

// -- Principal -----
int main ()
{
    Ventas ventas;
    leer_ventas(ventas);
    imprimir_sueldos(ventas);
}

```

Agregados Incompletos

Hay situaciones donde un array se define en tiempo de compilación con un tamaño mayor que el número de elementos actuales válidos que contendrá durante el Tiempo de Ejecución.

- Gestionar el array con huecos durante la ejecución del programa suele ser, en la mayoría de los casos, complejo e ineficiente.
- Mantener los elementos actuales válidos consecutivos al comienzo del array suele ser más adecuado:
 - Marcar la separación entre los elementos actuales válidos de los elementos vacíos con algún valor de adecuado suele ser, en la mayoría de los casos, complejo e ineficiente.
 - Definir un registro que contenga tanto el array, como el número de elementos actuales válidos consecutivos que contiene suele ser más adecuado.

Ejemplo

```

#include <iostream>
#include <tr1/array>
using namespace std;
using namespace std::tr1;
// -- Constantes -----
const unsigned MAX_AGENTES = 20;
const double SUELDO_FIJO = 1000.0;
const double INCENTIVO = 10.0;
const double PROMEDIO = 2.0 / 3.0;
// -- Tipos -----
typedef array<double, MAX_AGENTES> Datos;
struct Ventas {
    unsigned nelms;
    Datos elm;
};
// -- Subalgoritmos ----
double calc_media (const Ventas& v)
{
    double suma = 0.0;
    for (unsigned i = 0; i < v.nelms; ++i) {

```

```

        suma += v.elm[i];
    }
    return suma / double(v.nelms);
}
inline double porcentaje (double p, double valor)
{
    return (p * valor) / 100.0;
}
void leer_ventas_ag (unsigned i, double& v)
{
    cout << "Introduzca ventas Agente " << i << ": ";
    cin >> v;
}
// -----
// Dos métodos diferentes de leer un
// vector incompleto:
// -----
// Método-1: cuando se conoce a priori el número
//           de elementos que lo componen
// -----
void leer_ventas_2 (Ventas& v)
{
    unsigned nag;
    cout << "Introduzca total de agentes: ";
    cin >> nag;
    if (nag > v.elm.size()) {
        v.nelms = 0;
        cout << "Error" << endl;
    } else {
        v.nelms = nag;
        for (unsigned i = 0; i < v.nelms; ++i) {
            leer_ventas_ag(i, v.elm[i]);
        }
    }
}
// -----
// Método-2: cuando NO se conoce a priori el número
//           de elementos que lo componen, y este
//           número depende de la propia lectura de
//           los datos
// -----
void leer_ventas_1 (Ventas& v)
{
    double vent_ag;
    v.nelms = 0;
    leer_ventas_ag(v.nelms+1, vent_ag);
    while ((v.nelms < v.elm.size()) && (vent_ag > 0)) {
        v.elm[v.nelms] = vent_ag;
        ++v.nelms;
        leer_ventas_ag(v.nelms+1, vent_ag);
    }
}
// -----
void imprimir_sueldos (const Ventas& v)
{
    double umbral = PROMEDIO * calc_media(ventas);
    for (unsigned i = 0; i < v.nelms; ++i) {
        double sueldo = SUELDO_FIJO;
        if (v.elm[i] >= umbral) {

```

```

        sueldo += porcentaje(INCENTIVO, v.elm[i]);
    }
    cout << "Agente: " << i << " Sueldo: " << sueldo << endl;
}
}
// -- Principal -----
int main ()
{
    Ventas ventas;
    leer_ventas(ventas);
    imprimir_sueldos(ventas);
}

```

Agregados Multidimensionales

El *tipo Base* de un array puede ser tanto simple como compuesto, por lo tanto puede ser otro array, dando lugar a *arrays con múltiples dimensiones*. Así, cada elemento de un array puede ser a su vez otro array.

Los agregados anteriormente vistos se denominan de *una dimensión*. Así mismo, es posible declarar agregados de varias dimensiones. Un ejemplo de un agregado de dos dimensiones:

```

#include <iostream>
#include <tr1/array>
using namespace std;
using namespace std::tr1;
// -- Constantes -----
const unsigned NFILAS = 3;
const unsigned NCOLUMNAS = 5;
// -- Tipos -----
typedef array<unsigned, NCOLUMNAS> Fila;
typedef array<Fila, NFILAS> Matriz;
// -- Principal -----
int main ()
{
    Matriz m;
    for (unsigned f = 0; f < m.size(); ++f) {
        for (unsigned c = 0; c < m[f].size(); ++c) {
            m[f][c] = (f * 10) + c;
        }
    }
    Matriz mx = m;          // asigna a mx los valores de la Matriz m
    Fila fil = m[0];        // asigna a fil el array con valores {{ 00, 01, 02, 03, 04 }}
    unsigned n = m[2][4];   // asigna a n el valor 24
}

```

	m:				
0	00	01	02	03	04
1	10	11	12	13	14
2	20	21	22	23	24
	0	1	2	3	4

Donde *m* hace referencia a una variable de tipo *Matriz*, *m[f]* hace referencia a la fila *f* de la matriz *m* (que es de tipo *Fila*), y *m[f][c]* hace referencia al elemento *c* de la fila *f* de la matriz *m* (que es de tipo *unsigned*). Del mismo modo, el número de filas de la matriz *m* es igual a *m.size()*, y el número de elementos de la fila *f* de la matriz *m* es igual a *m[f].size()*.

Ejemplo 1

Diseñar un programa que lea una matriz de 3×5 de números enteros (fila a fila), almacenándolos en un array bidimensional, finalmente imprima la matriz según el siguiente formato:

```

a  a  a  a  a  b
a  a  a  a  a  b
a  a  a  a  a  b
c  c  c  c  c

```

donde *a* representa los elementos de la matriz leída desde el teclado, *b* representa el resultado de sumar todos los elementos de la fila correspondiente, y *c* representa el resultado de sumar todos los elementos de la columna donde se encuentran. Nótese en el ejemplo como es posible pasar como parámetro una única *fila*, y sin embargo no es posible pasar como parámetro una única *columna*.

```
#include <iostream>
#include <tr1/array>
using namespace std;
using namespace std::tr1;
// -- Constantes -----
const unsigned NFILAS = 3;
const unsigned NCOLUMNAS = 5;
// -- Tipos -----
typedef array<int, NCOLUMNAS> Fila;
typedef array<Fila, NFILAS> Matriz;
// -- Subalgoritmos ----
int sumar_fila (const Fila& fil)
{
    int suma = 0;
    for (unsigned c = 0; c < fil.size(); ++c) {
        suma += fil[c];
    }
    return suma;
}
int sumar_columna (const Matriz& m, unsigned c)
{
    int suma = 0;
    for (unsigned f = 0; f < m.size(); ++f) {
        suma += m[f][c];
    }
    return suma;
}
void escribir_fila (const Fila& fil)
{
    for (unsigned c = 0; c < fil.size(); ++c) {
        cout << fil[c] << " ";
    }
}
void escribir_matriz_formato (const Matriz& m)
{
    for (unsigned f = 0; f < m.size(); ++f) {
        escribir_fila(m[f]);
        cout << sumar_fila(m[f]);
        cout << endl;
    }
    for (unsigned c = 0; c < m[0].size(); ++c) {
        cout << sumar_columna(m, c) << " ";
    }
    cout << endl;
}
void leer_matriz (Matriz& m)
{
    cout << "Escribe fila a fila" << endl;
    for (unsigned f = 0; f < m.size(); ++f) {
        for (unsigned c = 0; c < m[f].size(); ++c) {
            cin >> m[f][c];
        }
    }
}
```

```
// -- Principal -----
int main ()
{
    Matriz m;
    leer_matriz(m);
    escribir_matriz_formato(m);
}
```

Ejemplo 2

Diseñe un programa que realice el producto de 2 matrices de máximo 10×10 elementos:

```
#include <iostream>
#include <cassert>
#include <tr1/array>
using namespace std;
using namespace std::tr1;
// -- Constantes -----
const unsigned MAX = 10;
// -- Tipos -----
typedef array<double, MAX> Fila;
typedef array<Fila, MAX> Tabla;
struct Matriz {
    unsigned n_fil;
    unsigned n_col;
    Tabla datos;
};
// -- Subalgoritmos ----
void leer_matriz (Matriz& m)
{
    cout << "Dimensiones?: ";
    cin >> m.n_fil >> m.n_col;
    assert(m.n_fil <= m.datos.size() && m.n_col <= m.datos[0].size());
    cout << "Escribe valores fila a fila:" << endl;
    for (unsigned f = 0; f < m.n_fil; ++f) {
        for (unsigned c = 0; c < m.n_col; ++c) {
            cin >> m.datos[f][c];
        }
    }
}

void escribir_matriz (const Matriz& m)
{
    for (unsigned f = 0; f < m.n_fil; ++f) {
        for (unsigned c = 0; c < m.n_col; ++c) {
            cout << m.datos[f][c] << " ";
        }
        cout << endl;
    }
}

double suma_fila_por_col (const Matriz& x, const Matriz& y, unsigned f, unsigned c)
{
    assert(x.n_col == y.n_fil); // PRE-COND
    double suma = 0.0;
    for (unsigned k = 0; k < x.n_col; ++k) {
        suma += x.datos[f][k] * y.datos[k][c];
    }
    return suma;
}

void mult_matriz (Matriz& m, const Matriz& a, const Matriz& b)
{
}
```

```

    assert(a.n_col == b.n_fil); // PRE-COND
    m.n_fil = a.n_fil;
    m.n_col = b.n_col;
    for (unsigned f = 0; f < m.n_fil; ++f) {
        for (unsigned c = 0; c < m.n_col; ++c) {
            m.datos[f][c] = suma_fil_por_col(a, b, f, c);
        }
    }
}
// -- Principal -----
int main ()
{
    Matriz a,b,c;
    leer_matriz(a);
    leer_matriz(b);
    if (a.n_col != b.n_fil) {
        cout << "No se puede multiplicar." << endl;
    } else {
        mult_matriz(c, a, b);
        escribir_matriz(c);
    }
}

```

6.5. Resolución de Problemas Utilizando Tipos Compuestos

Diseñe un programa para gestionar una agenda personal que contenga la siguiente información: Nombre, Teléfono, Dirección, Calle, Número, Piso, Código Postal y Ciudad, y las siguientes operaciones:

- Añadir los datos de una persona.
- Acceder a los datos de una persona a partir de su nombre.
- Borrar una persona a partir de su nombre.
- Modificar los datos de una persona a partir de su nombre.
- Listar el contenido completo de la agenda.

```

#include <iostream>
#include <string>
#include <cassert>
#include <tr1/array>
using namespace std;
using namespace std::tr1;
// -- Constantes -----
const unsigned MAX_PERSONAS = 50;
// -- Tipos -----
struct Direccion {
    unsigned num;
    string calle;
    string piso;
    string cp;
    string ciudad;
};
struct Persona {
    string nombre;
    string tel;
    Direccion direccion;
};
// -- Tipos -----

```

```

typedef array<Persona, MAX_PERSONAS> Personas;
struct Agenda {
    unsigned n_pers;
    Personas pers;
};
enum Cod_Error {
    OK, AG_LLENA, NO_ENCONTRADO, YA_EXISTE
};
// -- Subalgoritmos ----
void Inicializar (Agenda& ag)
{
    ag.n_pers = 0;
}
//-----
void Leer_Direccion (Direccion& dir)
{
    cin >> dir.calle;
    cin >> dir.num;
    cin >> dir.piso;
    cin >> dir.cp;
    cin >> dir.ciudad;
}
//-----
void Escribir_Direccion (const Direccion& dir)
{
    cout << dir.calle << " ";
    cout << dir.num << " ";
    cout << dir.piso << " ";
    cout << dir.cp << " ";
    cout << dir.ciudad << " ";
}
//-----
void Leer_Persona (Persona& per)
{
    cin >> per.nombre;
    cin >> per.tel;
    Leer_Direccion(per.direccion);
}
//-----
void Escribir_Persona (const Persona& per)
{
    cout << per.nombre << " ";
    cout << per.tel << " ";
    Escribir_Direccion(per.direccion);
    cout << endl;
}
//-----
// Busca una Persona en la Agenda
// Devuelve su posición si se encuentra, o bien >= ag.n_pers en otro caso
unsigned Buscar_Persona (const string& nombre, const Agenda& ag)
{
    unsigned i = 0;
    while ((i < ag.n_pers) && (nombre != ag.pers[i].nombre)) {
        ++i;
    }
    return i;
}
//-----
void Anyadir (Agenda& ag, const Persona& per)

```



```

{
    assert(ag.n_pers < ag.pers.size());
    ag.pers[ag.n_pers] = per;
    ++ag.n_pers;
}
//-----
void Eliminar (Agenda& ag, unsigned pos)
{
    assert(pos < ag.n_pers);
    if (pos < ag.npers-1) {
        ag.pers[pos] = ag.pers[ag.n_pers - 1];
    }
    --ag.n_pers;
}
//-----
void Anyadir_Persona (const Persona& per, Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(per.nombre, ag);
    if (i < ag.n_pers) {
        ok = YA_EXISTE;
    } else if (ag.n_pers >= ag.pers.size()) {
        ok = AG_LLENA;
    } else {
        ok = OK;
        Anyadir(ag, per);
    }
}
//-----
void Borrar_Persona (const string& nombre, Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {
        ok = OK;
        Eliminar(ag, i);
    }
}
//-----
void Modificar_Persona (const string& nombre, const Persona& nuevo, Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {
        Eliminar(ag, i);
        Anyadir_Persona(nuevo, ag, ok);
    }
}
//-----
void Imprimir_Persona (const string& nombre, const Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {
        ok = OK;
        Escribir_Persona(ag.pers[i]);
    }
}

```

```

}
//-----
void Imprimir_Agenda (const Agenda& ag, Cod_Error& ok)
{
    for (unsigned i = 0; i < ag.n_pers; ++i) {
        Escribir_Persona(ag.pers[i]);
    }
    ok = OK;
}
//-----
char Menu ()
{
    char opcion;
    cout << endl;
    cout << "a. - Añadir Persona" << endl;
    cout << "b. - Buscar Persona" << endl;
    cout << "c. - Borrar Persona" << endl;
    cout << "d. - Modificar Persona" << endl;
    cout << "e. - Imprimir Agenda" << endl;
    cout << "x. - Salir" << endl;
    do {
        cout << "Introduzca Opción: ";
        cin >> opcion;
    } while ( ! (((opcion >= 'a') && (opcion <= 'e')) || (opcion == 'x')));
    return opcion;
}
//-----
void Escribir_Cod_Error (Cod_Error cod)
{
    switch (cod) {
        case OK:
            cout << "Operación correcta" << endl;
            break;
        case AG_LLENA:
            cout << "Agenda llena" << endl;
            break;
        case NO_ENCONTRADO:
            cout << "La persona no se encuentra en la agenda" << endl;
            break;
        case YA_EXISTE:
            cout << "La persona ya se encuentra en la agenda" << endl;
            break;
    }
}
// -- Principal -----
int main ()
{
    Agenda ag;
    char opcion;
    Persona per;
    string nombre;
    Cod_Error ok;
    Inicializar(ag);
    do {
        opcion = Menu();
        switch (opcion) {
            case 'a':
                cout << "Introduzca los datos de la Persona" << endl;
                cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl;

```

```
        Leer_Persona(per);
        Anyadir_Persona(per, ag, ok);
        Escribir_Cod_Error(ok);
        break;
    case 'b':
        cout << "Introduzca Nombre" << endl;
        cin >> nombre;
        Imprimir_Persona(nombre, ag, ok);
        Escribir_Cod_Error(ok);
        break;
    case 'c':
        cout << "Introduzca Nombre" << endl;
        cin >> nombre;
        Borrar_Persona(nombre, ag, ok);
        Escribir_Cod_Error(ok);
        break;
    case 'd':
        cout << "Introduzca Nombre" << endl;
        cin >> nombre;
        cout << "Nuevos datos de la Persona" << endl;
        cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl;
        Leer_Persona(per);
        Modificar_Persona(nombre, per, ag, ok);
        Escribir_Cod_Error(ok);
        break;
    case 'e':
        Imprimir_Agenda(ag, ok);
        Escribir_Cod_Error(ok);
        break;
    }
} while (opcion != 'x' );
}
```


Capítulo 7

Búsqueda y Ordenación

Los algoritmos de búsqueda de un elemento en una colección de datos, así como los algoritmos de ordenación, son muy utilizados comúnmente, por lo que merecen un estudio explícito. En el caso de los algoritmos de búsqueda, éstos normalmente retornan la posición, dentro de la colección de datos, del elemento buscado, y en caso de no ser encontrado, retornan una posición *no válida*. Por otra parte, los algoritmos de ordenación organizan una colección de datos de acuerdo con algún criterio de ordenación. Los algoritmos de ordenación que se verán en este capítulo son los más fáciles de programar, pero sin embargo son los más ineficientes.

7.1. Búsqueda Lineal (Secuencial)

La búsqueda lineal es adecuada como mecanismo de búsqueda general en colecciones de datos *sin organización* conocida. Consiste en ir recorriendo secuencialmente la colección de datos hasta encontrar el elemento buscado, o en última instancia recorrer toda la colección completa, en cuyo caso el elemento buscado no habrá sido encontrado. En los siguientes ejemplos se presentan los algoritmos básicos, los cuales pueden ser adaptados según las circunstancias.

```
//-----
typedef array<int, MAXIMO> Vector;
//-----
// busca la posición del primer elemento == x
// si no encontrado, retorna v.size()
//-----
unsigned buscar(int x, const Vector& v)
{
    unsigned i = 0;
    while ((i < v.size()) && (x != v[i])) {
        ++i;
    }
    return i;
}
//-----
// busca la posición del primer elemento >= x
//-----
unsigned buscar_mayig(int x, const Vector& v)
{
    unsigned i = 0;
    while ((i < v.size()) && (x > v[i])) {
        ++i;
    }
    return i;
}
//-----
```

```
// busca la posición del primer elemento > x
//-----
unsigned buscar_may(int x, const Vector& v)
{
    unsigned i = 0;
    while ((i < v.size()) && (x >= v[i])) {
        ++i;
    }
    return i;
}
//-----
```

7.2. Búsqueda Binaria

La búsqueda binaria es adecuada como mecanismo de búsqueda cuando las colecciones de datos se encuentran *ordenadas* por algún criterio. Consiste en comprobar si el elemento buscado es igual, menor o mayor que el elemento que ocupa la posición central de la colección de datos, en caso de ser mayor o menor que dicho elemento, se descartan los elementos no adecuados de la colección de datos, y se repite el proceso hasta encontrar el elemento o hasta que no queden elementos adecuados en la colección, en cuyo caso el elemento no habrá sido encontrado. En los siguientes ejemplos se presentan los algoritmos básicos, los cuales pueden ser adaptados según las circunstancias.

```
//-----
typedef array<int, MAXIMO> Vector;
//-----
// busca la posición del primer elemento == x
// si no encontrado, retorna v.size()
//-----
unsigned buscar_bin(int x, const Vector& v)
{
    unsigned i = 0;
    unsigned f = v.size();
    unsigned res = v.size();
    while (i < f) {
        unsigned m = (i + f) / 2;
        if (x == v[m]) {
            res = i = f = m;
        } else if (x < v[m]) {
            f = m;
        } else {
            i = m + 1;
        }
    }
    return res;
}
//-----
// busca la posición del primer elemento >= x
//-----
unsigned buscar_bin_mayig(int x, const Vector& v)
{
    unsigned i = 0;
    unsigned f = v.size();
    while (i < f) {
        unsigned m = (i + f) / 2;
        if (x <= v[m]) {
            f = m;
        } else {
            i = m + 1;
        }
    }
}
```

```

    }
}
return i;
}
//-----
// busca la posición del primer elemento > x
//-----
unsigned buscar_bin_may(int x, const Vector& v)
{
    unsigned i = 0;
    unsigned f = v.size();
    while (i < f) {
        unsigned m = (i + f) / 2;
        if (x < v[m]) {
            f = m;
        } else {
            i = m + 1;
        }
    }
    return i;
}
//-----

```

7.3. Ordenación por Intercambio (Burbuja)

Se hacen múltiples recorridos sobre la *zona no ordenada* del array, ordenando los elementos *consecutivos*, trasladando en cada uno de ellos al elemento más pequeño hasta el inicio de dicha zona.

```

//-----
typedef array<int, MAXIMO> Vector;
//-----
inline void intercambio(int& x, int& y)
{
    int a = x;
    x = y;
    y = a;
}
//-----
void subir_menor(Vector& v, unsigned pos)
{
    for (unsigned i = v.size()-1; i > pos; --i) {
        if (v[i] < v[i-1]) {
            intercambio(v[i], v[i-1]);
        }
    }
}
//-----
void burbuja(Vector& v)
{
    for (unsigned pos = 0; pos < v.size()-1; ++pos) {
        subir_menor(v, pos);
    }
}
//-----

```

7.4. Ordenación por Selección

Se busca el elemento más pequeño de la *zona no ordenada* del array, y se traslada al inicio dicha zona, repitiendo el proceso hasta ordenar completamente el array.

```
//-----
typedef array<int, MAXIMO> Vector;
//-----
inline void intercambio(int& x, int& y)
{
    int a = x;
    x = y;
    y = a;
}
//-----
unsigned posicion_menor(const Vector& v, unsigned pos)
{
    unsigned pos_menor = pos;
    for (unsigned i = pos_menor+1; i < v.size(); ++i) {
        if (v[i] < v[pos_menor]) {
            pos_menor = i;
        }
    }
    return pos_menor;
}
//-----
inline void subir_menor(Vector& v, unsigned pos)
{
    unsigned pos_menor = posicion_menor(v, pos);
    if (pos != pos_menor) {
        intercambio(v[pos], v[pos_menor]);
    }
}
//-----
void seleccion(Vector& v)
{
    for (unsigned pos = 0; pos < v.size()-1; ++pos) {
        subir_menor(v, pos);
    }
}
//-----
```

7.5. Ordenación por Inserción

Se toma el primer elemento de la *zona no ordenada* del array, y se inserta en la posición adecuada de la *zona ordenada* del array, repitiendo el proceso hasta ordenar completamente el array.

```
//-----
typedef array<int, MAXIMO> Vector;
//-----
unsigned buscar_posicion(const Vector& v, unsigned posicion)
{
    unsigned i = 0;
    while (/*(i < posicion)&&*/ (v[posicion] > v[i])) {
        ++i;
    }
    return i;
}
```



```
//-----
void abrir_hueco(Vector& v, unsigned p_hueco, unsigned p_elm)
{
    for (unsigned i = p_elm; i > p_hueco; --i) {
        v[i] = v[i-1];
    }
}
//-----
void insercion(Vector& v)
{
    for (unsigned pos = 1; pos < v.size(); ++pos) {
        unsigned p_hueco = buscar_posicion(v, pos);
        if (p_hueco != pos) {
            int aux = v[pos];
            abrir_hueco(v, p_hueco, pos);
            v[p_hueco] = aux;
        }
    }
}
//-----
```

7.6. Ordenación por Inserción Binaria

Es igual que el algoritmo de ordenación por inserción, pero la posición del elemento a insertar se realiza mediante una búsqueda binaria.

```
//-----
typedef array<int, MAXIMO> Vector;
//-----
unsigned buscar_posicion_bin(const Vector& v, unsigned nelms,
                           int x)
{
    unsigned izq = 0;
    unsigned der = nelms;
    while (izq < der) {
        unsigned med = (izq + der) / 2;
        if (x < v[med]) {
            der = med;
        } else {
            izq = med + 1;
        }
    }
    return izq;
}
//-----
void abrir_hueco(Vector& v, unsigned p_hueco, unsigned p_elm)
{
    for (unsigned i = p_elm; i > p_hueco; --i) {
        v[i] = v[i-1];
    }
}
//-----
void insercion_bin(Vector& v)
{
    for (unsigned pos = 1; pos < v.size(); ++pos) {
        unsigned p_hueco = buscar_posicion_bin(v, pos, v[pos]);
        if (p_hueco != pos) {
            int aux = v[pos];
            abrir_hueco(v, p_hueco, pos);
        }
    }
}
```

```

        v[p_hueco] = aux;
    }
}
}
//-----

```

7.7. Aplicación de los Algoritmos de Búsqueda y Ordenación

Diseñe un programa para gestionar una agenda personal *ordenada* que contenga la siguiente información: Nombre, Teléfono, Dirección, Calle, Número, Piso, Código Postal y Ciudad, y las siguientes operaciones:

- Añadir los datos de una persona.
- Acceder a los datos de una persona a partir de su nombre.
- Borrar una persona a partir de su nombre.
- Modificar los datos de una persona a partir de su nombre.
- Listar el contenido completo de la agenda.

```

#include <iostream>
#include <string>
#include <cassert>
#include <tr1/array>
using namespace std;
using namespace std::tr1;
// -- Constantes -----
const unsigned MAX_PERSONAS = 50;
// -- Tipos -----
struct Direccion {
    unsigned num;
    string calle;
    string piso;
    string cp;
    string ciudad;
};
struct Persona {
    string nombre;
    string tel;
    Direccion direccion;
};
// -- Tipos -----
typedef array<Persona, MAX_PERSONAS> Personas;
struct Agenda {
    unsigned n_pers;
    Personas pers;
};
enum Cod_Error {
    OK, AG_LLENA, NO_ENCONTRADO, YA_EXISTE
};
// -- Subalgoritmos ----
void Inicializar (Agenda& ag)
{
    ag.n_pers = 0;
}
//-----
void Leer_Direccion (Direccion& dir)
{
    cin >> dir.calle;
}

```

```

        cin >> dir.num;
        cin >> dir.piso;
        cin >> dir.cp;
        cin >> dir.ciudad;
    }
    //-----
    void Escribir_Direccion (const Direccion& dir)
    {
        cout << dir.calle << " ";
        cout << dir.num << " ";
        cout << dir.piso << " ";
        cout << dir.cp << " ";
        cout << dir.ciudad << " ";
    }
    //-----
    void Leer_Persona (Persona& per)
    {
        cin >> per.nombre;
        cin >> per.tel;
        Leer_Direccion(per.direccion);
    }
    //-----
    void Escribir_Persona (const Persona& per)
    {
        cout << per.nombre << " ";
        cout << per.tel << " ";
        Escribir_Direccion(per.direccion);
        cout << endl;
    }
    //-----
    // Busca una Persona en la Agenda Ordenada
    // Devuelve su posición si se encuentra, o bien >= ag.n_pers en otro caso
    unsigned Buscar_Persona (const string& nombre, const Agenda& ag)
    {
        unsigned i = 0;
        unsigned f = ag.n_pers;
        unsigned res = ag.n_pers;
        while (i < f) {
            unsigned m = (i + f) / 2;
            int cmp = nombre.compare(ag.pers[m].nombre);
            if (cmp == 0) {
                res = i = f = m;
            } else if (cmp < 0) {
                f = m;
            } else {
                i = m + 1;
            }
        }
        return res;
    }
    //-----
    unsigned Buscar_Posicion (const string& nombre, const Agenda& ag)
    {
        unsigned i = 0;
        while ((i < ag.n_pers) && (nombre > ag.pers[i].nombre)) {
            ++i;
        }
        return i;
    }
}

```

```

//-----
void Anyadir_Ord (Agenda& ag, unsigned pos, const Persona& per)
{
    for (unsigned i = ag.n_pers; i > pos; --i) {
        ag.pers[i] = ag.pers[i - 1];
    }
    ag.pers[pos] = per;
    ++ag.n_pers;
}
//-----
void Eliminar_Ord (Agenda& ag, unsigned pos)
{
    --ag.n_pers;
    for (unsigned i = pos; i < ag.n_pers; ++i) {
        ag.pers[i] = ag.pers[i + 1];
    }
}
//-----
void Anyadir_Persona (const Persona& per, Agenda& ag, Cod_Error& ok)
{
    unsigned pos = Buscar_Posicion(per.nombre, ag);
    if ((pos < ag.n_pers) && (per.nombre == ag.pers[pos].nombre)) {
        ok = YA_EXISTE;
    } else if (ag.n_pers >= ag.pers.size()) {
        ok = AG_LLENA;
    } else {
        ok = OK;
        Anyadir_Ord(ag, pos, per);
    }
}
//-----
void Borrar_Persona (const string& nombre, Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {
        ok = OK;
        Eliminar_Ord(ag, i);
    }
}
//-----
void Modificar_Persona (const string& nombre, const Persona& nuevo, Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {
        ok = OK;
        Eliminar_Ord(ag, i);
        Anyadir_Persona(nuevo, ag, ok);
    }
}
//-----
void Imprimir_Persona (const string& nombre, const Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    }
}

```

```

    } else {
        ok = OK;
        Escribir_Persona(ag.pers[i]);
    }
}
//-----
void Imprimir_Agenda (const Agenda& ag, Cod_Error& ok)
{
    for (unsigned i = 0; i < ag.n_pers; ++i) {
        Escribir_Persona(ag.pers[i]);
    }
    ok = OK;
}
//-----
char Menu ()
{
    char opcion;
    cout << endl;
    cout << "a. - Añadir Persona" << endl;
    cout << "b. - Buscar Persona" << endl;
    cout << "c. - Borrar Persona" << endl;
    cout << "d. - Modificar Persona" << endl;
    cout << "e. - Imprimir Agenda" << endl;
    cout << "x. - Salir" << endl;
    do {
        cout << "Introduzca Opción: ";
        cin >> opcion;
    } while ( ! (((opcion >= 'a') && (opcion <= 'e')) || (opcion == 'x')));
    return opcion;
}
//-----
void Escribir_Cod_Error (Cod_Error cod)
{
    switch (cod) {
        case OK:
            cout << "Operación correcta" << endl;
            break;
        case AG_LLENA:
            cout << "Agenda llena" << endl;
            break;
        case NO_ENCONTRADO:
            cout << "La persona no se encuentra en la agenda" << endl;
            break;
        case YA_EXISTE:
            cout << "La persona ya se encuentra en la agenda" << endl;
            break;
    }
}
// -- Principal -----
int main ()
{
    Agenda ag;
    char opcion;
    Persona per;
    string nombre;
    Cod_Error ok;
    Inicializar(ag);
    do {
        opcion = Menu();

```

```
switch (opcion) {
case 'a':
    cout << "Introduzca los datos de la Persona"<<endl;
    cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl;
    Leer_Persona(per);
    Anyadir_Persona(per, ag, ok);
    Escribir_Cod_Error(ok);
    break;
case 'b':
    cout << "Introduzca Nombre" << endl;
    cin >> nombre;
    Imprimir_Persona(nombre, ag, ok);
    Escribir_Cod_Error(ok);
    break;
case 'c':
    cout << "Introduzca Nombre" << endl;
    cin >> nombre;
    Borrar_Persona(nombre, ag, ok);
    Escribir_Cod_Error(ok);
    break;
case 'd':
    cout << "Introduzca Nombre" << endl;
    cin >> nombre;
    cout << "Nuevos datos de la Persona" << endl;
    cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl;
    Leer_Persona(per);
    Modificar_Persona(nombre, per, ag, ok);
    Escribir_Cod_Error(ok);
    break;
case 'e':
    Imprimir_Agenda(ag, ok);
    Escribir_Cod_Error(ok);
    break;
}
} while (opcion != 'x' );
}
```

Capítulo 8

Otros Tipos Compuestos en C

OBS

Este capítulo muestra la utilización de otros tipos de datos compuestos usualmente utilizados en el *lenguaje de programación C*. Aunque se muestran aquí por completitud, no se recomienda el uso de agregados (o arrays) *predefinidos* ni de las cadenas de caracteres al *estilo-C* en programas escritos en C++, ya que C++ tiene tipos definidos para representar estas estructuras de datos de forma más robusta y conveniente, como se ha visto en el capítulo 6.

8.1. Agregados o Arrays Predefinidos

OBS

El tipo predefinido *array* se utiliza para la definición de un nuevo tipo mediante la **agregación** de entidades menores del *mismo tipo*, es decir, se define como una colección de un número determinado (definido en tiempo de compilación) de elementos de un mismo tipo de datos, de tal forma que se puede acceder a cada elemento individual de la colección de forma *parametrizada* mediante índices.

Los arrays son útiles en todas aquellas circunstancias en que necesitamos tener almacenados una colección de valores (un número fijo predeterminado en tiempo de compilación) a los cuales pretendemos acceder de forma parametrizada, normalmente para aplicar un proceso iterativo.

Un tipo *agregado* se especifica declarando el tipo base de los elementos que lo componen y el número de elementos (constante especificada en tiempo de compilación) de que consta dicha agregación. Así, por ejemplo, podemos definir un nuevo tipo **Vector** como un agregado de 5 elementos, cada uno del tipo **int**, y definir variables y constantes de dicho tipo (nótese que los elementos constantes del tipo array se especifican entre *llaves simples*):

```
// -- Constantes -----
const unsigned NELMS = 5;
// -- Tipos -----
typedef int Vector[NELMS];
// -- Constantes -----
const Vector PRIMOS = { 2, 3, 5, 7, 11 };
// -- Principal -----
int main ()
{
    Vector v;
    v: [ ?  ?  ?  ?  ? ]
        0  1  2  3  4
    PRIMOS: [ 2  3  5  7  11 ]
              0  1  2  3  4
```

El tipo base (de los elementos) del array puede ser de tipo simple o compuesto, así, por ejemplo, podemos definir un nuevo tipo **Citas** como un agregado de 4 elementos, cada uno del tipo **Fecha**, y definir variables y constantes de dicho tipo:

```

struct Fecha {
    unsigned dia;
    unsigned mes;
    unsigned anyo;
};
const int N_CITAS = 4;
typedef Fecha Citas[N_CITAS];
const Citas CUMPLEANYOS = {
    { 1, 1, 2001 },
    { 2, 2, 2002 },
    { 3, 3, 2003 },
    { 4, 4, 2004 }
};
int main()
{
    Citas cit;
}

```

CUMPLEANYOS:

1	2	3	4
1	2	3	4
2001	2002	2003	2004
0	1	2	3

No es posible asignar variables de tipo array predefinido. Del mismo modo, **tampoco** se le pueden aplicar los operadores relacionales (`==`, `!=`, `>`, `>=`, `<`, `<=`) a entidades de tipo array predefinido.

Para acceder a un elemento concreto del agregado, especificaremos entre corchetes (`[` y `]`) el índice de la posición que ocupa el mismo, teniendo en cuenta que el primer elemento ocupa la posición 0 (cero) y el último elemento ocupa la posición del número de elementos menos 1. Por ejemplo `cit[0]` y `cit[N_CITAS-1]` aluden al primer y último elemento del agregado respectivamente. Un determinado elemento puede utilizarse en cualquier lugar donde sea válido una variable de su mismo tipo base.

El lenguaje de programación C no comprueba que los accesos a los elementos de un agregado son correctos y se encuentran dentro de los límites válidos del array predefinido, por lo que será responsabilidad del programador comprobar que así sea.

```

struct Fecha {
    unsigned dia;
    unsigned mes;
    unsigned anyo;
};
const int N_CITAS = 4;
typedef Fecha Citas[N_CITAS];
int main()
{
    Citas cit;
    cit[0].dia = 18;
    cit[0].mes = 10;
    cit[0].anyo = 2001;
    for (int i = 0; i < N_CITAS; ++i) {
        cit[i].dia = 1;
        cit[i].mes = 1;
        cit[i].anyo = 2002;
    }
    cit[N_CITAS] = { 1, 1, 2002 };    // ERROR. Acceso fuera de los límites
    // ...
}

```

Ejemplo 1

```

#include <iostream>
using namespace std;
// -- Constantes -----
const unsigned NELMS = 5;
// -- Tipos -----

```



```

typedef int Vector[NELMS];
// -- Subalgoritmos ----
void leer (Vector& v)
{
    for (unsigned i = 0; i < NELMS; ++i) {
        cin >> v[i];
    }
}
int sumar (const Vector& v)
{
    int suma = 0;
    for (unsigned i = 0; i < NELMS; ++i) {
        suma += v[i];
    }
    return suma;
}
// -- Principal -----
int main ()
{
    Vector v1, v2;
    leer(v1);
    leer(v2);
    if (sumar(v1) == sumar(v2)) {
        cout << "Misma suma" << endl;
    }
}

```

Ejemplo 2

Programa que lee las ventas de cada “agente” e imprime su sueldo que se calcula como una cantidad fija (1000 €) más un incentivo que será un 10% de las ventas que ha realizado. Dicho incentivo sólo será aplicable a aquellos agentes cuyas ventas superen los 2/3 de la media de ventas del total de los agentes.

```

#include <iostream>
using namespace std;
// -- Constantes -----
const unsigned NAGENTES = 20;
const double SUELDO_FIJO = 1000.0;
const double INCENTIVO = 10.0;
const double PROMEDIO = 2.0 / 3.0;
// -- Tipos -----
typedef double Ventas[NAGENTES];
// -- Subalgoritmos ----
double calc_media (const Ventas& v)
{
    double suma = 0.0;
    for (unsigned i = 0; i < NAGENTES; ++i) {
        suma += v[i];
    }
    return suma / double(NAGENTES);
}
inline double porcentaje (double p, double valor)
{
    return (p * valor) / 100.0;
}
void leer_ventas (Ventas& v)
{
    for (unsigned i = 0; i < NAGENTES; ++i) {

```

```

        cout << "Introduzca ventas del Agente " << i << ": ";
        cin >> v[i];
    }
}

void imprimir_sueldos (const Ventas& v)
{
    double umbral = PROMEDIO * calc_media(ventas);
    for (unsigned i = 0; i < NAGENTES; ++i) {
        double sueldo = SUELDO_FIJO;
        if (v[i] >= umbral) {
            sueldo += porcentaje(INCENTIVO, v[i]);
        }
        cout << "Agente: " << i << " Sueldo: " << sueldo << endl;
    }
}

// -- Principal -----
int main ()
{
    Ventas ventas;
    leer_ventas(ventas);
    imprimir_sueldos(ventas);
}

```

Agregados Incompletos

Hay situaciones donde un array se define en tiempo de compilación con un tamaño mayor que el número de elementos actuales válidos que contendrá durante el Tiempo de Ejecución.

- Gestionar el array con huecos durante la ejecución del programa suele ser, en la mayoría de los casos, complejo e ineficiente.
- Mantener los elementos actuales válidos consecutivos al comienzo del array suele ser más adecuado:
 - Marcar la separación entre los elementos actuales válidos de los elementos vacíos con algún valor de adecuado suele ser, en la mayoría de los casos, complejo e ineficiente.
 - Definir un registro que contenga tanto el array, como el número de elementos actuales válidos consecutivos que contiene suele ser más adecuado.

Ejemplo

```

#include <iostream>
using namespace std;
// -- Constantes -----
const unsigned MAX_AGENTES = 20;
const double SUELDO_FIJO = 1000.0;
const double INCENTIVO = 10.0;
const double PROMEDIO = 2.0 / 3.0;
// -- Tipos -----
typedef double Datos[MAX_AGENTES];
struct Ventas {
    unsigned nelms;
    Datos elm;
};
// -- Subalgoritmos ----
double calc_media (const Ventas& v)
{
    double suma = 0.0;
    for (unsigned i = 0; i < v.nelms; ++i) {
        suma += v.elm[i];
    }
}

```

```

        return suma / double(v.nelms);
    }
    inline double porcentaje (double p, double valor)
    {
        return (p * valor) / 100.0;
    }
    void leer_ventas_ag (unsigned i, double& v)
    {
        cout << "Introduzca ventas Agente " << i << ": ";
        cin >> v;
    }
    // -----
    // Dos métodos diferentes de leer un
    // vector incompleto:
    // -----
    // Método-1: cuando se conoce a priori el número
    //           de elementos que lo componen
    // -----
    void leer_ventas_2 (Ventas& v)
    {
        unsigned nag;
        cout << "Introduzca total de agentes: ";
        cin >> nag;
        if (nag > MAX_AGENTES) {
            v.nelms = 0;
            cout << "Error" << endl;
        } else {
            v.nelms = nag;
            for (unsigned i = 0; i < v.nelms; ++i) {
                leer_ventas_ag(i, v.elm[i]);
            }
        }
    }
    // -----
    // Método-2: cuando NO se conoce a priori el número
    //           de elementos que lo componen, y este
    //           número depende de la propia lectura de
    //           los datos
    // -----
    void leer_ventas_1 (Ventas& v)
    {
        double vent_ag;
        v.nelms = 0;
        leer_ventas_ag(v.nelms+1, vent_ag);
        while ((v.nelms < MAX_AGENTES)&&(vent_ag > 0)) {
            v.elm[v.nelms] = vent_ag;
            ++v.nelms;
            leer_ventas_ag(v.nelms+1, vent_ag);
        }
    }
    // -----
    void imprimir_sueldos (const Ventas& v)
    {
        double umbral = PROMEDIO * calc_media(ventas);
        for (unsigned i = 0; i < v.nelms; ++i) {
            double sueldo = SUELDO_FIJO;
            if (v.elm[i] >= umbral) {
                sueldo += porcentaje(INCENTIVO, v.elm[i]);
            }
        }
    }

```

```

        cout << "Agente: " << i << " Sueldo: " << sueldo << endl;
    }
}
// -- Principal -----
int main ()
{
    Ventas ventas;
    leer_ventas(ventas);
    imprimir_sueldos(ventas);
}

```

Agregados Multidimensionales

El *tipo Base* de un array puede ser tanto simple como compuesto, por lo tanto puede ser otro array, dando lugar a *arrays con múltiples dimensiones*. Así, cada elemento de un array puede ser a su vez otro array.

Los agregados anteriormente vistos se denominan de *una dimensión*. Así mismo, es posible declarar agregados de varias dimensiones. Un ejemplo de un agregado de dos dimensiones:

```

#include <iostream>
using namespace std;
// -- Constantes -----
const unsigned NFILAS = 3;
const unsigned NCOLUMNAS = 5;
// -- Tipos -----
typedef unsigned Fila[NCOLUMNAS];
typedef Fila Matriz[NFILAS];
// -- Principal -----
int main ()
{
    Matriz m;
    for (unsigned f = 0; f < NFILAS; ++f) {
        for (unsigned c = 0; c < NCOLUMNAS; ++c) {
            m[f][c] = (f * 10) + c;
        }
    }
}

```

m:					
0	00	01	02	03	04
1	10	11	12	13	14
2	20	21	22	23	24
	0	1	2	3	4

Donde `m` hace referencia a una variable de tipo `Matriz`, `m[f]` hace referencia a la fila `f` de la matriz `m` (que es de tipo `Fila`), y `m[f][c]` hace referencia al elemento `c` de la fila `f` de la matriz `m` (que es de tipo `unsigned`).

Ejemplo 1

Diseñar un programa que lea una matriz de 3×5 de números enteros (fila a fila), almacenándolos en un array bidimensional, finalmente imprima la matriz según el siguiente formato:

```

a  a  a  a  a  b
a  a  a  a  a  b
a  a  a  a  a  b
c  c  c  c  c

```

donde *a* representa los elementos de la matriz leída desde el teclado, *b* representa el resultado de sumar todos los elementos de la fila correspondiente, y *c* representa el resultado de sumar todos los elementos de la columna donde se encuentran. Nótese en el ejemplo como es posible pasar como parámetro una única *fila*, y sin embargo no es posible pasar como parámetro una única *columna*.

```

#include <iostream>
using namespace std;
// -- Constantes -----

```

```

const unsigned NFILAS = 3;
const unsigned NCOLUMNAS = 5;
// -- Tipos -----
typedef int Fila[NCOLUMNAS];
typedef Fila Matriz[NFILAS];
// -- Subalgoritmos ----
int sumar_fila (const Fila& fil)
{
    int suma = 0;
    for (unsigned c = 0; c < NCOLUMNAS; ++c) {
        suma += fil[c];
    }
    return suma;
}
int sumar_columna (const Matriz& m, unsigned c)
{
    int suma = 0;
    for (unsigned f = 0; f < NFILAS; ++f) {
        suma += m[f][c];
    }
    return suma;
}
void escribir_fila (const Fila& fil)
{
    for (unsigned c = 0; c < NCOLUMNAS; ++c) {
        cout << fil[c] << " ";
    }
}
void escribir_matriz_formato (const Matriz& m)
{
    for (unsigned f = 0; f < NFILAS; ++f) {
        escribir_fila(m[f]);
        cout << sumar_fila(m[f]);
        cout << endl;
    }
    for (unsigned c = 0; c < NCOLUMNAS; ++c) {
        cout << sumar_columna(m, c) << " ";
    }
    cout << endl;
}
void leer_matriz (Matriz& m)
{
    cout << "Escribe fila a fila" << endl;
    for (unsigned f = 0; f < NFILAS; ++f) {
        for (unsigned c = 0; c < NCOLUMNAS; ++c) {
            cin >> m[f][c];
        }
    }
}
// -- Principal -----
int main ()
{
    Matriz m;
    leer_matriz(m);
    escribir_matriz_formato(m);
}

```

Ejemplo 2

Diseñe un programa que realice el producto de 2 matrices de máximo 10×10 elementos:

```
#include <iostream>
#include <cassert>
using namespace std;
// -- Constantes -----
const unsigned MAX = 10;
// -- Tipos -----
typedef double Fila[MAX];
typedef Fila Tabla[MAX];
struct Matriz {
    unsigned n_fil;
    unsigned n_col;
    Tabla datos;
};
// -- Subalgoritmos ----
void leer_matriz (Matriz& m)
{
    cout << "Dimensiones?: ";
    cin >> m.n_fil >> m.n_col;
    assert(m.n_fil <= MAX && m.n_col <= MAX);
    cout << "Escribe valores fila a fila:" << endl;
    for (unsigned f = 0; f < m.n_fil; ++f) {
        for (unsigned c = 0; c < m.n_col; ++c) {
            cin >> m.datos[f][c];
        }
    }
}

void escribir_matriz (const Matriz& m)
{
    for (unsigned f = 0; f < m.n_fil; ++f) {
        for (unsigned c = 0; c < m.n_col; ++c) {
            cout << m.datos[f][c] << " ";
        }
        cout << endl;
    }
}

double suma_fila_por_col (const Matriz& x, const Matriz& y, unsigned f, unsigned c)
{
    assert(x.n_col == y.n_fil); // PRE-COND
    double suma = 0.0;
    for (unsigned k = 0; k < x.n_col; ++k) {
        suma += x.datos[f][k] * y.datos[k][c];
    }
    return suma;
}

void mult_matriz (Matriz& m, const Matriz& a, const Matriz& b)
{
    assert(a.n_col == b.n_fil); // PRE-COND
    m.n_fil = a.n_fil;
    m.n_col = b.n_col;
    for (unsigned f = 0; f < m.n_fil; ++f) {
        for (unsigned c = 0; c < m.n_col; ++c) {
            m.datos[f][c] = suma_fila_por_col(a, b, f, c);
        }
    }
}

// -- Principal -----
```

```

int main ()
{
    Matriz a,b,c;
    leer_matriz(a);
    leer_matriz(b);
    if (a.n_col != b.n_fil) {
        cout << "No se puede multiplicar." << endl;
    } else {
        mult_matriz(c, a, b);
        escribir_matriz(c);
    }
}

```

Paso de Parámetros de Arrays Predefinidos

Con respecto al *paso de agregados “predefinidos” como parámetros*, tanto el paso por valor como el paso por referencia actúan de igual forma: por referencia, por lo que no es posible pasarlos por valor. Por ello, los parámetros de entrada de tipo array predefinido se realizarán mediante *paso por referencia constante* y los parámetros de salida y de entrada/salida de tipo array predefinido se realizarán mediante *paso por referencia*. Por ejemplo:

```

const unsigned NELMS = 9;
typedef int Vector[NELMS];
void copiar(Vector& destino, const Vector& origen)
{
    for (unsigned i = 0; i < NELMS; ++i) {
        destino[i] = origen[i];
    }
}
int main()
{
    Vector v1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    Vector v2;
    copiar(v2, v1);
}

```

Hay que tener en cuenta que si se utiliza la sintaxis del paso por valor, se pasarán realmente por referencia, pero de este modo será más propenso a errores, ya que en este caso no se comprueban que los parámetros sean del mismo tipo, *siendo posible* pasar un array de un tamaño diferente al especificado, con los errores que ello conlleva. Por ejemplo:

```

const unsigned NELMS_1 = 9;
const unsigned NELMS_2 = 5;
typedef int Vector_1[NELMS_1];
typedef int Vector_2[NELMS_2];
void copiar(Vector_1 destino, const Vector_1 origen)
{
    for (unsigned i = 0; i < NELMS_1; ++i) {
        destino[i] = origen[i];
    }
}
int main()
{
    Vector_1 v1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    Vector_2 v2;
    copiar(v2, v1); // Error al pasar una variable de tipo Vector_2
}

```

Paso de Parámetros de Arrays Predefinidos de Tamaño Variable

Hay situaciones en las cuales es conveniente que un subprograma trabaje con agregados de un tamaño que dependa de la invocación del mismo. Para ello se declara el parámetro mediante el tipo base, el identificador del parámetro y el indicativo (`[]`) de que es un agregado sin tamaño especificado (dependerá de la invocación al subprograma).

Sólo es posible pasar agregados de una dimensión sin especificar su tamaño. Además, la información sobre el tamaño del agregado se pierde al pasarlo como agregado abierto, por lo que normalmente dicho tamaño se deberá también pasar como parámetro.

Además, el paso se realiza siempre por referencia sin necesidad de especificar el símbolo `&`, y para asegurar que no sea modificado en caso de información de entrada, se realizará el paso de parámetros constante. Ejemplo:

```
#include <iostream>
using namespace std;
const unsigned MAX = 20;
const unsigned NDATOS = 10;
typedef int Numeros[MAX];
typedef int Datos[NDATOS];
void imprimir(unsigned n, const int vct[])
{
    for (unsigned i = 0; i < n; ++i) {
        cout << vct[i] << " ";
    }
    cout << endl;
}
void asignar_valores(unsigned n, int vct[])
{
    for (unsigned i = 0; i < n; ++i) {
        vct[i] = i;
    }
}
int main()
{
    Numeros nm;
    Datos dt;
    asignar_valores(MAX, nm);
    imprimir(MAX, nm);
    asignar_valores(NDATOS, dt);
    imprimir(NDATOS, dt);
}
```

Nota: Sólo es válido declarar agregados abiertos como parámetros. No es posible declarar variables como agregados abiertos.

Comparación de Arrays Predefinidos Respecto al Tipo Array de la Biblioteca Estándar

El tratamiento de los arrays predefinidos en el lenguaje C no es consistente con el tratamiento de los otros tipos de datos, y presenta además algunas anomalías. Por ejemplo:

- No es posible la asignación directa de variables de tipo array predefinido, sin embargo, si el array predefinido se encuentra dentro de un registro, el registro sí se puede asignar.
- No existe el constructor de copia para el tipo array predefinido, a diferencia de todos los otros tipos. Este hecho tiene además consecuencias cuando se define el constructor de copia para `class`.
- El tipo array predefinido no puede ser el tipo de retorno de una función.
- Si se aplican los operadores relacionales (`==`, `!=`, `>`, `>=`, `<`, `<=`) a variables de tipo array predefinido, el comportamiento y los resultados producidos no son los esperados.

- El paso de parámetros de arrays predefinidos sin el símbolo ampersand (&) es propenso a errores no detectables en tiempo de compilación.
- En caso de paso de parámetros de arrays predefinidos sin el símbolo ampersand (&), dentro del subprograma se pueden utilizar los operadores de asignación (=) y relacionales, pero su comportamiento y los resultados producidos no son los esperados.
- Los arrays predefinidos tienen una conversión automática al tipo puntero, que es indeseable en muchos casos.

```
#include <iostream>
using namespace std;
const unsigned NELMS_1 = 9;
const unsigned NELMS_2 = 5;
typedef int Vector_1[NELMS_1];
typedef int Vector_2[NELMS_2];
void escribir(const Vector_1 v)
{
    for (unsigned i = 0; i < NELMS_1; ++i) {
        cout << v[i] << " ";
    }
    cout << endl;
}

void copiar(Vector_1 destino, Vector_1 origen)
{
    destino = origen; // Error: comportamiento inesperado
}

bool es_menor(const Vector_1 v1, const Vector_1 v2)
{
    return v1 < v2; // Error: comportamiento inesperado
}

int main()
{
    Vector_2 v2;
    Vector_1 vx;
    Vector_1 v1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    //-----
    // vx = v1; // Error de Compilación
    // v2 = v1; // Error de Compilación
    if (v1 < vx) {
        cout << "v1 < vx" << endl;
    }
    if (v1 < v2) {
        cout << "v1 < v2" << endl;
    }
    //-----
    copiar(vx, v1);
    copiar(v2, v1); // Error al pasar una variable de tipo Vector_2
    //-----
    if (es_menor(v1, vx)) {
        cout << "v1 es menor que vx" << endl;
    }
    if (es_menor(v1, v2)) {
        cout << "v1 es menor que v2" << endl;
    }
    //-----
    escribir(v1);
    escribir(vx);
    escribir(v2); // Error al pasar una variable de tipo Vector_2
}
```

Cuya ejecución produce el siguiente resultado:

```

v1 < vx
v1 < v2
v1 es menor que vx
v1 es menor que v2
1 2 3 4 5 6 7 8 9
134519591 1 65535 3221154088 134524397 3076926329 134530372 3221154104 134519004
3077695113 134530372 3221154136 134524297 3078529616 3221154144 3221154232 3075855445 134524272

```

Por otra parte, el tratamiento del tipo `array` de la biblioteca estándar es consistente con los otros tipos de datos. Es posible la asignación (=) de variables de dicho tipo, y los operadores relacionales tienen el comportamiento esperado, así como es posible su devolución desde una función, y el paso de parámetros, tanto por valor como por referencia y referencia constante, es seguro y mantiene su semántica previamente establecida. Además, es posible acceder al número de elementos que lo componen mediante el operador `size()`,¹ así como también es posible acceder a un determinado elemento con comprobación de rango (`at(i)`).

8.2. Cadenas de Caracteres al Estilo-C OBS

Las *cadenas de caracteres al estilo-C* se definen como un array predefinido de caracteres (tipo `char`) de un determinado tamaño especificado en tiempo de compilación, donde la cadena de caracteres en sí está formada por los caracteres comprendidos entre el principio del array y un carácter especial que marca el final de la cadena. Este carácter es el `'\0'`. Esto sucede tanto con cadenas de caracteres constantes como con las variables. Por ejemplo:

```

#include <iostream>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];
int main()
{
    Cadena nombre = "Pepe";
    Cadena apellidos;
}

```

nombre:	P	e	p	e	\0	?	?	?	?	?
	0	1	2	3	4	5	6	7	8	9

apellidos:	?	?	?	?	?	?	?	?	?	?
	0	1	2	3	4	5	6	7	8	9

No es posible asignar variables de tipo array predefinido, salvo en la inicialización con una cadena literal constante entre comillas dobles. Del mismo modo, **tampoco** se le pueden aplicar los operadores relacionales (`==`, `!=`, `>`, `>=`, `<`, `<=`) a entidades de tipo array predefinido.

Para acceder a un carácter concreto de la cadena de caracteres, especificaremos entre corchetes (`[` y `]`) el índice de la posición que ocupa el mismo, teniendo en cuenta que el primer elemento ocupa la posición 0 (cero) y el último elemento ocupa la posición del número de elementos menos 1. Por ejemplo `nombre[0]` y `nombre[MAX_CADENA-1]` aluden al primer y último elemento del agregado respectivamente. Un determinado elemento puede utilizarse en cualquier lugar donde sea válido una variable de su mismo tipo base.

Hay que tener en cuenta que la secuencia de caracteres se almacena en el array predefinido desde la primera posición, y consecutivamente hasta el carácter terminador (`'\0'`). Por lo tanto, siempre se debe reservar espacio en toda cadena de caracteres para contener este carácter terminador además de los caracteres que componen la secuencia.

El lenguaje de programación C no comprueba que los accesos a los elementos de un agregado son correctos y se encuentran dentro de los límites válidos del array predefinido, por lo que será responsabilidad del programador comprobar que así sea.

```

#include <iostream>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];

```

¹La utilización del operador `size()` en vez de la constante disminuye el acoplamiento en el programa respecto a un valor constante global, y hace que el programa sea más robusto.

```

int main()
{
    Cadena nombre = "Pepe";
    for (unsigned i = 0; nombre[i] != '\0'; ++i) {
        cout << nombre[i];
    }
    cout << endl;
}

```

Paso de Parámetros de Cadenas de Caracteres al Estilo-C

Las cadenas de caracteres al estilo-C se definen como arrays predefinidos de tipo base `char`, por lo que le son aplicables todo lo mencionado en dicha sección. Sin embargo, las cadenas de caracteres al estilo-C tienen algunas peculiaridades que hace que sea más adecuado pasarlas para parámetros como arrays abiertos, ya que puede haber diversos arrays con distintos tamaños que representen cadenas de caracteres, siendo su procesamiento equivalente, e independiente de dicho tamaño. Así mismo, las cadenas de caracteres literales constantes tienen el tamaño dependiente del número de caracteres que la componen. Es importante que en el caso del paso de cadenas de caracteres como arrays abiertos, en caso de que la cadena sea modificable, es conveniente pasar también el tamaño máximo que la cadena puede alcanzar, para evitar acceder más allá de sus límites. Sin embargo, si el parámetro es un array abierto constante entonces no es necesario pasar el tamaño del array, ya que el final de la cadena viene marcado por el carácter terminador.

```

#include <iostream>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];
void escribir(const char cad[])
{
    for (unsigned i = 0; cad[i] != '\0'; ++i) {
        cout << cad[i] ;
    }
}
unsigned longitud(const char cad[])
{
    unsigned i = 0;
    while (cad[i] != '\0') {
        ++i;
    }
    return i;
}
void copiar(char destino[], const char origen[], unsigned sz)
{
    unsigned i;
    for (i = 0; (i < sz-1)&&(origen[i] != '\0'); ++i) {
        destino[i] = origen[i];
    }
    destino[i] = '\0';
}
int main()
{
    Cadena c1 = "Pepe";
    Cadena c2;
    unsigned l = longitud(c1);
    copiar(c2, c1, MAX_CADENA);
    escribir(c2);
    copiar(c2, "Luis", MAX_CADENA);
    escribir(c2);
}

```

Nota: también es posible realizar el paso de parámetros de cadenas de caracteres como `char*` y `const char*` con la misma semántica que la especificada anteriormente, sin embargo la sintaxis con los corchetes (`[]`) representa mejor el significado de este paso de parámetros en estas circunstancias, por lo que se **desaconseja** la siguiente sintaxis para el paso de cadenas de caracteres como parámetros:

```
#include <iostream>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];
void escribir(const char* cad)
{
    for (unsigned i = 0; cad[i] != '\0'; ++i) {
        cout << cad[i] ;
    }
}
unsigned longitud(const char* cad)
{
    unsigned i = 0;
    while (cad[i] != '\0') {
        ++i;
    }
    return i;
}
void copiar(char* destino, const char* origen, unsigned sz)
{
    unsigned i;
    for (i = 0; (i < sz-1)&&(origen[i] != '\0'); ++i) {
        destino[i] = origen[i];
    }
    destino[i] = '\0';
}
int main()
{
    Cadena c1 = "Pepe";
    Cadena c2;
    unsigned l = longitud(c1);
    copiar(c2, c1, MAX_CADENA);
    escribir(c2);
    copiar(c2, "Luis", MAX_CADENA);
    escribir(c2);
}
```

Entrada y Salida de Cadenas de Caracteres

El operador `<<` aplicado a un flujo de salida (`cout` para el flujo de salida estándar, usualmente el terminal) permite mostrar el contenido de las cadenas de caracteres, tanto constantes como variables. Por ejemplo:

```
#include <iostream>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];
const Cadena AUTOR = "José Luis";
int main()
{
    Cadena nombre = "Pepe";
    cout << "Nombre: " << nombre << " " << AUTOR << endl;
}
```

El operador `>>` aplicado a un flujo de entrada (`cin` para el flujo de entrada estándar, usualmente el teclado) permite leer secuencias de caracteres y almacenarlas en variables del tipo cadena de caracteres. Sin embargo, para evitar sobrepasar el límite del array predefinido durante la lectura, es conveniente utilizar el manipulador `setw`. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];
int main()
{
    cout << "Introduzca el nombre: ";
    Cadena nombre;
    cin >> setw(MAX_CADENA) >> nombre;
    cout << "Nombre: " << nombre << endl;
}
```

Este operador de entrada (`>>`) se comporta (como se especificó en el capítulo 3.2 dedicado a la Entrada y Salida básica) de la siguiente forma: elimina los espacios en blanco que hubiera al principio de la entrada de datos, y lee dicha entrada hasta que lea tantos caracteres como se especifica en `setw`, o hasta que encuentre algún carácter de espacio en blanco, que no será leído y permanecerá en el buffer de entrada (véase 3.3) hasta la próxima operación de entrada. En caso de que durante la entrada surja alguna situación de error, dicha entrada se detiene y el flujo de entrada se pondrá en un estado erróneo. Se consideran espacios en blanco los siguientes caracteres: espacio en blanco, tabuladores, retorno de carro y nueva línea (' ', '\t', '\v', '\f', '\r', '\n').

También es posible leer una línea completa, hasta leer el fin de línea, desde el flujo de entrada, sin eliminar los espacios iniciales:

```
#include <iostream>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];
int main()
{
    cout << "Introduzca el nombre: ";
    Cadena nombre;
    cin.getline(nombre, MAX_CADENA);
    cout << "Nombre: " << nombre << endl;
}
```

También es posible leer una línea completa, hasta leer un delimitador especificado, desde el flujo de entrada, sin eliminar los espacios iniciales:

```
#include <iostream>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];
const char DELIMITADOR = '.';
int main()
{
    cout << "Introduzca el nombre: ";
    Cadena nombre;
    cin.getline(nombre, MAX_CADENA, DELIMITADOR);
    cout << "Nombre: " << nombre << endl;
}
```

Nótese que realizar una operación `getline` después de una operación con `>>` puede tener complicaciones, ya que `>>` dejara los espacios en blanco (y fin de línea) en el buffer, que serán leídos por `getline`. Por ejemplo:

```

#include <iostream>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];
int main()
{
    cout << "Introduzca número: ";
    int n;
    cin >> n;
    cout << "Introduzca el nombre: ";
    Cadena nombre;
    cin.getline(nombre, MAX_CADENA);
    cout << "Número: " << n << " Nombre: " << nombre << endl;
}

```

Para evitar este problema, el siguiente subprograma leerá una cadena que sea distinta de la vacía:

```

#include <iostream>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];
inline void leer_linea_no_vacia(istream& ent, char linea[], unsigned sz)
{
    ent >> ws;          // salta los espacios en blanco y fin de línea
    ent.getline(linea, sz); // leerá la primera línea no vacía
}
int main()
{
    cout << "Introduzca número: ";
    int n;
    cin >> n;
    cout << "Introduzca el nombre (NO puede ser vacío): ";
    Cadena nombre;
    leer_linea_no_vacia(cin, nombre, MAX_CADENA);
    cout << "Número: " << n << " Nombre: " << nombre << endl;
}

```

Por el contrario, en caso de que la cadena vacía sea una entrada válida, será necesario eliminar el resto de caracteres (incluyendo los espacios en blanco y fin de línea) del buffer de entrada, después de leer un dato con `>>`, de tal forma que el buffer esté limpio antes de realizar la entrada de la cadena de caracteres con `getline`. Por ejemplo, el subprograma `leer_int` elimina los caracteres del buffer después de leer un dato de tipo `int`:

```

#include <iostream>
#include <limits>
using namespace std;
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];
inline void leer_int(istream& ent, int& dato)
{
    ent >> dato;          // lee el dato
    ent.ignore(numeric_limits<streamsize>::max(), '\n'); // elimina los caracteres del buffer
    // ent.ignore(10000, '\n'); // otra posibilidad de eliminar los caracteres del buffer
}
int main()
{
    cout << "Introduzca número: ";
    int n;
    leer_int(cin, n);
    cout << "Introduzca el nombre (puede ser vacío): ";
}

```

```

Cadena nombre;
cin.getline(nombre, MAX_CADENA);
cout << "Número: " << n << " Nombre: " << nombre << endl;
}

```

Téngase en cuenta que para utilizar `numeric_limits<...>`, es necesario incluir la biblioteca estándar `<limits>`.

Operaciones con Cadenas de Caracteres al Estilo-C

La biblioteca `cstring` proporciona una serie de subprogramas adecuados para el tratamiento más común de las cadenas de caracteres al estilo-C. Para utilizarla se debe incluir dicha biblioteca, así como utilizar el espacio de nombres `std`.

```

#include <cstring>
using namespace std;

unsigned strlen(const char s1[]);
    // devuelve la longitud de la cadena s1

char* strcpy(char dest[], const char orig[]);
char* strncpy(char dest[], const char orig[], unsigned n);
    // Copia la cadena orig a dest (incluyendo el terminador '\0').
    // Si aparece 'n', hasta como máximo 'n' caracteres
    // (si alcanza el límite, no incluye el terminador '\0')

char* strcat(char dest[], const char orig[]);
char* strncat(char dest[], const char orig[], unsigned n);
    // Concatena la cadena orig a la cadena dest.
    // Si aparece 'n', hasta como máximo 'n' caracteres
    // (si alcanza el límite, no incluye el terminador '\0')

int strcmp(const char s1[], const char s2[]);
int strncmp(const char s1[], const char s2[], unsigned n);
    // Compara lexicográficamente las cadenas s1 y s2.
    // Si aparece 'n', hasta como máximo 'n' caracteres
    // devuelve <0 si s1<s2, ==0 si s1==s2, >0 si s1>s2

const char* strchr(const char s1[], char ch);
    // devuelve un puntero a la primera ocurrencia de ch en s1
    // NULL si no se encuentra
const char* strrchr(const char s1[], char ch);
    // devuelve un puntero a la última ocurrencia de ch en s1
    // NULL si no se encuentra

unsigned strspn(const char s1[], const char s2[]);
    // devuelve la longitud del prefijo de s1 de caracteres
    // que se encuentran en s2
unsigned strcspn(const char s1[], const char s2[]);
    // devuelve la longitud del prefijo de s1 de caracteres
    // que NO se encuentran en s2

const char* strpbrk(const char s1[], const char s2[]);
    // devuelve un puntero a la primera ocurrencia en s1
    // de cualquier carácter de s2. NULL si no se encuentra

const char* strstr(const char s1[], const char s2[]);
    // devuelve un puntero a la primera ocurrencia en s1
    // de la cadena s2. NULL si no se encuentra

```

Comparación del Tipo String y de Cadenas de Caracteres al Estilo-C

El tratamiento de las cadenas de caracteres al estilo-C no es consistente con el tratamiento de los otros tipos de datos, y presenta además algunas anomalías, algunas debidas a la utilización de arrays predefinidos como soporte para su implementación. Por ejemplo:

- No es posible la asignación directa de variables, sin embargo, si la cadena se encuentra dentro de un registro, el registro si se puede asignar. Así mismo, también es posible asignar una cadena de caracteres constante literal (entre comillas dobles).
- El tipo array predefinido no puede ser el tipo de retorno de una función, se debe retornar un puntero a carácter.
- Si se aplican los operadores relacionales (==, !=, >, >=, <, <=) a cadenas de caracteres, el comportamiento y los resultados producidos no son los esperados.
- El paso de parámetros debe realizarse mediante arrays abiertos. En este caso, dentro del subprograma se pueden utilizar los operadores de asignación (=) y relacionales, pero su comportamiento y los resultados producidos no son los esperados. Así mismo, en numerosas ocasiones se debe también pasar como parámetro el tamaño del array, para evitar desbordamientos.
- Las cadenas de caracteres tienen una conversión automática al tipo puntero a carácter, que es indeseable en muchos casos.

Las cadenas de caracteres también presentan otros problemas adicionales con respecto al tipo `string` de la biblioteca estándar. Por ejemplo:

- Es necesario especificar *a priori* un tamaño límite para la cadena de caracteres, siendo necesario definir múltiples tipos con diferentes tamaños para representar conceptos diferentes. Al definir los tamaños, hay que considerar el carácter terminador (`'\0'`).
- El procesamiento de las cadenas de caracteres es más complicado, ya que requiere un procesamiento secuencial hasta encontrar el terminador.
- El procesamiento de las cadenas de caracteres es propenso a errores de *desbordamiento*, especialmente al añadir caracteres y concatenar cadenas.
- La entrada de datos simple (`cin >> cadena;`) es susceptible a errores de desbordamiento. Es necesario especificar el límite de la lectura (`cin >> setw(MAX) >> cadena;`).

Por otra parte, el tratamiento del tipo `string` de la biblioteca estándar es consistente con los otros tipos de datos. No requiere la especificación *a priori* de un límite en tiempo de compilación, es posible la asignación (=) de variables de dicho tipo, y los operadores relacionales tienen el comportamiento esperado, así como es posible su devolución desde una función, y el paso de parámetros, tanto por valor como por referencia y referencia constante, es seguro y mantiene su semántica previamente establecida. Además, es posible acceder al número de elementos que lo componen mediante el operador `size()`, así como también es posible acceder a un determinado elemento con comprobación de rango (`at(i)`). Es robusto ante errores de desbordamiento y posee un amplio conjunto de operaciones útiles predefinidas.

8.3. Uniones

Otra construcción útil, aunque no muy utilizada, son las *uniones*, y sirven para compactar varias entidades de diferentes tipos en la misma zona de memoria. Es decir, todas las entidades definidas dentro de una unión compartirán la misma zona de memoria, y por lo tanto su utilización será excluyente. Se utilizan dentro de las estructuras:

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

enum Tipo {
    COCHE,
    MOTOCICLETA,
    BICICLETA
}
```



```

};
struct Vehiculo {
    Tipo vh;
    union {
        double capacidad; // para el caso de tipo COCHE
        int cilindrada; // para el caso de tipo MOTOCICLETA
        int talla; // para el caso de tipo BICICLETA
    };
};
int main()
{
    Vehiculo xx;
    Vehiculo yy;

    xx.vh = COCHE;
    xx.capacidad = 1340.25;
    yy.vh = MOTOCICLETA;
    yy.cilindrada = 600;
}

```

obviamente, los tipos de los campos de la unión pueden ser tanto simples como compuestos. Es responsabilidad del programador utilizar los campos adecuados en función del tipo que se esté almacenando.

8.4. Campos de Bits

Permiten empaquetar secuencias de bits dentro de un determinado tipo, y acceder a ellas mediante su identificador.

```

struct PPN {
    unsigned PFN : 22; // 22 bits sin signo
    unsigned : 3; // 3 bits sin signo [unused]
    unsigned CCA : 3; // 3 bits sin signo
    bool dirty : 1; // 1 bit booleano
};

```

8.5. Resolución de Problemas Utilizando Tipos Compuestos

Diseña un programa para gestionar una agenda personal que contenga la siguiente información: Nombre, Teléfono, Dirección, Calle, Número, Piso, Código Postal y Ciudad, y las siguientes operaciones:

- Añadir los datos de una persona.
- Acceder a los datos de una persona a partir de su nombre.
- Borrar una persona a partir de su nombre.
- Modificar los datos de una persona a partir de su nombre.
- Listar el contenido completo de la agenda.

```

#include <iostream>
#include <iomanip>
#include <cstring>
#include <cassert>
using namespace std;
// -- Constantes -----
const unsigned MAX_CADENA = 60;
const unsigned MAX_PERSONAS = 50;
// -- Tipos -----

```

```

typedef char Cadena[MAX_CADENA];
struct Direccion {
    unsigned num;
    Cadena calle;
    Cadena piso;
    Cadena cp;
    Cadena ciudad;
};
struct Persona {
    Cadena nombre;
    Cadena tel;
    Direccion direccion;
};
// -- Tipos -----
typedef Persona Personas[MAX_PERSONAS];
struct Agenda {
    unsigned n_pers;
    Personas pers;
};
enum Cod_Error {
    OK, AG_LLENA, NO_ENCONTRADO, YA_EXISTE
};
// -- Subalgoritmos ----
void Inicializar (Agenda& ag)
{
    ag.n_pers = 0;
}
//-----
void Leer_Direccion (Direccion& dir)
{
    cin >> setw(MAX_CADENA) >> dir.calle;
    cin >> dir.num;
    cin >> setw(MAX_CADENA) >> dir.piso;
    cin >> setw(MAX_CADENA) >> dir.cp;
    cin >> setw(MAX_CADENA) >> dir.ciudad;
}
//-----
void Escribir_Direccion (const Direccion& dir)
{
    cout << dir.calle << " ";
    cout << dir.num << " ";
    cout << dir.piso << " ";
    cout << dir.cp << " ";
    cout << dir.ciudad << " ";
}
//-----
void Leer_Persona (Persona& per)
{
    cin >> setw(MAX_CADENA) >> per.nombre;
    cin >> setw(MAX_CADENA) >> per.tel;
    Leer_Direccion(per.direccion);
}
//-----
void Escribir_Persona (const Persona& per)
{
    cout << per.nombre << " ";
    cout << per.tel << " ";
    Escribir_Direccion(per.direccion);
    cout << endl;
}

```

```

}
//-----
// Busca una Persona en la Agenda
// Devuelve su posición si se encuentra, o bien >= ag.n_pers en otro caso
unsigned Buscar_Persona (const char nombre[], const Agenda& ag)
{
    unsigned i = 0;
    while ((i < ag.n_pers) && (strcmp(nombre, ag.pers[i].nombre) != 0)) {
        ++i;
    }
    return i;
}
//-----
void Anyadir (Agenda& ag, const Persona& per)
{
    assert(ag.n_pers < MAX_PERSONAS);
    ag.pers[ag.n_pers] = per;
    ++ag.n_pers;
}
//-----
void Eliminar (Agenda& ag, unsigned pos)
{
    assert(pos < ag.n_pers);
    if (pos < ag.n_pers-1) {
        ag.pers[pos] = ag.pers[ag.n_pers - 1];
    }
    --ag.n_pers;
}
//-----
void Anyadir_Persona (const Persona& per, Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(per.nombre, ag);
    if (i < ag.n_pers) {
        ok = YA_EXISTE;
    } else if (ag.n_pers == MAX_PERSONAS) {
        ok = AG_LLENA;
    } else {
        ok = OK;
        Anyadir(ag, per);
    }
}
//-----
void Borrar_Persona (const char nombre[], Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {
        ok = OK;
        Eliminar(ag, i);
    }
}
//-----
void Modificar_Persona (const char nombre[], const Persona& nuevo, Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {

```

```

        Eliminar(ag, i);
        Anyadir_Persona(nuevo, ag, ok);
    }
}
//-----
void Imprimir_Persona (const char nombre[], const Agenda& ag, Cod_Error& ok)
{
    unsigned i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {
        ok = OK;
        Escribir_Persona(ag.pers[i]);
    }
}
//-----
void Imprimir_Agenda (const Agenda& ag, Cod_Error& ok)
{
    for (unsigned i = 0; i < ag.n_pers; ++i) {
        Escribir_Persona(ag.pers[i]);
    }
    ok = OK;
}
//-----
char Menu ()
{
    char opcion;
    cout << endl;
    cout << "a. - Añadir Persona" << endl;
    cout << "b. - Buscar Persona" << endl;
    cout << "c. - Borrar Persona" << endl;
    cout << "d. - Modificar Persona" << endl;
    cout << "e. - Imprimir Agenda" << endl;
    cout << "x. - Salir" << endl;
    do {
        cout << "Introduzca Opción: ";
        cin >> opcion;
    } while ( ! ((opcion >= 'a') && (opcion <= 'e')) || (opcion == 'x')));
    return opcion;
}
//-----
void Escribir_Cod_Error (Cod_Error cod)
{
    switch (cod) {
        case OK:
            cout << "Operación correcta" << endl;
            break;
        case AG_LLENA:
            cout << "Agenda llena" << endl;
            break;
        case NO_ENCONTRADO:
            cout << "La persona no se encuentra en la agenda" << endl;
            break;
        case YA_EXISTE:
            cout << "La persona ya se encuentra en la agenda" << endl;
            break;
    }
}
// -- Principal -----

```

```
int main ()
{
    Agenda ag;
    char opcion;
    Persona per;
    Cadena nombre;
    Cod_Error ok;
    Inicializar(ag);
    do {
        opcion = Menu();
        switch (opcion) {
            case 'a':
                cout << "Introduzca los datos de la Persona" << endl;
                cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl;
                Leer_Persona(per);
                Anyadir_Persona(per, ag, ok);
                Escribir_Cod_Error(ok);
                break;
            case 'b':
                cout << "Introduzca Nombre" << endl;
                cin >> setw(MAX_CADENA) >> nombre;
                Imprimir_Persona(nombre, ag, ok);
                Escribir_Cod_Error(ok);
                break;
            case 'c':
                cout << "Introduzca Nombre" << endl;
                cin >> setw(MAX_CADENA) >> nombre;
                Borrar_Persona(nombre, ag, ok);
                Escribir_Cod_Error(ok);
                break;
            case 'd':
                cout << "Introduzca Nombre" << endl;
                cin >> setw(MAX_CADENA) >> nombre;
                cout << "Nuevos datos de la Persona" << endl;
                cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl;
                Leer_Persona(per);
                Modificar_Persona(nombre, per, ag, ok);
                Escribir_Cod_Error(ok);
                break;
            case 'e':
                Imprimir_Agenda(ag, ok);
                Escribir_Cod_Error(ok);
                break;
        }
    } while (opcion != 'x' );
}
```


Capítulo 9

Algunas Bibliotecas Útiles

En este capítulo se muestra superficialmente algunas funciones básicas de la biblioteca estándar.

cmath

La biblioteca `<cmath>` proporciona principalmente algunas funciones matemáticas útiles:

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

<code>double sin(double r);</code>	seno, $\sin r$ (en radianes)
<code>double cos(double r);</code>	coseno, $\cos r$ (en radianes)
<code>double tan(double r);</code>	tangente, $\tan r$ (en radianes)
<code>double asin(double x);</code>	arco seno, $\arcsin x, x \in [-1, 1]$
<code>double acos(double x);</code>	arco coseno, $\arccos x, x \in [-1, 1]$
<code>double atan(double x);</code>	arco tangente, $\arctan x$
<code>double atan2(double y, double x);</code>	arco tangente, $\arctan y/x$
<code>double sinh(double r);</code>	seno hiperbólico, $\sinh r$
<code>double cosh(double r);</code>	coseno hiperbólico, $\cosh r$
<code>double tanh(double r);</code>	tangente hiperbólica, $\tanh r$
<code>double sqrt(double x);</code>	$\sqrt{x}, x \geq 0$
<code>double pow(double x, double y);</code>	x^y
<code>double exp(double x);</code>	e^x
<code>double log(double x);</code>	logaritmo neperiano, $\ln x, x > 0$
<code>double log10(double x);</code>	logaritmo decimal, $\log x, x > 0$
<code>double ceil(double x);</code>	menor entero $\geq x, \lceil x \rceil$
<code>double floor(double x);</code>	mayor entero $\leq x, \lfloor x \rfloor$
<code>double fabs(double x);</code>	valor absoluto de $x, x $
<code>double ldexp(double x, int n);</code>	$x2^n$
<code>double frexp(double x, int* exp);</code>	inversa de <code>ldexp</code>
<code>double modf(double x, double* ip);</code>	parte entera y fraccionaria
<code>double fmod(double x, double y);</code>	resto de x/y

cctype

La biblioteca `<cctype>` proporciona principalmente características sobre los valores de tipo `char`:

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

<code>bool isalnum(char ch);</code>	<code>(isalpha(ch) isdigit(ch))</code>
<code>bool isalpha(char ch);</code>	<code>(isupper(ch) islower(ch))</code>
<code>bool iscntrl(char ch);</code>	caracteres de control
<code>bool isdigit(char ch);</code>	dígito decimal
<code>bool isgraph(char ch);</code>	caracteres imprimibles excepto espacio
<code>bool islower(char ch);</code>	letra minúscula
<code>bool isprint(char ch);</code>	caracteres imprimibles incluyendo espacio
<code>bool ispunct(char ch);</code>	carac. impr. excepto espacio, letra o dígito
<code>bool isspace(char ch);</code>	espacio, <code>'\r'</code> , <code>'\n'</code> , <code>'\t'</code> , <code>'\v'</code> , <code>'\f'</code>
<code>bool isupper(char ch);</code>	letra mayúscula
<code>bool isxdigit(char ch);</code>	dígito hexadecimal
<code>char tolower(char ch);</code>	retorna la letra minúscula correspondiente a <code>ch</code>
<code>char toupper(char ch);</code>	retorna la letra mayúscula correspondiente a <code>ch</code>

cstdlib

La biblioteca `<cstdlib>` proporciona principalmente algunas funciones generales útiles:

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

<code>int system(const char orden[]);</code>	orden a ejecutar por el sistema operativo
<code>int abs(int n);</code>	retorna el valor absoluto del número <code>int n</code>
<code>long labs(long n);</code>	retorna el valor absoluto del número <code>long n</code>
<code>void srand(unsigned semilla);</code>	inicializa el generador de números aleatorios
<code>int rand();</code>	retorna un aleatorio entre 0 y <code>RAND_MAX</code> (ambos inclusive)

```
#include <cstdlib>
#include <ctime>
using namespace std;
// -----
// inicializa el generador de números aleatorios
inline unsigned ini_aleatorio()
{
    srand(time(0));
}
// -----
// Devuelve un número aleatorio entre 0 y max (exclusive)
inline unsigned aleatorio(unsigned max)
{
    return unsigned(max*double(rand())/(RAND_MAX+1.0));
}
// -----
// Devuelve un número aleatorio entre min y max (ambos inclusive)
inline unsigned aleatorio(unsigned min, unsigned max)
{
    return min + aleatorio(max-min+1);
}
// -----
```


Parte II

Programación Intermedia

Capítulo 10

Almacenamiento en Memoria Secundaria: Ficheros

Los programas de ordenador usualmente trabajan con datos almacenados en la *memoria principal* (RAM). Esta memoria principal tiene como principales características que tiene un tiempo de acceso (para lectura y escritura) muy eficiente, sin embargo este tipo de memoria es *volátil*, en el sentido de que los datos almacenados en ella desaparecen cuando termina la ejecución del programa o se apaga el ordenador. Los ordenadores normalmente almacenan su información de manera *permanente* en dispositivos de almacenamiento de *memoria secundaria*, tales como dispositivos magnéticos (discos duros, cintas), discos ópticos (CDROM, DVD), memorias permanentes de estado sólido (memorias flash USB), etc.

Estos dispositivos suelen disponer de gran capacidad de almacenamiento, por lo que es necesario alguna organización que permita gestionar y acceder a la información allí almacenada. A esta organización se la denomina el *sistema de ficheros*, y suele estar organizado jerárquicamente en directorios (a veces denominados también carpetas) y ficheros (a veces denominados también archivos), donde los directorios permiten organizar jerárquicamente y acceder a los ficheros, y estos últimos almacenan de forma permanente la información, que puede ser tanto programas (software) como datos que serán utilizados por los programas. Así, los programas acceden y almacenan la información de manera permanente por medio de los ficheros, que son gestionados por el Sistema Operativo dentro de la jerarquía del sistema de ficheros.

Tipos de Ficheros

Los ficheros se pueden clasificar de múltiples formas dependiendo de los criterios seleccionados. En nuestro caso, nos centraremos en la clasificación por la codificación o formato en el que almacenan la información que contienen. Así, podemos distinguir los *ficheros de texto* y los *ficheros binarios*. En los ficheros de texto, la información se almacena utilizando una codificación y formato adecuados para que puedan ser procesados (y leídos), además de por un programa de ordenador, por un *ser humano*. Por lo tanto, los ficheros de texto almacenan la información utilizando una codificación textual como *secuencia de caracteres* (usualmente basada en la codificación ASCII, UTF-8, etc), y en un formato que permita su legibilidad y procesamiento. Por otra parte, los ficheros binarios son procesados automáticamente por la ejecución de programas, sin la intervención humana, por lo que no necesitan representar la información en un formato legible para el ser humano. Por ello, suelen codificar la información en un formato orientado a ser procesado eficientemente por los ordenadores, y en ese caso utilizan la representación en el *código binario* que utilizan internamente los ordenadores para representar los datos. En este caso, pueden surgir problemas de compatibilidad en aquellos casos en los que estos ficheros son procesados por programas ejecutándose en ordenadores que utilizan distintas representaciones internas de los datos.

Así por ejemplo, un fichero de texto denominado `fechas.txt` podría estar almacenado en una determinada posición en la jerarquía del sistema de ficheros (`/home/alumno/documentos/fechas.txt`)

1. *Incluir* la biblioteca `<fstream>` donde se definen los tipos correspondientes, y utilizar el espacio de nombres `std`.
2. *Declarar* variables del tipo de flujo adecuado (para entrada, salida, o ambos) para que actúen como manejadores de fichero.
3. *Abrir* el flujo de datos vinculando la variable correspondiente con el fichero especificado. Esta operación establece un vínculo entre la variable (manejador de fichero) definida en nuestro programa con el fichero gestionado por el sistema operativo, de tal forma que toda transferencia de información que el programa realice con el fichero, se realizará a través de la variable manejador de fichero vinculada con el mismo.
4. *Comprobar* que la apertura del fichero del paso previo se realizó correctamente. Si la vinculación con el fichero especificado no pudo realizarse por algún motivo (por ejemplo, el fichero no existe, en el caso de entrada de datos, o no es posible crear el fichero, en el caso de salida de datos), entonces la operación de apertura fallaría (véase 3.5).
5. *Realizar* la transferencia de información (de entrada o de salida) con el fichero a través de la variable de flujo vinculada al fichero. Para esta transferencia de información (entrada y salida) se pueden utilizar los mecanismos vistos en los capítulos anteriores (3, 5.10, 6.2, 8.2). En el caso de salida de datos, éstos deberán escribirse siguiendo un formato adecuado que permita su posterior lectura, por ejemplo escribiendo los separadores adecuados para ello entre los diferentes valores almacenados.
Normalmente, tanto la entrada como la salida de datos implican un proceso *iterativo*, que en el caso de entrada se suele realizar hasta leer y procesar todo el contenido del fichero.
6. *Comprobar* que el procesamiento del fichero del paso previo se realizó correctamente, de tal forma que si el procesamiento consistía en entrada de datos, el estado de la variable vinculada al fichero se encuentre en un estado indicando que se ha alcanzado el final del fichero, y si el procesamiento era de salida, el estado de la variable vinculada al fichero se deberá encontrar en un estado correcto (véase 3.5).
7. Finalmente *cerrar* el flujo liberando la variable de su vinculación con el fichero. Si no se cierra el flujo de fichero, cuando termine el ámbito de vida de la variable vinculada, el flujo será cerrado automáticamente, y su vinculación liberada.

Cuando sea necesario, una variable de tipo flujo, tanto de entrada, salida o ambos, puede ser pasada como *parámetro por referencia* (**no** constante) a cualquier subprograma.

Nota: si se va a abrir un fichero utilizando la misma variable que ya ha sido previamente utilizada para procesar otro fichero (pero posteriormente desvinculada tras cerrar el flujo), se deberá utilizar el método `clear()` para limpiar los flags de estado (véase 3.5). Aunque esta práctica de utilizar la misma variable para procesar distintos ficheros está totalmente desaconsejada, siendo una técnica mejor definir y vincular variables diferentes, en distintos subprogramas que representen adecuadamente la abstracción del programa.

10.2. Entrada de Datos desde Ficheros de Texto

Para realizar la entrada de datos desde un fichero de texto, el programa debe:

1. Incluir la biblioteca `<fstream>` donde se definen los tipos correspondientes, y utilizar el espacio de nombres `std`.

```
#include <fstream>
using namespace std;
```
2. Definir una variable manejador de fichero del tipo de flujo de entrada (`ifstream` –*input file stream*).

```
ifstream f_ent;
```
3. Abrir el flujo vinculando la variable correspondiente con el fichero especificado.

```
f_ent.open(nombre_fichero.c_str());
```
4. Comprobar que la apertura del fichero se realizó correctamente.

```
if (f_ent.fail()) { ... }
```

5. Realizar la entrada de datos con los operadores y subprogramas correspondientes, así como procesar la información leída.

```
f_ent >> nombre >> apellidos >> dia >> mes >> anyo;
f_ent.ignore(1000, '\n');
f_ent >> ws;
getline(f_ent, linea);
f_ent.get(c);
```

Usualmente es un proceso iterativo que se realiza hasta que la operación de entrada de datos falla, usualmente debido a haber alcanzado el final del fichero. Este proceso iterativo usualmente consiste en la iteración del siguiente proceso:

- Lectura de datos
- Si la lectura no ha sido correcta, entonces terminar el proceso iterativo.
- En otro caso, procesamiento de los datos leídos, y vuelta al proceso iterativo, leyendo nuevos datos

```
{
    ...
    f_ent >> datos;
    while (! f_ent.fail() ... ) {
        procesar(datos, ...);
        f_ent >> datos;
    }
}
```

6. Comprobar que el procesamiento del fichero se realizó correctamente, es decir, el fichero se leyó completamente hasta el final de mismo (`eof` representa *end-of-file*).

```
if (f_ent.eof()) { ... }
```

7. Finalmente cerrar el flujo liberando la variable de su vinculación.

```
f_ent.close();
```

Por ejemplo, un programa que lee números desde un fichero de texto y los procesa (en este caso simplemente los muestra por pantalla):

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
enum Codigo {
    OK, ERROR_APERTURA, ERROR_FORMATO
};
void procesar(int num)
{
    cout << num << endl;
}
void leer_fich(const string& nombre_fichero, Codigo& ok)
{
    ifstream f_ent;
    f_ent.open(nombre_fichero.c_str());
    if (f_ent.fail()) {
        ok = ERROR_APERTURA;
    } else {
        int numero;
        f_ent >> numero;
        while (! f_ent.fail()) {
            procesar(numero);
            f_ent >> numero;
        }
    }
}
```

```

    }
    if (f_ent.eof()) {
        ok = OK;
    } else {
        ok = ERROR_FORMATO;
    }
    f_ent.close();
}
}
void codigo_error(Codigo ok)
{
    switch (ok) {
    case OK:
        cout << "Fichero procesado correctamente" << endl;
        break;
    case ERROR_APERTURA:
        cout << "Error en la apertura del fichero" << endl;
        break;
    case ERROR_FORMATO:
        cout << "Error de formato en la lectura del fichero" << endl;
        break;
    }
}
int main()
{
    Codigo ok;
    string nombre_fichero;
    cout << "Introduzca el nombre del fichero: ";
    cin >> nombre_fichero;
    leer_fich(nombre_fichero, ok);
    codigo_error(ok);
}

```

10.3. Salida de Datos a Ficheros de Texto

Para realizar la salida de datos a un fichero de texto, el programa debe:

1. Incluir la biblioteca `<fstream>` donde se definen los tipos correspondientes, y utilizar el espacio de nombres `std`.

```

#include <fstream>
using namespace std;

```

2. Definir una variable manejador de fichero del tipo de flujo de salida (`ofstream` –*output file stream*).

```

ofstream f_sal;

```

3. Abrir el flujo vinculando la variable correspondiente con el fichero especificado.

```

f_sal.open(nombre_fichero.c_str());

```

4. Comprobar que la apertura del fichero se realizó correctamente.

```

if (f_sal.fail()) { ... }

```

5. Realizar la salida de datos con los operadores y subprogramas correspondientes, teniendo en cuenta los separadores que se deben escribir para que puedan ser leídos adecuadamente.

```

f_sal << nombre << " " << apellidos << " " << dia << " " << mes << " " << anyo << endl;

```

Usualmente éste es un proceso iterativo que se realiza hasta que se escriben en el fichero todos los datos apropiados y mientras el estado del flujo sea correcto.

```

while ( ... f_sal.good() ) { ... }

```

6. Comprobar que el procesamiento del fichero se realizó correctamente, es decir, el fichero se encuentra en buen estado.

```
if (f_sal.good()) { ... }
```

7. Finalmente cerrar el flujo liberando la variable de su vinculación.

```
f_sal.close();
```

Por ejemplo, un programa que lee números de teclado (hasta introducir un cero) y los escribe a un fichero de texto:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
enumCodigo {
    OK, ERROR_APERTURA, ERROR_FORMATO
};
void escribir_fich(const string& nombre_fichero, Codigo& ok)
{
    ofstream f_sal;
    f_sal.open(nombre_fichero.c_str());
    if (f_sal.fail()) {
        ok = ERROR_APERTURA;
    } else {
        int numero;
        cin >> numero
        while ((numero > 0) && ! cin.fail() && f_sal.good()) {
            f_sal << numero << endl;
            cin >> numero
        }
        if (f_sal.good()) {
            ok = OK;
        } else {
            ok = ERROR_FORMATO;
        }
        f_sal.close();
    }
}
void codigo_error(Codigo ok)
{
    switch (ok) {
        case OK:
            cout << "Fichero guardado correctamente" << endl;
            break;
        case ERROR_APERTURA:
            cout << "Error en la apertura del fichero" << endl;
            break;
        case ERROR_FORMATO:
            cout << "Error de formato al escribir al fichero" << endl;
            break;
    }
}
int main()
{
    Codigo ok;
    string nombre_fichero;
    cout << "Introduzca el nombre del fichero: ";
    cin >> nombre_fichero;
    escribir_fich(nombre_fichero, ok);
}
```



```

        codigo_error(ok);
    }

```

10.4. Ejemplos

Ejemplo 1

Ejemplo de un programa que copia el contenido de un fichero a otro, carácter a carácter:

```

#include <iostream>
#include <fstream>
#include <string>
using namespace std;
enum Codigo {
    OK, ERROR_APERTURA_ENT, ERROR_APERTURA_SAL, ERROR_FORMATO
};
void copiar_fichero(const string& salida, const string& entrada, Codigo& ok)
{
    ifstream f_ent;
    f_ent.open(entrada.c_str());
    if (f_ent.fail()) {
        ok = ERROR_APERTURA_ENT;
    } else {
        ofstream f_sal;
        f_sal.open(salida.c_str());
        if (f_sal.fail()) {
            ok = ERROR_APERTURA_SAL;
        } else {
            char ch;
            f_ent.get(ch);
            while (! f_ent.fail() && f_sal.good()) {
                f_sal.put(ch);
                f_ent.get(ch);
            }
            if (f_ent.eof() && f_sal.good()) {
                ok = OK;
            } else {
                ok = ERROR_FORMATO;
            }
            f_sal.close(); // no es necesario
        }
        f_ent.close(); // no es necesario
    }
}

void codigo_error(Codigo ok)
{
    switch (ok) {
        case OK:
            cout << "Fichero procesado correctamente" << endl;
            break;
        case ERROR_APERTURA_ENT:
            cout << "Error en la apertura del fichero de entrada" << endl;
            break;
        case ERROR_APERTURA_SAL:
            cout << "Error en la apertura del fichero de salida" << endl;
            break;
        case ERROR_FORMATO:
            cout << "Error de formato en la lectura del fichero" << endl;
            break;
    }
}

```

```

    }
}
int main()
{
    Codigo ok;
    string entrada, salida;
    cout << "Introduzca el nombre del fichero de entrada: ";
    cin >> entrada;
    cout << "Introduzca el nombre del fichero de salida: ";
    cin >> salida;
    copiar_fichero(salida, entrada, ok);
    codigo_error(ok);
}

```

Ejemplo 2

Ejemplo de un programa que crea, guarda y carga una agenda personal.

```

//-----
#include <iostream>
#include <fstream>
#include <string>
#include <tr1/array>
#include <cctype>
using namespace std;
using namespace std::tr1;
//-----
struct Fecha {
    unsigned dia;
    unsigned mes;
    unsigned anyo;
};
struct Persona {
    string nombre;
    string tfn;
    Fecha fnac;
};
const unsigned MAX = 100;
typedef array<Persona, MAX> APers;
struct Agenda {
    unsigned nelms;
    APers elm;
};
//-----
void inic_agenda(Agenda& ag)
{
    ag.nelms = 0;
}
void anyadir_persona(Agenda& ag, const Persona& p, bool& ok)
{
    if (ag.nelms < ag.elm.size()) {
        ag.elm[ag.nelms] = p;
        ++ag.nelms;
        ok = true;
    } else {
        ok = false;
    }
}
//-----
void leer_fecha(Fecha& f)

```

```

{
    cout << "Introduza fecha de nacimiento (dia mes año): ";
    cin >> f.dia >> f.mes >> f.anyo;
}
void leer_persona(Persona& p)
{
    cout << "Introduza nombre: ";
    cin >> ws;
    getline(cin, p.nombre);
    cout << "Introduza teléfono: ";
    cin >> p.tfn;
    leer_fecha(p.fnac);
}
void nueva_persona(Agenda& ag)
{
    bool ok;
    Persona p;
    leer_persona(p);
    if (!cin.fail()) {
        anyadir_persona(ag, p, ok);
        if (!ok) {
            cout << "Error al introducir la nueva persona" << endl;
        }
    } else {
        cout << "Error al leer los datos de la nueva persona" << endl;
        cin.clear();
        cin.ignore(1000, '\n');
    }
}
//-----
void escribir_fecha(const Fecha& f)
{
    cout << f.dia << '/' << f.mes << '/' << f.anyo;
}
void escribir_persona(const Persona& p)
{
    cout << "Nombre: " << p.nombre << endl;
    cout << "Teléfono: " << p.tfn << endl;
    cout << "Fecha nac: ";
    escribir_fecha(p.fnac);
    cout << endl;
}
void escribir_agenda(const Agenda& ag)
{
    for (unsigned i = 0; i < ag.nelms; ++i) {
        cout << "-----" << endl;
        escribir_persona(ag.elm[i]);
    }
    cout << "-----" << endl;
}
//-----
// FORMATO DEL FICHERO DE ENTRADA:
//
// <nombre> <RC>
// <teléfono> <dia> <mes> <año> <RC>
// <nombre> <RC>
// <teléfono> <dia> <mes> <año> <RC>
// ...
//-----

```

```

void leer_fecha(istream& fich, Fecha& f)
{
    fich >> f.dia >> f.mes >> f.anyo;
}
void leer_persona(istream& fich, Persona& p)
{
    fich >> ws;
    getline(fich, p.nombre);
    fich >> p.tfn;
    leer_fecha(fich, p.fnac);
}
//-----
// Otra posible implementación
// void leer_persona(istream& fich, Persona& p)
// {
//     getline(fich, p.nombre);
//     fich >> p.tfn;
//     leer_fecha(fich, p.fnac);
//     fich.ignore(1000, '\n');
// }
//-----
void leer_agenda(const string& nombre_fich, Agenda& ag, bool& ok)
{
    ifstream fich;
    Persona p;

    fich.open(nombre_fich.c_str());
    if (fich.fail()) {
        ok = false;
    } else {
        ok = true;
        inic_agenda(ag);
        leer_persona(fich, p);
        while (!fich.fail() && ok) {
            anyadir_persona(ag, p, ok);
            leer_persona(fich, p);
        }
        ok = ok && fich.eof();
        fich.close();
    }
}

void cargar_agenda(Agenda& ag)
{
    bool ok;
    string nombre_fich;
    cout << "Introduce el nombre del fichero: ";
    cin >> nombre_fich;
    leer_agenda(nombre_fich, ag, ok);
    if (!ok) {
        cout << "Error al cargar el fichero" << endl;
    }
}
//-----
// FORMATO DEL FICHERO DE SALIDA:
//
// <nombre> <RC>
// <teléfono> <dia> <mes> <año> <RC>
// <nombre> <RC>
// <teléfono> <dia> <mes> <año> <RC>

```

```

// ...
//-----
void escribir_fecha(ofstream& fich, const Fecha& f)
{
    fich << f.dia << ' ' << f.mes << ' ' << f.anyo;
}
void escribir_persona(ofstream& fich, const Persona& p)
{
    fich << p.nombre << endl;
    fich << p.tfn << ' ';
    escribir_fecha(fich, p.fnac);
    fich << endl;
}
void escribir_agenda(const string& nombre_fich, const Agenda& ag, bool& ok)
{
    ofstream fich;

    fich.open(nombre_fich.c_str());
    if (fich.fail()) {
        ok = false;
    } else {
        unsigned i = 0;
        while ((i < ag.nelms) && (fich.good())) {
            escribir_persona(fich, ag.elm[i]);
            ++i;
        }
        ok = fich.good();
        fich.close();
    }
}
void guardar_agenda(const Agenda& ag)
{
    bool ok;
    string nombre_fich;
    cout << "Introduce el nombre del fichero: ";
    cin >> nombre_fich;
    escribir_agenda(nombre_fich, ag, ok);
    if (!ok) {
        cout << "Error al guardar el fichero" << endl;
    }
}
//-----
char menu()
{
    char op;
    cout << endl;
    cout << "C. Cargar Agenda" << endl;
    cout << "M. Mostrar Agenda" << endl;
    cout << "N. Nueva Persona" << endl;
    cout << "G. Guardar Agenda" << endl;
    cout << "X. Fin" << endl;
    do {
        cout << endl << "    Opción: ";
        cin >> op;
        op = char(toupper(op));
    } while (!(op == 'C') || (op == 'M') || (op == 'N') || (op == 'G') || (op == 'X')));
    cout << endl;
    return op;
}

```

```

//-----
int main()
{
    Agenda ag;
    char op;
    inic_agenda(ag);
    do {
        op = menu();
        switch (op) {
            case 'C':
                cargar_agenda(ag);
                break;
            case 'M':
                escribir_agenda(ag);
                break;
            case 'N':
                nueva_persona(ag);
                break;
            case 'G':
                guardar_agenda(ag);
                break;
        }
    } while (op != 'X');
}
//-----

```

10.5. Otros Tipos de Flujos de Ficheros

10.5.1. Ficheros Binarios

Como se mostró al inicio del capítulo, los ficheros binarios permiten almacenar la información utilizando una codificación basada en la representación interna de los datos en la memoria principal por el procesador, el código binario. Cuando queremos trabajar con un fichero codificado en formato binario, tanto para entrada como para salida de datos, al realizar la operación de apertura para vincular la variable manipulador del fichero con el fichero, se deberá especificar que el fichero está en modo binario mediante el símbolo `ios::binary` en dicha operación.

Salida Binaria de Datos

La salida binaria de variables de tipos simples (o variables de tipo array de tipos simples) se realiza como se indica a continuación. Para ello es necesario abrir el fichero en modo binario, y utilizar los métodos adecuados:

<code>ofstream f_sal;</code>	Definir variable flujo de salida
<code>f_sal.open(nombre_fich.c_str(), ios::binary);</code>	Vincular el flujo con un fichero en modo <i>binario</i>
<code>f_sal.write((const char*)&x, sizeof(x));</code>	Escribe en modo binario el contenido de una variable x de tipo simple, o array de tipo simple
<code>long n = f_sal.pcount();</code>	Retorna el número de caracteres (bytes) leídos en la última operación de escritura
<code>f_sal.flush();</code>	Fuerza el volcado del flujo de salida al dispositivo de almacenamiento

El formato binario de las cadenas de caracteres coincide con su representación textual, por lo que utilizaremos el mismo mecanismo que para la salida textual de cadenas de caracteres, pero deberemos ahora incluir un delimitador al final de la cadena adecuado para el formato binario tal como `'\0'`.

```
f_sal << nombre << '\0';
```

La salida de variables de tipo estructurado (registros, clases y arrays de registros y clases) debe realizarse componente a componente.

Entrada Binaria de Datos

La entrada binaria de variables de tipos simples (o variables de tipo array de tipos simples) se realiza como se indica a continuación. Para ello es necesario abrir el fichero en modo binario, y utilizar los métodos adecuados:

<code>ifstream f_ent;</code>	Definir variable flujo de entrada
<code>f_ent.open(nombre_fich.c_str(), ios::binary);</code>	Vincular el flujo con un fichero en modo <i>binario</i>
<code>f_ent.read((char*)&x, sizeof(x));</code>	Lee en modo binario el contenido de una variable <i>x</i> de tipo simple, o <i>array</i> de tipo simple
<code>long n = f_ent.gcount();</code>	Retorna el número de caracteres (bytes) leídos en la última operación de lectura

El formato binario de las cadenas de caracteres coincide con su representación textual, por lo que utilizaremos el mismo mecanismo que para la entrada textual de cadenas de caracteres, pero considerando que el final de la cadena de caracteres está marcado por la inclusión en el fichero de un determinado delimitador según se realizó el proceso de almacenamiento en el fichero ('`\0`').

```
getline(f_ent, nombre, '\0');
```

La entrada de variables de tipo estructurado (registros, clases y arrays de registros y clases) debe realizarse componente a componente.

Ⓐ Entrada y Salida Binaria Avanzada

La entrada y salida binaria de datos utiliza la representación interna (en memoria principal) de los datos para su almacenamiento en memoria secundaria, y esta representación interna depende de como el hardware (el procesador) representa dicha información internamente. Por este motivo, la entrada y salida binaria de datos puede dar lugar a diversos errores de *compatibilidad* cuando se realiza con ordenadores que utilizan diferentes representaciones internas de los datos (*big-endian* y *little-endian*).

Valor numérico		Representación interna	
<i>Decimal</i>	<i>Hexadecimal</i>	<i>Binario (bytes)</i>	<i>byte order</i>
987654321	0x3ade68b1	00111010:11011110:01101000:10110001	<i>big-endian (xdr, network-byte-order)</i>
987654321	0x3ade68b1	10110001:01101000:11011110:00111010	<i>little-endian (Intel)</i>

Por ello, y para evitar este tipo de problemas, en determinadas circunstancias se puede optar por realizar la entrada y salida binaria en una determinada representación independiente de la representación interna utilizada por el procesador (*big-endian* o *little-endian*). Por ejemplo, para entrada y salida de un dato de tipo `int` (es válida para los tipos simples integrales):

```
#include <fstream>
#include <climits>
using namespace std;
//-----
typedef int Tipo;
//-----
void escribir_big_endian(ostream& out, Tipo x)
{
    unsigned char memoria[sizeof(x)];
    for (int i = sizeof(x)-1; i >= 0; --i) {
        memoria[i] = (unsigned char)(x & ((1U<<CHAR_BIT)-1U));
        x = Tipo(x >> CHAR_BIT);
    }
    out.write((const char*)memoria, sizeof(x));
}
void leer_big_endian(istream& in, Tipo& x)
{
    unsigned char memoria[sizeof(x)];
    in.read((char*)memoria, sizeof(x));
```

```

    x = 0;
    for (unsigned i = 0; i < sizeof(x); ++i) {
        x = Tipo((x << CHAR_BIT) | unsigned(memoria[i]));
    }
}
//-----
void escribir_little_endian(ostream& out, Tipo x)
{
    unsigned char memoria[sizeof(x)];
    for (unsigned i = 0; i < sizeof(x); ++i) {
        memoria[i] = (unsigned char)(x & ((1U<<CHAR_BIT)-1U));
        x = Tipo(x >> CHAR_BIT);
    }
    out.write((const char*)memoria, sizeof(x));
}
void leer_little_endian(istream& in, Tipo& x)
{
    unsigned char memoria[sizeof(x)];
    in.read((char*)memoria, sizeof(x));
    x = 0;
    for (int i = sizeof(x)-1; i >= 0; --i) {
        x = Tipo((x << CHAR_BIT) | unsigned(memoria[i]));
    }
}
//-----

```

Para entrada y salida de cadenas de caracteres en un formato más avanzado, se añadirá al principio el número de caracteres de la cadena en formato binario, y se incluirá un delimitador al final de la cadena adecuado para el formato binario tal como '\0':

```

//-----
void escribir_big_endian(ostream& out, const string& x)
{
    escribir_big_endian(out, x.size());
    out << nombre << '\0';
}
void leer_big_endian(istream& in, string& x)
{
    unsigned sz;
    leer_big_endian(in, sz);
    x.reserve(sz);
    getline(f_ent, x, '\0');
    assert(sz == x.size());
}
//-----
void escribir_little_endian(ostream& out, const string& x)
{
    escribir_little_endian(out, x.size());
    out << nombre << '\0';
}
void leer_little_endian(istream& in, string& x)
{
    unsigned sz;
    leer_little_endian(in, sz);
    x.reserve(sz);
    getline(f_ent, x, '\0');
    assert(sz == x.size());
}
//-----

```


Así, almacenar el número de caracteres permite poder alojar el espacio en memoria suficiente para almacenar la secuencia de caracteres que vaya a ser leída, aunque ésto no es necesario si se utiliza el tipo `string` de C++, no obstante es conveniente.

Nota: estos subprogramas son adecuados si la cadena de caracteres no contiene el carácter `'\0'` como un elemento válido de la secuencia de caracteres.

10.5.2. Acceso Directo en Ficheros

En muchas ocasiones el fichero sobre el que realizamos la entrada o salida de datos se encuentra almacenado en un dispositivo de memoria secundaria con soporte para el acceso directo, tales como discos magnéticos, ópticos, discos de memoria de estado sólido, etc. Estos dispositivos permiten acceder a determinadas posiciones del fichero directamente, sin necesidad de realizar un acceso secuencial sobre todo el contenido del fichero (como podría suceder si el fichero se encontrase almacenado en un dispositivo de cinta magnética. En estas circunstancias, el lenguaje de programación C++ ofrece una serie de operaciones para poder poner el índice de lectura o escritura en una determinada posición del fichero, de tal forma que la próxima operación de lectura o de escritura se realice a partir de esa posición.

Acceso Directo en Flujos de Entrada

Las siguientes operaciones, cuando se aplican a un flujo de entrada de datos, permiten acceder para conocer y modificar la posición donde se encuentra en índice de lectura del flujo, a partir del cual se realizará la próxima operación de lectura.

La operación `tellg()` aplicada a un flujo de entrada permite obtener la posición donde se encuentra en índice de lectura:

```
n = cin.tellg();
```

La operación `seekg(pos)` aplicada a un flujo de entrada permite poner la posición del índice de lectura, en una posición *absoluta* desde el inicio del fichero:

```
cin.seekg(pos);
```

La operación `seekg(offset, desde)` aplicada a un flujo de entrada permite poner la posición del índice de lectura, en una posición *relativa* como un desplazamiento desde una posición conocida del fichero, el cual puede ser desde el inicio (`ios::beg`), desde el final (`ios::end`) o desde la posición actual (`ios::cur`):

```
cin.seekg(offset, desde);
```

Es importante considerar que en el caso de flujos abiertos en modo de lectura y escritura simultánea, puede suceder que el índice de lectura sea el mismo que el índice de escritura, por lo que cualquier operación realizada sobre el índice de lectura también afectará al índice de escritura, y viceversa.

Acceso Directo en Flujos de Salida

Las siguientes operaciones, cuando se aplican a un flujo de salida de datos, permiten acceder para conocer y modificar la posición donde se encuentra en índice de escritura del flujo, a partir del cual se realizará la próxima operación de escritura.

La operación `tellp()` aplicada a un flujo de salida permite obtener la posición donde se encuentra en índice de escritura:

```
n = cout.tellp();
```

La operación `seekp(pos)` aplicada a un flujo de salida permite poner la posición del índice de escritura, en una posición *absoluta* desde el inicio del fichero:

```
cout.seekp(pos);
```

La operación `seekp(offset, desde)` aplicada a un flujo de salida permite poner la posición del índice de lectura, en una posición *relativa* como un desplazamiento desde una posición conocida del fichero, el cual puede ser desde el inicio (`ios::beg`), desde el final (`ios::end`) o desde la posición actual (`ios::cur`):

```
cout.seekp(offset, desde);
```

Es importante considerar que en el caso de flujos abiertos en modo de lectura y escritura simultánea, puede suceder que el índice de lectura sea el mismo que el índice de escritura, por lo que cualquier operación realizada sobre el índice de lectura también afectará al índice de escritura, y viceversa.

10.5.3. Flujos de Entrada y Salida

Además de tener flujos dedicados exclusivamente para realizar entrada de datos y salida de datos de forma independiente, también es posible abrir un fichero para realizar sobre el mismo tanto operaciones de entrada como operaciones de salida sobre el mismo flujo de datos. Para trabajar con *flujos de entrada y salida* simultánea, deberemos declarar las variables manejadores de fichero del tipo `fstream`:

```
fstream fich_ent_sal;
```

posteriormente vincularemos la variable manejador del fichero con un determinado fichero

```
fich_ent_sal.open(nombre_fichero.c_str());
```

y podremos realizar tanto operaciones de entrada como de salida de datos sobre el mismo. Todo lo explicado en los capítulos anteriores es aplicable a este tipo de flujos, especialmente las operaciones de acceso directo para posicionar adecuadamente el índice de lectura y el de escritura.

Es importante considerar que en el caso de flujos abiertos en modo de lectura y escritura simultánea, puede suceder que el índice de lectura sea el mismo que el índice de escritura, por lo que cualquier operación realizada sobre el índice de lectura también afectará al índice de escritura, y viceversa.

10.6. Flujos de Entrada y Salida Vinculados a Cadenas de Caracteres

Hemos visto como hay flujos vinculados con la entrada y salida estándares, así como flujos vinculados a ficheros almacenados en *memoria secundaria*. El lenguaje de programación C++ también permite vincular flujos de entrada y salida con cadenas de caracteres (strings) contenidos en *memoria principal*. Para ello, se deberá incluir la biblioteca estándar `<sstream>`:

```
#include <sstream>
```

Entrada de Datos desde Cadenas de Caracteres

Para utilizar un flujo de entrada de datos desde cadenas de caracteres, se declara una variable de tipo `istreamstringstream` sobre la que se puede realizar la entrada de datos como se realiza de forma habitual con los flujos de entrada. El contenido del flujo desde el que se realizará la entrada de datos se puede especificar durante la propia definición de la variable, o mediante el operador `str("...")`. Por ejemplo:

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

int main()
```

```

{
    string datos = "123 456";
    istream fent(datos);
    int val;
    fent >> val;
    while (fent) {
        cout << "Valor: " << val<<endl;
        fent >> val;
    }
    bool ok = fent.eof();
    cout << boolalpha << "Estado: " << ok << endl;
    //-----
    fent.clear();
    fent.str(" 789 345 ");
    fent >> val;
    while (fent) {
        cout << "Valor: " << val<<endl;
        fent >> val;
    }
    cout << boolalpha << "Estado: " << fent.eof() << endl;
}

```

Ⓐ Salida de Datos a Cadenas de Caracteres

Para utilizar un flujo de salida de datos a cadenas de caracteres, se declara una variable de tipo `ostream` sobre la que se puede realizar la salida de datos como se realiza de forma habitual con los flujos de salida. El operador `str()` permite obtener la cadena de caracteres correspondiente a la salida de datos realizada. Así mismo, una llamada a `str("")` reinicializa el flujo de salida de datos:

```

#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main()
{
    ostream fsal;
    for (int i = 0; i < 10; ++i) {
        fsal << i << " ";
    }
    string salida = fsal.str();
    cout << "Salida: " << salida << endl;
    //-----
    fsal.str("");
    for (int i = 0; i < 10; ++i) {
        fsal << i << " ";
    }
    salida = fsal.str();
    cout << "Salida: " << salida << endl;
}

```

Ⓐ Entrada y Salida de Datos a Cadenas de Caracteres

Por último, también es posible realizar entrada y salida de datos simultáneamente utilizando el tipo `stringstream`, y utilizando en operador `str("...")` para dar y el operador `str()` para obtener valores de tipo cadena de caracteres. La entrada y salida de datos sobre el flujo se realiza de forma habitual.

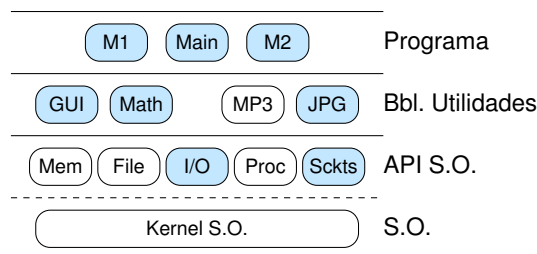
Capítulo 11

Módulos y Bibliotecas

Cuando se desarrollan programas de complejidad media/alta, el código fuente normalmente no se encuentra en un único fichero, sino que se encuentra distribuido entre varios módulos. Una primera ventaja de la existencia de módulos es que permiten aumentar la localidad y cohesión del código y aislarlo del exterior, es decir, poner todo el código encargado de resolver un determinado problema en un módulo nos permite aislarlo del resto, con lo que futuras modificaciones serán más fáciles de realizar.

Otra ventaja adicional de los módulos es el hecho de que si se modifica algo interno en un determinado módulo, sólo será necesario volver a compilar dicho módulo, y no todo el programa completo, lo que se convierte en una ventaja indudable en caso de programas grandes (*compilación separada*).

Además, esta división modular es una pieza fundamental para la *reutilización del código*, ya que permite la utilización de bibliotecas del sistema, así como la creación y distribución de bibliotecas de utilidades que podrán ser utilizadas por múltiples programas. Esta distribución de bibliotecas se puede hacer en *código objeto*, por lo que no es necesario distribuir el código fuente de la misma.



Así, vemos que en la figura un determinado programa se compone de varios módulos de programa (Main, M1 y M2) en los cuales está dividida la solución principal del problema, varios módulos de bibliotecas proporcionan utilidades gráficas, matemáticas y tratamiento de imágenes; así como varios módulos de biblioteca dan acceso a servicios de entrada/salida y comunicaciones por Internet proporcionados por el sistema operativo.

11.1. Interfaz e Implementación del Módulo

En el lenguaje de programación C++, normalmente un módulo se compone de dos ficheros: uno donde aparece el código que resuelve un determinado problema o conjunto de problemas (implementación – parte privada), y un fichero que contiene las definiciones de tipos, constantes y prototipos de subprogramas que el módulo ofrece (interfaz – parte pública). Así, se denomina la *implementación* del módulo al fichero que contiene la parte privada del módulo, y se denomina la *interfaz* del módulo al fichero que contiene la parte pública del mismo. A este fichero también se le denomina “fichero de encabezamiento” o “fichero de cabecera” (*header file* en inglés).

<pre>main.cpp (Principal) #include "vector.hpp" // utilización de vector int main() { ... }</pre>	<pre>vector.hpp (Interfaz) #ifndef _vector_hpp_ #define _vector_hpp_ // interfaz de vector // público ... #endif</pre>	<pre>vector.cpp (Implementación) #include "vector.hpp" // implementación de vector // privado</pre>
---	--	---

Por lo tanto, un programa completo normalmente se compone de varios módulos, cada uno con su fichero de encabezamiento (interfaz) y de implementación, y de un módulo principal donde reside la función principal `main`. Para que un determinado módulo pueda hacer uso de las utilidades que proporciona otro módulo, deberá incluir el fichero de encabezamiento (interfaz) del módulo que se vaya a utilizar, de tal forma que tenga acceso a las declaraciones públicas de éste. Así mismo, el fichero de implementación de un determinado módulo también deberá especificar *al comienzo* del mismo la inclusión del fichero de encabezamiento de su propio módulo, con objeto de obtener y contrastar las definiciones allí especificadas, es decir, el fichero de encabezamiento del propio módulo debe ser el primer fichero que se incluya en el fichero de implementación del mismo módulo. Normalmente los ficheros de implementación tendrán una extensión “.cpp” (también suelen utilizarse otras extensiones como “.cxx” y “.cc”) y los ficheros de encabezamiento tendrán una extensión “.hpp” (también suelen utilizarse otras extensiones como “.hxx”, “.hh” y “.h”). Así, para incluir el fichero de encabezamiento (interfaz) de un módulo `vector` se utiliza la siguiente directiva:

```
#include "vector.hpp"
```

Nótese que cuando se incluyen ficheros de encabezamiento (interfaz) de la biblioteca estándar (o del sistema), el nombre del fichero se especifica entre `<...>`, pero cuando se incluyen ficheros de encabezamiento de módulos y bibliotecas locales (no estándares), entonces el nombre del fichero se especifica entre `"..."`. De esta forma, los ficheros de la biblioteca estándar y del sistema se buscan en directorios del sistema, pero los ficheros y bibliotecas locales se buscan en el directorio local de trabajo.

Guardas en un Fichero de Encabezamiento

Las definiciones en los ficheros de encabezamiento (interfaz) serán especificadas entre las *guardas* (directivas de compilación condicional) para evitar la inclusión duplicada de las definiciones allí contenidas. El nombre de la guarda usualmente se deriva del nombre del fichero, como se indica en el siguiente ejemplo donde el módulo `vector` tendrá los siguientes ficheros de encabezamiento y de implementación (en determinadas circunstancias, puede ser conveniente que al nombre de la guarda se le añada también el nombre del espacio de nombres que se explicará en la siguiente sección):

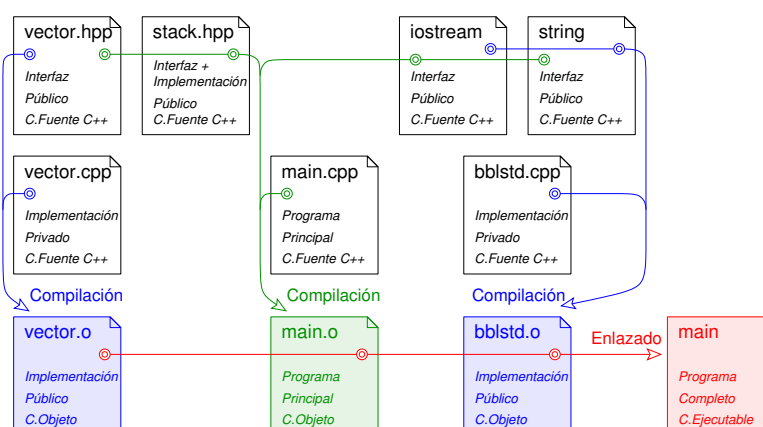
Fichero: <code>vector.hpp</code> (Interfaz)	Fichero: <code>vector.cpp</code> (Implementación)
<pre>// Guarda para evitar inclusión duplicada #ifndef _vector_hpp_ #define _vector_hpp_ // Definiciones Públicas de: // * Constantes // * Tipos (Enum, Registros, Clases) // * Prototipos de Subprogramas #endif // Fin de guarda</pre>	<pre>#include "vector.hpp" // Implementaciones Privadas de: // * Constantes Privadas // * Tipos Privados // * Subprogramas // * Clases</pre>

Directrices para el Diseño de Ficheros de Encabezamiento

Las siguientes directrices deben ser tenidas en cuenta con objeto de organizar adecuadamente el diseño de los ficheros de encabezamiento de los módulos:

- Un fichero de encabezamiento sólo deberá contener definiciones de constantes, definiciones de tipos y prototipos de subprogramas que exporta (parte pública) el propio módulo. No deberá contener definiciones de variables globales, ni la implementación de código (de subprogramas y métodos). Esto último salvo algunas excepciones, tales como la definición de subprogramas “en línea” (véase 5.6), la definición de clases *simples* definidas “en línea” (véase 13.1.1) y finalmente la definición de subprogramas y clases genéricas (véase 14).
 - El mecanismo de inclusión de ficheros de encabezamiento debe ser robusto ante posibles inclusiones duplicadas. Para ello siempre se utilizará el mecanismo de guardas explicado anteriormente.
 - Un fichero de encabezamiento debe incluir todos los ficheros de encabezamiento de otros módulos que necesite para su propia definición, de forma tal que el orden de inclusión de los ficheros de encabezamiento no sea importante.
- Ⓐ Un cuidadoso diseño de los módulos normalmente evitará la inclusión y dependencias circulares entre módulos, sin embargo excepcionalmente pueden surgir dichas dependencias circulares, y en estas circunstancias una *declaración adelantada* de un *tipo incompleto* (véase 15.6) puede sustituir la inclusión de un fichero de encabezamiento y evitar la dependencia circular.

11.2. Compilación Separada y Enlazado



Cuando se compila un módulo de forma independiente (compilación separada), se compila su fichero de implementación, por ejemplo `vector.cpp`, y produce como resultado un fichero en *código objeto*, por ejemplo `vector.o`, considerando que el código fuente en C++ compilado es el contenido del fichero de implementación junto con el contenido de todos los ficheros de encabezamiento incluidos durante el proceso de compilación. Por ejemplo, mediante el siguiente comando se compila un módulo de implementación de código fuente en C++ utilizando el compilador *GNU GCC* para generar el correspondiente código objeto:

```
g++ -ansi -Wall -Werror -c vector.cpp
g++ -ansi -Wall -Werror -c main.cpp
```

y el enlazado de los códigos objeto para generar el código ejecutable:

```
g++ -ansi -Wall -Werror -o main main.o vector.o
```

Aunque también es posible realizar la compilación y enlazado en el mismo comando:

```
g++ -ansi -Wall -Werror -o main main.cpp vector.cpp
```

o incluso mezclar compilación de código fuente y enlazado de código objeto:

```
g++ -ansi -Wall -Werror -o main main.cpp vector.o
```

Hay que tener en cuenta que el compilador enlaza automáticamente el código generado con las bibliotecas estándares de C++, y por lo tanto no es necesario que éstas se especifiquen explícitamente. Sin embargo, en caso de ser necesario, también es posible especificar el enlazado con bibliotecas externas:

```
g++ -ansi -Wall -Werror -o main main.cpp vector.cpp -ljpeg
```

Estas bibliotecas no son más que una agregación de módulos compilados a código objeto, y organizadas adecuadamente para que puedan ser reutilizados por muy diversos programas.

Así mismo, en sistemas *Unix* existe otro mecanismo (**make** y *makefiles*) que permite establecer relaciones de dependencia entre diversos módulos, de tal forma que es capaz de discernir cuando es necesario compilar los diferentes módulos dependiendo de si se produce algún cambio en los módulos de los que dependen (véase 11.4).

11.3. Espacios de Nombre

Cuando se trabaja con múltiples módulos y bibliotecas, es posible que se produzcan *colisiones* en la definición de entidades diferentes con los mismos identificadores proporcionadas por diferentes módulos y bibliotecas. Este hecho no está permitido por el lenguaje de programación C++. Para evitar estas posibles colisiones existen los *espacios de nombre* (*namespace* en inglés), que permiten agrupar bajo una misma denominación un conjunto de declaraciones y definiciones, de tal forma que dicha denominación será necesaria para identificar y diferenciar cada entidad declarada. Estos espacios de nombre pueden ser únicos para un determinado módulo, o por el contrario pueden abarcar múltiples módulos y bibliotecas gestionados por el mismo proveedor, por ejemplo todas las entidades definidas en la biblioteca estándar se encuentran bajo el espacio de nombres **std**. Así, el nombre del espacio de nombres puede ser derivado del propio nombre del fichero, puede incluir una denominación relativa al proveedor del módulo, o alguna otra denominación más compleja que garantice que no habrá colisiones en el nombre del espacio de nombres.

main.cpp (Principal)	vector.hpp (Interfaz)	vector.cpp (Implementación)
<pre>#include <iostream> #include "vector.hpp" using namespace std; using namespace umalcc; // utilización de vector int main() { ... }</pre>	<pre>#ifndef _vector.hpp_ #define _vector_hpp_ #include <...otros...> // interfaz de vector namespace umalcc { } #endif</pre>	<pre>#include "vector.hpp" #include <...otros...> // implementación de vector namespace umalcc { }</pre>

Nótese que la inclusión de ficheros de encabezamiento se debe realizar externamente a la definición de los espacios de nombre.

- Ⓐ Hay que tener en cuenta que un mismo espacio de nombres puede especificarse múltiples veces, desde diversos ficheros, para añadir nuevas entidades al mismo.
- Ⓐ Adicionalmente, también es posible definir espacios de nombre anidados, pudiendo, de esta forma, definir jerarquías de espacios de nombre.
- Ⓐ Así mismo, también existen *espacios de nombre anónimos* que permiten definir entidades privadas internas a los módulos de implementación, de tal forma que no puedan producir colisiones con las entidades públicas del sistema completo. De esta forma, cualquier declaración y definición realizada dentro de un espacio de nombres anónimo será únicamente visible en el módulo de implementación donde se encuentre (privada), pero no será visible en el exterior del módulo.

Utilización de Espacios de Nombre

Es posible utilizar las entidades (identificadores) definidas dentro de espacios de nombres de varias formas, dependiendo de las circunstancias donde se produzca esta utilización, teniendo en cuenta que todos los identificadores definidos dentro de un espacio de nombres determinado son visibles y accesibles directamente dentro de él mismo.

- En la implementación de los módulos (archivos de implementación `.cpp`), mediante la directiva `using namespace` se ponen disponibles (accesibles) todos los identificadores de dicho espacio de nombres completo, que podrán ser accedidos directamente, sin necesidad de cualificación explícita. Por ejemplo:

```
using namespace std;
using namespace umalcc;
```

Si se utilizan (mediante `using namespace`) varios espacios de nombre simultáneamente y ambos definen el mismo identificador, si dicho identificador no se utiliza entonces no se produce colisión. Sin embargo en caso de que se utilice dicho identificador, entonces se produce una colisión ya que el compilador no puede discernir a qué entidad se refiere. En este último caso, el programador debe utilizar la *cualificación explícita* (explicada a continuación) para este identificador y eliminar de esta forma la ambigüedad en su utilización.

Sin embargo, no es adecuado aplicar la directiva `using namespace` dentro de archivos de encabezamiento, ya que si es utilizada en un archivo de encabezamiento que se incluye por múltiples módulos, entonces pondría disponible (accesible) todos los identificadores de dicho espacio de nombres para todos los archivos que incluyan (`include`) dicho archivo de encabezamiento, algo que podría provocar colisiones inesperadas, y esto sería un efecto colateral no deseado para aquellos que utilizaran dicho módulo (incluyeran dicho archivo de encabezamiento). Por lo tanto, en archivos de encabezamiento se deben utilizar los siguientes métodos explicados a continuación.

- En los archivos de encabezamiento (`.hpp`) cada identificador externo (perteneciente a otro espacio de nombres) se debe utilizar cualificado con el espacio de nombres al que pertenece (*cualificación explícita*) utilizando para ello el *nombre* del espacio de nombres, seguido por el operador `::` y del nombre del *identificador*, como en el siguiente ejemplo para utilizar el tipo `array` del espacio de nombres `std::tr1`:

```
namespace umalcc {
    typedef std::tr1::array<int, 20> Vector;
}
```

- Por conveniencia, existe un mecanismo intermedio entre los dos métodos generales explicados anteriormente. El método consiste en utilizar la declaración `using` seguido por la cualificación explícita del identificador, de tal forma que la utilización de dicho identificador queda disponible para ser utilizada directamente sin cualificación (*cualificación implícita*). Este mecanismo se puede utilizar en archivos de implementación, y en el caso de archivos de encabezamiento siempre se debe utilizar dentro de otro espacio de nombres, para hacer público dicho identificador dentro de este nuevo espacio de nombres. Por ejemplo:

```
namespace umalcc {
    using std::tr1::array;
}
```

- Finalmente, también es posible crear un alias para un espacio de nombres para facilitar la cualificación:

```
namespace lcc = umalcc;
```

Ejemplo

Módulo Vector

Ejemplo de un módulo `vector`, que implementa el concepto de *array incompleto*. Debido a su simplicidad, toda la implementación del módulo se proporciona como *código en línea*, por lo que no es necesario un fichero aparte que contenga la implementación independiente. En este ejemplo, se puede apreciar como se hace uso de una entidad definida en otro módulo (espacio de nombres `std::tr1`) mediante cualificación explícita:

```
//- fichero: vector.hpp -----
#ifndef _vector_hpp_
#define _vector_hpp_
#include <cassert>
#include <tr1/array>
namespace umalcc {
    const unsigned MAX_ELMS = 30;
    typedef std::tr1::array<int, MAX_ELMS> Datos;
    struct Vector {
        unsigned nelms;
        Datos elm;
    };
    inline void inicializar(Vector& v)
    {
        v.nelms = 0;
    }
    inline bool vacio(const Vector& v) {
        return (v.nelms == 0);
    }
    inline bool lleno(const Vector& v) {
        return (v.nelms >= v.elm.size());
    }
    inline unsigned nelms(const Vector& v)
    {
        return v.nelms;
    }
    inline void anyadir_elm(Vector& v, int e)
    {
        assert(! lleno(v)); // Pre-condición
        v.elm[v.nelms] = e;
        ++v.nelms;
    }
    inline void eliminar_ultimo(Vector& v)
    {
        assert(! vacio(v)); // Pre-condición
        --v.nelms;
    }
    inline int obtener_elm(const Vector& v, unsigned i)
    {
        assert(i < size(v)); // Pre-condición
        return v.elm[i];
    }
    inline void asignar_elm(Vector& v, unsigned i, int e)
    {
        assert(i < size(v)); // Pre-condición
        v.elm[i] = e;
    }
}
#endif
//- fin: vector.hpp -----
```

Una posible utilización del módulo vector definido anteriormente podría ser:

```
//- fichero: main.cpp -----
#include "vector.hpp"
#include <iostream>
using namespace std;
using namespace umalcc;
void anyadir(Vector& v)
{
    for (int x = 0; x < 10; ++x) {
        if (!lleno(v)) {
            anyadir_elm(v, x);
        }
    }
}
void imprimir(const Vector& v)
{
    for (unsigned i = 0; i < nelms(v); ++i) {
        cout << obtener_elm(v, i) << " ";
    }
    cout << endl;
}
void eliminar(Vector& v, unsigned i)
{
    if (i < nelms(v)) {
        asignar_elm(v, i, obtener_elm(v, nelms(v)-1));
        eliminar_ultimo(v);
    }
}
int main()
{
    Vector v;
    inicializar(v);
    anyadir(v);
    imprimir(v);
    eliminar(v, 3);
}
//- fin: main.cpp -----
```

Su compilación y enlazado en *GNU GCC*:

```
g++ -ansi -Wall -Werror -o main main.cpp
```

Módulo Números Complejos

Otro ejemplo de módulo para operaciones con números complejos, donde el fichero de encabezamiento podría ser:

```
//- fichero: complejos.hpp -----
#ifndef _complejos_hpp_
#define _complejos_hpp_
/*
 * Declaraciones públicas del módulo
 */
namespace umalcc {
    struct Complejo {
        double real;
        double imag;
    };
    void crear(Complejo& num, double real, double imag);
}
```

```

    void sumar(Complejo& res, const Complejo& x, const Complejo& y);
    void mult(Complejo& res, const Complejo& x, const Complejo& y);
    void escribir(const Complejo& c);
}
#endif
// fin: complejos.hpp -----

```

La implementación del módulo se realiza en un fichero independiente, donde además se puede apreciar la utilización de un espacio de nombres anónimo:

```

// fichero: complejos.cpp -----
#include "complejos.hpp"
#include <iostream>
#include <cmath>
using namespace std;
using namespace umalcc;
/*
 * Implementación privada del módulo (namespace anónimo)
 */
namespace {
    struct Polar {
        double rho;
        double theta;
    };
    inline double sq(double x)
    {
        return x*x;
    }
    void cartesiana_a_polar(Polar& pol, const Complejo& cmp)
    {
        pol.rho = sqrt(sq(cmp.real) + sq(cmp.imag));
        pol.theta = atan2(cmp.imag, cmp.real);
    }
    void polar_a_cartesiana(Complejo& cmp, const Polar& pol)
    {
        cmp.real = pol.rho * cos(pol.theta);
        cmp.imag = pol.rho * sin(pol.theta);
    }
}
/*
 * Implementación correspondiente a la parte pública del módulo
 */
namespace umalcc {
    void crear(Complejo& num, double real, double imag)
    {
        num.real = real;
        num.imag = imag;
    }
    void sumar(Complejo& res, const Complejo& x, const Complejo& y)
    {
        res.real = x.real + y.real;
        res.imag = x.imag + y.imag;
    }
    void mult(Complejo& res, const Complejo& x, const Complejo& y)
    {
        Polar pr, p1, p2;
        cartesiana_a_polar(p1, x);
        cartesiana_a_polar(p2, y);
        pr.rho = p1.rho * p2.rho;
        pr.theta = p1.theta + p2.theta;
    }
}

```

```

        polar_a_cartesiana(res, pr);
    }
    void escribir(const Complejo& c)
    {
        cout << c.real << (c.imag >= 0 ? " + ":" ") << c.imag << " i";
    }
}
// - fin: complejos.cpp -----

```

Un ejemplo de utilización del módulo de números complejos podría ser:

```

// - fichero: main.cpp -----
#include <iostream>
#include "complejos.hpp"
using namespace std;
using namespace umalcc;
/*
 * utilización de números complejos
 */
int main()
{
    Complejo c1, c2, c3;
    crear(c2, 2.5, 4.7);
    crear(c3, 7.2, 6.3);
    multiplicar(c1, c2, c3);
    escribir(c1);
}
// - fin: main.cpp -----

```

Su compilación separada y enlazado en *GNU GCC*:

```

g++ -ansi -Wall -Werror -c complejos.cpp
g++ -ansi -Wall -Werror -c main.cpp
g++ -ansi -Wall -Werror -o main main.o complejos.o

```

Alternativamente se puede realizar en dos pasos:

```

g++ -ansi -Wall -Werror -c complejos.cpp
g++ -ansi -Wall -Werror -o main main.cpp complejos.o

```

o incluso en un único paso:

```

g++ -ansi -Wall -Werror -o main main.cpp complejos.cpp

```

11.4. Herramientas de Ayuda a la Gestión de la Compilación Separada

En los sistemas Unix (Linux, MacOS-X, etc.) existe una herramienta para la ayuda en la gestión del proceso de compilación separada. Esta herramienta es capaz de generar automáticamente las dependencias de los ficheros (utilizando para ello *make**depend*), de tal forma que cuando algún fichero de encabezamiento cambia, la herramienta *make* invocará la compilación de sólo aquellos módulos que se vean afectados por los cambios realizados desde la última compilación realizada. Por ejemplo, para los siguientes ficheros que componen un programa completo:

<pre> //- fichero: vector.hpp #ifndef _vector_hpp_ #define _vector_hpp_ #include <tr1/array> namespace umalcc { // ... } #endif </pre>	<pre> //- fichero: stack.hpp #ifndef _stack_hpp_ #define _stack_hpp_ #include "vector.hpp" namespace umalcc { // ... } #endif </pre>
<pre> //- fichero: vector.cpp #include "vector.hpp" #include <iostream> namespace umalcc { // ... } </pre>	<pre> //- fichero: main.cpp #include <iostream> #include <jpeglib.h> #include "vector.hpp" #include "stack.hpp" int main() { // ... } </pre>

Se define el siguiente fichero **Makefile** que contiene la enumeración de los ficheros fuente (**SRCS**) que componen el programa completo, así como las librerías externas necesarias (**LIBS**). Este ejemplo sigue la sintaxis utilizada por *GNU Make* (donde el símbolo $\boxed{\rightarrow}$ representa el carácter *tabulador*):

```

#- Makefile -----
SRCS=main.cpp vector.cpp
LIBS=-ljpeg
#-----
PROGNAME=
#-----
ifeq ($(strip $(PROGNAME)),)
    PROGNAME=$(basename $(notdir $(firstword $(SRCS))))
endif
ifdef NDEBUG
    CXXDBGFLAGS=-DNDEBUG -O2
else
    CXXDBGFLAGS=-g
endif
#-----
CXXFLAGS=-ansi -Wall -Werror $(CXXDBGFLAGS)
ALLDEFINES=$(CXXFLAGS)
DEPENDFLAGS=-Y
DEPEND=makedepend
OBJS=$(SRCS:.cpp=.o)
#-----
all: $(PROGNAME)
ndebug:
 $\boxed{\rightarrow}$  make NDEBUG=1
$(PROGNAME): $(OBJS)
 $\boxed{\rightarrow}$  $(LINK.cc) -o $$ $(OBJS) $(LIBS)
depend:
 $\boxed{\rightarrow}$  -c$(DEPEND) $(DEPENDFLAGS) -- $(ALLDEFINES) $(DEPEND_DEFINES) -- $(SRCS) >& /dev/null
# -----
# DO NOT DELETE THIS LINE -- make depend depends on it.

```

donde la siguiente invocación a **make** desde un terminal (shell) de Unix:

```
$ make depend
```

añade automáticamente las siguientes líneas de dependencias al final del fichero **Makefile**:

```

main.o: stack.hpp vector.hpp
vector.o: vector.hpp

```

La siguiente invocación a `make` desde un terminal (shell) de Unix:

```
$ make
```

realiza la siguiente compilación:

```
g++ -ansi -Wall -Werror -g -c -o main.o main.cpp
g++ -ansi -Wall -Werror -g -c -o vector.o vector.cpp
g++ -ansi -Wall -Werror -g -o main main.o vector.o -ljpeg
```

y la siguiente invocación a `make` desde un terminal (shell) de Unix:

```
$ make ndebug
```

realiza la siguiente compilación:

```
g++ -ansi -Wall -Werror -DNDEBUG -O2 -c -o main.o main.cpp
g++ -ansi -Wall -Werror -DNDEBUG -O2 -c -o vector.o vector.cpp
g++ -ansi -Wall -Werror -DNDEBUG -O2 -o main main.o vector.o -ljpeg
```

Posteriormente, algún cambio en los ficheros `main.cpp` o `stack.hpp` hará que la invocación a `make` realice la siguiente compilación:

```
g++ -ansi -Wall -Werror -g -c -o main.o main.cpp
g++ -ansi -Wall -Werror -g -o main main.o vector.o -ljpeg
```

Sin embargo, si se modifica el fichero `vector.hpp`, entonces se compilará todo de nuevo, ya que `vector.hpp` es incluido por todos los módulos del programa.

```
g++ -ansi -Wall -Werror -g -c -o main.o main.cpp
g++ -ansi -Wall -Werror -g -c -o vector.o vector.cpp
g++ -ansi -Wall -Werror -g -o main main.o vector.o -ljpeg
```


Capítulo 12

Manejo de Errores. Excepciones

Durante la ejecución de un programa, éste debe tratar con situaciones anómalas que suelen ser excepcionales, es decir, no forman parte del flujo de ejecución normal del programa en situaciones normales. Sin embargo, estas situaciones, a pesar de ser excepcionales, ocurren, y el programa debe estar preparado para tratar con ellas.

Dentro de estas situaciones anómalas, podríamos diferenciar dos grandes grupos, por un lado están aquellas situaciones que se producen debido a *errores de programación*, que deben ser evitadas mediante un adecuado diseño y programación, y por otro lado están aquellas producidas por *situaciones anómalas excepcionales*, que no pueden ser anticipadas, evitadas o tratadas hasta que dicha situación sucede durante la ejecución del programa.

Respecto a los errores de programación, es algo que no debería suceder en un programa bien diseñado, pero la realidad enseña que el desarrollador de software debe convivir con ellos. Por lo tanto, debe estar preparado para tratar con ellos, y diseñar estrategias que minimicen los errores de programación, así como su influencia.

12.1. Errores de Programación y Asertos

Los *asertos* constituyen una ayuda importante para la detección de errores de programación durante la fase de desarrollo y depuración del proyecto software. Los asertos comprueban que el estado del programa sea consistente frente a errores de programación. Hacen referencia a pre-condiciones, post-condiciones, invariantes, etc. especificando, mediante expresiones lógicas, que el estado del programa en un determinado punto de su ejecución debería cumplir con unas determinadas condiciones especificadas según el correcto comportamiento teórico del programa (véase 5.11).

En caso de que las condiciones especificadas en los asertos no se cumplan, entonces reflejan errores de programación, por lo que el aserto que no se cumpla *abortará* la ejecución del programa indicando la situación donde falló dicha comprobación del estado.

Para poder utilizar asertos, hay que incluir la biblioteca estándar `<cassert>`. Así, en el siguiente ejemplo, el aserto que expresa la pre-condición del subprograma fallará si su invocación no respeta las restricciones especificadas por la definición del mismo. Por ello es necesario que el programa que hace la invocación implemente algún mecanismo (en este caso la sentencia `if (...) { ... }`) que garantice que la llamada cumplirá la restricción especificada por la pre-condición. Por otra parte, si no se cumple el aserto que expresa la post-condición del subprograma, entonces habrá algún error en su programación que deberá ser subsanado.

```

#include <iostream>
#include <cassert>
using namespace std;
//-----
// retorna el cociente y resto resultado de dividir
// dividendo entre divisor
// Precond: (divisor != 0)
//-----
void dividir(int dividendo, int divisor, int& cociente, int& resto)
{
    assert(divisor != 0); // PRE-CONDICIÓN
    cociente = dividendo / divisor;
    resto = dividendo % divisor;
    assert(dividendo == (divisor * cociente + resto)); // POST-CONDICIÓN
}
//-----
int main()
{
    int dividendo, divisor, cociente, resto;
    cout << "Introduzca Dividendo y Divisor ";
    cin >> dividendo >> divisor;
    if (divisor == 0) {
        cout << "Error, división por cero no permitida" << endl;
    } else {
        dividir(dividendo, divisor, cociente, resto);
        cout << "Dividendo: " << dividendo << endl;
        cout << "Divisor: " << divisor << endl;
        cout << " Cociente: " << cociente << endl;
        cout << " Resto: " << resto << endl;
    }
}

```

Los asertos son útiles durante el desarrollo y depuración del programa, sin embargo se suelen desactivar cuando se han corregido *todos* los errores de programación y se implanta el software desarrollado. En *GNU GCC* es posible desactivar la comprobación de asertos mediante la siguiente directiva de compilación, que será utilizada para la generación del código final a implantar:

```
g++ -DNDEBUG -ansi -Wall -Werror -o programa programa.cpp
```

12.2. Situaciones Anómalas Excepcionales

Hay situaciones en las cuales el error (o comportamiento anómalo) no puede ser previsto y evitado con antelación, y debe ser gestionado adecuadamente por el programa en el momento en el que éste suceda, por ejemplo una entrada de datos errónea desde el exterior, un fichero para entrada de datos que no existe, un fichero para salida de datos que no puede ser creado, agotamiento de la reserva de memoria dinámica en caso de ser necesaria, una entrada de datos que excede de la capacidad de gestión del programa, etc.

Estas situaciones anómalas usualmente impiden a una determinada parte del software cumplir sus especificaciones y llevar a cabo sus tareas. Así, si el entorno del software donde se detecta la situación anómala tiene información y capacidad suficiente para tratar y gestionar dicha situación, de tal forma que después de dicho tratamiento le sea posible cumplir con sus especificaciones y llevar a cabo su tarea, entonces el tratamiento de la situación anómala se realizará en dicho lugar.

Por otra parte, si el entorno del software donde se detecta la situación anómala no tiene capacidad para su tratamiento, entonces deberá informar al *entorno externo superior* (normalmente el programa invocante) de que no pudo cumplir sus especificaciones por la existencia de una determinada situación anómala, un *error*. Dadas estas situaciones anómalas donde una determinada pieza de software detecta una determinada situación de error, existen diversos enfoques para informar al entorno externo superior:

1. Informar del error mediante un código de retorno (retorno de función o parámetro de salida). Este es el método usualmente utilizado en el lenguaje de programación C, su librería estándar y API del sistema operativo.

Este método es quizás el más fácil de gestionar “a priori”, donde el tratamiento de las situaciones de error son tratadas dentro del propio flujo de ejecución normal del programa. Sin embargo, la utilización de este método puede dar lugar a errores de programación debidos a que el software minimice e incluso ignore y elimine la gestión y tratamiento de determinados errores.

2. En caso de situación errónea, poner el estado de un determinado objeto en modo erróneo, de tal forma que el comportamiento del objeto refleje dicho estado. Este método es el seguido por los flujos de entrada y salida de la biblioteca estándar de C++.

Este método puede ser visto como una extensión del primer método anterior en un escenario de abstracción de datos (tipos abstractos de datos y programación orientada a objetos). En este caso, como en el anterior, el tratamiento de errores se encuentra integrado con el flujo de ejecución normal del programa, y además, también es vulnerable a errores de programación debidos a la posibilidad de que el software ignore el tratamiento de estos errores.

3. En caso de situación errónea, lanzar una determinada *excepción* especificando el error, interrumpiendo el flujo de ejecución normal del programa, de tal forma que sea *capturada* en un determinado entorno con capacidad e información suficiente para que pueda ser gestionada adecuadamente.

Este método es adecuado para el tratamiento de situaciones anómalas excepcionales, cuyo tratamiento debe ser gestionado de forma ajena al flujo de ejecución normal del programa. Este método proporciona un sistema robusto para el manejo de situaciones excepcionales, cuyo tratamiento no puede ser ignorado.

Estos métodos presentan diferentes enfoques para informar al entorno exterior de la existencia de un determinado error, así como diferentes enfoques para su tratamiento, el cual se puede gestionar dentro del propio flujo de ejecución normal del programa, o mediante un flujo alternativo ajeno al flujo de ejecución normal.

Así, integrar el tratamiento de errores dentro del flujo de ejecución normal del programa puede ser adecuado cuando la situación anómala se considera como parte integrada dentro del comportamiento normal del programa. Por otra parte, cuando la situación anómala es excepcional y requiere de un tratamiento excepcional al comportamiento normal del programa, entonces puede ser más adecuado gestionar estas situaciones en un flujo de ejecución diferente, alternativo al flujo de ejecución normal del programa. Este mecanismo, además, presenta la ventaja de que el software para el tratamiento de los errores está claramente diferenciado del software para la ejecución normal del programa.

Sin embargo, el desarrollo de software y la gestión de recursos en un entorno con presencia de excepciones puede resultar más complicado, debido a la existencia, a veces invisible, de un posible flujo de ejecución alternativo al flujo de ejecución normal del programa.¹

12.3. Gestión de Errores Mediante Excepciones

Las excepciones surgen como una forma de manejar situaciones de error excepcionales en los programas, no son por lo tanto un mecanismo para controlar el flujo de ejecución normal del programa, ni son un mecanismo para gestionar un error de programación. Son muy adecuadas porque permiten diferenciar el código correspondiente a la resolución del problema del código encargado de manejar situaciones anómalas.

Como norma general de diseño, un subprograma (o método) debe suponer que su precondición se cumple a su invocación, y así, en este caso, si no es capaz de realizar su tarea

¹RAII (cap. 22.1) es una técnica para la gestión automática de los recursos en presencia de excepciones.

y cumplir con su post-condición, entonces deberá lanzar una excepción. Normalmente ésto sucede cuando hay una situación excepcional de error que el subprograma no tiene información suficiente para manejar (resolver), por lo que deberá lanzar la excepción para que sea manejada a otro nivel superior.

Para lanzar (también conocido como elevar) una excepción desde una determinada parte del código, la siguiente sentencia lanza una excepción del tipo de **valor**, con el valor especificado:

```
throw valor;
```

Cuando se lanza una excepción, el flujo de ejecución se interrumpe y salta directamente al manejador que es capaz de capturarla según su tipo, retornando automáticamente de los subprogramas anidados donde se encuentre, destruyéndose todas las variables que han sido creadas en los ámbitos y subprogramas de los que se salga, hasta llegar al manejador.

Por otra parte, el código diseñado para gestionar el tratamiento de excepciones debe estar en un bloque **try/catch**, de tal forma que una excepción lanzada por un determinado código dentro del bloque **try** será capturada por el bloque **catch** correspondiente, según el tipo de la excepción lanzada. En caso de no existir un bloque **catch** del tipo adecuado, el manejador de excepciones buscará un bloque **catch** de tipo adecuado en un nivel externo superior. Si finalmente no se encontrase ningún bloque **catch** adecuado, el programa abortaría.

```
try {
    // código que puede lanzar una excepción
} catch (const Error1& e) {
    // código que maneja la excepción 'e' de tipo Error1
} catch (const Error2& e) {
    // código que maneja la excepción 'e' de tipo Error2
} catch ( ... ) { // captura cualquier excepción
    // código que maneja una excepción sin tipo especificado
}
```

Es así mismo también posible, dentro de un bloque **catch**, *relanzar* la excepción que está siendo manejada, para que pueda ser tratada en un nivel externo superior, de la siguiente forma:

```
} catch ( ... ) { // captura cualquier excepción
    // código que maneja una excepción sin tipo especificado
    throw; // relanza la excepción
}
```

El tipo de la excepción lanzada podrá ser simple o compuesto. Por ejemplo:

```
#include <iostream>
#include <cassert>
using namespace std;
//-----
enum Excepcion {          // Tipo de la Excepción
    DIVISION_POR_CERO
};
//-----
// retorna el cociente y resto resultado de dividir
// dividendo entre divisor
// Lanza excep. DIVISION_POR_CERO si (divisor == 0)
//-----
void dividir(int dividendo, int divisor, int& cociente, int& resto)
{
    if (divisor == 0) {
        throw DIVISION_POR_CERO;
    }
    cociente = dividendo / divisor;
    resto = dividendo % divisor;
```

```

    assert(dividendo == (divisor * cociente + resto)); // POST-CONDICIÓN
}
//-----
int main()
{
    try {
        int dividendo, divisor, cociente, resto;
        cout << "Introduzca Dividendo y Divisor ";
        cin >> dividendo >> divisor;
        dividir(dividendo, divisor, cociente, resto);
        cout << "Dividendo: " << dividendo << endl;
        cout << "Divisor: " << divisor << endl;
        cout << "  Cociente: " << cociente << endl;
        cout << "  Resto: " << resto << endl;
    } catch (const Excepcion& e) {
        switch (e) {
            case DIVISION_POR_CERO:
                cerr << "Error: División por Cero" << endl;
                break;
            default:
                cerr << "Error inesperado" << endl;
                break;
        }
    } catch ( ... ) {
        cerr << "Error inesperado" << endl;
    }
}

```

También es posible lanzar excepciones de tipos compuestos, los cuales podrán ser sin campos internos si sólo interesa el tipo de la excepción. Por ejemplo:

```

#include <iostream>
#include <cassert>
using namespace std;
//-----
struct Division_por_Cero {}; // Tipo de la Excepción
//-----
// retorna el cociente y resto resultado de dividir
// dividendo entre divisor
// Lanza excep. Division_por_Cero si (divisor == 0)
//-----
void dividir(int dividendo, int divisor, int& cociente, int& resto)
{
    if (divisor == 0) {
        throw Division_por_Cero();
    }
    cociente = dividendo / divisor;
    resto = dividendo % divisor;
    assert(dividendo == (divisor * cociente + resto)); // POST-CONDICIÓN
}
//-----
int main()
{
    try {
        int dividendo, divisor, cociente, resto;
        cout << "Introduzca Dividendo y Divisor ";
        cin >> dividendo >> divisor;
        dividir(dividendo, divisor, cociente, resto);
        cout << "Dividendo: " << dividendo << endl;
        cout << "Divisor: " << divisor << endl;
    }
}

```

```

        cout << " Cociente: " << cociente << endl;
        cout << " Resto: " << resto << endl;
    } catch (const Division_por_Cero& e) {
        cerr << "Error: División por Cero" << endl;
    } catch ( ... ) {
        cerr << "Error inesperado" << endl;
    }
}

```

Sin embargo, también es posible que las excepciones de tipos compuestos contengan valores que puedan informar al entorno superior de la naturaleza del error. Por ejemplo:

```

#include <iostream>
#include <cassert>
using namespace std;
//-----
struct Division_por_Cero {    // Tipo de la Excepción
    int dividendo;
    int divisor;
};
//-----
// retorna el cociente y resto resultado de dividir
// dividendo entre divisor
// Lanza excep. Division_por_Cero si (divisor == 0)
//-----
void dividir(int dividendo, int divisor, int& cociente, int& resto)
{
    if (divisor == 0) {
        Division_por_Cero e;
        e.dividendo = dividendo;
        e.divisor = divisor;
        throw e;
    }
    cociente = dividendo / divisor;
    resto = dividendo % divisor;
    assert(dividendo == (divisor * cociente + resto)); // POST-CONDICIÓN
}
//-----
int main()
{
    try {
        int dividendo, divisor, cociente, resto;
        cout << "Introduzca Dividendo y Divisor ";
        cin >> dividendo >> divisor;
        dividir(dividendo, divisor, cociente, resto);
        cout << "Dividendo: " << dividendo << endl;
        cout << "Divisor: " << divisor << endl;
        cout << " Cociente: " << cociente << endl;
        cout << " Resto: " << resto << endl;
    } catch (const Division_por_Cero& e) {
        cerr << "Error: División por Cero: "
            << e.dividendo << " / " << e.divisor << endl;
    } catch ( ... ) {
        cerr << "Error inesperado" << endl;
    }
}

```

Ⓐ Directrices en el Diseño del Tratamiento de Errores

Considerando el diseño del software, es muy importante tomar una adecuada decisión respecto a si una determinada circunstancia debe ser especificada mediante pre-condiciones (y por lo tanto gestionada dentro del flujo normal de ejecución del programa para evitar la invocación en dichas situaciones), o por el contrario no especificar la pre-condición y lanzar una excepción en el caso de que surja la situación. Normalmente, si la situación es excepcional dentro de las circunstancias de ejecución del programa, y requiere un tratamiento excepcional, entonces puede ser más adecuado considerar su tratamiento mediante excepciones. Sin embargo, si la situación es bastante usual dentro de las circunstancias de ejecución del programa, entonces probablemente requiera un tratamiento dentro del flujo de ejecución normal del programa. Una técnica usual de programación robusta respecto a la gestión de recursos en un entorno con excepciones es RAII (véase 22.1).

Nótese que es normalmente un síntoma de mal diseño el hecho de que el código del programa esté *repleto* de bloques `try/catch`, ya que en ese caso mostraría una situación donde dichas excepciones no son tan excepcionales, sino que forman parte del flujo normal de ejecución. No sería natural que cada llamada a un subprograma tuviera un bloque `try/catch` para capturar las excepciones lanzadas por éste, ya que normalmente dichas excepciones serán *excepcionales* y normalmente serán tratadas a otro nivel.

Por ejemplo, en el diseño de una estructura de datos *stack* (pila), es una situación usual comprobar si la pila está vacía, y en caso contrario, sacar un elemento de ella para su procesamiento. Por lo tanto, el caso de que la pila esté vacía es una situación normal, y se deberá tratar en el flujo de ejecución normal del programa. Sin embargo, es inusual el hecho de que la pila se llene, ya que normalmente se dimensiona lo suficientemente amplia para poder resolver un determinado problema. En el caso excepcional de que la pila se llene, este hecho será un escollo para resolver el problema, por lo que se puede indicar mediante una excepción.

```
//- fichero: stack.hpp -----
#ifndef _stack_hpp_
#define _stack_hpp_
#include <cassert>
#include <tr1/array>
namespace umalcc {
    const unsigned MAX_ELMS = 30;
    std::tr1::array<int, MAX_ELMS> Datos;
    struct Stack {
        unsigned nelms;
        Datos elm;
    };
    //-----
    struct Stack_Full {};          // Tipo de la Excepción
    //-----
    inline void init(Stack& s)
    {
        s.nelms = 0;
    }
    inline bool empty(const Stack& s) {
        return (s.nelms == 0);
    }
    inline void push(Stack& s, int e)
    {
        if (s.nelms >= s.elm.size()) {
            throw Stack_Full();
        }
        s.elm[s.nelms] = e;
        ++s.nelms;
    }
    inline int pop(Stack& s)
    {

```

```

        assert(! empty(s)); // Pre-condición
        --s.nelms;
        return s.elm[s.nelms];
    }
}
#endif
// - fin: stack.hpp -----

```

Una posible utilización del módulo `stack` definido anteriormente podría ser:

```

#include "stack.hpp"
#include <iostream>
using namespace std;
using namespace umalcc;
void almacenar_datos(Stack& s)
{
    for (int x = 0; x < 10; ++x) {
        push(s, x);
    }
}
void imprimir(Stack& s)
{
    while (! empty(s)) {
        cout << pop(s) << " ";
    }
    cout << endl;
}
int main()
{
    Stack s;
    try {
        init(s);
        almacenar_datos(s);
        imprimir(s);
    } catch (const Stack_Full& e) {
        cout << "Error: Pila Llena"<< endl;
    } catch ( ... ) {
        cout << "Error: Inesperado"<< endl;
    }
}

```

12.4. Excepciones Estándares

Las excepciones estándares que el sistema lanza están basadas en el tipo `exception` que se obtiene incluyendo el fichero `<exception>` y del cual podemos obtener un mensaje mediante la llamada a `what()`. Ejemplo:

```

try {
    . . .
} catch (const exception& e) {
    cout << "Error: " << e.what() << endl;
} catch ( ... ) {
    cout << "Error: Inesperado"<< endl;
}

```

Así mismo, el sistema tiene un número de excepciones predefinidas (derivadas del tipo `exception` anterior), que pueden, a su vez, ser base para nuevas excepciones definidas por el programador (cap. 17):

- `logic_error(str)`: especifican errores debidos a la lógica interna del programa. Son errores predecibles, detectables antes de que el programa se ejecute y por lo tanto evitables mediante chequeos adecuados en determinados lugares. (`#include <stdexcept>`)

`domain_error(str), invalid_argument(str), length_error(str), out_of_range(str)`

- `runtime_error(str)`: errores debidos a eventos más allá del ámbito del programa. Son errores impredecibles, sólo detectables cuando el programa se ejecuta y la única alternativa es su manejo en tiempo de ejecución. (`#include <stdexcept>`)

`range_error(str), overflow_error(str), underflow_error(str)`

- `bad_alloc()` lanzada en fallo en `new` (`#include <new>`)
- `bad_cast()` lanzada en fallo en `dynamic_cast` (`#include <typeinfo>`)
- `bad_typeid()` lanzada en fallo en `typeid` (`#include <typeinfo>`)
- `bad_exception()` lanzada en fallo en la especificación de excepciones lanzadas por una función. (`#include <exception>`)
- `ios::failure()` lanzada en fallo en operaciones de Entrada/Salida. (`#include <iostream>`)

- Ⓐ El programador puede extender este conjunto de excepciones derivando nuevas clases basadas en estas excepciones estándares.
- Ⓐ Nota: para ver una descripción de los requisitos que deben cumplir las clases para comportarse adecuadamente en un entorno de manejo de excepciones véase [13.3](#)

```
#include <iostream>
#include <stdexcept>
#include <cassert>
using namespace std;
//-----
// retorna el cociente y resto resultado de dividir
// dividendo entre divisor
// Lanza excep. domain_error si (divisor == 0)
//-----
void dividir(int dividendo, int divisor, int& cociente, int& resto)
{
    if (divisor == 0) {
        throw domain_error("división por cero");
    }
    cociente = dividendo / divisor;
    resto = dividendo % divisor;
    assert(dividendo == (divisor * cociente + resto)); // POST-CONDICIÓN
}
//-----
int main()
{
    try {
        int dividendo, divisor, cociente, resto;
        cout << "Introduzca Dividendo y Divisor ";
        cin >> dividendo >> divisor;
        dividir(dividendo, divisor, cociente, resto);
        cout << "Dividendo: " << dividendo << endl;
        cout << "Divisor: " << divisor << endl;
        cout << "Cociente: " << cociente << endl;
        cout << "Resto: " << resto << endl;
    } catch (const domain_error& e) {
        cerr << e.what() << endl;
    }
}
```

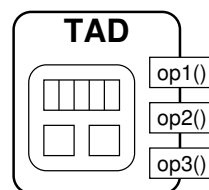
```
    } catch (const exception& e) {  
        cerr << e.what() << endl;  
    } catch ( ... ) {  
        cerr << "Error inesperado" << endl;  
    }  
}
```

Capítulo 13

Tipos Abstractos de Datos

A medida que aumenta la complejidad del problema a resolver, del mismo modo deben aumentar los niveles de abstracción necesarios para diseñar y construir su solución algorítmica. Así, la abstracción procedimental permite aplicar adecuadamente técnicas de *diseño descendente* y *refinamientos sucesivos* en el desarrollo de algoritmos y programas. La programación modular permite aplicar la abstracción a mayor escala, permitiendo abstraer sobre conjuntos de operaciones y los datos sobre los que se aplican. De esta forma, a medida que aumenta la complejidad del problema a resolver, aumenta también la complejidad de las estructuras de datos necesarias para su resolución, y este hecho requiere, así mismo, la aplicación de la abstracción a las estructuras de datos.

La aplicación de la abstracción a las estructuras de datos da lugar a los *Tipos Abstractos de Datos* (TAD), donde se especifica el concepto que representa un determinado tipo de datos, y la semántica (el significado) de las operaciones que se le pueden aplicar, pero donde su representación e implementación internas permanecen ocultas e inaccesibles desde el exterior, de tal forma que no son necesarias para su utilización. Así, podemos considerar que un tipo abstracto de datos *encapsula* una determinada estructura abstracta de datos, impidiendo su manipulación directa, permitiendo solamente su manipulación a través de las operaciones especificadas. De este modo, los tipos abstractos de datos proporcionan un mecanismo adecuado para el diseño y reutilización de software fiable y robusto.



Para un determinado tipo abstracto de datos, se pueden distinguir tres niveles:

- Nivel de utilización, donde se utilizan objetos de un determinado tipo abstracto de datos, basándose en la especificación del mismo, de forma independiente a su implementación y representación concretas. Así, estos objetos se manipulan mediante la invocación a las operaciones especificadas en el TAD.
- Nivel de especificación, donde se especifica el tipo de datos, el concepto abstracto que representa y la semántica y restricciones de las operaciones que se le pueden aplicar. Este nivel representa el *interfaz* público del tipo abstracto de datos.
- Nivel de implementación, donde se define e implementa tanto las estructuras de datos que soportan la abstracción, como las operaciones que actúan sobre ella según la semántica especificada. Este nivel interno permanece privado, y no es accesible desde el exterior del tipo abstracto de datos.

Utilización de TAD
Especificación de TAD
Implementación de TAD

Nótese que para una determinada especificación de un tipo abstracto de datos, su implementación puede cambiar sin que ello afecte a la utilización del mismo.

13.1. Tipos Abstractos de Datos en C++: Clases

En el lenguaje de programación C++, las clases dan la posibilidad al programador de definir tipos abstractos de datos que se comporten de igual manera que los tipos predefinidos, de tal forma que permiten definir su representación interna (compuesta por sus atributos miembros), la forma en la que se crean y se destruyen, como se asignan y se pasan como parámetros, las conversiones de tipos aplicables, y las operaciones que se pueden aplicar (denominadas funciones miembros o simplemente métodos). De esta forma se hace el lenguaje extensible. Así mismo, la definición de tipos abstractos de datos mediante clases puede ser combinada con la definición de módulos (véase 11), haciendo de este modo posible la reutilización de estos nuevos tipos de datos.

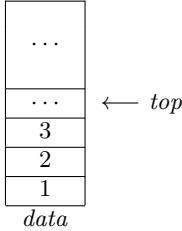
13.1.1. Definición e Implementación de Clases “en Línea”

En el caso de definición de clases simples, es posible realizar la definición e implementación de la clase *en línea*. Para ello se especifica la palabra reservada `class` seguida por el nombre del nuevo tipo que se está definiendo, y entre llaves la definición de los atributos (miembros) que lo componen y de los métodos (funciones miembros) que se le pueden aplicar directamente. Por ejemplo, el TAD *pila de números enteros* representa el siguiente concepto abstracto:

Una *pila* es una colección ordenada (según el orden de inserción) de elementos homogéneos donde se pueden introducir elementos (manteniendo el orden de inserción) y sacar elementos de ella (en orden inverso al orden de inserción), de tal forma que el primer elemento que sale de la pila es el último elemento que ha sido introducido en ella. Además, también es posible comprobar si la pila contiene elementos, de tal forma que no se podrá sacar ningún elemento de una pila vacía.

y se puede definir la clase `Stack` dentro del espacio de nombres `umalcc` de la siguiente forma:¹

```
//-stack.hpp -----
#ifndef _stack_hpp_
#define _stack_hpp_
#include <tr1/array>
#include <cassert>
#include <stdexcept>
namespace umalcc {
    class Stack {
    private:
        // -- Constantes Privadas --
        static const unsigned MAX = 256;
        // -- Tipos Privados --
        typedef std::tr1::array<int, MAX> Datos;
        // -- Atributos Privados --
        unsigned top;
        Datos data;
    public:
        // -- Constructor por Defecto Público --
        Stack() : top(0), data() {}
        // -- Métodos Públicos --
        bool empty() const {
            return (top == 0);
        }
        void push(int e) {
            if (top >= data.size()) {
                throw std::length_error("Stack::push length error");
            }
            data[top] = e;
        }
    };
}
```



¹Nótese que el delimitador punto y coma (;) debe especificarse después de la llave de cierre de la definición de la clase.

```

        ++top;
    }
    int pop() {          // Pre-condición: (! empty())
        assert(! empty());
        --top;
        return data[top];
    }
};                      // El delimitador ; termina la definición
}
#endif
//-----

```

Zona Privada y Zona Pública

En la definición de una clase, se pueden distinguir dos ámbitos de visibilidad (accesibilidad), la parte *privada*, cuyos miembros sólo serán accesibles desde un ámbito *interno* a la propia clase, y la parte *pública*, cuyos miembros son accesibles tanto desde un ámbito *interno* como desde un ámbito *externo* a la clase.

La parte privada comprende desde el principio de la definición de la clase hasta la etiqueta **public**, y la parte pública comprende desde esta etiqueta hasta que se encuentra otra etiqueta **private**. Cada vez que se especifica una de las palabras reservadas **public** o **private**, las declaraciones que la siguen adquieren el atributo de visibilidad dependiendo de la etiqueta especificada.

En este manual usualmente se definirá primero la parte pública de la clase, y se definirá posteriormente la parte privada del mismo, aunque en este primer ejemplo, con objeto de facilitar la explicación y comprensión del mismo, se ha definido primero la parte privada y posteriormente la parte pública. Nótese que con respecto a la implementación *en línea* de los métodos de la clase, el orden en el que se declaran los miembros (constantes, tipos, atributos y métodos) de la clase es irrelevante.

Constantes y Tipos de Ámbito de Clase

Las constantes de ámbito de clase se definen especificando los cualificadores **static const**, seguidos por el tipo, el identificador y el valor de la constante. Estas constantes serán accesibles por todos las instancias (objetos) de la clase. En este ejemplo, se define la constante **MAX** con un valor de 256.

También se pueden definir tipos internos de ámbito de clase de igual forma a como se hace externamente a la clase. En este ejemplo se define un tipo **Datos** como un array de 256 números enteros. Estos tipos serán útiles en la definición de los atributos miembros de la clase, o para definir elementos auxiliares en la implementación del tipo abstracto de datos.

Atributos

Los atributos componen la representación interna de la clase, y se definen de igual forma a los campos de los registros (véase 6.3). En nuestro ejemplo, el atributo **top** es un número natural que representa el índice de la cima de la pila, y el atributo **data** es un array de números enteros que contiene los elementos almacenados en la pila, de tal forma que un nuevo elemento se inserta en la cima de la pila, y el último elemento se extrae también desde la cima, incrementando o disminuyendo la cima de la pila en estas operaciones.

De igual modo a los registros y sus campos, cada instancia de la clase (objeto) que se defina almacenará su propia representación interna de los atributos de forma independiente a las otras instancias de la clase (véase el apartado más adelante referido a *instancias de clase: objetos*).

Constructores

El *constructor* de una clase permite construir e inicializar una instancia de la clase (un *objeto*). En este ejemplo se ha definido el constructor por defecto, que es utilizado como mecanismo por defecto para construir objetos de este tipo cuando no se especifica otro método de construcción.

Los constructores se denominan con el mismo identificador de la clase, seguidamente se especifican entre paréntesis los parámetros necesarios para la construcción, que en el caso del constructor por defecto, serán vacíos. Seguidamente, tras el delimitador (`:`) se especifica la *lista de inicialización*, donde aparecen según el orden de declaración todos los atributos miembros del objeto, así como los valores que toman especificados entre paréntesis (se invoca al constructor adecuado según los parámetros especificados entre paréntesis, de tal forma que los paréntesis vacíos representan la construcción por defecto). A continuación se especifican entre llaves las sentencias pertenecientes al cuerpo del constructor para realizar las acciones adicionales necesarias para la construcción del objeto.

El lenguaje de programación C++ definirá por defecto el constructor de copia, que permite inicializar una variable de tipo `stack` con una copia del valor de otra variable del mismo tipo. Así mismo, también definirá por defecto la operación de asignación como una copia entre variables del mismo tipo.

Métodos Generales y Métodos Constantes

Los métodos de una clase pueden tener el cualificador `const` especificado después de los parámetros, en cuyo caso indica que el método no modifica el estado interno del objeto, por lo que se puede aplicar tanto a objetos constantes como variables. Sin embargo, si dicho cualificador no aparece entonces significa que el método sí modifica el estado interno del objeto, por lo que sólo podrá ser aplicado a objetos variables, y por el contrario no podrá ser aplicado a objetos constantes. Por ejemplo, el método `empty()` que comprueba si la pila está vacía, no modifica el estado del objeto, y por lo tanto se define como un método constante.

El método `push(...)` inserta un elemento en la cima de la pila, salvo en el caso excepcional de que la pila esté llena, en cuyo caso lanza una excepción. Por otra parte, el método `pop()` extrae y retorna el elemento de la cima de la pila, con la pre-condición de que la pila no debe estar vacía. Por lo tanto es responsabilidad del programador realizar las comprobaciones oportunas antes de invocar a este método.

Desde la implementación interna de una clase, se pueden utilizar internamente, mediante sus identificadores, tanto las constantes, tipos, atributos y métodos de esa clase. En caso de utilización de atributos e invocación de métodos, se corresponden con los atributos y métodos aplicados a la instancia de la clase a la que se haya aplicado el método en concreto. Así, cuando se utilizan los atributos `data` y `top` en el ejemplo, se está accediendo a esos atributos para la instancia de la clase a la que le sea aplicado el método correspondiente.

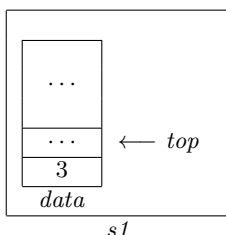
Por ejemplo, si se invoca al método `pop()` aplicado a un determinado objeto `s1` de tipo `Stack` mediante el operador punto `s1.pop()` (véase el siguiente apartado), entonces la implementación de `pop()` se aplica a los atributos `data` y `top` correspondientes a la variable `s1`. Sin embargo, si se aplica el mismo método `pop()` a otro objeto `s2` de tipo `Stack`, entonces la implementación de `pop()` se aplica a los atributos `data` y `top` correspondientes a la variable `s2`.

Instancias de Clase: Objetos

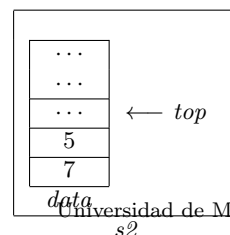
Un objeto es una instancia de una clase, y podremos definir tantos objetos cuyo tipo sea de una determinada clase como sea necesario, de tal modo que cada objeto contiene su propia representación interna de forma independiente del resto. Dado un determinado objeto, se podrá manipular externamente invocando a sus métodos públicos mediante el operador punto (`.`). Estos métodos manipularán la representación interna (atributos) de la instancia (objeto) de la clase a la que sean aplicados. Por ejemplo el siguiente código define dos objetos (`s1` y `s2`) como instancias de la clase `Stack`, es decir, define dos objetos de tipo `Stack` y le aplica varios métodos que lo manipulan:

```
int main()
{
    Stack s1, s2;
    s1.push(3);
    s1.push(4);
    if ( ! s1.empty() ) {
```

4



s1



s2

```

        int x = s1.pop();
    }
    s2.push(7);
    s2.push(5);

    Stack s3 = s1; // construcción por copia
    s3 = s2;       // asignación por copia
}

```

Otro ejemplo de utilización de la clase `Stack` definida anteriormente puede ser el siguiente, donde se define en `main()` un objeto `s` de tipo `Stack`, que se construye mediante el constructor por defecto, y se pasa como parámetro por referencia a subprogramas, en los cuales se manipula el objeto `s` invocando a sus métodos públicos. Nótese como la invocación al método `pop()` está precedida por una pregunta para garantizar que la pila no esté vacía, así como la manipulación de la pila se encuentra englobada dentro de un bloque `try/catch` para capturar la posible excepción en caso de que se llene la pila.

```

#include "stack.hpp"
#include <iostream>
using namespace std;
using namespace umalcc;
void almacenar_datos(Stack& s)
{
    for (int x = 0; x < 10; ++x) {
        s.push(x);
    }
}
void imprimir(Stack& s)
{
    while (! s.empty() ) {
        cout << s.pop() << " ";
    }
    cout << endl;
}
int main()
{
    try {
        Stack s;
        almacenar_datos(s);
        imprimir(s);
    } catch (const length_error& e) {
        cout << "Error: " << e.what() << endl;
    } catch ( ... ) {
        cout << "Error: Inesperado"<< endl;
    }
}

```

13.1.2. Definición de Clases e Implementación Separada

El lenguaje de programación C++ también permite realizar por separado tanto la definición de una clase por una parte, como su implementación por otra parte, y distribuir el interfaz de la clase en un fichero de encabezamiento, y la implementación de la clase en un fichero de implementación.

Vamos a definir el interfaz y la implementación del tipo abstracto de datos *Vector* como un array incompleto de números enteros, al cual se le pueden añadir elementos hasta un determinado límite. Algunos métodos simples se definirán *en línea*, mientras que otros métodos más complejos se definirán aparte en el fichero de implementación.

```

//- vector.hpp -----

```

```

#ifndef _vector_hpp_
#define _vector_hpp_
#include <iostream>
#include <tr1/array>
#include <cassert>
namespace umalcc {
    class Vector {
    public:
        ~Vector() {} // destructor
        Vector() : sz(0), v() {} // constructor por defecto
        Vector(const Vector& orig); // constructor de copia
        Vector(const Vector& orig, unsigned i, unsigned nelms = MAX); // constructor
        explicit Vector(unsigned n); // constructor explícito
        Vector& operator =(const Vector& orig); // asignación
        unsigned size() const { return sz; };
        void clear() { sz = 0; }
        void push_back(int e);
        void pop_back(); // Pre-condición: (size() > 0)
        const int& operator [] (unsigned i) const { // Pre-condición: (i < size())
            assert(i < size());
            return v[i];
        }
        int& operator [] (unsigned i) { // Pre-condición: (i < size())
            assert(i < size());
            return v[i];
        }
        Vector subvector(unsigned i, unsigned nelms = MAX) const;
        Vector& operator +=(const Vector& vect);
        Vector& operator +=(int e) { push_back(e); return *this; }
        friend Vector operator + (const Vector& v1, const Vector& v2);
        friend bool operator == (const Vector& v1, const Vector& v2);
        friend bool operator < (const Vector& v1, const Vector& v2);
        friend bool operator != (const Vector& v1, const Vector& v2) { return ! (v1 == v2); }
        friend bool operator > (const Vector& v1, const Vector& v2) { return (v2 < v1); }
        friend bool operator >= (const Vector& v1, const Vector& v2) { return ! (v1 < v2); }
        friend bool operator <= (const Vector& v1, const Vector& v2) { return ! (v2 < v1); }
        friend std::ostream& operator << (std::ostream& out, const Vector& vect);
        friend std::istream& operator >> (std::istream& in, Vector& vect);
    private:
        static const unsigned MAX = 256;
        typedef std::tr1::array<int, MAX> Datos;
        // -- Atributos --
        unsigned sz;
        Datos v;
    };
}
#endif
//-----

```

Constantes y Tipos de Ámbito de Clase

Se ha visto anteriormente como es posible definir tipos y constantes de ámbito de clase. Estas constantes son compartidas por todas las instancias de la clase (objetos), y es por ello por lo que es necesario poner la palabra reservada **static** delante de la definición de la constante, para indicar que esa constante es única y compartida por todas las instancias de la clase.

Atributos

Los atributos componen la representación interna de la clase, de tal forma que cada objeto se compone de un número natural (**sz**) que indica cuantos elementos tiene almacenados, y de un array (**v**) donde se almacenan los elementos de forma consecutiva desde el comienzo del mismo. Es importante recordar que cada instancia de la clase (objeto) contiene su propia representación interna (atributos) independiente de las otras instancias.

Destructor

El *destructor* de la clase se define mediante el símbolo `~` seguido del identificador de la clase y una lista de parámetros vacía (`~Vector()`).

Este destructor será invocado automáticamente (sin parámetros actuales) para una determinada instancia (objeto) de esta clase cuando dicho objeto deba ser destruido, normalmente sucederá cuando el flujo de ejecución del programa salga del ámbito de visibilidad de dicho objeto.

```
{
    Vector v1; // construcción del objeto 'v1'
    ...       // utilización del objeto 'v1'
}
```

En el cuerpo del destructor se especificarán las sentencias necesarias para destruir el objeto, tales como liberaciones de los recursos que contenga, etc. Posteriormente, el destructor invoca automáticamente a los destructores de los atributos miembros del objeto para que éstos sean destruidos. En el caso del ejemplo no es necesario realizar ninguna acción adicional, por lo que el cuerpo del destructor se define en línea como vacío. Esto último es equivalente a no definir el destructor de la clase, en cuyo caso el compilador lo define automáticamente como vacío. Nótese que un destructor nunca deberá lanzar excepciones.

Constructores

Los *constructores* de la clase especifican las diferentes formas de crear (construir) un determinado objeto y establecen su *invariante*. Los constructores se denominan con el mismo identificador de la clase, seguidamente se especifican entre paréntesis los parámetros necesarios para la construcción.

Para su implementación, tras el delimitador (`:`) se especifica la lista de inicialización, donde aparecen según el orden de declaración todos los atributos miembros del objeto, así como los valores que toman especificados entre paréntesis (se invoca al constructor adecuado según los parámetros especificados entre paréntesis, de tal forma que los paréntesis vacíos representan la construcción por defecto). A continuación se especifican entre llaves las sentencias pertenecientes al cuerpo del constructor para realizar las acciones adicionales necesarias para la construcción del objeto.

Para una determinada clase, pueden existir múltiples definiciones de constructores, diferenciándose entre ellos por los parámetros. De entre todos ellos, hay dos constructores especiales: el constructor por defecto, y el constructor de copia.

Constructor por Defecto

El constructor por defecto no recibe ningún parámetro y define como se creará un objeto de dicha clase si no se especifica ninguna forma explícita de construcción. Si el programador no define ningún constructor, entonces el compilador definirá automáticamente un constructor por defecto que invoque automáticamente a los constructores por defecto de cada atributo miembro del objeto. En nuestro ejemplo, el constructor por defecto se ha implementado en línea debido a su simplicidad.

Se puede crear un objeto de tipo **Vector** construido por defecto de la siguiente forma:

```
Vector v1;
```

o también:

```
Vector v2 = Vector();
```

Nótese como en este segundo ejemplo se invoca directamente al constructor para construir un objeto.

Constructor de Copia

El constructor de copia recibe como parámetro un objeto de la misma clase y especifica como crear un nuevo objeto que sea una copia del objeto que se recibe. Este constructor se utiliza en las inicializaciones de objetos de este tipo y en el *paso de parámetros por valor* a subprogramas (aunque este caso está desaconsejado, prefiriéndose en esta situación el paso por referencia constante). Si el programador no define explícitamente el constructor de copia, entonces el compilador define automáticamente un constructor de copia que realiza una copia individual de todos los atributos del objeto, invocando implícitamente a los constructores de copia de cada atributo miembro.

Por ejemplo, se puede crear un objeto de tipo **Vector** mediante copia de otro objeto de tipo **Vector** de la siguiente forma:

```
Vector v3(v1);
```

o también:

```
Vector v4 = v1;
```

o también:

```
Vector v5 = Vector(v1);
```

Nótese como en este último ejemplo se invoca directamente al constructor para construir un objeto.

Ⓐ Constructores Explícitos

Si se especifica la palabra reservada **explicit** delante de la declaración de un constructor, entonces se elimina la posibilidad de que el constructor se aplique de forma implícita, haciendo necesario que sea invocado explícitamente. De hecho, es una forma de evitar conversiones de tipo implícitas, y además, si el especificador **explicit** se aplica al constructor de copia, entonces se evita el paso de parámetros por valor.

En este ejemplo, se define un constructor explícito del tipo **Vector** con un parámetro de tipo **unsigned**, de tal forma que en caso de ser invocado, creará un vector inicial con tantos elementos (con valor inicial cero) como especifique dicho parámetro. Por ejemplo, para crear un vector con cinco elementos nulos se puede hacer de la siguiente forma:

```
Vector v6(5);
```

o también:

```
Vector v7 = Vector(5);
```

Sin embargo, si no se hubiese cualificado el constructor como **explicit**, entonces el siguiente código también sería válido, aunque en nuestro ejemplo no lo es, ya que ha sido declarado **explicit**.

```
Vector v8 = 5; // Error: requiere que el constructor no sea explicit
```

Operador de Asignación

El operador de asignación define como se realiza esta operación para objetos de esta clase. En caso de que el programador no defina este método, entonces el compilador define automáticamente un operador de asignación que invoca automáticamente al operador de asignación para cada atributo miembro de la clase.

No se debe confundir el operador de asignación con el constructor de copia, ya que el constructor de copia construye un nuevo objeto que no tiene previamente ningún valor, mientras que en el caso del operador de asignación, el objeto ya tiene previamente un valor que deberá ser sustituido por

el nuevo valor. Este valor previo deberá, en ocasiones, ser destruido antes de realizar la asignación del nuevo valor.

- Ⓐ En el lenguaje de programación C++, el operador de asignación devuelve una *referencia* al propio objeto que está siendo asignado. Esto es así porque se permite encadenar sucesivas asignaciones en la misma sentencia, aunque realizar este encadenamiento puede traer consigo problemas de legibilidad, por lo que en general se desaconseja su utilización.
- Ⓐ Nótese que devolver una referencia es diferente de la devolución normal donde se devuelve un determinado *valor temporal*. En caso de devolver una referencia, se devuelve el propio objeto y se le puede aplicar un método directamente, o incluso puede serle asignado un nuevo valor (puede aparecer en la parte izquierda de una sentencia de asignación).

Métodos y Operadores Generales y Constantes

Como se explicó anteriormente, los métodos de una clase pueden tener el cualificador **const** especificado después de los parámetros, en cuyo caso indica que el método no modifica el estado interno del objeto, por lo que se puede aplicar tanto a objetos constantes como variables. Sin embargo, si dicho cualificador no aparece entonces significa que el método sí modifica el estado interno del objeto, por lo tanto sólo podrá ser aplicado a objetos variables, y por el contrario no podrá ser aplicado a objetos constantes.

El método constante **size()** no modifica el estado interno del objeto, y devuelve el número de elementos actuales almacenados en el vector. En nuestro ejemplo, se ha implementado en línea debido a su simplicidad.

El método **subvector(...)** devuelve un nuevo vector creado con los elementos del vector actual desde una determinada posición, un número determinado de elementos.

El método **clear()** re-inicializa el vector al estado vacío, es decir, sin elementos almacenados. El método **push_back()** permite añadir un elemento al final del vector, mientras que el método **pop_back()** elimina el último elemento del mismo.

Por otra parte, el operador de incremento (**+=**) se define para añadir elementos al vector, de tal forma que añadir un sólo elemento es equivalente al método **push_back(...)**, mientras que también existe la posibilidad de añadir los elementos que contenga otro vector. Este operador de incremento devuelve una referencia a este objeto, de igual manera que lo hace el operador de asignación.

Así, los métodos de una clase se aplican a los objetos mediante el operador punto. Por ejemplo:

```
Vector v1(7);
v1.push_back(5);
Vector v2 = v1.subvector(3, 2);
```

mientras que los operadores se aplican utilizando la notación correspondiente (infija, post-fija o prefija) según el operador especificado. Por ejemplo:

```
Vector v1(7);
v1.push_back(5);
Vector v2;
v2 = v1.subvector(3, 2);
v2 += Vector(3);
```

Ⓐ Métodos y Operadores que Devuelven Referencias

El ejemplo muestra como se define el operador de indexación (**[]**) para el tipo **Vector**, el cual se define de forma independiente para ser aplicado a objetos constantes y para ser aplicado a objetos variables. Ambos reciben como parámetro un índice, y devuelven una *referencia* al elemento que ocupa dicha posición en el vector (una *referencia constante* en caso de ser aplicado a un objeto constante).

Como se ha explicado anteriormente, devolver una referencia es diferente de la devolución normal donde se devuelve un determinado *valor temporal*. En caso de devolver una referencia, se devuelve el propio objeto y se le puede aplicar un método directamente, o incluso, en caso de

no ser una *referencia constante*, puede serle asignado un nuevo valor (puede aparecer en la parte izquierda de una sentencia de asignación).

Ambos métodos devuelven una referencia a un elemento, este hecho hace posible que el resultado de dicho método pueda aparecer en la parte izquierda de una asignación, por ejemplo

```
v1[3] = 5;
```

además de poder utilizarse para obtener su valor como en

```
int x = v1[3];
```

Es importante tener en cuenta que un subprograma o método no debe devolver una referencia a un objeto local, ya que su ámbito desaparece, y en ese caso se devolvería una referencia a un objeto que no existe.

Ⓐ Subprogramas y Operadores Amigos

Los subprogramas *amigos* (*friend*) son subprogramas y operadores externos a la definición de la clase, es decir, no son métodos pertenecientes a la clase. Por lo tanto se utilizan como subprogramas externos y no se aplican al objeto mediante el operador punto.

Sin embargo, al ser subprogramas *amigos*, tienen acceso a las definiciones privadas internas de la clase, tales como su representación interna (atributos miembros), métodos privados, etc., un hecho que no sería posible en caso de no ser *amigos*.

Los subprogramas amigos que se definen son para la concatenación de vectores, que devuelve un nuevo vector resultado de la concatenación de los vectores recibidos como parámetros, los operadores relacionales que devuelven un valor lógico dependiendo del resultado de la comparación, y los operadores de entrada y salida. Nótese que los operadores se utilizan según su notación estándar. Un posible ejemplo de su utilización podría ser:

```
Vector v1(7);
v1.push_back(5);
Vector v2;
v2 = v1.subvector(3, 2);
Vector v3 = v1 + v2;
if (v1 < v2) {
    cout << "Menor" << endl;
}
cout << "Introduzca el contenido de un vector entre llaves:" << endl;
cin >> v1;
cout << "Vector: " << v1 << endl;
```

Implementación Separada

En este ejemplo, realizaremos la implementación de los métodos de la clase **Vector** en un fichero independiente, diferenciando claramente los tres niveles de los tipos abstractos de datos (interfaz, implementación y utilización). Además, se facilita la ocultación de la implementación, el proceso de compilación separada, y la distribución del módulo de biblioteca.

Se deberán definir los métodos, operadores y amigos de la clase en el mismo espacio de nombres donde está definida la clase, y el identificador de cada método y operador debe ser cualificado con el identificador de la clase a la que corresponde (utilizando el delimitador `::`). No obstante, los subprogramas y operadores amigos no serán cualificados, ya que no son miembros de la clase.

En nuestro ejemplo, la implementación del módulo del tipo abstracto de datos **Vector** comienza con un *espacio de nombres anónimo* donde se definen los subprogramas privados necesarios para la implementación del módulo. Dentro del espacio de nombres `umalcc` se definen los métodos de la clase **Vector**. Para ello, cada nombre de método o operador se cualifica con el nombre de la clase a la que pertenece (**Vector** seguido por el delimitador `::`).

```

//vector.cpp -----
#include "vector.hpp"
#include <stdexcept>
namespace { // anónimo. zona privada al módulo
    inline unsigned minimo(unsigned x, unsigned y)
    {
        return (x < y) ? x : y ;
    }
}
namespace umalcc {
    Vector::Vector(const Vector& orig)
        : sz(0), v()
    {
        for (sz = 0; sz < orig.sz; ++sz) {
            v[sz] = orig.v[sz];
        }
    }
    Vector::Vector(const Vector& orig, unsigned inicio, unsigned nelms)
        : sz(0), v()
    {
        if (inicio > orig.sz) {
            throw std::out_of_range("Vector::Vector out of range");
        }
        unsigned n = minimo(nelms, orig.sz - inicio);
        for (sz = 0; sz < n; ++sz) {
            v[sz] = orig.v[inicio + sz];
        }
    }
    Vector::Vector(unsigned n)
        : sz(0), v()
    {
        if (n > v.size()) {
            throw std::length_error("Vector::Vector length error");
        }
        for (sz = 0; sz < n; ++sz) {
            v[sz] = 0;
        }
    }
    Vector& Vector::operator = (const Vector& orig)
    {
        if (this != &orig) { // comprobar autoasignación
            // destruir el valor antiguo y copiar el nuevo
            for (sz = 0; sz < orig.sz; ++sz) {
                v[sz] = orig.v[sz];
            }
        }
        return *this;
    }
    void Vector::push_back(int e) {
        if (sz >= v.size()) {
            throw std::length_error("Vector::push_back length error");
        }
        v[sz] = e;
        ++sz;
    }
    void Vector::pop_back() {
        assert(size() > 0); // Pre-condición
        --sz;
    }
}

```

```

Vector Vector::subvector(unsigned inicio, unsigned nelms) const
{
    return Vector(*this, inicio, nelms);
}
Vector& Vector::operator += (const Vector& vect)
{
    if (sz + vect.sz > v.size()) {
        throw std::length_error("Vector::operator+= length error");
    }
    unsigned nelms = vect.sz; // para considerar autoasignación
    for (unsigned i = 0; i < nelms; ++i) {
        v[sz] = vect.v[i];
        ++sz;
    }
    return *this;
}
Vector operator + (const Vector& v1, const Vector& v2)
{
    Vector res = v1;
    res += v2;
    return res;
}
bool operator == (const Vector& v1, const Vector& v2)
{
    bool res;
    if (v1.sz != v2.sz) {
        res = false;
    } else {
        unsigned i;
        for (i = 0; (i < v1.sz)&&(v1.v[i] == v2.v[i]); ++i) {
            // vacío
        }
        res = (i == v1.sz);
    }
    return res;
}
bool operator < (const Vector& v1, const Vector& v2)
{
    unsigned i;
    for (i = 0; (i < v1.sz)&&(i < v2.sz)
        &&(v1.v[i] == v2.v[i]); ++i) {
        // vacío
    }
    return (((i == v1.sz) && (i < v2.sz))
        || ((i < v1.sz) && (i < v2.sz) && (v1.v[i] < v2.v[i])));
}
std::ostream& operator << (std::ostream& out, const Vector& vect)
{
    out << vect.sz << " ";
    for (unsigned i = 0; i < vect.sz; ++i) {
        out << vect.v[i] << " ";
    }
    return out;
}
std::istream& operator >> (std::istream& in, Vector& vect)
{
    vect.clear();
    unsigned nelms;
    in >> nelms;

```

```

        for (unsigned i = 0; (i < nelms)&&(!in.fail()); ++i) {
            int x;
            in >> x;
            if ( ! in.fail() ) {
                vect.push_back(x);
            }
        }
        return in;
    }
}
//-----

```

La implementación de los constructores, los métodos y los operadores sigue la misma pauta que lo explicado anteriormente, considerando que cuando se hace referencia a un determinado atributo o método, éstos se refieren (o aplican) a los del propio objeto sobre los que se aplique el método en cuestión.

Además, en caso de que un objeto de la misma clase se reciba como parámetro, entonces es posible acceder a sus atributos e invocar a sus métodos mediante el operador punto (`.`), como por ejemplo en el constructor de copia de la clase `Vector`, donde se accede a los atributos `sz` y `v` del objeto recibido como parámetro (`orig`) de la siguiente forma: `orig.sz` y `orig.v[...]`.

Implementación de Constructores

El constructor de copia se define creando un vector vacío en la lista de inicialización, donde se inicializa cada atributo, según el orden en que han sido declarados, con el valor adecuado invocando al constructor especificado.²

Posteriormente se van asignando los elementos del vector especificado como parámetro (`orig`) al vector que se está construyendo. Nótese que se accede a los atributos del vector especificado como parámetro utilizando el operador punto (`orig.sz` y `orig.v[...]`), sin embargo, se accede a los atributos del vector que se está creando especificando únicamente sus identificadores (`sz` y `v`).

También se define una constructor que copia una parte del contenido de otro vector, en este caso, la implementación comprueba que los valores especificados son correctos y copia los elementos especificados del vector `orig` al vector que se está construyendo.

En la implementación separada del constructor explícito no se debe utilizar el especificador `explicit`, ya que fue especificada durante la definición de la clase. Esta implementación comprueba que los valores especificados como parámetros son correctos, y añade al vector el número de elementos especificado con el valor cero (0).

Implementación del Operador de Asignación

En la implementación del operador de asignación hay que tener en cuenta las siguientes consideraciones:

1. Hay que comprobar y evitar que se produzca una auto-asignación, ya que si no se evita, se podría destruir el objeto antes de copiarlo.

```

    if (this != &orig) { ... }

```

Así, `this` representa la dirección en memoria del objeto que recibe la asignación, y `&orig` representa la dirección en memoria del objeto que se recibe como parámetro. Si ambas direcciones son diferentes, entonces significa que son variables diferentes y se puede realizar la asignación.

2. Una vez que se ha comprobado que se está asignando un objeto diferente, se debe destruir el valor antiguo del objeto receptor de la asignación. Esta operación a veces no es necesaria realizarla explícitamente, ya que el siguiente paso puede realizarla de forma implícita.

²Nótese que no es posible invocar explícitamente al constructor de copia u operador de asignación para un atributo miembro de tipo array predefinido, sin embargo si es posible hacerlo si es de tipo array de `tr1`.

En el ejemplo, no es necesario destruir el valor anterior, ya que la asignación de cada elemento del vector se encargará de realizarla individualmente.

3. Se debe copiar el valor nuevo a la representación interna del objeto.

En el ejemplo, se asigna individualmente el valor de cada elemento del vector original al objeto que recibe la asignación. Esta asignación individual se deberá encargar de destruir el valor anterior que tuviese cada elemento.

4. Finalmente, el operador de asignación debe devolver el objeto actual (***this**) sobre el que recae la asignación.

- Ⓐ La siguiente implementación es muy habitual en caso de haber definido un método eficiente **swap(...)** para el intercambio de objetos, ya que realiza la copia y destrucción del valor anterior invocando explícitamente al constructor de copia e implícitamente al destructor de la clase (cuando se destruye la variable temporal).

```
Clase& Clase::operator = (const Clase& orig)
{
    if (this != &orig) {
        Clase(orig).swap(*this);
    }
    return *this;
}
```

Así, se invoca al constructor de copia (**Clase(orig)**) para copiar el valor a asignar a una variable temporal, y después se intercambian los valores de esta variable temporal con los del propio objeto (**.swap(*this)**). De esta forma el objeto recibirá la copia de los valores de **orig**, y pasado sus antiguos valores a la variable temporal. Posteriormente, la variable temporal se destruye, destruyéndose de esta forma el valor anterior del objeto asignado.

Implementación de Métodos y Operadores

Tanto los métodos **push_back()** como **pop_back()** se implementan igual que en el caso del tipo **Stack**, pero al estar definidos de forma separada, se deben cualificar con el nombre de la clase (**Vector::**).

El método **subvector()** devuelve un nuevo vector con el contenido de una parte del vector al que se aplica el método. Para realizar su implementación, simplemente se devuelve un nuevo objeto construido invocando al constructor explicado anteriormente que copia una parte del objeto actual (***this**).

En la implementación del operador de incremento (**+=**), primero se comprueba que hay espacio suficiente para albergar a los nuevos elementos. Posteriormente se añaden al final una copia de los elementos del vector especificado como parámetro, considerando que en caso de auto asignación (**v += v;**), la implementación del método debe funcionar adecuadamente añadiendo al final una copia de los elementos del propio vector. Finalmente, como en el caso del operador de asignación, se devuelve una referencia al propio objeto (***this**), para poder encadenar la asignaciones.

Implementación de Subprogramas Amigos

Para implementar los subprogramas y operadores amigos (**friend**), al no ser funciones miembro de la clase, no se cualifican con el nombre de la clase, ya que no pertenecen a ella, por lo que se definen como subprogramas y operadores comunes, pero con la particularidad de que al ser amigos de una determinada clase, tienen acceso a los miembros privados de los objetos pasados como parámetros de dicha clase. Este acceso a los miembros privados de la clase se realiza mediante el operador punto (**.**) aplicado a los objetos recibidos como parámetros de dicha clase.

Así, se implementa el operador de concatenación (**+**) invocando al constructor de copia para copiar el contenido del primer vector al vector resultado, y posteriormente se invoca al operador

de incremento (+=) para que añada el contenido del segundo vector al vector resultado que será finalmente devuelto como resultado de la función.

También se definen los operadores de comparación de igualdad (==) y menor (<), realizando una comparación lexicográfica de los elementos de ambos vectores recibidos como parámetros. Los demás operadores relacionales se implementan utilizando éstos dos operadores implementados.

Por último se implementan los operadores de entrada y salida, de tal forma que considerando que el número de elementos del vector es variable, se deben mostrar y leer el número de elementos contenidos en el vector, seguido por una secuencia de los elementos del mismo, separados por espacios. Por ejemplo, si el vector contiene los elementos

```
{ 1, 2, 3, 4, 5, 6 }
```

entonces mostrará en el flujo de salida los siguientes valores:

```
6 1 2 3 4 5 6
```

la entrada de datos realizará la operación inversa a la anterior, es decir, leerá el número de elementos del vector, y a continuación leerá tantos elementos como se haya especificado.

Nótese que al comienzo de la entrada de datos, es necesario inicializar el vector a vacío mediante la llamada al método `clear()`. Así mismo, nótese como estos operadores devuelven el propio flujo de datos que se recibió como parámetro, y sobre el que se han realizado las operaciones de entrada y salida, de tal forma que sea posible encadenar en cascada las invocaciones a estos operadores, como se realiza habitualmente:

Ⓐ Implementación Alternativa de los Operadores de Entrada y Salida de Datos

Es posible realizar una implementación alternativa (un poco más complicada) donde la secuencia de longitud indeterminada de elementos que contiene el vector se encuentra delimitada entre llaves, por ejemplo:

```
{ 1 2 3 4 5 6 }
```

```
//-vector.cpp -----
namespace umalcc {
    // .....
    std::ostream& operator << (std::ostream& out, const Vector& vect)
    {
        out << "{ ";
        for (unsigned i = 0; i < vect.sz; ++i) {
            out << vect.v[i] << " ";
        }
        out << "}";
        return out;
    }
    std::istream& operator >> (std::istream& in, Vector& vect)
    {
        char ch = '\0';
        in >> ch;
        if (ch != '{') {
            in.unget();
            in.setstate(std::ios::failbit);
        } else if ( ! in.fail() ) {
            vect.clear();
            in >> ch;
            while (( ! in.fail() ) && (ch != '}') ) {
                in.unget();
                int x;
                in >> x;
                if ( ! in.fail() ) {
```

```

        vect.push_back(x);
    }
    in >> ch;
}
}
return in;
}
}
//-----

```

Así, en primer lugar el subprograma de entrada de datos lee un carácter desde el flujo de entrada, el cual si no es igual al carácter '{' significa que la entrada de datos es errónea (no concuerda con el formato de entrada especificado), por lo que el carácter previamente leído se *devuelve* al flujo de entrada y el flujo de entrada se pone al estado *erróneo*.

```

std::istream& operator >> (std::istream& in, Vector& vect)
{
    char ch = '\0';
    in >> ch;
    if (ch != '{') {
        in.unget();
        in.setstate(std::ios::failbit);
    }
}

```

En otro caso, si el carácter leído es igual a '{' entonces se procede a leer los datos que se añadirán al vector. Para ello, se inicializa el vector a vacío mediante la llamada al método `clear()`. Posteriormente se lee un carácter desde el flujo de entrada, con el objetivo de poder detectar el final de los datos de entrada mediante el carácter '}' :

```

    } else if ( ! in.fail() ) {
        vect.clear();
        in >> ch;
    }
}

```

Si la lectura fue correcta y distinta del carácter '}' entonces significa que hay datos para leer, por lo que el carácter previamente leído se *devuelve* al flujo de entrada, y se intenta ahora leer el dato (un número entero) desde el flujo de entrada, para añadirlo al vector en caso de que se realice una lectura correcta:

```

while ((! in.fail()) && (ch != '}')) {
    in.unget();
    int x;
    in >> x;
    if ( ! in.fail() ) {
        vect.push_back(x);
    }
}

```

A continuación se vuelve a leer otro carácter con el objetivo de poder detectar el final de los datos de entrada mediante el carácter '}', y se vuelve al comienzo del bucle `while`:

```

        in >> ch;
    }
}

```

Cuando el carácter leído sea igual al carácter '}' entonces se sale del bucle de lectura y los datos leídos se habrán añadido al vector. En caso de que se haya producido algún error de formato durante la entrada de datos, entonces el flujo de entrada se pondrá en un estado erróneo.

Finalmente, se devuelve el flujo de entrada para que se puedan encadenar en cascada las invocaciones a este operador, como se realiza habitualmente:

```

    }
    return in;
}

```

Instancias de Clase: Objetos

El siguiente programa muestra una utilización del *Tipo Abstracto de Datos Vector* donde se comprueba el normal funcionamiento de los objetos declarados de dicho tipo. Así, los operadores se utilizan con la sintaxis esperada, pero las funciones miembro se utilizan a través de la notación punto. Además, vemos también como se crean los objetos de la clase utilizando diferentes constructores y diferente sintaxis.

```
#include "vector.hpp"
#include <iostream>
using namespace std;
using namespace umalcc;
void anyadir(Vector& v, unsigned n)
{
    for (unsigned i = 0; i < n; ++i) {
        v.push_back(i);
    }
}
void eliminar(Vector& v, unsigned i)
{
    if (i < v.size()) {
        v[i] = v[v.size()-1];
        v.pop_back();
    }
}
int main()
{
    try {
        Vector v1;
        cout << "Introduzca un vector entre llaves" << endl;
        cin >> v1;
        cout << "v1: " << v1 << endl;
        Vector v2(5);
        cout << "v2: " << v2 << endl;
        const Vector v3 = v1;
        cout << "v3: " << v3 << endl;
        Vector v4(v1);
        cout << "v4: " << v4 << endl;
        Vector v5(v1, 3, 2);
        cout << "v5: " << v5 << endl;
        anyadir(v5, 4);
        cout << "v5: " << v5 << endl;
        eliminar(v5, 1);
        cout << "v5: " << v5 << endl;

        v2 = v1.subvector(3, 2);
        cout << "v2: " << v2 << endl;
        //v3 = v4; // Error de Compilación

        if (3 < v1.size()) {
            v1[3] = 9;
            cout << "v1: " << v1 << endl;
        }
        //v3[3] = 9; // Error de Compilación

        v5 += v1;
        cout << "v5: " << v5 << endl;

        Vector v6 = v1 + v2;
        cout << "v6: " << v6 << endl;
    }
}
```

```

    cout << (v1 == v1) << endl;
    cout << (v1 == v2) << endl;

    for (int i = 0; i < 50; ++i) {
        v6 += v6; // hasta overflow
    }
} catch (const length_error& e) {
    cout << "Error: " << e.what() << endl;
} catch (const exception& e) {
    cout << "Error: " << e.what() << endl;
} catch ( ... ) {
    cout << "Error: Inesperado"<< endl;
}
}

```

13.2. Métodos Definidos Automáticamente por el Compilador

Los siguientes métodos pueden ser definidos automáticamente por el compilador si el programador no los define. Por ejemplo, para la siguiente clase:

```

class X {
public:
    // ...
private:
    T1 a1; // atributo a1 de tipo T1
    T2 a2; // atributo a2 de tipo T2
};

```

- El constructor por defecto (sin parámetros) será definido automáticamente por el compilador *si el programador no define ningún constructor*. El comportamiento predefinido consistirá en la llamada al constructor por defecto para cada atributo miembro de la clase. En caso de que se invoque explícitamente, el constructor por defecto de los tipos predefinidos es la inicialización a cero. Sin embargo, si no se invoca explícitamente para los tipos predefinidos, su valor queda inespecificado.

```
X::X() : a1(), a2() {}
```

- El constructor de copia se definirá automáticamente por el compilador en caso de que el programador no lo proporcione. El comportamiento predefinido consistirá en la llamada al constructor de copia para cada atributo miembro de la clase. El constructor de copia por defecto de los tipos predefinidos realizará una copia byte a byte de un objeto origen al objeto destino.

```
X::X(const X& o) : a1(o.a1), a2(o.a2) {}
```

- El operador de asignación será definido automáticamente por el compilador si no es proporcionado por el programador. Su comportamiento predefinido será llamar al operador de asignación para cada atributo miembro de la clase. El operador de asignación por defecto de los tipos predefinidos realizará una asignación byte a byte de un objeto origen al objeto destino.

```
X& X::operator=(const X& o) { a1 = o.a1; a2 = o.a2; return *this; }
```

- El destructor de la clase se definirá automáticamente por el compilador si no es definido por el programador, y su comportamiento predefinido será llamar a los destructores de los atributos miembros de la clase.

```
~X::X() {}
```

13.3. Requisitos de las Clases Respecto a las Excepciones

Con objeto de diseñar clases que se comporten adecuadamente en su ámbito de utilización, es conveniente seguir los siguientes consejos en su diseño:

- No se deben lanzar excepciones desde los destructores.
- Las operaciones de comparación no deben lanzar excepciones.
- Cuando se actualiza un objeto, no se debe destruir la representación antigua antes de haber creado completamente la nueva representación y pueda reemplazar a la antigua sin riesgo de excepciones.
- Antes de lanzar una excepción, se deberá liberar todos los recursos adquiridos que no pertenezcan a ningún otro objeto. (Utilizar la técnica RAII “*adquisición de recursos es inicialización*” sec. 22.1).
- Antes de lanzar una excepción, se deberá asegurar de que cada atributo se encuentra en un estado “*válido*”. Es decir, dejar cada objeto en un estado en el que pueda ser destruido por su destructor de forma coherente y sin lanzar ninguna excepción.

Nótese que un constructor es un caso especial, ya que cuando lanza una excepción no deja ningún objeto “*creado*” para destruirse posteriormente, por lo que se debe asegurar de liberar todos los recursos adquiridos durante la construcción fallida antes de lanzar la excepción (RAII “*adquisición de recursos es inicialización*”).

Cuando se lanza una excepción dentro de un constructor, se ejecutarán los destructores asociados a los atributos que hayan sido previamente inicializados al llamar a sus constructores en la lista de inicialización. Sin embargo, los recursos obtenidos dentro del cuerpo del constructor deberán liberarse explícitamente en caso de que alguna excepción sea lanzada. (RAII “*adquisición de recursos es inicialización*”).

13.4. Más sobre Métodos y Atributos

Atributos Constantes

Es posible definir atributos miembros *constantes* (no estáticos) mediante el cualificador `const`. En este caso, el valor del atributo constante toma su valor en el momento de la creación del objeto, y no varía durante la vida del objeto hasta su destrucción. Hay que notar que, en este caso, estos atributos constantes pueden tener diferentes valores para cada instancia (objeto) de la clase, a diferencia de las constantes estáticas, que contienen el mismo valor compartido para todas las instancias de la clase. En caso de que una clase tenga atributos miembros constantes, entonces no se le podrá aplicar el operador de asignación por defecto.

```
#include <iostream>
using namespace std;
class X {
public:
    X(int v) : valor(v) {}
    friend ostream& operator << (ostream& out, const X& x) {
        return out << x.valor ;
    }
private:
    const int valor;
};
int main()
{
    X x1(1);
    X x2 = 2;
```

```

    cout << x1 << " " << x2 << endl;

    // x1 = x2;    // Error, asignación por defecto no permitida
}

```

Ⓐ Atributos Mutables

Es posible, además, definir atributos miembro *mutables* mediante el cualificador `mutable`. Este cualificador significa que el atributo mutable puede ser modificado, incluso aunque pertenezca a un objeto constante, por medio de un método u operador constante.

```

#include <iostream>
using namespace std;
class X {
public:
    X(int v) : valor(v) {}
    void incremento(int i) const {
        valor += i;
    }
    friend ostream& operator << (ostream& out, const X& x) {
        return out << x.valor ;
    }
private:
    mutable int valor;
};
int main()
{
    const X x1(1);
    X x2 = 2;
    cout << x1 << " " << x2 << endl;
    x1.incremento(5);
    x2.incremento(7);
    cout << x1 << " " << x2 << endl;
}

```

Ⓐ Atributos y Métodos Estáticos

También es posible declarar atributos *estáticos* mediante el cualificador `static`. Este cualificador significa que el valor de dicho atributo es compartido por todas las instancias (objetos) de la clase, de tal forma que si un objeto modifica su valor, este nuevo valor será compartido por todos los objetos.

Cada objeto que se declare de una determinada clase tendrá diferentes instancias de los atributos de la clase, permitiendo así que diferentes objetos de la misma clase tengan atributos con valores diferentes, pero si algún atributo es estático, entonces dicho atributo será único y compartido por todos los objetos de la clase.

Cuando se declara un atributo `static`, se deberá definir en el módulo de implementación (cualificándolo con el nombre de la clase) y asignarle un valor inicial.

También es posible definir un método que sea estático, en cuyo caso se podrá invocar directamente sobre la clase, es decir, no será necesario que se aplique a ningún objeto específico, y sólo podrá acceder a atributos estáticos (no puede acceder a los atributos que corresponden a objetos instancias de la clase).

```

#include <iostream>
using namespace std;
class X {
public:
    X(int v) : valor(v) { ++cnt; }
    static unsigned cuenta() {          // Método estático

```

```

        return cnt;
    }
    friend ostream& operator << (ostream& out, const X& x) {
        return out << x.valor ;
    }
private:
    static unsigned cnt;           // Atributo estático
    int valor;
};
//-----
unsigned X::cnt = 0; // definición e inicialización del atributo estático
//-----
int main()
{
    X x1(1);
    X x2 = 2;
    unsigned num_objs = X::cuenta(); // invocación de método estático
    cout << "Número de Objetos de Clase X creados: " << num_objs << endl;
    cout << x1 << " " << x2 << endl;
}

```

Ⓐ Operadores de Incremento y Decremento

Se pueden definir los operadores de incremento y decremento, tanto prefijo, como post-fijo.

```

#include <iostream>
using namespace std;
class X {
public:
    X(int v) : valor(v) {}
    X& operator++ () { // Incremento Prefijo
        ++valor;
        return *this;
    }
    X& operator-- () { // Decremento Prefijo
        --valor;
        return *this;
    }
    X operator++ (int) { // Incremento Postfijo
        X anterior(*this); // copia del valor anterior
        ++valor;
        return anterior; // devolución del valor anterior
    }
    X operator-- (int) { // Decremento Postfijo
        X anterior(*this); // copia del valor anterior
        --valor;
        return anterior; // devolución del valor anterior
    }
    friend ostream& operator << (ostream& out, const X& x) {
        return out << x.valor ;
    }
private:
    int valor;
};
int main()
{
    X x1 = 1;
    ++x1;
    x1++;
    cout << x1 << endl;
}

```

```

--x1;
x1--;
cout << x1 << endl;
}

```

Ⓐ Conversiones de Tipo

Es posible definir conversiones de tipo implícitas para un determinado objeto de una clase. El tipo destino puede ser cualquier tipo, ya sea predefinido o definido por el usuario. Por ejemplo, se puede definir una conversión directa e implícita de un objeto de la clase `X` definida a continuación a un tipo entero.

```

#include <iostream>
using namespace std;
class X {
public:
    X(int v) : valor(v) {}           // constructor
    operator int () const {         // operador de conversión a int
        return valor;
    }
private:
    int valor;
};
int main()
{
    const X x1(1);                  // invocación explícita al constructor
    X x2 = 2;                       // invocación implícita al constructor
    cout << x1 << " " << x2 << endl; // invocación implícita al op de conversión
    int z1 = x1;                   // invocación implícita al op de conversión
    int z2 = int(x2);              // invocación explícita al op de conversión
    cout << z1 << " " << z2 << endl;
    X x3 = x1 + x2;                // inv. impl. al op de conversión y constructor
    cout << x3 << endl;            // invocación implícita al op de conversión
}

```

En este ejemplo se aprecia como los valores de `x1` y `x2` se convierten tanto implícitamente como explícitamente a valores enteros al mostrarlos por pantalla, y también al asignarlos a variables de tipo entero (`z1` y `z2`). Además, también hay una conversión implícita cuando se suman los valores de `x1` y `x2`, y posteriormente el resultado de tipo `int` se convierte y asigna a un objeto de clase `X` mediante el constructor que recibe un entero como parámetro.

Así, se puede apreciar que los métodos de conversión de tipos permiten definir la conversión implícita de un determinado tipo abstracto de datos a otro determinado tipo especificado por el operador de conversión correspondiente. Además, Los constructores también ofrecen otro tipo de conversión implícita (o explícita si se define mediante el especificador **explicit**) desde valores de otros tipos al tipo abstracto de datos para el que se define el constructor.

Por lo tanto, en el caso del ejemplo anterior, el constructor `X(int v)` ofrece una conversión implícita de un valor de tipo `int` a un objeto de tipo `X`. Así mismo, el operador de conversión `operator int ()` ofrece una conversión implícita de un objeto de tipo `X` a un valor de tipo `int`.

Sin embargo, si el constructor de la clase `X` se define como **explicit**, entonces es necesario invocarlo explícitamente para realizar la conversión de tipo `int` al tipo `X`, por ejemplo:

```

#include <iostream>
using namespace std;
class X {
public:
    explicit X(int v) : valor(v) {} // constructor explícito
    operator int () const {         // operador de conversión a int
        return valor;
    }
}

```



```

private
    int valor;
};
int main()
{
    const X x1(1);           // invocación explícita al constructor
    X x2(2);                 // invocación explícita al constructor
    cout << x1 << " " << x2 << endl; // invocación implícita al op de conversión
    int z1 = x1;             // invocación implícita al op de conversión
    int z2 = int(x2);        // invocación explícita al op de conversión
    cout << z1 << " " << z2 << endl;
    X x3 = X(x1 + x2);       // inv. impl. al op de conv. y expl. al ctor
    cout << x3 << endl;      // invocación implícita al op de conversión
}

```

13.5. Sobrecarga de Operadores \mathbb{A}

El lenguaje de programación C++ permite sobrecargar los operadores para tipos definidos por el programador. Sin embargo, no es posible definir nuevos operadores ni cambiar la sintaxis, la aridad, la precedencia o la asociatividad de los operadores existentes. Se puede definir el comportamiento de los siguientes operadores para tipos definidos por el programador:

Operadores	Aridad	Asociatividad
() [] -> ->*	<i>binario</i>	izq. a dch.
++ -- tipo()	<i>unario</i>	dch. a izq.
! ~ + - * &	<i>unario</i>	dch. a izq.
* / %	<i>binario</i>	izq. a dch.
+ -	<i>binario</i>	izq. a dch.
<< >>	<i>binario</i>	izq. a dch.
< <= > >=	<i>binario</i>	izq. a dch.
== !=	<i>binario</i>	izq. a dch.
&	<i>binario</i>	izq. a dch.
^	<i>binario</i>	izq. a dch.
	<i>binario</i>	izq. a dch.
&&	<i>binario</i>	izq. a dch.
	<i>binario</i>	izq. a dch.
= += -= *= /= %= &= ^= = <<= >>=	<i>binario</i>	dch. a izq.
,	<i>binario</i>	izq. a dch.
new new[] delete delete[]	<i>unario</i>	izq. a dch.

Sin embargo, los siguientes operadores no podrán ser definidos:

:: . .* ?: sizeof typeid

Los siguientes operadores ya tienen un significado predefinido para cualquier tipo que se defina, aunque pueden ser redefinidos:

operator= operator& operator,

La conversión de tipos y los siguientes operadores sólo podrán definirse como funciones miembros de objetos:

operator= operator[] operator() operator->

Capítulo 14

Introducción a la Programación Genérica. Plantillas

El lenguaje de programación C++ proporciona soporte a la programación genérica mediante las plantillas (“*templates*” en inglés). Las plantillas proporcionan un mecanismo eficaz para definir código (constantes, tipos y subprogramas) genéricos parametrizados, que puedan ser instanciados en “*tiempo de compilación*”. Estos parámetros genéricos de las plantillas podrán ser instanciados con tipos y valores constantes concretos especificados en tiempo de compilación.

Las definiciones genéricas deberán, por lo general, estar visibles en el lugar donde sean instanciadas, por lo que en el caso de definirse en módulos diferentes, deberán estar definidas en los ficheros de encabezamiento, para que puedan ser incluidas por todos aquellos módulos que las necesiten.

La definición de plantillas, tanto de subprogramas como de tipos comienza con la palabra reservada **template**, seguida entre delimitadores `< ... >` por los parámetros genéricos de la definición. Estos parámetros genéricos pueden ser tanto tipos (precedidos por la palabra reservada **typename**), como constantes de tipos integrales (**char**, **short**, **int**, **unsigned**, **long**) o de tipos genéricos parametrizados con anterioridad (que deben ser instanciados a tipos integrales).

14.1. Subprogramas Genéricos

Los subprogramas genéricos son útiles cuando definen procesamientos genéricos que son independientes de los tipos concretos sobre los que se aplican. Veamos algunos ejemplos de definición de subprogramas genéricos:

```
template <typename Tipo>
inline Tipo maximo(const Tipo& x, const Tipo& y)
{
    return (x > y) ? x : y ;
}
template <typename Tipo>
void intercambio(Tipo& x, Tipo& y)
{
    Tipo aux = x;
    x = y;
    y = aux;
}
int main()
{
    int x = 4;
    int y = maximo(x, 8);
    intercambio(x, y);

    double a = 7.5;
```

```

double b = maximo(a, 12.0);
intercambio(a, b);

double c = maximo(a, 12); // Error: maximo(double, int) no esta definido
}

```

En el ejemplo se puede ver que los parámetros de entrada a los subprogramas se pasan por referencia constante, ya que al ser un tipo genérico podría ser tanto un tipo simple como un tipo estructurado.

También puede apreciarse que la instanciación de subprogramas genéricos a tipos concretos se realiza automáticamente a partir de la invocación a los mismos, de tal forma que la instanciación de los parámetros genéricos se realiza por deducción a partir del tipo de los parámetros especificados en la invocación a los subprogramas.

Sin embargo, hay situaciones donde los parámetros genéricos no pueden ser deducidos de la propia invocación al subprograma. En este caso, los parámetros genéricos involucrados deben ser especificados explícitamente en la llamada. Nota: el siguiente ejemplo se proporciona para ilustrar esta característica, aunque no debe ser tomado como ejemplo de diseño, ya que este ejemplo en concreto carece de utilidad práctica.

```

template <typename TipoDestino, typename TipoOrigen>
inline TipoDestino convertir(const TipoOrigen& x)
{
    return TipoDestino(x);
}
int main()
{
    int x = convertir<int>(3.14);
}

```

El siguiente ejemplo de subprograma genérico muestra la utilización de parámetros genéricos constantes (de tipo integral), y es útil para buscar elementos en un array genérico. Nótese que el tipo del elemento a buscar puede ser diferente del tipo base del array siempre y cuando esté definido el operador de comparación (!=).

```

#include <iostream>
#include <tr1/array>
using namespace std;
using namespace std::tr1;
//-----
template <typename TipoBusc, typename TipoBase, unsigned SIZE>
unsigned buscar(const TipoBusc& x, const array<TipoBase, SIZE>& v)
{
    unsigned i = 0;
    while ((i < v.size())&&(x != v[i])) {
        ++i;
    }
    return i;
}
//-----
typedef array<int, 5> AInt;
void prueba1()
{
    AInt a = {{ 1, 2, 3, 4, 5 }};
    unsigned i = buscar(4, a);
    cout << i << endl;
}
//-----
struct Persona {
    string nombre;
    string telefono;
}

```

```

};
inline bool operator!=(const string& n, const Persona& p)
{
    return n != p.nombre;
}
typedef array<Persona, 5> APers;
void prueba2()
{
    APers a = {{ { "pepe", "111" }, { "juan", "222" }, { "maría", "333" },
                  { "marta", "444" }, { "lucas", "555" }
    }};
    unsigned i = buscar("maría", a);
    cout << i << endl;
}
//-----

```

Errores de Instanciación de Parámetros Genéricos

En el caso de que la definición de un determinado subprograma sea incorrecta para una determinada instanciación concreta de los parámetros genéricos, se producirá un error de compilación indicando el tipo de error. Por ejemplo, el siguiente código produce un error de compilación, ya que el tipo `Persona` no tiene definido el operador de comparación (`>`).

```

#include <string>
using namespace std;
struct Persona {
    string nombre;
    string telefono;
};
template <typename Tipo>
inline Tipo maximo(const Tipo& x, const Tipo& y)
{
    return (x > y) ? x : y ;
}
int main()
{
    Persona p1 = { "pepe", "1111" };
    Persona p2 = { "juan", "2222" };
    Persona p3 = maximo(p1, p2);
}

```

Produce el siguiente mensaje de error de compilación (con *GNU GCC*):

```

main.cpp: In function 'Tipo maximo(const Tipo&, const Tipo&) [with Tipo = Persona]':
main.cpp:16:   instantiated from here
main.cpp:10: error: no match for 'operator>' in 'x > y'

```

Cuando se trabaja con plantillas en C++, hay que tener presente que, a veces, los mensajes de error pueden ser bastante complicados de interpretar en el caso de errores producidos por instanciaciones de parámetros genéricos.

14.2. Tipos Abstractos de Datos Genéricos

Las plantillas también pueden ser utilizadas para la definición de tipos genéricos, tanto registros como clases genéricas. No obstante, a diferencia de los subprogramas genéricos que son capaces de instanciar los parámetros genéricos a partir de la invocación al subprograma, en el caso de tipos genéricos será necesaria la instanciación explícita de los parámetros de los mismos.

Como se explicó al comienzo del capítulo, las definiciones genéricas deben, por lo general, estar visibles en el lugar donde sean instanciadas, y por lo tanto, en el caso de tipos abstractos genéricos

tanto la definición como la implementación se realizará en ficheros de encabezamiento. Además, en esta sección introductoria a la programación genérica sólo veremos los tipos abstractos de datos genéricos definidos e implementados “*en línea*”, y en su versión más simple.

Veamos un ejemplo de definición e implementación de un tipo **Vector** genérico, como un contenedor de elementos donde éstos se pueden añadir hasta un determinado límite. En este ejemplo, se ha simplificado la definición e implementación del tipo vector con respecto a la definición del capítulo anterior, para hacer más fácil la comprensión de los conceptos relativos a la programación genérica. Así, se utilizan el constructor de copia, el operador de asignación y el destructor de la clase generados automáticamente por el compilador.

```
//-vector.hpp -----
#ifndef _vector_hpp_
#define _vector_hpp_
#include <tr1/array>
#include <cassert>
#include <stdexcept>
namespace umalcc {
    template <typename Tipo, unsigned SIZE>
    class Vector {
    public:
        Vector() : sz(0), v() {}
        unsigned size() const {
            return sz;
        }
        void clear() {
            sz = 0;
        }
        void push_back(const Tipo& e) {
            if (sz >= v.size()) {
                throw std::runtime_error("Vector::push_back");
            }
            v[sz] = e;
            ++sz;
        }
        void pop_back() {
            assert(size() > 0); // Pre-condición
            --sz;
        }
        const Tipo& operator [] (unsigned i) const {
            assert(i < size()); // Pre-condición
            return v[i];
        }
        Tipo& operator [] (unsigned i) {
            assert(i < size()); // Pre-condición
            return v[i];
        }
        friend std::ostream& operator << (std::ostream& out, const Vector& vect) {
            out << vect.sz << " ";
            for (unsigned i = 0; i < vect.sz; ++i) {
                out << vect.v[i] << " ";
            }
            return out;
        }
        friend std::istream& operator >> (std::istream& in, Vector& vect)
        {
            vect.clear();
            unsigned nelms;
            in >> nelms;
            for (unsigned i = 0; (i < nelms)&&(!in.fail()); ++i) {
```

```

        Tipo x;
        in >> x;
        if ( ! in.fail() ) {
            vect.push_back(x);
        }
    }
    return in;
}
private:
    typedef std::tr1::array<Tipo, SIZE> Datos;
    // -- Atributos --
    unsigned sz;
    Datos v;
};
}
#endif

```

Así, se puede utilizar el vector definido anteriormente como base para implementar un tipo **Stack**. En este caso, se utilizan los constructores por defecto y de copia, así como el operador de asignación y el destructor de la clase generados automáticamente por el compilador. En el ejemplo puede apreciarse como se realiza una instanciación explícita de un tipo genérico para la definición del atributo `v`.

```

//--stack.hpp -----
#ifndef _stack_hpp_
#define _stack_hpp_
#include <cassert>
#include "vector.hpp"
namespace umalcc {
    template <typename Tipo, unsigned SIZE>
    class Stack {
    public:
        bool empty() const {
            return v.size() == 0;
        }
        void clear() {
            v.clear();
        }
        void push(const Tipo& e) {
            v.push_back(e);
        }
        void pop(Tipo& e) {
            assert(size() > 0); // Pre-condición
            e = v[v.size() - 1];
            v.pop_back();
        }
    private:
        // -- Atributos --
        Vector<Tipo, SIZE> v;
    };
}
#endif

```

Ambos tipos abstractos de datos genéricos se pueden utilizar de la siguiente forma, donde se definen dos tipos (`Vect` y `Pila`) como instanciaciones explícitas de los tipos genéricos previamente definidos.

```

#include <iostream>
#include "vector.hpp"
#include "stack.hpp"

```

```

using namespace std;
using namespace umalcc;

typedef Vector<double, 30> Vect;
typedef Stack<int, 20> Pila;

void almacenar_datos(Pila& s, unsigned n)
{
    for (int x = 0; x < n; ++x) {
        s.push(x);
    }
}

void imprimir(Pila& s)
{
    while (! s.empty() ) {
        int x;
        s.pop(x);
        cout << x << " ";
    }
    cout << endl;
}

void anyadir(Vect& v, unsigned n)
{
    for (unsigned i = 0; i < n; ++i) {
        v.push_back(i);
    }
}

void eliminar(Vect& v, unsigned i)
{
    if (i < v.size()) {
        v[i] = v[v.size()-1];
        v.pop_back();
    }
}

int main()
{
    try {
        Pila s;
        almacenar_datos(s, 10);
        imprimir(s);

        Vect v;
        anyadir(v, 4);
        eliminar(v, 1);
    } catch (const length_error& e) {
        cout << "Error: " << e.what() << endl;
    } catch ( ... ) {
        cout << "Error: Inesperado"<< endl;
    }
}

```

14.3. Parámetros Genéricos por Defecto

Es posible especificar instanciaciones *por defecto* para los parámetros genéricos en la definición de tipos genéricos. Estos parámetros con valores por defecto deben ser los últimos de la lista de parámetros genéricos. Por ejemplo:


```

template <typename T, typename C = vector<T> >
class Stack {
    // ...
};

template <typename T, unsigned SIZE=512>
class Array {
    // ...
};

typedef Stack<int, list> SInt; // instancia el segundo parámetro a list<int>
typedef Stack<int> StckInt;   // instancia el segundo parámetro a vector<int>

typedef Array<int, 126> AInt; // instancia el segundo parámetro a 126
typedef Array<int> ArrayInt;  // instancia el segundo parámetro a 512

```

14.4. Definición de Tipos dentro de la Definición de Tipos Genéricos

Si en algún ámbito se utiliza un determinado tipo cuya especificación depende de un tipo genérico no instanciado, entonces será necesario cualificar la utilización de dicho tipo con la palabra reservada `typename`. Así mismo, para utilizar desde un ámbito externo un tipo definido dentro de una clase (o estructura), dicho tipo deberá cualificarse con el nombre de la clase (o estructura) a la que pertenece. Por ejemplo:

```

//-----
#include <iostream>
using namespace std;
//-----
struct A {
    typedef int Tipo;
};
//-----
template <typename TT>
struct B {
    typedef int Tipo;
    typedef TT Base;
};
//-----
template <typename TT>
void prueba1()
{
    A::Tipo x = 3;
    typename B<TT>::Tipo z = 5;
    typename B<TT>::Base y = 4;
    cout << x << " " << y << " " << z << endl;
}
//-----
void prueba2()
{
    A::Tipo x = 3;
    B<int>::Tipo z = 5;
    B<int>::Base y = 4;
    cout << x << " " << y << " " << z << endl;
}
//-----
int main()
{

```

```

    prueba1<int>();
    prueba2();
}
//-----

```

14.5. Separación de Definición e Implementación

También es posible hacer la definición e implementación del tipo abstracto de datos genérico “fuera de línea”, de tal forma que por una parte se realiza la definición del tipos, y por otra parte se realiza la implementación del mismo. No obstante, hay que tener en cuenta que ambas partes deben ser visibles para poder instanciar el tipo y utilizar sus métodos, por lo que ambas partes residirán en el mismo fichero de encabezamiento.

Por ejemplo, es posible definir la clase **Vector** y su implementación de forma separada como se indica a continuación:

```

//--vector.hpp -----
#ifndef _vector_hpp_
#define _vector_hpp_
#include <iostream>
#include <tr1/array>
#include <cassert>
#include <stdexcept>
namespace umalcc {
    //-----
    // Declaración adelantada de la clase Vector
    template <typename Tipo, unsigned SIZE>
    class Vector;
    //-----
    // Prototipo del operador de entrada y salida para la clase Vector
    template <typename Tipo, unsigned SIZE>
    std::ostream& operator <<(std::ostream&, const Vector<Tipo,SIZE>&);
    template <typename Tipo, unsigned SIZE>
    std::istream& operator >>(std::istream&, Vector<Tipo,SIZE>&);
    //-----
    template <typename Tipo, unsigned SIZE>
    class Vector {
    public:
        Vector() ;
        unsigned size() const ;
        void clear() ;
        void push_back(const Tipo& e) ;
        void pop_back() ;
        const Tipo& operator [] (unsigned i) const ;
        Tipo& operator [] (unsigned i) ;
        friend std::ostream& operator << <<(std::ostream&, const Vector<Tipo,SIZE>&);
        friend std::istream& operator >> <<(std::istream&, Vector<Tipo,SIZE>&);
    private:
        typedef std::tr1::array<Tipo, SIZE> Datos;
        // -- Atributos --
        unsigned sz;
        Datos v;
    };
    //-----
    template <typename Tipo, unsigned SIZE>
    Vector<Tipo,SIZE>::Vector()
        : sz(0), v()
    {}
    //-----

```

```

template <typename Tipo, unsigned SIZE>
unsigned Vector<Tipo,SIZE>::size() const {
    return sz;
}
//-----
template <typename Tipo, unsigned SIZE>
void Vector<Tipo,SIZE>::clear() {
    sz = 0;
}
//-----
template <typename Tipo, unsigned SIZE>
void Vector<Tipo,SIZE>::push_back(const Tipo& e) {
    if (sz >= v.size()) {
        throw std::runtime_error("Vector::push_back");
    }
    v[sz] = e;
    ++sz;
}
//-----
template <typename Tipo, unsigned SIZE>
void Vector<Tipo,SIZE>::pop_back() {
    assert(size() > 0); // Pre-condición
    --sz;
}
//-----
template <typename Tipo, unsigned SIZE>
const Tipo& Vector<Tipo,SIZE>::operator [] (unsigned i) const {
    assert(i < size()); // Pre-condición
    return v[i];
}
//-----
template <typename Tipo, unsigned SIZE>
Tipo& Vector<Tipo,SIZE>::operator [] (unsigned i) {
    assert(i < size()); // Pre-condición
    return v[i];
}
//-----
template <typename Tipo, unsigned SIZE>
std::ostream& operator << (std::ostream& out, const Vector<Tipo,SIZE>& vect) {
    out << vect.sz << " ";
    for (unsigned i = 0; i < vect.sz; ++i) {
        out << vect.v[i] << " ";
    }
    return out;
}
//-----
template <typename Tipo, unsigned SIZE>
std::istream& operator >> (std::istream& in, Vector<Tipo,SIZE>& vect) {
    vect.clear();
    unsigned nelms;
    in >> nelms;
    for (unsigned i = 0; (i < nelms)&&(!in.fail()); ++i) {
        Tipo x;
        in >> x;
        if ( ! in.fail() ) {
            vect.push_back(x);
        }
    }
    return in;
}

```

```

    }
    //-----
}
#endif

```

Por ejemplo, para definir e implementar un tipo `Subrango` de otro tipo básico integral, de tal forma que si se asigna un valor incorrecto fuera del rango especificado, se lance una excepción. En este ejemplo, el compilador genera automáticamente el constructor de copia, el operador de asignación y el destructor de la clase.

```

//-archivo: subrango.hpp -----
#include <iostream>
#include <stdexcept>
namespace umalcc {
    // Declaración adelantada de la clase Subrango
    template<typename Tipo, Tipo menor, Tipo mayor>
    class Subrango;

    // Prototipo del operador de entrada y salida para la clase Subrango
    template<typename Tipo, Tipo menor, Tipo mayor>
    std::ostream& operator <<(std::ostream&, const Subrango<Tipo, menor, mayor>&);
    template<typename Tipo, Tipo menor, Tipo mayor>
    std::istream& operator >>(std::istream&, Subrango<Tipo, menor, mayor>&);

    template<typename Tipo, Tipo menor, Tipo mayor>
    class Subrango {
    public:
        Subrango();
        Subrango(const T_Base& i);
        operator T_Base();
        friend std::ostream& operator << <>(std::ostream&, const Subrango<Tipo, menor, mayor>&);
        friend std::istream& operator >> <>(std::istream&, Subrango<Tipo, menor, mayor>&);
    private:
        typedef Tipo T_Base;
        // -- Atributos --
        T_Base valor;
    };

    template<typename Tipo, Tipo menor, Tipo mayor>
    std::ostream& operator <<(std::ostream& sal, const Subrango<Tipo, menor, mayor>& i)
    {
        return sal << i.valor;
    }

    template<typename Tipo, Tipo menor, Tipo mayor>
    std::istream& operator >>(std::istream& in, Subrango<Tipo, menor, mayor>& i)
    {
        Tipo val;
        in >> val;
        i = val;
        return in;
    }

    template<typename Tipo, Tipo menor, Tipo mayor>
    Subrango<Tipo, menor, mayor>::Subrango()
        : valor(menor)
    {}

    template<typename Tipo, Tipo menor, Tipo mayor>
    Subrango<Tipo, menor, mayor>::Subrango(const T_Base& i)

```

```

        : valor(i)
    {
        if ((valor < menor) || (valor > mayor)) {
            throw std::range_error("Subrango::Subrango range error");
        }
    }

    template<typename Tipo, Tipo menor, Tipo mayor>
    Subrango<Tipo,menor,mayor>::operator Tipo()
    {
        return valor;
    }
}
//--fin: subrango.hpp -----

```

Una posible utilización del tipo Subrango podría ser:

```

//--fichero: main.cpp -----
#include "subrango.hpp"
#include <iostream>
using namespace std;
using namespace umalcc;

typedef Subrango<int, 3, 20> Dato;

int main()
{
    try {
        Dato x, z;
        Subrango<char, 'a', 'z'> y;

        x = 17;
        x = 25; // fuera de rango

        y = 'm';
        y = 'M'; // fuera de rango

        z = x;
        x = x + 5; // fuera de rango

        cout << x << " " << y << " " << z << endl;

    } catch (const std::range_error& e) {
        cerr << "Error: " << e.what() << endl;
    } catch ( ... ) {
        cerr << "Excepción inesperada" << endl;
    }
}

```


Capítulo 15

Memoria Dinámica. Punteros

Hasta ahora, todos los programas que se han visto en capítulos anteriores almacenan su estado interno por medio de variables que son automáticamente gestionadas por el compilador. Las variables son *creadas* cuando el flujo de ejecución entra en el ámbito de su definición (se reserva espacio en memoria y se crea el valor de su estado inicial), posteriormente se *manipula* el estado de la variable (accediendo o modificando su valor almacenado), y finalmente se *destruye* la variable cuando el flujo de ejecución sale del ámbito donde fue declarada la variable (liberando los recursos asociados a ella y la zona de memoria utilizada). A este tipo de variables gestionadas automáticamente por el compilador se las suele denominar *variables automáticas* (también variables locales), y residen en una zona de memoria gestionada automáticamente por el compilador, la *pila* de ejecución, donde se alojan y desalojan las variables locales (automáticas) pertenecientes al ámbito de ejecución de cada subprograma.

Así, el tiempo de vida de una determinada variable está condicionado por el ámbito de su declaración. Además, el número de variables automáticas utilizadas en un determinado programa está especificado explícitamente en el propio programa, y por lo tanto su capacidad de almacenamiento está también especificada y predeterminada por lo especificado explícitamente en el programa. Es decir, con la utilización única de variables automáticas, la capacidad de almacenamiento de un determinado programa está predeterminada desde el momento de su programación (tiempo de compilación), y no puede adaptarse a las necesidades reales de almacenamiento surgidas durante la ejecución del programa (tiempo de ejecución).¹

La gestión de *memoria dinámica* surge como un mecanismo para que el propio programa, durante su ejecución (tiempo de ejecución), pueda solicitar (alojar) y liberar (desalojar) memoria según las necesidades surgidas durante una determinada ejecución, dependiendo de las circunstancias reales de cada momento de la ejecución del programa en un determinado entorno. Esta ventaja adicional viene acompañada por un determinado coste asociado a la mayor complejidad que requiere su gestión, ya que en el caso de las variables automáticas, es el propio compilador el encargado de su gestión, sin embargo en el caso de las *variables dinámicas* es el propio programador el que debe, mediante código software, gestionar el tiempo de vida de cada variable dinámica, cuando debe ser alojada y creada, como será utilizada, y finalmente cuando debe ser destruida y desalojada. Adicionalmente, como parte de esta gestión de la memoria dinámica por el propio programador, la memoria dinámica pasa a ser un *recurso* que debe gestionar el programador, y se debe preocupar de su alojamiento y de su liberación, poniendo especial cuidado y énfasis en no perder recursos (perder zonas de memoria sin liberar y sin capacidad de acceso).

¹En realidad esto no es completamente cierto, ya que en el caso de subprogramas recursivos, cada invocación recursiva en tiempo de ejecución tiene la capacidad de alojar nuevas variables que serán posteriormente desalojadas automáticamente cuando la llamada recursiva finaliza.

15.1. Punteros

El *tipo puntero* es un tipo *simple* que permite a un determinado programa acceder a posiciones concretas de memoria, y más específicamente a determinadas zonas de la memoria dinámica. Aunque el lenguaje de programación C++ permite otras utilizaciones más diversas del tipo puntero, en este capítulo sólo se utilizará el tipo puntero para acceder a zonas de memoria dinámica.

Así, una determinada variable de tipo puntero apunta (o referencia) a una determinada entidad (variable) de un determinado tipo alojada en la zona de memoria dinámica. Por lo tanto, para un determinado tipo puntero, se debe especificar también el tipo de la variable (en memoria dinámica) a la que apunta, el cual define el espacio que ocupa en memoria y las operaciones (y métodos) que se le pueden aplicar, entre otras cosas.

De este modo, cuando un programa gestiona la memoria dinámica a través de punteros, debe manejar y gestionar por una parte la propia variable de tipo puntero, y por otra parte la variable dinámica apuntada por éste.

Un tipo puntero se define utilizando la palabra reservada **typedef** seguida del tipo de la variable dinámica apuntada, un asterisco para indicar que es un **puntero** a una variable de dicho tipo, y el identificador que denomina al tipo. Por ejemplo:

```
typedef int* PInt;

struct Persona {
    string nombre;
    string telefono;
    unsigned edad;
};
typedef Persona* PPersona;
```

Así, el tipo PInt es el tipo de una variable que apunta a una variable dinámica de tipo `int`. Del mismo modo, el tipo PPersona es el tipo de una variable que apunta a una variable dinámica de tipo `Persona`.

Es posible definir variables de los tipos especificados anteriormente. Nótese que estas variables (`p1` y `p2` en el siguiente ejemplo) son variables automáticas (gestionadas automáticamente por el compilador), es decir, se crean automáticamente al entrar el flujo de ejecución en el ámbito de visibilidad de la variable, y posteriormente se destruyen automáticamente cuando el flujo de ejecución sale del ámbito de visibilidad de la variable. Por otra parte, las variables apuntadas por ellos son variables dinámicas (gestionadas por el programador), es decir el programador se encargará de solicitar la memoria dinámica cuando sea necesaria y de liberarla cuando ya no sea necesaria, durante la ejecución del programa. En el siguiente ejemplo, si las variables se definen sin inicializar, entonces tendrán un valor inicial inespecificado:

```
int main()
{
    PInt p1;           p1: ?
    PPersona p2;       p2: ?
}
```

La constante `NULL` es una constante especial de tipo puntero que indica que una determinada variable de tipo puntero no apunta a nada, es decir, especifica que la variable de tipo puntero que contenga el valor `NULL` no apunta a ninguna zona de la memoria dinámica. Así, se pueden definir las variables `p1` y `p2` e inicializarlas a un valor indicando que no apuntan a nada.

```
int main()
{
    PInt p1 = NULL;    p1: /
    PPersona p2 = NULL; p2: /
}
```

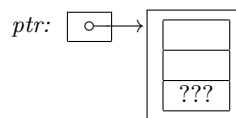
- Ⓐ Alternativamente, también es posible declarar directamente las variables de tipo puntero sin necesidad de definir explícitamente un tipo puntero:


```
int main()
{
    int* p1 = NULL;
    Persona* p2 = NULL;
}
```

15.2. Gestión de Memoria Dinámica

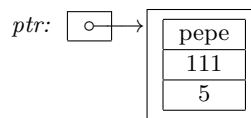
La memoria dinámica la debe gestionar el propio programador, por lo que cuando necesite crear una determinada variable dinámica, debe *solicitar memoria dinámica* con el operador **new** seguido por el tipo de la variable dinámica a crear. Este operador realiza dos acciones principales, primero aloja (reserva) espacio en memoria dinámica para albergar a la variable, y después crea (invocando al constructor especificado) el contenido de la variable dinámica. Finalmente, a la variable **ptr** se le asigna el valor del puntero (una dirección de memoria) que apunta a la variable dinámica creada por el operador **new**. Por ejemplo, para crear una variable dinámica del tipo **Persona** definido anteriormente utilizando el constructor por defecto de dicho tipo.

```
int main()
{
    PPersona ptr = new Persona;
}
```



En caso de que el tipo de la variable dinámica tenga otros constructores definidos, es posible utilizarlos en la construcción del objeto en memoria dinámica. Por ejemplo, suponiendo que el tipo **Persona** tuviese un constructor que reciba el nombre, teléfono y edad de la persona:

```
int main()
{
    PPersona ptr = new Persona("pepe", "111", 5);
}
```

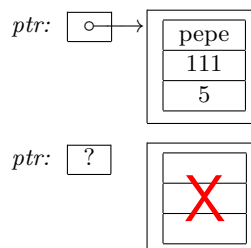


Posteriormente, tras manipular adecuadamente, según las características del programa, la memoria dinámica alojada, llegará un momento en que dicha variable dinámica ya no sea necesaria, y su tiempo de vida llegue a su fin. En este caso, el programador debe *liberar* explícitamente dicha variable dinámica mediante el operador **delete** de la siguiente forma:

```
int main()
{
    PPersona ptr = new Persona("pepe", "111", 5);

    // manipulación

    delete ptr;
}
```



La sentencia **delete ptr** realiza dos acciones principales, primero destruye la variable dinámica (invocando a su destructor), y después desaloja (libera) la memoria dinámica reservada para dicha variable. Finalmente la variable local **ptr** queda con un valor inespecificado, y será destruida automáticamente por el compilador cuando el flujo de ejecución salga de su ámbito de declaración.

Si se ejecuta la operación **delete** sobre una variable de tipo puntero que tiene el valor **NULL**, entonces esta operación no hace nada.

En caso de que no se libere (mediante el operador **delete**) la memoria dinámica apuntada por la variable **ptr**, y esta variable sea destruida al terminar su tiempo de vida (su ámbito de visibilidad), entonces se *perderá* la memoria dinámica a la que apunta.

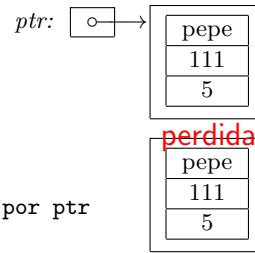
```

int main()
{
    PPersona ptr = new Persona("pepe", "111", 5);

    // manipulación

    // no se libera la memoria dinámica apuntada por ptr
    // se destruye la variable local ptr
}

```



15.3. Operaciones con Variables de Tipo Puntero

Desreferenciación de una Variable de Tipo Puntero

Para acceder a una variable dinámica apuntada por una variable de tipo puntero, se utiliza el operador unario *asterisco* (*) precediendo al nombre de la variable de tipo puntero a través de la cual es apuntada. Por ejemplo, si `ptr` es una variable local de tipo puntero que apunta a una variable dinámica de tipo `Persona`, entonces `*ptr` es la variable dinámica apuntada, y se trata de igual forma que cualquier otra variable de tipo `Persona`.

```

int main()
{
    PPersona ptr = new Persona("pepe", "111", 5);

    Persona p = *ptr;

    *ptr = p;

    delete ptr;
}

```

Sin embargo, si una variable de tipo puntero tiene el valor `NULL`, entonces *desreferenciar* la variable produce un error en tiempo de ejecución que aborta la ejecución del programa.

Es posible, así mismo, acceder a los elementos de la variable apuntada mediante el operador de desreferenciación. Por ejemplo:

```

int main()
{
    PPersona ptr = new Persona;

    (*ptr).nombre = "pepe";
    (*ptr).telefono = "111";
    (*ptr).edad = 5;

    delete ptr;
}

```

Nótese que el uso de los paréntesis es obligatorio debido a que el operador punto (.) tiene mayor precedencia que el operador de desreferenciación (*). Por ello, en el caso de acceder a los campos de un registro en memoria dinámica a través de una variable de tipo puntero, se puede utilizar el operador de desreferenciación (->) más utilizado comúnmente. Por ejemplo:

```

int main()
{
    PPersona ptr = new Persona;

    ptr->nombre = "pepe";
    ptr->telefono = "111";
    ptr->edad = 5;
}

```

```

    delete ptr;
}

```

Este operador también se utiliza para invocar a métodos de un objeto si éste se encuentra alojado en memoria dinámica. Por ejemplo:

```

#include <iostream>
using namespace std;
class Numero {
public:
    Numero(int v) : val(v) {}
    int valor() const { return val; }
private:
    int val;
};
typedef Numero* PNumero;
int main()
{
    PNumero ptr = new Numero(5);

    cout << ptr->valor() << endl;

    delete ptr;
}

```

Asignación de Variables de Tipo Puntero

El puntero nulo (NULL) se puede asignar a cualquier variable de tipo puntero. Por ejemplo:

```

int main()
{
    PPersona p1;           p1: [?]
    // ...
    p1 = NULL;             p1: [ / ]
    // ...
}

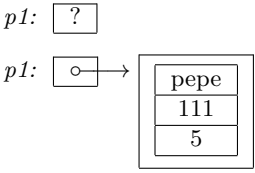
```

El resultado de crear una variable dinámica con el operador **new** se puede asignar a una variable de tipo puntero al tipo de la variable dinámica creada. Por ejemplo:

```

int main()
{
    PPersona p1;
    // ...
    p1 = new Persona("pepe", "111", 5);
    // ...
}

```

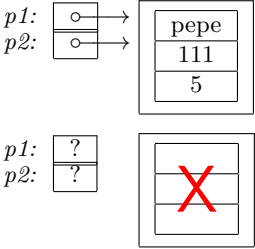


Así mismo, a una variable de tipo puntero se le puede asignar el valor de otra variable puntero. En este caso, ambas variables de tipo puntero apuntarán a la misma variable dinámica, que será compartida por ambas. Si se libera la variable dinámica apuntada por una de ellas, la variable dinámica compartida se destruye, su memoria se desaloja y ambas variables locales de tipo puntero quedan con un valor inespecificado.

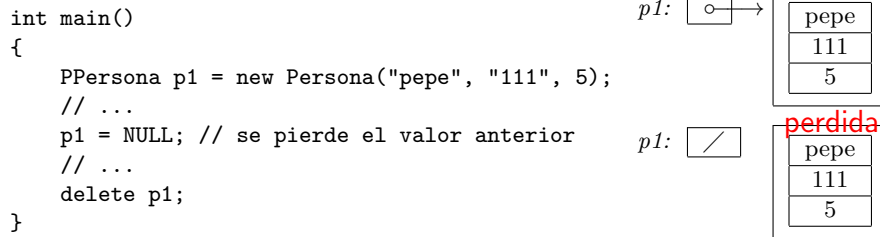
```

int main()
{
    PPersona p1 = new Persona("pepe", "111", 5);
    PPersona p2;
    // ...
    p2 = p1;
    // ...
    delete p1;
}

```



En la operación de asignación, el valor anterior que tuviese la variable de tipo puntero se pierde, por lo que habrá que tener especial cuidado de que no se pierda la variable dinámica que tuviese asignada, si tuviese alguna.



Comparación de Variables de Tipo Puntero

Las variables del mismo tipo puntero se pueden comparar entre ellas por igualdad (==) o desigualdad (!=), para comprobar si apuntan a la misma variable dinámica. Así mismo, también se pueden comparar por igualdad o desigualdad con el puntero nulo (NULL) para saber si apunta a alguna variable dinámica, o por el contrario no apunta a nada. Por ejemplo:

```

int main()
{
    PPersona p1, p2;
    // ...
    if (p1 == p2) {
        // ...
    }
    if (p1 != NULL) {
        // ...
    }
    // ...
}

```

15.4. Paso de Parámetros de Variables de Tipo Puntero

El tipo puntero es un *tipo simple*, y por lo tanto se tratará como tal. En caso de paso de parámetros de tipo puntero, si es un parámetro de entrada, entonces se utilizará el *paso por valor*, y si es un parámetro de salida o de entrada/salida, entonces se utilizará el *paso por referencia*.

Hay que ser consciente de que un parámetro de tipo puntero puede apuntar a una variable dinámica, y en este caso, a partir del parámetro se puede acceder a la variable apuntada.

Así, si el parámetro se pasa *por valor*, entonces se copia el valor del puntero del parámetro actual (en la invocación) al parámetro formal (en el subprograma), por lo que ambos apuntarán a la misma variable dinámica compartida, y en este caso, si se modifica el valor almacenado en la variable dinámica, este valor se verá afectado, así mismo, en el exterior del subprograma, aunque el parámetro haya sido pasado por valor.

Por otra parte, las funciones también pueden devolver valores de tipo puntero.

```

void modificar(PPersona& p);

PPersona buscar(PPersona l, const string& nombre);

```

- Ⓐ Alternativamente, también es posible declarar directamente los parámetros de tipo puntero sin necesidad de definir explícitamente un tipo puntero:

```

void modificar(Persona* & p);

Persona* buscar(Persona* l, const string& nombre);

```

Ⓐ Punteros a Variables Dinámicas Constantes

A veces es interesante garantizar que cuando se pasa un puntero a un subprograma, éste no pueda modificar los valores almacenados en la variable dinámica apuntada. Ya que el paso por valor no puede proporcionar dicha garantía, en el lenguaje de programación C++ existe el tipo *puntero a una variable dinámica constante*, de tal forma que se puede realizar un paso por valor de una variable de tipo puntero a un determinado tipo, a un parámetro de tipo puntero a una variable dinámica constante. Por ejemplo:

```
struct Persona {
    string nombre;
    string telefono;
};
typedef Persona* PPersona;
typedef const Persona* PCPersona;
void datos(PCPersona ptr)
{
    ptr->nombre = "pepe"; // error, no es posible modificar entidad constante
    ptr->telefono = "111"; // error, no es posible modificar entidad constante
}
int main()
{
    PPersona ptr = new Persona;
    datos(ptr);
}
```

- Ⓐ Alternativamente, también es posible declarar directamente los parámetros de tipo puntero constante sin necesidad de definir explícitamente un tipo puntero:

```
struct Persona {
    string nombre;
    string telefono;
};
void datos(const Persona* ptr)
{
    ptr->nombre = "pepe"; // error, no es posible modificar entidad constante
    ptr->telefono = "111"; // error, no es posible modificar entidad constante
}
int main()
{
    Persona* ptr = new Persona;
    datos(ptr);
}
```

15.5. Abstracción en la Gestión de Memoria Dinámica

La gestión de memoria dinámica por parte del programador se basa en estructuras de programación de *bajo nivel*, las cuales son propensas a errores de programación y pérdida de recursos de memoria. Además, entremezclar sentencias de gestión de memoria, de bajo nivel, con sentencias aplicadas al dominio de problema a resolver suele dar lugar a código no legible y propenso a errores.

Por lo tanto se hace necesario aplicar niveles de abstracción que aislen la gestión de memoria dinámica (de bajo nivel) del resto del código más directamente relacionado con la solución del problema. Para ello, los tipos abstractos de datos proporcionan el mecanismo adecuado para aplicar la abstracción a estas estructuras de datos basadas en la gestión de memoria dinámica, además de proporcionar una herramienta adecuada para la gestión de memoria dinámica, ya que los destructores se pueden encargar de liberar los recursos asociados a un determinado objeto. Por ejemplo:

```

template <typename Tipo>
class Dato {
public:
    Dato(const Tipo& d) : ptr(NULL) {
        ptr = new Nodo;
        ptr->dato = d;
    }
    ~Dato() {
        delete ptr;
    }
    // ... Otros métodos
private:
    struct Nodo {
        Tipo dato;
    };
    typedef Nodo* PNodo;
    //-- Atributos --
    PNodo ptr;
};

```

15.6. Estructuras Enlazadas

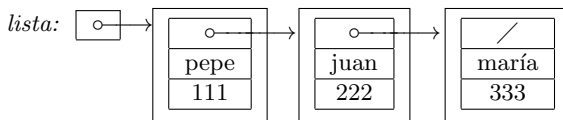
Una de las principales aplicaciones de la Memoria Dinámica es el uso de estructuras enlazadas, de tal forma que un campo o atributo de la variable dinámica es a su vez también de tipo puntero, por lo que puede apuntar a otra variable dinámica que también tenga un campo o atributo de tipo puntero, el cual puede volver a apuntar a otra variable dinámica, y así sucesivamente, tantas veces como sea necesario, hasta que un puntero con el valor NULL indique el final de la estructura enlazada (lista enlazada).

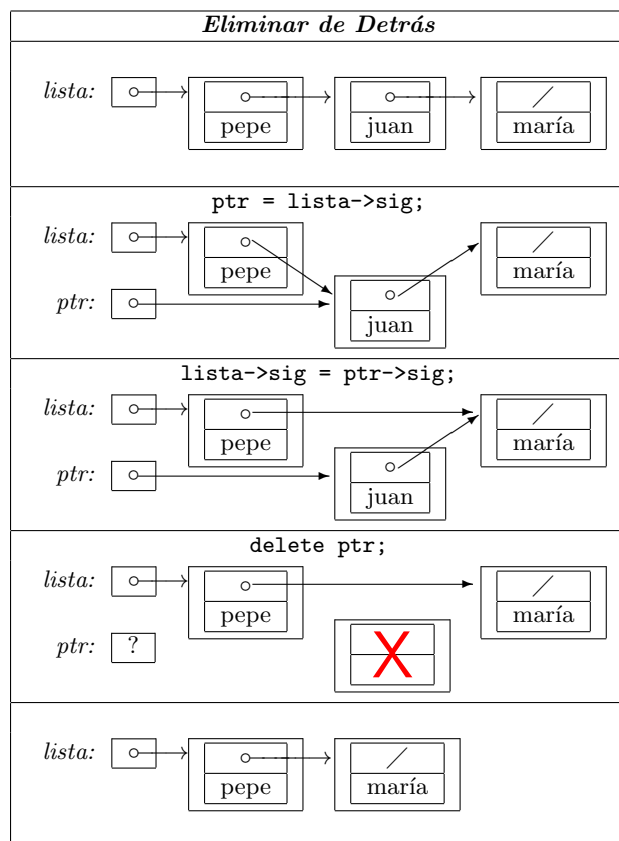
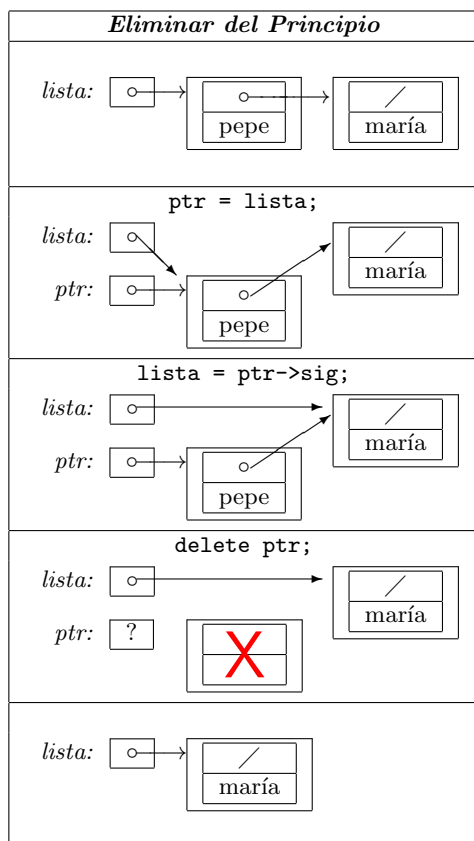
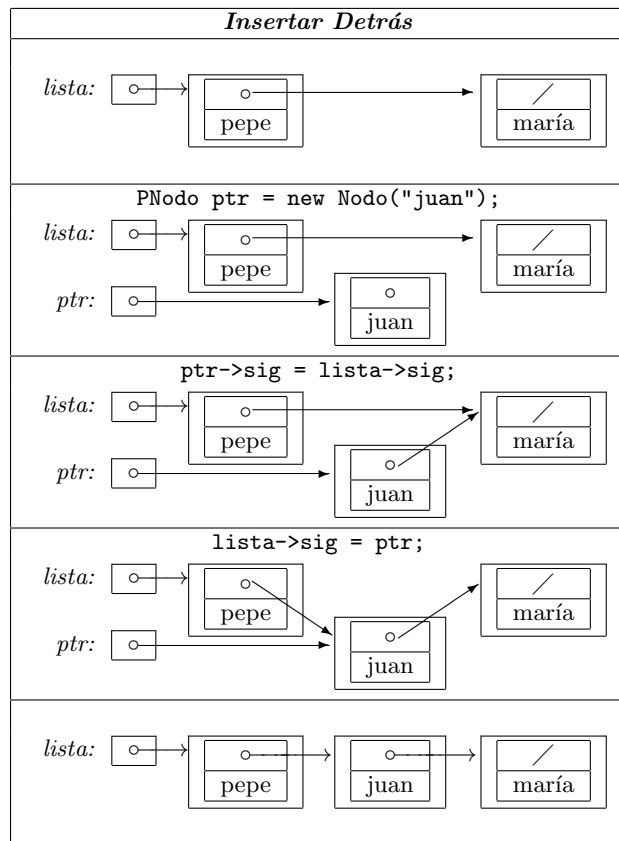
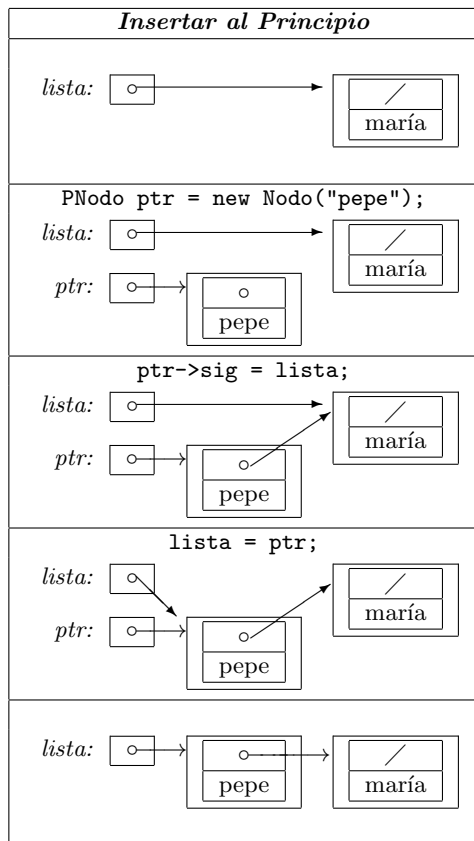
Así, en este caso, vemos que un campo de la estructura es de tipo puntero a la propia estructura, por lo que es necesario definir el tipo puntero antes de definir la estructura. Sin embargo, la estructura todavía no ha sido definida, por lo que no se puede definir un puntero a ella. Por ello es necesario realizar una *declaración adelantada* de un *tipo incompleto* del tipo de la variable dinámica, donde se declara que un determinado identificador es una estructura o clase, pero no se definen sus componentes.

```

struct Persona;           // Declaración adelantada del tipo incompleto Persona
typedef Persona* PPersona; // Definición de tipo Puntero a tipo incompleto Persona
struct Persona {           // Definición del tipo Persona
    PPersona sig;           // enlace a la siguiente estructura dinámica
    string nombre;
    string telefono;
};
int main()
{
    PPersona lista = NULL;
    string nm, tlf;
    cin >> nm;
    while (n != "fin") {
        cin >> tlf;
        PPersona ptr = new Persona;
        ptr->nombre = nm;
        ptr->telefono = tlf;
        ptr->sig = lista;
        lista = ptr;
        cin >> nm;
    }
    // ...
}

```





15.7. Operaciones con Listas Enlazadas

Aunque las listas enlazadas se pueden programar directamente entremezcladas con el código de resolución del problema en cuestión, es conveniente que su gestión se realice dentro de una *abstracción* que aisle su tratamiento y permita una mejor gestión de sus recursos.

Con objeto de facilitar su estudio, el siguiente ejemplo es una implementación *simplificada* del tipo abstracto de datos *lista*. Nótese, sin embargo, que otras implementaciones pueden mejorar notablemente su eficiencia.

Es importante considerar que en presencia de excepciones, esta implementación del TAD lista puede *perder* memoria en el caso de excepciones elevadas en el constructor de copia al duplicar una lista enlazada (véase 15.8).

```
//- lista.hpp -----
#ifndef _lista_hpp_
#define _lista_hpp_
#include <iostream>
#include <cassert>
namespace umalcc {
    //-----
    namespace lista_impl {
        template <typename Tipo>
        inline void intercambiar(Tipo& a, Tipo& b)
        {
            Tipo x = a;
            a = b;
            b = x;
        }
    }
    //-----
    template <typename Tipo>
    class Lista {
    public:
        ~Lista() { destruir(); }
        Lista() : sz(0), lista(NULL) {}
        //-----
        Lista(const Lista& o) : sz(0), lista(NULL) { duplicar(o); }
        //-----
        Lista& operator = (const Lista& o)
        {
            if (this != &o) {
                duplicar(o);
            }
            return *this;
        }
        //-----
        unsigned size() const { return sz; }
        //-----
        void clear()
        {
            destruir();
        }
        //-----
        void swap(Lista& o)
        {
            lista_impl::intercambiar(sz, o.sz);
            lista_impl::intercambiar(lista, o.lista);
        }
        //-----
        unsigned buscar(const Tipo& d) const
```



```

{
    unsigned i = 0;
    PNode ptr = lista;
    while ((ptr != NULL)&&(d != ptr->dato)) {
        ptr = ptr->sig;
        ++i;
    }
    return i;
}

//-----
void insertar_inicio(const Tipo& d)
{
    insertar_nodo(lista, d);
}

//-----
void insertar_final(const Tipo& d)
{
    if (lista == NULL) {
        insertar_nodo(lista, d);
    } else {
        PNode ptr = lista;
        while (ptr->sig != NULL) {
            ptr = ptr->sig;
        }
        insertar_nodo(ptr->sig, d);
    }
}

//-----
void insertar_ordenado(const Tipo& d)
{
    if ((lista == NULL)|| (d <= lista->dato)) {
        insertar_nodo(lista, d);
    } else {
        PNode ant = lista;
        PNode ptr = lista->sig;
        while ((ptr != NULL)&&(d > ptr->dato)) {
            ant = ptr;
            ptr = ptr->sig;
        }
        insertar_nodo(ant->sig, d);
    }
}

//-----
void insertar(unsigned n, const Tipo& d)
{
    if ((lista == NULL)|| (n == 0)) {
        insertar_nodo(lista, d);
    } else {
        unsigned i = 1;
        PNode ptr = lista;
        while ((ptr->sig != NULL)&&(i < n)) {
            ptr = ptr->sig;
            ++i;
        }
        insertar_nodo(ptr->sig, d);
    }
}

//-----
void eliminar_primer()

```

```

{
    if (lista != NULL) {
        eliminar_nodo(lista);
    }
}
//-----
void eliminar_ultimo()
{
    if (lista != NULL) {
        if (lista->sig == NULL) {
            eliminar_nodo(lista);
        } else {
            PNode ant = lista;
            PNode ptr = lista->sig;
            while (ptr->sig != NULL) {
                ant = ptr;
                ptr = ptr->sig;
            }
            eliminar_nodo(ant->sig);
        }
    }
}
//-----
void eliminar_elm(const Tipo& d)
{
    if (lista != NULL) {
        if (d == lista->dato) {
            eliminar_nodo(lista);
        } else {
            PNode ant = lista;
            PNode ptr = lista->sig;
            while ((ptr != NULL)&&(d != ptr->dato)) {
                ant = ptr;
                ptr = ptr->sig;
            }
            if (ptr != NULL) {
                eliminar_nodo(ant->sig);
            }
        }
    }
}
//-----
void eliminar(unsigned p)
{
    if (lista != NULL) {
        if (p == 0) {
            eliminar_nodo(lista);
        } else {
            unsigned i = 1;
            PNode ptr = lista;
            while ((ptr != NULL)&&(i < p)) {
                ptr = ptr->sig;
                ++i;
            }
            if ((ptr != NULL)&&(ptr->sig != NULL)) {
                eliminar_nodo(ptr->sig);
            }
        }
    }
}

```

```

}
//-----
void eliminar_todos(const Tipo& d)
{
    // borrar el primero si tiene dato
    while ((lista != NULL)&&(d == lista->dato)) {
        eliminar_nodo(lista);
    }
    if (lista != NULL) {
        // borrar todos los nodos que tienen dato a partir del segundo
        PNode ant = lista;
        PNode ptr = lista->sig;
        while (ptr != NULL) {
            if (d == ptr->dato) {
                eliminar_nodo(ant->sig);
            } else {
                ant = ptr;
            }
            ptr = ant->sig;
        }
    }
}
//-----
const Tipo& operator [] (unsigned i) const
{
    assert(i < size()); // Pre-condición
    PNode ptr = situar(i);
    return ptr->dato;
}
//-----
Tipo& operator [] (unsigned i)
{
    assert(i < size()); // Pre-condición
    PNode ptr = situar(i);
    return ptr->dato;
}
//-----
friend std::ostream& operator <<(std::ostream& out, const Lista& lista)
{
    out << vect.sz << " ";
    PNode ptr = lista.lista;
    while (ptr != NULL) {
        out << ptr->dato << " ";
        ptr = ptr->sig;
    }
    return out;
}
//-----
friend std::istream& operator >> (std::istream& in, Lista& lista)
{
    lista.clear();
    unsigned nelms;
    in >> nelms;
    for (unsigned i = 0; (i < nelms)&&(!in.fail()); ++i) {
        Tipo x;
        in >> x;
        if ( ! in.fail() ) {
            lista.insertar_final(x);
        }
    }
}

```

```

    }
    return in;
}
//-----
private:
//-----
struct Nodo;          // declaración adelantada
typedef Nodo* PNode;
struct Nodo {
    PNode sig;         // enlace
    Tipo dato;
    //-----
    // Nodo(const Tipo& d, PNode s = NULL) : sig(s), dato(d) {} // Insertar_Nodo Alternativo
    //-----
};
//-----
//-- Atributos --
//-----
unsigned sz;
PNode lista;
//-----
// Métodos Privados
//-----
void insertar_nodo(PNode& ptr, const Tipo& d)
{
    //-----
    // ptr = new Nodo(d, ptr); // Insertar_Nodo Alternativo
    //-----
    PNode aux = new Nodo;
    aux->dato = d;
    aux->sig = ptr;
    ptr = aux;
    //-----
    ++sz;
    //-----
}
//-----
void eliminar_nodo(PNode& ptr)
{
    assert(ptr != NULL);
    //-----
    --sz;
    //-----
    PNode aux = ptr;
    ptr = ptr->sig;
    delete aux;
    //-----
}
//-----
void destruir()
{
    while (lista != NULL) {
        eliminar_nodo(lista);
    }
}
//-----
void duplicar(const Lista& o)
{
    assert(this != &o);

```

```

        destruir();
        if (o.lista != NULL) {
            insertar_nodo(lista, o.lista->dato);
            PNode u = lista;
            PNode p = o.lista->sig;
            while (p != NULL) {
                insertar_nodo(u->sig, p->dato);
                u = u->sig;
                p = p->sig;
            }
        }
    }
    //-----
    PNode situar(unsigned n) const
    {
        unsigned i = 0;
        PNode ptr = lista;
        while ((ptr != NULL)&&(i < n)) {
            ptr = ptr->sig;
            ++i;
        }
        return ptr;
    }
    //-----
};
}
#endif
//-----

```

Se puede apreciar como tanto el constructor de copia, como el operador de asignación duplican la lista almacenada, y por el contrario tanto el destructor como el método `clear()` liberan todos los recursos que el objeto tenga asignados. Así mismo, el método `duplicar` invoca a la destrucción de los recursos que tuviese antes de duplicar la lista. El método `swap(...)` intercambia los contenidos almacenados en dos listas. Este método se puede utilizar para pasar el contenido de una lista a otra sin realizar duplicación del contenido.

A continuación se puede ver un ejemplo de utilización del tipo abstracto de datos *lista* definido anteriormente.

```

#include "lista.hpp"
#include <iostream>
using namespace std;
using namespace umalcc;

int main()
{
    try {
        Lista<int> lista;
        int dato;

        cout << "Introduzca lista entre llaves: ";
        cin >> lista;
        cout << lista << endl;

        cout << "Insertar al inicio. " ;
        cin >> dato;
        lista.insertar_inicio(dato);

        cout << "Ctor de Copia" << endl;
        Lista<int> copia = lista;
    }
}

```

```

    cout << "Copia:" << copia << endl;

    cout << "Insertar al final. " ;
    cin >> dato;
    lista.insertar_final(dato);

    cout << "Insertar Posición Dato. " ;
    unsigned pos;
    cin >> pos >> dato;
    lista.insertar(pos, dato);

    cout << "Lista:" << lista << endl;
    cout << "Copia:" << copia << endl;

    cout << "buscar un nodo. " ;
    cin >> dato;
    unsigned i = lista.buscar(dato);
    if (i < lista.size()) {
        cout << "Encontrado: " << lista[i] << endl;
    } else {
        cout << "No encontrado: " << dato << endl;
    }

    copia = lista;

    cout << "borrar un nodo. " ;
    cin >> dato;
    lista.eliminar_elm(dato);
    cout << lista << endl;

    cout << "Eliminar Posición. " ;
    cin >> pos;
    lista.eliminar(pos);

    cout << "eliminar todos. " ;
    cin >> dato;
    copia.eliminar_todos(dato);

    cout << "Lista:" << lista << endl;
    cout << "Copia:" << copia << endl;
} catch (const exception& e) {
    cerr << "Error: " << e.what() << endl;
} catch (...) {
    cerr << "Error: Inesperado" << endl;
}
}

```

Ⓐ Implementación Alternativa de los Operadores de Entrada y Salida de Datos

Es posible realizar una implementación alternativa (un poco más complicada) donde la secuencia de longitud indeterminada de elementos que contiene el vector se encuentra delimitada entre llaves, por ejemplo:

```
{ 1 2 3 4 5 6 }
```

```

//-lista.hpp -----
namespace umalcc {
    // .....
    template <typename Tipo>
    class Lista {

```

```

// .....
friend std::ostream& operator <<(std::ostream& out, const Lista& lista)
{
    out << "{ ";
    PNode ptr = lista.lista;
    while (ptr != NULL) {
        out << ptr->dato << " ";
        ptr = ptr->sig;
    }
    out << "}";
    return out;
}
//-----
friend std::istream& operator >> (std::istream& in, Lista& lista)
{
    char ch = '\0';
    in >> ch;
    if (ch != '{') {
        in.unget();
        in.setstate(std::ios::failbit);
    } else if ( ! in.fail() ) {
        lista.clear();
        in >> ch;
        while ((! in.fail()) && (ch != '}')) {
            in.unget();
            Tipo x;
            in >> x;
            if ( ! in.fail() ) {
                lista.insertar_final(x);
            }
            in >> ch;
        }
    }
    return in;
}
// .....
};
//-----

```

15.8. Gestión de Memoria Dinámica en Presencia de Excepciones

En secciones anteriores se ha explicado que la gestión de memoria dinámica se basa en estructuras de programación de *bajo nivel*, que debido a su complejidad son propensas a errores y pérdidas de recursos, por lo que es adecuado aislar su programación dentro de una abstracción de datos.

Además, si en una determinada pieza de software es posible que se eleven excepciones, ya sea explícitamente por el propio código del programador, como por código de bibliotecas y módulos externos, entonces el flujo de ejecución del programa se vuelve más impredecible, y puede suceder que determinadas zonas de código encargadas de la liberación de recursos no se ejecuten finalmente en el flujo de ejecución del programa.

Por lo tanto, en caso de que el software se ejecute en un entorno con presencia de excepciones, se hace incluso más necesario que toda la gestión de la memoria dinámica se realice dentro de abstracciones cuyos destructores se encarguen de liberar los recursos adquiridos, ya que cuando se lanza una excepción se destruyen todas las variables (invocando a su destructor) que hayan

sido definidas dentro de cada ámbito del que se salga hasta encontrar un capturador adecuado para la excepción elevada. De esta forma se liberan automáticamente todos los recursos asociados a dichas variables (véase RAI 22.1), que de otra forma probablemente no serían gestionados adecuadamente.

Así, la lista enlazada implementada en la sección anterior (véase 15.7) se comporta adecuadamente en presencia de excepciones, ya que en caso de que se destruya una determinada lista por la elevación de una excepción, sus recursos serán liberados por la invocación a su destructor.

Sin embargo, existe una excepción a este hecho. Si la excepción se eleva durante la propia construcción del objeto de tipo `Lista`, entonces el objeto no está completamente creado, por lo que en este caso no se invocará a su destructor, aunque sin embargo si se invocará a los destructores de sus componentes que hayan sido correctamente creados. Este hecho puede hacer que la memoria dinámica reservada durante la construcción del objeto se pierda, por lo que requiere que deba ser manejada atendiendo a estas circunstancias.

Además, en la gestión de memoria dinámica, la creación de una variable en memoria dinámica es susceptible a elevar excepciones por dos motivos fundamentales, en primer lugar puede suceder que la memoria dinámica se agote y no quede memoria dinámica suficiente para alojar al nuevo objeto, en cuyo caso se lanzará una excepción estándar (`bad_alloc`), y en segundo lugar, cuando la construcción de un objeto falla por alguna circunstancia, entonces elevará una excepción para indicar la causa del fallo. Por ejemplo, si durante la construcción de un objeto de tipo `Lista` mediante el constructor de copia se eleva una excepción, entonces la memoria alojada al duplicar la lista se perderá, ya que no se invocará al destructor de la clase `Lista`.

Por lo tanto, para que nuestro tipo abstracto de datos sea robusto también ante las excepciones elevadas durante su propia construcción, es necesario garantizar que se destruya la lista aunque el constructor de copia de la clase `Lista` eleve alguna excepción. Para ello, se puede definir una clase auxiliar `RAII_Lista` cuyo destructor libere los recursos asociados a una lista, de tal forma que se utilice para garantizar dicha destrucción durante la duplicación de la lista.

```
//- lista_raii.hpp -----
#ifndef _lista_raii_hpp_
#define _lista_raii_hpp_
#include <iostream>
#include <cassert>
namespace umalcc {
    //-----
    namespace lista_raii_impl {
        template <typename Tipo>
        inline void intercambiar(Tipo& a, Tipo& b)
        {
            Tipo x = a;
            a = b;
            b = x;
        }
    }
    //-----
    template <typename Tipo>
    class Lista {
    public:
        ~Lista() {
            RAI_Lista aux;
            aux.swap(lista);
        }
        //-----
        Lista() : sz(0), lista(NULL) {}
        //-----
        Lista(const Lista& o)
            : sz(0), lista(NULL)
        {
```



```

        RAII_Lista aux;
        aux.duplicar(o.lista);
        aux.swap(lista);
        sz = o.sz;
    }
    //-----
    Lista& operator = (const Lista& o)
    {
        if (this != &o) {
            Lista(o).swap(*this);
        }
        return *this;
    }
    //-----
    unsigned size() const { return sz; }
    //-----
    void clear()
    {
        Lista().swap(*this);
    }
    //-----
    void swap(Lista& o)
    {
        lista_raii_impl::intercambiar(sz, o.sz);
        lista_raii_impl::intercambiar(lista, o.lista);
    }
    //-----
    unsigned buscar(const Tipo& d) const
    {
        unsigned i = 0;
        PNode ptr = lista;
        while ((ptr != NULL)&&(d != ptr->dato)) {
            ptr = ptr->sig;
            ++i;
        }
        return i;
    }
    //-----
    void insertar_inicio(const Tipo& d)
    {
        insertar_nodo(lista, d);
    }
    //-----
    void insertar_final(const Tipo& d)
    {
        if (lista == NULL) {
            insertar_nodo(lista, d);
        } else {
            PNode ptr = lista;
            while (ptr->sig != NULL) {
                ptr = ptr->sig;
            }
            insertar_nodo(ptr->sig, d);
        }
    }
    //-----
    void insertar_ordenado(const Tipo& d)
    {
        if ((lista == NULL)|| (d <= lista->dato)) {

```

```

        insertar_nodo(lista, d);
    } else {
        PNode ant = lista;
        PNode ptr = lista->sig;
        while ((ptr != NULL)&&(d > ptr->dato)) {
            ant = ptr;
            ptr = ptr->sig;
        }
        insertar_nodo(ant->sig, d);
    }
}
//-----
void insertar(unsigned n, const Tipo& d)
{
    if ((lista == NULL)||(n == 0)) {
        insertar_nodo(lista, d);
    } else {
        unsigned i = 1;
        PNode ptr = lista;
        while ((ptr->sig != NULL)&&(i < n)) {
            ptr = ptr->sig;
            ++i;
        }
        insertar_nodo(ptr->sig, d);
    }
}
//-----
void eliminar_primer()
{
    if (lista != NULL) {
        eliminar_nodo(lista);
    }
}
//-----
void eliminar_ultimo()
{
    if (lista != NULL) {
        if (lista->sig == NULL) {
            eliminar_nodo(lista);
        } else {
            PNode ant = lista;
            PNode ptr = lista->sig;
            while (ptr->sig != NULL) {
                ant = ptr;
                ptr = ptr->sig;
            }
            eliminar_nodo(ant->sig);
        }
    }
}
//-----
void eliminar_elm(const Tipo& d)
{
    if (lista != NULL) {
        if (d == lista->dato) {
            eliminar_nodo(lista);
        } else {
            PNode ant = lista;
            PNode ptr = lista->sig;

```

```

        while ((ptr != NULL)&&(d != ptr->dato)) {
            ant = ptr;
            ptr = ptr->sig;
        }
        if (ptr != NULL) {
            eliminar_nodo(ant->sig);
        }
    }
}

//-----
void eliminar(unsigned p)
{
    if (lista != NULL) {
        if (p == 0) {
            eliminar_nodo(lista);
        } else {
            unsigned i = 1;
            PNode ptr = lista;
            while ((ptr != NULL)&&(i < p)) {
                ptr = ptr->sig;
                ++i;
            }
            if ((ptr != NULL)&&(ptr->sig != NULL)) {
                eliminar_nodo(ptr->sig);
            }
        }
    }
}

//-----
void eliminar_todos(const Tipo& d)
{
    // borrar el primero si tiene dato
    while ((lista != NULL)&&(d == lista->dato)) {
        eliminar_nodo(lista);
    }
    if (lista != NULL) {
        // borrar todos los nodos que tienen dato a partir del segundo
        PNode ant = lista;
        PNode ptr = lista->sig;
        while (ptr != NULL) {
            if (d == ptr->dato) {
                eliminar_nodo(ant->sig);
            } else {
                ant = ptr;
            }
            ptr = ant->sig;
        }
    }
}

//-----
const Tipo& operator [] (unsigned i) const
{
    assert(i < size()); // Pre-condición
    PNode ptr = situar(i);
    return ptr->dato;
}

//-----
Tipo& operator [] (unsigned i)

```

```

{
    assert(i < size()); // Pre-condición
    PNode ptr = situar(i);
    return ptr->dato;
}
//-----
friend std::ostream& operator <<(std::ostream& out, const Lista& lista)
{
    out << "{ ";
    PNode ptr = lista.lista;
    while (ptr != NULL) {
        out << ptr->dato << " ";
        ptr = ptr->sig;
    }
    out << "}";
    return out;
}
//-----
friend std::istream& operator >> (std::istream& in, Lista& lista)
{
    char ch = '\0';
    in >> ch;
    if (ch != '{') {
        in.unget();
        in.setstate(std::ios::failbit);
    } else if ( ! in.fail() ) {
        lista.clear();
        in >> ch;
        while ((! in.fail()) && (ch != '}')) {
            in.unget();
            Tipo x;
            in >> x;
            if ( ! in.fail() ) {
                lista.insertar_final(x);
            }
            in >> ch;
        }
    }
    return in;
}
//-----
private:
//-----
struct Nodo;          // declaración adelantada
typedef Nodo* PNode;
struct Nodo {
    PNode sig;          // enlace
    Tipo dato;
    //-----
    Nodo(const Tipo& d, PNode s = NULL) : sig(s), dato(d) {}
};
//-----
//-- Atributos --
//-----
unsigned sz;
PNode lista;
//-----
// Métodos Privados
//-----

```

```

void insertar_nodo(PNodo& ptr, const Tipo& d)
{
    ptr = new Nodo(d, ptr);
    ++sz;
}
//-----
void eliminar_nodo(PNodo& ptr)
{
    --sz;
    PNodo aux = ptr;
    ptr = ptr->sig;
    delete aux;
}
//-----
PNodo situar(unsigned n) const
{
    unsigned i = 0;
    PNodo ptr = lista;
    while ((ptr != NULL)&&(i < n)) {
        ptr = ptr->sig;
        ++i;
    }
    return ptr;
}
//-----
class RAII_Lista {
public:
    //-----
    ~RAII_Lista() { destruir(); }
    //-----
    RAII_Lista() : lista(NULL) {}
    //-----
    void swap(PNodo& l) {
        PNodo x = lista;
        lista = l;
        l = x;
    }
    //-----
    void duplicar(PNodo l)
    {
        destruir();
        if (l != NULL) {
            lista = new Nodo(l->dato, lista);
            PNodo u = lista;
            PNodo p = l->sig;
            while (p != NULL) {
                u->sig = new Nodo(p->dato, u->sig);
                u = u->sig;
                p = p->sig;
            }
        }
    }
    //-----
private:
    //-----
    PNodo lista;
    //-----
    RAII_Lista(const RAII_Lista& o);           // copia no permitida
    RAII_Lista& operator=(const RAII_Lista& o); // asignación no permitida

```

```

//-----
void destruir()
{
    while (lista != NULL) {
        PNode aux = lista;
        lista = lista->sig;
        delete aux;
    }
}
//-----
};
//-----
};
}
#endif
//-----

```

Construcción del Objeto en Memoria Dinámica

El operador `new` desempeña principalmente dos acciones, primero aloja espacio suficiente en memoria dinámica para almacenar la representación de un objeto, y segundo construye el objeto invocando al constructor adecuado según lo especificado en la sentencia. En caso de que falle la construcción del objeto, entonces automáticamente libera la memoria dinámica que había alojado previamente.

Por lo tanto, alojar la memoria dinámica utilizando en un único paso el constructor adecuado para construir el objeto con los valores necesarios es más robusto que alojar la memoria utilizando el constructor por defecto y posteriormente asignar los valores necesarios. Ya que si esta asignación falla y lanza una excepción, entonces la memoria alojada se perderá, mientras que si se hace en un único paso, si el constructor lanza una excepción entonces se desaloja automáticamente la memoria solicitada. Por ejemplo, si el método `insertar_nodo` de la clase `Lista` se hubiese definido de la siguiente forma:

```

void insertar_nodo(PNode& ptr, const Tipo& d)
{
    PNode aux = new Nodo;
    aux->dato = d;
    aux->sig = ptr;
    ptr = aux;
    ++sz;
}

```

sería susceptible de perder la memoria alojada para el nodo en caso de que la asignación del dato lanzase una excepción. Por el contrario, la siguiente definición es más robusta ante excepciones:

```

void insertar_nodo(PNode& ptr, const Tipo& d)
{
    ptr = new Nodo(d, ptr);
    ++sz;
}

```

ya que en caso de que la construcción del objeto fallase, se liberaría automáticamente la memoria previamente alojada para dicho objeto.

Utilización de Lista en Presencia de Excepciones

Las definiciones del tipo `Lista` anteriores se pueden comprobar en presencia de excepciones mediante el siguiente programa que utiliza un objeto que explícitamente eleva una excepción durante su construcción.

```

//- main.cpp -----
#include "lista.hpp"
#include "lista_raii.hpp"
#include <iostream>
#include <stdexcept>
using namespace std;
using namespace umalcc;

class X {
public:
    ~X() { ++cnt_dtor; }
    X(int v = 0) : valor(v) {
        if (cnt_ctor == 11) {
            cerr << "Error" << endl;
            throw runtime_error("X:X");
        }
        ++cnt_ctor;
    }
    X(const X& v) : valor(v.valor) {
        if (cnt_ctor == 11) {
            cerr << "Error" << endl;
            throw runtime_error("X:X");
        }
        ++cnt_ctor;
    }
    friend bool operator == (const X& a, const X& b) {
        return a.valor == b.valor;
    }
    friend bool operator != (const X& a, const X& b) {
        return a.valor != b.valor;
    }
    friend ostream& operator << (ostream& out, const X& x) {
        return out << x.valor ;
    }
    friend istream& operator >> (istream& in, X& x) {
        return in >> x.valor ;
    }
    //-----
    static unsigned cnt_construidos() { return cnt_ctor; }
    static unsigned cnt_destruidos() { return cnt_dtor; }
    //-----
private:
    static unsigned cnt_ctor;
    static unsigned cnt_dtor;
    int valor;
    //-----
};
//-----
unsigned X::cnt_ctor = 0; // definición e inicialización del atributo estático
unsigned X::cnt_dtor = 0; // definición e inicialización del atributo estático
//-----

int main()
{
    try {
        Lista<X> lista;
        int dato;

        cout << "Introduzca lista entre llaves: ";
    }
}

```

```

cin >> lista;
cout << lista << endl;

cout << "Insertar al inicio. " ;
cin >> dato;
lista.insertar_inicio(dato);

cout << "Ctor de Copia" << endl;
Lista<X> copia = lista;
cout << "Copia:" << copia << endl;

cout << "Insertar al final. " ;
cin >> dato;
lista.insertar_final(dato);

cout << "Insertar Posición Dato. " ;
unsigned pos;
cin >> pos >> dato;
lista.insertar(pos, dato);

cout << "Lista:" << lista << endl;
cout << "Copia:" << copia << endl;

cout << "buscar un nodo. " ;
cin >> dato;
unsigned i = lista.buscar(dato);
if (i < lista.size()) {
    cout << "Encontrado: " << lista[i] << endl;
} else {
    cout << "No encontrado: " << dato << endl;
}

cout << "Asignar: " << endl;
copia = lista;

cout << "borrar un nodo. " ;
cin >> dato;
lista.eliminar_elm(dato);
cout << lista << endl;

cout << "Eliminar Posición. " ;
cin >> pos;
lista.eliminar(pos);

cout << "eliminar todos. " ;
cin >> dato;
copia.eliminar_todos(dato);

cout << "Lista:" << lista << endl;
cout << "Copia:" << copia << endl;
} catch (const exception& e) {
    cerr << "Error: " << e.what() << endl;
} catch (...) {
    cerr << "Error: Inesperado" << endl;
}
cout << "Objetos contruidos: " << X::cnt_construidos() << endl;
cout << "Objetos destruidos: " << X::cnt_destruidos() << endl;
}
//-----

```


Donde la ejecución con la implementación de `Lista` sin considerar las excepciones (véase 15.7) produce el siguiente resultado:

```

Introduzca lista entre llaves: {1 2 3}
{ 1 2 3 }
Insertar al inicio. 0
Ctor de Copia
Error
Error: X::X
Objetos construidos: 11
Objetos destruidos: 8
ERROR, el número de NEW [9] es diferente del número de DELETE [6]

```

Mientras que la implementación de `Lista` utilizando la clase auxiliar `RAII_Lista` (véase 15.8) produce el siguiente resultado:

```

Introduzca lista entre llaves: {1 2 3}
{ 1 2 3 }
Insertar al inicio. 0
Ctor de Copia
Error
Error: X::X
Objetos construidos: 11
Objetos destruidos: 11
OK, el número de NEW [9] es igual al número de DELETE [9]

```

Con ayuda del programa de la sección 15.9 se puede realizar fácilmente la comprobación del número total de nodos alojados y desalojados.

15.9. Comprobación de Gestión de Memoria Dinámica

El siguiente código C++ se puede compilar junto con cualquier programa C++ de forma externa *no intrusiva*, y automáticamente realiza en tiempo de ejecución una cuenta del número de nodos alojados y liberados, de tal forma que al finalizar el programa escribe unas estadísticas comprobando si coinciden o no. Nótese que no es necesario modificar el programa principal para que se pueda comprobar automáticamente la cuenta de nodos alojados y liberados, es suficiente con la invocación al compilador.

```

g++ -ansi -Wall -Werror -o main main.cpp newdelcnt.cpp

//-----
//- newdelcnt.cpp -----
//-----
#include <iostream>
#include <cstdlib>
#include <new>
//-----
namespace {
    static unsigned inc_new(unsigned n = 0) throw()
    {
        static unsigned cnt = 0;
        return (cnt += n);
    }
    //-----
    static unsigned inc_delete(unsigned n = 0) throw()
    {
        static unsigned cnt = 0;
        return (cnt += n);
    }
}

```

```

//-----
static void check_new_delete() throw()
{
    if (inc_new() != inc_delete()) {
        std::cerr << "ERROR, el número de NEW [" << inc_new()
            << "] es diferente del número de DELETE ["
            << inc_delete() << "]" << std::endl;
#ifdef __WIN32__
        std::system("pause");
#endif
        std::exit(1234);
    } else {
        std::cerr << "OK, el número de NEW [" << inc_new()
            << "] es igual al número de DELETE ["
            << inc_delete() << "]" << std::endl;
    }
}
//-----
struct ExecAtEnd { ~ExecAtEnd() throw() { check_new_delete(); } };
static ExecAtEnd x;
//-----
} //end namespace
//-----
void* operator new (std::size_t sz) throw (std::bad_alloc)
{
    if (sz == 0) {
        sz = 1;    // malloc(0) es impredecible; lo evitamos.
    }
    void* p = static_cast<void*>(std::malloc(sz));
    if (p == 0) {
        std::cerr << "Error: New: Memory failure" << std::endl;
        throw std::bad_alloc();
    }
    inc_new(1);
    return p;
}
//-----
void operator delete (void* ptr) throw()
{
    if (ptr) {
        inc_delete(1);
        std::free(ptr);
    }
}
//-----

```

15.10. Operador de Dirección

En el lenguaje de programación C++ una variable de tipo puntero, además de apuntar a variables alojadas en memoria dinámica, también puede apuntar a variables alojadas en otras zonas de memoria gestionadas por el compilador (variables estáticas y automáticas). Para ello, es necesario poder obtener la dirección de dichas variables, de tal forma que la variable puntero pueda acceder a ellas. El operador que nos devuelve la dirección donde se encuentra una determinada variable es el operador prefijo unario “*ampersand*” (&). Por ejemplo:

```

struct Fecha {
    unsigned dia, mes, anyo;
};

```

```
int main()
{
    int x = 3;
    int* p = &x;

    cout << *p << endl; // escribe 3 en pantalla
    *p = 5;
    cout << x << endl; // escribe 5 en pantalla

    Fecha f;
    Fecha* pf = &f;
    pf->dia = 21;
    pf->mes = 02;
    pf->anyo = 2011;
}
```

Hay que tener presente que en estos casos, la variable de tipo puntero está apuntando a una zona de memoria gestionada por el compilador, por lo que el programador no debe utilizar los operadores **new** ni **delete**. Así mismo, también puede suceder que la variable de tipo puntero apunte a una variable automática que haya desaparecido debido a que su ámbito y tiempo de vida hayan expirado con anterioridad.

Nótese que esta característica es una estructura de programación de muy *bajo nivel* que aumenta aún más la complejidad de los programas, ya que a la complejidad de la gestión de los punteros y la memoria dinámica se añade la complejidad de gestionar también punteros a direcciones de memoria gestionadas por el compilador, y el programador debe discernir entre ambos tipos, así como sobre el tiempo de vida de las variables gestionadas por el compilador.

Capítulo 16

Introducción a los Contenedores de la Biblioteca Estándar (STL)

Los *contenedores* de la biblioteca estándar proporcionan un método general para almacenar y acceder a elementos homogéneos, proporcionando cada uno de ellos diferentes características que los hacen adecuados a diferentes necesidades.

En este capítulo introductorio se mostrarán las principales operaciones que se pueden realizar con los siguientes contenedores: el tipo `vector`, el tipo `stack`, y el tipo `queue` de la biblioteca estándar.

Paso de Parámetros de Contenedores

Los contenedores de la biblioteca estándar se pueden pasar como parámetros a subprogramas como cualquier otro tipo compuesto, y por lo tanto se aplican los mecanismos de paso de parámetros para tipos compuestos explicados en la sección 6.1. Es decir, los parámetros de entrada se pasarán por referencia constante, mientras que los parámetros de salida y entrada/salida se pasarán por referencia.

Así mismo, como norma general, salvo excepciones, no es adecuado que las funciones retornen valores de tipos de los contenedores, debido a la sobrecarga que generalmente conlleva dicha operación para el caso de los tipos compuestos. En estos casos suele ser más adecuado que el valor se devuelva como un parámetro por referencia.

16.1. Vector

El contenedor de tipo `vector<...>` representa una secuencia de elementos homogéneos optimizada para el acceso directo a los elementos según su posición. Para utilizar un contenedor de tipo `vector` se debe incluir la biblioteca estándar `<vector>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <vector>
```

El tipo `vector` es similar al tipo `array`, salvo en el hecho de que los vectores se caracterizan porque su tamaño puede crecer en tiempo de ejecución dependiendo de las necesidades surgidas durante la ejecución del programa. Por ello, a diferencia de los arrays, no es necesario especificar un tamaño fijo y predeterminado en tiempo de compilación respecto al número de elementos que pueda contener.

El número máximo de elementos que se pueden almacenar en una variable de tipo `vector` no está especificado, y se pueden almacenar elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Nótese que en los siguientes ejemplos, por simplicidad, tanto el número de elementos como el valor inicial de los mismos están especificados mediante valores constantes, sin embargo, también se pueden especificar como valores de variables y expresiones calculados en tiempo de ejecución.

Instanciación del Tipo Vector

Se pueden definir explícitamente instanciaciones del tipo `vector` para tipos de elementos concretos mediante la declaración `typedef`. Por ejemplo la siguiente definición declara el tipo `Vect_Int` como un tipo vector de números enteros.

```
typedef std::vector<int> Vect_Int;
```

Las siguientes definiciones declaran el tipo `Matriz` como un vector de dos dimensiones de números enteros.

```
typedef std::vector<int> Fila;
typedef std::vector<Fila> Matriz;
```

Construcción de un Objeto de Tipo Vector

Se pueden definir variables de un tipo vector previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (`v1` y `v2`) de tipo vector de números enteros, así como la variable `m` de tipo vector de dos dimensiones de números enteros.

```
int main()
{
    Vect_Int v1;           // vector de enteros vacío
    std::vector<int> v2;    // vector de enteros vacío
    Matriz m;              // vector de dos dimensiones de enteros vacío
    // ...
}
```

El constructor por defecto del tipo `vector` crea un objeto `vector` inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible crear un objeto vector con un número inicial de elementos con un valor inicial por defecto, al que posteriormente se le podrán añadir nuevos elementos. Este número inicial de elementos puede ser tanto una constante, como el valor de una variable calculado en tiempo de ejecución.

```
int main()
{
    Vect_Int v1(10);        // vector con 10 enteros con valor inicial 0
    Matriz m(10, Fila(5));  // matriz de 10x5 enteros con valor inicial 0
    // ...
}
```

Así mismo, también se puede especificar el valor por defecto que tomarán los elementos creados inicialmente.

```
int main()
{
    Vect_Int v1(10, 3);      // vector con 10 enteros con valor inicial 3
    Matriz m(10, Fila(5, 3)); // matriz de 10x5 enteros con valor inicial 3
    // ...
}
```

También es posible inicializar un vector con el contenido de otro vector de igual tipo:

```

int main()
{
    Vect_Int v1(10, 3);    // vector con 10 enteros con valor inicial 3
    Vect_Int v2(v1);       // vector con el mismo contenido de v1
    Vect_Int v3 = v1;      // vector con el mismo contenido de v1
    // ...
}

```

Asignación de un Objeto de Tipo Vector

Es posible la asignación de vectores de igual tipo. En este caso, se destruye el valor anterior del vector destino de la asignación.

```

int main()
{
    Vect_Int v1(10, 3);    // vector con 10 enteros con valor inicial 3
    Vect_Int v2;           // vector de enteros vacío

    v2 = v1;               // asigna el contenido de v1 a v2
    v2.assign(5, 7);       // asigna 5 enteros con valor inicial 7
}

```

Así mismo, también es posible intercambiar (*swap* en inglés) el contenido entre dos vectores utilizando el método `swap`. Por ejemplo:

```

int main()
{
    Vect_Int v1(10, 5);    // v1 = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }
    Vect_Int v2(5, 7);     // v2 = { 7, 7, 7, 7, 7 }

    v1.swap(v2);           // v1 = { 7, 7, 7, 7, 7 }
                          // v2 = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }
}

```

Control sobre los Elementos de un Vector

El número de elementos actualmente almacenados en un vector se obtiene mediante el método `size()`. Por ejemplo:

```

int main()
{
    Vect_Int v1(10, 3);    // vector con 10 enteros con valor inicial 3
    unsigned n = v1.size(); // número de elementos de v1
}

```

Es posible tanto añadir un elemento al final de un vector mediante el método `push_back(...)`, como eliminar el último elemento del vector mediante el método `pop_back()` (en este caso el vector no debe estar vacío). Así mismo, el método `clear()` elimina todos los elementos del vector. Por ejemplo:

```

int main()
{
    Vect_Int v(5);          // v = { 0, 0, 0, 0, 0 }
    for (int i = 1; i <= 3; ++i) {
        v.push_back(i);
    }                      // v = { 0, 0, 0, 0, 0, 1, 2, 3 }
    for (unsigned i = 0; i < v.size(); ++i) {
        cout << v[i] << " ";
    }                      // muestra: 0 0 0 0 0 1 2 3
    cout << endl;
}

```

```

    while (v.size() > 3) {
        v.pop_back();
    }
    v.clear();
}
// v = { 0, 0, 0 }
// v = { }

```

También es posible cambiar el tamaño del número de elementos almacenados en el vector. Así, el método `resize(...)` reajusta el número de elementos contenidos en un vector. Si el número especificado es menor que el número actual de elementos, se eliminarán del final del vector tantos elementos como sea necesario para reducir el vector hasta el número de elementos especificado. Si por el contrario, el número especificado es mayor que el número actual de elementos, entonces se añadirán al final del vector tantos elementos como sea necesario para alcanzar el nuevo número de elementos especificado (con el valor especificado o con el valor por defecto). Por ejemplo:

```

int main()
{
    Vect_Int v(10, 1); // v = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }
    v.resize(5);       // v = { 1, 1, 1, 1, 1 }
    v.resize(9, 2);    // v = { 1, 1, 1, 1, 1, 2, 2, 2, 2 }
    v.resize(7, 3);    // v = { 1, 1, 1, 1, 1, 2, 2 }
    v.resize(10);      // v = { 1, 1, 1, 1, 1, 2, 2, 0, 0, 0 }
}

```

Acceso a los Elementos de un Vector

Es posible acceder a cada elemento del vector individualmente, según el índice de la posición que ocupe, tanto para obtener su valor almacenado, como para modificarlo mediante el operador de indexación `[]`. El primer elemento ocupa la posición cero (0), y el último elemento almacenado en el vector `v` ocupa la posición `v.size()-1`. Por ejemplo:

```

int main()
{
    Vect_Int v(10);
    for (unsigned i = 0; i < v.size(); ++i) {
        v[i] = i;
    }
    for (unsigned i = 0; i < v.size(); ++i) {
        cout << v[i] << " ";
    }
    cout << endl;
}

```

El lenguaje de programación C++ no comprueba que los accesos a los elementos de un vector sean correctos y se encuentren dentro de los límites válidos del vector, por lo que será responsabilidad del programador comprobar que así sea.

También es posible acceder a un determinado elemento mediante el método `at(i)`, de tal forma que si el valor del índice `i` está fuera del rango válido, entonces se lanzará una excepción `out_of_range`. Se puede tanto utilizar como modificar el valor de este elemento.

```

int main()
{
    Vect_Int v(10);
    for (unsigned i = 0; i < v.size(); ++i) {
        v.at(i) = i;
    }
    for (unsigned i = 0; i < v.size(); ++i) {
        cout << v.at(i) << " ";
    }
    cout << endl;
}

```


Comparación Lexicográfica entre Vectores

Es posible realizar la comparación lexicográfica (`==`, `!=`, `>`, `>=`, `<`, `<=`) entre vectores del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes del vector. Por ejemplo:

```
int main()
{
    Vect_Int v1(10, 7); // v1 = { 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 }
    Vect_Int v2(5, 3);  // v2 = { 3, 3, 3, 3, 3 }
    if (v1 == v2) {
        cout << "Iguales" << endl;
    } else {
        cout << "Distintos" << endl;
    }
    if (v1 < v2) {
        cout << "Menor" << endl;
    } else {
        cout << "Mayor o Igual" << endl;
    }
}
```

16.2. Stack

El contenedor de tipo `stack<...>` representa el tipo abstracto de datos *Pila*, como una colección ordenada (según el orden de inserción) de elementos homogéneos donde se pueden introducir elementos (manteniendo el orden de inserción) y sacar elementos de ella (en orden inverso al orden de inserción), de tal forma que el primer elemento que sale de la pila es el último elemento que ha sido introducido en ella. Además, también es posible comprobar si la pila contiene elementos, de tal forma que no se podrá sacar ningún elemento de una pila vacía. Para utilizar un contenedor de tipo `stack` se debe incluir la biblioteca estándar `<stack>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <stack>
```

El número máximo de elementos que se pueden almacenar en una variable de tipo `stack` no está especificado, y se pueden introducir elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Instanciación del Tipo Stack

Se pueden definir explícitamente instancias del tipo `stack` para tipos de elementos concretos mediante la declaración `typedef`. Por ejemplo la siguiente definición declara el tipo `Stack_Int` como un tipo pila de números enteros.

```
typedef std::stack<int> Stack_Int;
```

Construcción de un Objeto de Tipo Pila

Se pueden definir variables de un tipo pila previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (`s1` y `s2`) de tipo pila de números enteros.

```
int main()
{
    Stack_Int s1;          // stack de enteros vacío
    std::stack<int> s2;    // stack de enteros vacío
    // ...
}
```

El constructor por defecto del tipo `stack` crea un objeto `stack` inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible inicializar una pila con el contenido de otra pila de igual tipo:

```
int main()
{
    Stack_Int s1;          // stack de enteros vacío
    // ...
    Stack_Int s2(s1);      // stack con el mismo contenido de s1
    Stack_Int s3 = s1;     // stack con el mismo contenido de s1
    // ...
}
```

Asignación de un Objeto de Tipo Pila

Es posible la asignación de pilas de igual tipo. En este caso, se destruye el valor anterior de la pila destino de la asignación.

```
int main()
{
    Stack_Int s1;          // stack de enteros vacío
    Stack_Int s2;          // stack de enteros vacío

    s2 = s1;               // asigna el contenido de s1 a s2
}
```

Control y Acceso a los Elementos de una Pila

Es posible tanto añadir un elemento una pila mediante el método `push(...)`, como eliminar el último elemento introducido en la pila mediante el método `pop()` (en este caso la pila no debe estar vacía).

Por otra parte, el método `empty()` indica si una pila está vacía o no, mientras que el número de elementos actualmente almacenados en una pila se obtiene mediante el método `size()`.

Así mismo, se puede acceder al último elemento introducido en la pila mediante el método `top()`. Se puede tanto utilizar como modificar el valor de este elemento (en este caso la pila no debe estar vacía).

Por ejemplo:

```
int main()
{
    Stack_Int s;                                // s = { }
    for (int i = 1; i <= 3; ++i) {
        s.push(i);
    }                                           // s = { 1, 2, 3 }

    s.top() = 5;                               // s = { 1, 2, 5 }

    s.pop();                                   // s = { 1, 2 }
    s.pop();                                   // s = { 1 }
    s.push(7);                                 // s = { 1, 7 }
    s.push(9);                                 // s = { 1, 7, 9 }

    cout << s.size() << endl;                 // muestra: 3

    while (! s.empty()) {
        cout << s.top() << " ";               // muestra: 9 7 1
        s.pop();
    }                                           // s = { }
    cout << endl;
}
```

Comparación Lexicográfica entre Pilas

Es posible realizar la comparación lexicográfica (`==`, `!=`, `>`, `>=`, `<`, `<=`) entre pilas del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes de la pila. Por ejemplo:

```
int main()
{
    Stack_Int s1;
    Stack_Int s2;
    // ...
    if (s1 == s2) {
        cout << "Iguales" << endl;
    } else {
        cout << "Distintos" << endl;
    }
    if (s1 < s2) {
        cout << "Menor" << endl;
    } else {
        cout << "Mayor o Igual" << endl;
    }
}
```

16.3. Queue

El contenedor de tipo `queue<...>` representa el tipo abstracto de datos *Cola*, como una colección ordenada (según el orden de inserción) de elementos homogéneos donde se pueden introducir elementos (manteniendo el orden de inserción) y sacar elementos de ella (en el mismo orden al orden de inserción), de tal forma que el primer elemento que sale de la cola es el primer elemento que ha sido introducido en ella. Además, también es posible comprobar si la cola contiene elementos, de tal forma que no se podrá sacar ningún elemento de una cola vacía. Para utilizar un contenedor de tipo `queue` se debe incluir la biblioteca estándar `<queue>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <queue>
```

El número máximo de elementos que se pueden almacenar en una variable de tipo `queue` no está especificado, y se pueden introducir elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Instanciación del Tipo Queue

Se pueden definir explícitamente instanciaciones del tipo `queue` para tipos de elementos concretos mediante la declaración `typedef`. Por ejemplo la siguiente definición declara el tipo `Queue_Int` como un tipo cola de números enteros.

```
typedef std::queue<int> Queue_Int;
```

Construcción de un Objeto de Tipo Cola

Se pueden definir variables de un tipo cola previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (`c1` y `c2`) de tipo cola de números enteros.

```
int main()
{
    Queue_Int c1;           // queue de enteros vacío
    std::queue<int> c2;     // queue de enteros vacío
    // ...
}
```

El constructor por defecto del tipo `queue` crea un objeto `queue` inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible inicializar una cola con el contenido de otra cola de igual tipo:

```
int main()
{
    Queue_Int c1;          // queue de enteros vacío
    // ...
    Queue_Int c2(c1);      // queue con el mismo contenido de c1
    Queue_Int s3 = c1;     // queue con el mismo contenido de c1
    // ...
}
```

Asignación de un Objeto de Tipo Cola

Es posible la asignación de colas de igual tipo. En este caso, se destruye el valor anterior de la cola destino de la asignación.

```
int main()
{
    Queue_Int c1;          // queue de enteros vacío
    Queue_Int c2;          // queue de enteros vacío

    c2 = c1;               // asigna el contenido de c1 a c2
}
```

Control y Acceso a los Elementos de una Cola

Es posible tanto añadir un elemento una cola mediante el método `push(...)`, como eliminar el primer elemento introducido en la cola mediante el método `pop()` (en este caso la cola no debe estar vacía).

Por otra parte, el método `empty()` indica si una cola está vacía o no, mientras que el número de elementos actualmente almacenados en una cola se obtiene mediante el método `size()`.

Así mismo, se puede acceder al último elemento introducido en la cola mediante el método `back()`, así como al primer elemento introducido en ella mediante el método `front()`. Se pueden tanto utilizar como modificar el valor de estos elementos (en este caso la cola no debe estar vacía).

Por ejemplo:

```
int main()
{
    Queue_Int c;           // c = { }
    for (int i = 1; i <= 3; ++i) {
        c.push(i);
    }                       // c = { 1, 2, 3 }

    c.front() = 6;         // c = { 6, 2, 3 }
    c.back() = 5;          // c = { 6, 2, 5 }

    c.pop();               // c = { 2, 5 }
    c.pop();               // c = { 5 }
    c.push(7);             // c = { 5, 7 }
    c.push(9);             // c = { 5, 7, 9 }

    cout << c.size() << endl; // muestra: 3

    while (! c.empty()) {
        cout << c.front() << " "; // muestra: 5 7 9
        c.pop();
    }                       // c = { }
```

```
    cout << endl;
}
```

Comparación Lexicográfica entre Colas

Es posible realizar la comparación lexicográfica (`==`, `!=`, `>`, `>=`, `<`, `<=`) entre colas del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes de la cola. Por ejemplo:

```
int main()
{
    Queue_Int c1;
    Queue_Int c2;
    // ...
    if (c1 == c2) {
        cout << "Iguales" << endl;
    } else {
        cout << "Distintos" << endl;
    }
    if (c1 < c2) {
        cout << "Menor" << endl;
    } else {
        cout << "Mayor o Igual" << endl;
    }
}
```

16.4. Resolución de Problemas Utilizando Contenedores

Ejemplo 1: Agentes de Ventas

Diseñe un programa que lea y almacene las ventas realizadas por unos *agentes de ventas*, de tal forma que se eliminen aquellos agentes cuyas ventas sean inferiores a la media de las ventas realizadas.

```
//-----
#include <iostream>
#include <vector>
#include <string>
using namespace std;

struct Agente {
    string nombre;
    double ventas;
};

typedef vector<Agente> VAgentes;

void leer (VAgentes& v)
{
    v.clear();
    Agente a;
    cout << "Introduzca Nombre: ";
    getline(cin, a.nombre);
    while (( ! cin.fail()) && (a.nombre.size() > 0)) {
        cout << "Introduzca Ventas: ";
        cin >> a.ventas;
        cin.ignore(1000, '\n');
        v.push_back(a);
        cout << "Introduzca Nombre: ";
    }
}
```

```

        getline(cin, a.nombre);
    }
}

double media(const VAgentes& v)
{
    double suma=0.0;
    for (unsigned i = 0; i < v.size(); ++i) {
        suma += v[i].ventas;
    }
    return suma/double(v.size());
}

void eliminar(VAgentes& v, double media)
{
    unsigned i = 0;
    while (i < v.size()) {
        if (v[i].ventas < media) {
            v[i] = v[v.size()-1];
            v.pop_back();
        } else {
            ++i;
        }
    }
}

void eliminar_ordenado(VAgentes& v, double media)
{
    unsigned k = 0;
    for (unsigned i = 0; i < v.size(); ++i) {
        if(v[i].ventas >= media) {
            if(i != k) {
                v[k] = v[i];
            }
            ++k;
        }
    }
    v.resize(k);
}

void imprimir(const VAgentes& v)
{
    for (unsigned i = 0; i < v.size(); ++i) {
        cout << v[i].nombre << " " << v[i].ventas << endl;
    }
}

int main ()
{
    VAgentes v;
    leer(v);
    eliminar(v, media(v));
    imprimir(v);
}
//-----

```

Ejemplo 2: Multiplicación de Matrices

Diseñe un programa que lea dos matrices de tamaños arbitrarios y muestre el resultado de multiplicar ambas matrices.

```
//-----
#include <vector>
#include <iostream>
#include <iomanip>
using namespace std;

typedef vector <double> Fila;
typedef vector <Fila> Matriz;

void imprimir(const Matriz& m)
{
    for (unsigned f = 0; f < m.size(); ++f) {
        for (unsigned c = 0; c < m[f].size(); ++c) {
            cout << setw(10) << setprecision(4)
                << m[f][c] << " ";
        }
        cout << endl;
    }
}

void leer(Matriz& m)
{
    unsigned nf, nc;
    cout << "Introduzca el numero de filas: ";
    cin >> nf;
    cout << "Introduzca el numero de columnas: ";
    cin >> nc;
    m = Matriz(nf, Fila(nc));
    cout << "Introduzca los elementos: " << endl;
    for (unsigned f = 0; f < m.size(); ++f){
        for (unsigned c = 0; c < m[f].size(); ++c){
            cin >> m[f][c];
        }
    }
}

// otra opción más eficiente para la lectura de vectores
void leer_2(Matriz& m)
{
    unsigned nf, nc;
    cout << "Introduzca el numero de filas: ";
    cin >> nf;
    cout << "Introduzca el numero de columnas: ";
    cin >> nc;
    Matriz aux(nf, Fila(nc));
    cout << "Introduzca los elementos: " << endl;
    for (unsigned f = 0; f < aux.size(); ++f){
        for (unsigned c = 0; c < aux[f].size(); ++c){
            cin >> aux[f][c];
        }
    }
    m.swap(aux);
}

void multiplicar(const Matriz& m1, const Matriz& m2, Matriz& m3)
```

```

{
    m3.clear();
    if ((m1.size() > 0) && (m2.size() > 0) && (m2[0].size() > 0)
        && (m1[0].size() == m2.size())){
        Matriz aux(m1.size(), Fila(m2[0].size()));
        for (unsigned f = 0; f < aux.size(); ++f){
            for (unsigned c = 0; c < aux[f].size(); ++c){
                double suma = 0.0;
                for (unsigned k = 0; k < m2.size(); ++k){
                    suma += m1[f][k] * m2[k][c];
                }
                aux[f][c] = suma;
            }
        }
        m3.swap(aux);
    }
}

int main()
{
    Matriz m1, m2, m3;
    leer(m1);
    leer(m2);
    multiplicar(m1, m2, m3);
    if (m3.size() == 0) {
        cout << "Error en la multiplicación de Matrices" << endl;
    } else {
        imprimir(m3);
    }
}
//-----

```


Parte III

Programación Avanzada

**A PARTIR DE ESTE PUNTO,
EL TEXTO NECESITA SER
REVISADO EN
PROFUNDIDAD.
EN EL ESTADO ACTUAL, SE
ENCUENTRA EN UNA
VERSIÓN PRELIMINAR**

Capítulo 17

Programación Orientada a Objetos

```
#include <iostream>
using namespace std;
//-----
//-----
class Cloneable {
public:
    virtual Cloneable* clone() const =0;
};
//-----
class Volador {    // : public virtual XXX
public:
    virtual void despegar() =0;
    virtual void vuela(int ix, int iy) =0;
    virtual void aterriza() =0;
};
//-----
class Insectivoro {
public:
    virtual void comer() =0;
};
//-----
//-----
class Animal : public virtual Cloneable {
public:
    virtual ~Animal() {
        cout << "Muerto en ";
        print();
    }
    virtual void print() const {
        cout << "[ " << x << ", " << y << " ]" << endl;
    }
    virtual void mover(int ix, int iy) {
        x += ix; y += iy;
    }
    virtual void comer() =0;
protected:
    Animal(int _x = 0, int _y = 0) : x(_x), y(_y) {
        cout << "Nace en ";
        print();
    }
    Animal(const Animal& a) : x(a.x), y(a.y) {    // C++0x
        cout << "Nace en ";
        print();
    }
};
```

```

    }
private:
    int x, y;
    const Animal& operator=(const Animal& o);
};
//-----
class Mamifero : public Animal {
public:
    virtual void mover(int ix, int iy) {
        cout << "Andando" << endl;
        Animal::mover(ix, iy);
    }
    virtual void print() const {
        cout << "Estoy en ";
        Animal::print();
    }
};
//-----
class Murcielago : public Mamifero,
                  public virtual Volador, public virtual Insectivoro {
public:
    Murcielago() : Mamifero() {}
    virtual Murcielago* clone() const {
        return new Murcielago(*this);
    }
    virtual void mover(int ix, int iy) {
        cout << "Volando" << endl;
        Animal::mover(ix, iy);
    }
    virtual void comer() {
        cout << "Comiendo mosquitos ";
        Animal::print();
    }
    virtual void despega() {
        cout << "Despega ";
        Animal::print();
    }
    virtual void vuela(int ix, int iy) {
        mover(ix, iy);
    }
    virtual void aterrizo() {
        cout << "Aterrizo ";
        Animal::print();
    }
}
protected:
    Murcielago(const Murcielago& o) : Mamifero(o) {}
};
//-----
//-----
int main()
{
    Murcielago* m = new Murcielago();
    m->despega();
    m->vuela(5, 7);
    m->comer();
    m->aterrizo();
    m->print();
    Animal* a = m->clone();
    a->mover(11, 13);
}

```

```
a->comer();
a->print();
Murcielago* m2 = dynamic_cast<Murcielago*>(a);
if (m2 != NULL) {
    m2->vuela(5, 7);
}
delete m;
delete a;
}
//-----
```


Capítulo 18

Memoria Dinámica Avanzada

18.1. Memoria Dinámica de Agregados

Otra posibilidad consiste en solicitar *memoria dinámica para un agregado* de elementos. En tal caso, se genera un puntero al primer elemento del agregado, siendo el tipo devuelto por el operador `new` un puntero al tipo base. Por ejemplo para solicitar un agregado de 20 Personas:

```
typedef Persona* AD_Personas;
const int N_PERS = 20;

AD_Personas pers = new Persona[N_PERS];
```

accedemos a cada elemento de igual forma como si fuera un agregado normal:

```
pers[i].nombre
pers[i].fec_nacimiento
.....
```

Para *liberar un agregado dinámico* previamente solicitado:

```
delete [] pers;
```

Nota: En el caso de punteros, no hay diferencia en el tipo entre un puntero a un elemento simple, y un puntero a un agregado de elementos, por lo que será el propio programador el responsable de diferenciarlos, y de hecho liberarlos de formas diferentes (con los corchetes en caso de agregados). Una posibilidad para facilitar dicha tarea al programador consiste en definir el tipo de tal forma que indique que es un agregado dinámico, y no un puntero a un elemento.

Paso de Parámetros de Variables de Tipo Puntero a Agregado

Respecto al paso de agregados dinámicos como parámetros, y con objeto de diferenciar el paso de un puntero a una variable del paso de un puntero a un agregado, el paso de agregados dinámicos se realizará como el paso de un agregado abierto (constante o no), incluyendo o no, dependiendo de su necesidad el número de elementos del agregado, como se indica a continuación:

```
int ad_enteros[]          // agregado dinámico de enteros
const int adc_enteros[]   // agregado dinámico constante de enteros
```

Ejemplo:

```
#include <iostream>
typedef int* AD_Enteror;

void leer(int n, int numeros[])
{
```

```

    cout << "Introduce " << n << " elementos " << endl;
    for (int i = 0; i < n; ++i) {
        cin >> numeros[i];
    }
}
double media(int n, const int numeros[])
{
    double suma = 0.0;
    for (int i = 0; i < n; ++i) {
        suma += numeros[i];
    }
    return suma / double(n);
}
int main()
{
    int nelm;
    AD_Enteros numeros;    // agregado dinámico

    cout << "Introduce el número de elementos ";
    cin >> nelm;
    numeros = new int[nelm];
    leer(nelm, numeros);
    cout << "Media: " << media(nelm, numeros) << endl;
    delete [] numeros;
}

```

18.2. Punteros a Subprogramas

Es posible, así mismo, declarar *punteros a funciones*:

```

typedef void (*PtrFun)(int dato);

void imprimir_1(int valor)
{
    cout << "Valor: " << valor << endl;
}

void imprimir_2(int valor)
{
    cout << "Cuenta: " << valor << endl;
}

int main()
{
    PtrFun salida;        // vble puntero a función (devuelve void y recibe int)

    salida = imprimir_1; // asigna valor al puntero

    salida(20);           // llamada. no es necesario utilizar (*salida)(20);

    salida = imprimir_2; // asigna valor al puntero

    salida(20);           // llamada. no es necesario utilizar (*salida)(20);
}

```

18.3. Punteros Inteligentes

```
auto_ptr<...> unique_ptr<...>
```

`shared_ptr<...> weak_ptr<...>`
otros definidos por los usuarios

Capítulo 19

Tipos Abstractos de Datos Avanzados

19.1. Punteros a Miembros

Punteros a miembros de clases (atributos y métodos). Nótese los paréntesis en la “llamada a través de puntero a miembro” por cuestiones de precedencia.

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

struct Clase_X {
    int v;
    void miembro() { cout << v << " " << this << endl; }
};

typedef void (Clase_X::*PointerToClaseXMemberFunction) ();
typedef int Clase_X::*PointerToClaseXMemberAttr;

int main()
{
    PointerToClaseXMemberFunction pmf = &Clase_X::miembro;
    PointerToClaseXMemberAttr pma = &Clase_X::v;

    Clase_X x;

    x.*pma = 3; // acceso a traves de puntero a miembro
    x.miembro(); // llamada directa a miembro
    (x.*pmf)(); // llamada a traves de puntero a miembro

    Clase_X* px = &x;

    px->*pma = 5; // acceso a traves de puntero a miembro
    px->miembro(); // llamada directa a miembro
    (px->*pmf)(); // llamada a traves de puntero a miembro
}
```

19.2. Ocultar la Implementación

Aunque C++ no soporta directamente tipos opacos, dicha característica se puede simular muy fácilmente con las herramientas que C++ ofrece. Hay dos corrientes principales a la hora de

solucionar el problema, aunque hay otras menores que no comentaremos:

Puntero a la Implementación

```
//-- elem.hpp -----
#ifndef _elem_hpp_
#define _elem_hpp_
namespace elem {
    class Elem {
    public:
        ~Elem() throw();
        Elem();
        Elem(const Elem& o);
        Elem& operator=(const Elem& o);
        int get_val() const;
        void set_val(int);
        void swap(Elem& o);
    private:
        class ElemImpl;
        ElemImpl* pimpl;
    }; // class Elem
} // namespace elem
#endif

/-- main.cpp -----
#include <iostream>
#include "elem.hpp"
using namespace std;
using namespace elem;
int main()
{
    Elem e;
    e.set_val(7);
    cout << e.get_val() << endl;
    Elem e2(e);
    e2.set_val(e2.get_val()+2);
    cout << e2.get_val() << endl;
    e2 = e;
    cout << e2.get_val() << endl;
}

/-- elem.cpp -----
#include "elem.hpp"
#include <iostream>
using namespace std;
namespace elem {
    class Elem::ElemImpl {
    public:
        ~ElemImpl() throw() { cout << "destrucción de: " << val << endl; }
        ElemImpl() : val(0) {}
        ElemImpl(const Elem& o) : val(o.val) {}
        ElemImpl& operator=(const ElemImpl& o) { val = o.val; return *this; }
        int get_val() const { return val; }
        void set_val(int v) { val = v; }
    private:
        int val;
    }; // class ElemImpl
    Elem::~Elem() throw()
    {
        delete pimpl;
    }
}
```

```

Elem::Elem()
    : pimpl(new ElemImpl)
{}
Elem::Elem(const Elem& o)
    : pimpl(new ElemImpl(*o.pimpl))
{}
Elem& Elem::operator=(const Elem& o)
{
    if (this != &o) {
        Elem(o).swap(*this);
    }
    return *this;
}
int Elem::get_val() const
{
    return pimpl->get_val();
}
void Elem::set_val(int v)
{
    pimpl->set_val(v);
}
void Elem::swap(Elem& o)
{
    ElemImpl* aux = o.pimpl;
    o.pimpl = pimpl;
    pimpl = aux;
}
} // namespace elem
//-- fin -----

```

Puntero a la Implementación

```

//-- complejo.hpp -----
#ifndef _complejo_hpp_
#define _complejo_hpp_
#include <iostream>
namespace umalcc {
    class Complejo {
    public:
        ~Complejo();
        Complejo();
        Complejo(double r, double i);
        Complejo(const Complejo& c);
        Complejo& operator=(const Complejo& c);
        double get_real() const;
        double get_imag() const;
        Complejo& operator+=(const Complejo& c);
        Complejo& swap(Complejo& c);
    private:
        struct Hidden* impl;
    };
    inline Complejo operator+(const Complejo& c1, const Complejo& c2) {
        Complejo res = c1;
        res += c2;
        return res;
    }
    inline std::ostream& operator<<(std::ostream& out, const Complejo& c) {
        out << "[ " << c.get_real() << ", " << c.get_imag() << " ]";
        return out;
    }
}

```

```

    }
} // namespace umalcc
#endif
// - main.cpp -----
#include <iostream>
#include <string>
#include "complejo.hpp"
using namespace std;
using namespace umalcc;

int main()
{
    Complejo c1;
    Complejo c2;
    Complejo c3 = c1 + c2;
    cout << c1 << " " << c2 << " " << c3 << endl;
    c3 += Complejo(4, 5);
    cout << c3 << endl;
    c3 = Complejo(411, 535);
    cout << c3 << endl;
}

// - complejo.cpp -----
#include "complejo.hpp"
namespace umalcc {
    struct Hidden {
        double real, imag;
    };
    Complejo::~Complejo() {
        delete impl;
    }
    Complejo::Complejo() : impl() {}
    Complejo::Complejo(double r, double i) : impl() {
        impl = new Hidden;
        impl->real = r;
        impl->imag = i;
    }
    Complejo::Complejo(const Complejo& c) : impl() {
        if (c.impl != NULL) {
            impl = new Hidden;
            impl->real = c.impl->real;
            impl->imag = c.impl->imag;
        }
    }
    Complejo& Complejo::operator=(const Complejo& c) {
        if (this != &c) {
            Complejo(c).swap(*this);
        }
        return *this;
    }
    Complejo& Complejo::swap(Complejo& c) {
        swap(impl, c.impl);
    }
    double Complejo::get_real() const {
        return (impl == NULL) ? 0.0 : impl->real;
    }
    double Complejo::get_imag() const {
        return (impl == NULL) ? 0.0 : impl->imag;
    }
    Complejo& Complejo::operator+=(const Complejo& c) {

```



```

        if (impl == NULL) {
            *this = c;
        } else {
            impl->real += c.get_real();
            impl->imag += c.get_imag();
        }
        return *this;
    }
} // namespace umalcc
//-----

```

Base Abstracta

```

//-- elem.hpp -----
#ifndef _elem_hpp_
#define _elem_hpp_
namespace elem {
    class Elem {
    public:
        static Elem* create();
        virtual Elem* clone() const = 0;
        virtual ~Elem() throw();
        virtual int get_val() const = 0;
        virtual void set_val(int) = 0;
    }; // class Elem
} // namespace elem
#endif

//-- main.cpp -----
#include <iostream>
#include <memory>
#include "elem.hpp"
using namespace std;
using namespace elem;
int main()
{
    auto_ptr<Elem> e(Elem::create());
    e->set_val(7);
    cout << e->get_val() << endl;
    auto_ptr<Elem> e2(e->clone());
    e2->set_val(e2->get_val()+2);
    cout << e2->get_val() << endl;
    e2 = auto_ptr<Elem>(e->clone());
    cout << e2->get_val() << endl;
}

//-- elem.cpp -----
// #include "elem.hpp"
#include <iostream>
using namespace std;
namespace {
    class ElemImpl : public elem::Elem {
    public:
        virtual ElemImpl* clone() const { return new ElemImpl(*this); }
        virtual ~ElemImpl() throw()
        { cout << "destrucción de: " << val << endl; }
        ElemImpl() : _ClaseBase(), val(0) {}
        // ElemImpl(const Elem& o) : _ClaseBase(o), val(o.val) {}
        // ElemImpl& operator=(const ElemImpl& o) { val = o.val; return *this; }
        virtual int get_val() const { return val; }
        virtual void set_val(int v) { val = v; }
    };
}

```

```

private:
    typedef elem::Elem _ClaseBase;
    int val;
}; // class ElemImpl
}
namespace elem {
    Elem* Elem::create() { return new ElemImpl(); }
    Elem::~Elem() throw() {}
} // namespace elem
//-- fin -----

```

19.3. Control de Elementos de un Contenedor

En esta sección veremos como acceder a elementos de un contenedor definido por nosotros, pero manteniendo el control sobre las operaciones que se realizan sobre el mismo:

```

//-- matriz.hpp -----
#ifndef _matriz_hpp_
#define _matriz_hpp_
/*
 * Matriz Dispersa
 *
 * Definición de tipos
 *
 *     typedef Matriz<int> Mat_Int;
 *
 * Definición de Variables
 *
 *     Mat_Int mat(NFIL, NCOL);
 *
 * Definición de Constantes
 *
 *     const Mat_Int aux2(mat);
 *
 * Operaciones
 *
 *     aux1 = mat;           // asignación
 *     print(mat);          // paso de parámetros
 *     unsigned nfil = mat.getnfil(); // número de filas
 *     unsigned ncol = mat.getncol(); // número de columnas
 *     unsigned nelm = mat.getnelm(); // número de elementos almacenados
 *     mat(1U, 3U) = 4;      // asignación a elemento
 *     x = mat(1U, 3U);     // valor de elemento
 *
 * Excepciones
 *
 *     OutOfRange
 */
#include <map>
#include <iterator>

namespace matriz {

    class OutOfRange {};

    template <typename Tipo>
    class Ref_Elm;
    template <typename Tipo>

```

```

class RefC_Elm;

/-- Matriz -----
template <typename Tipo>
class Matriz {
private:
    friend class Ref_Elm<Tipo>;
    friend class RefC_Elm<Tipo>;
    //
    typedef Tipo                Tipo_Elm;
    typedef Ref_Elm<Tipo_Elm>    Tipo_Ref_Elm;
    typedef RefC_Elm<Tipo_Elm>  Tipo_RefC_Elm;

    typedef std::map<unsigned,Tipo_Elm>    Lista_Elm;
    typedef std::map<unsigned,Lista_Elm>    Lista_Fila;

    typedef typename Lista_Elm::iterator    ItElm;
    typedef typename Lista_Fila::iterator    ItFil;
    typedef typename Lista_Elm::const_iterator ItCElm;
    typedef typename Lista_Fila::const_iterator ItCFil;
    // atributos
    unsigned nfil;
    unsigned ncol;
    Lista_Fila elm;
public:
    Matriz(unsigned nf, unsigned nc) : nfil(nf), ncol(nc) {}
    unsigned getnfil() const { return nfil; }
    unsigned getncol() const { return ncol; }
    unsigned size() const;
    unsigned getnfilalm() const { return elm.size(); } //DEBUG
    const Tipo_RefC_Elm operator() (unsigned f, unsigned c) const
        throw(OutOfRangeException);
    Tipo_Ref_Elm operator() (unsigned f, unsigned c)
        throw(OutOfRangeException);
}; // class Matriz

/-- RefC_Elm -----
template <typename Tipo>
class RefC_Elm {
private:
    friend class Matriz<Tipo>;
    //
    typedef Tipo                Tipo_Elm;
    typedef Matriz<Tipo_Elm>    Matriz_Elm;
    //
    const Matriz_Elm* mat;
    unsigned fil;
    unsigned col;
    //
    RefC_Elm(const Matriz_Elm* m, unsigned f, unsigned c)
        : mat(m), fil(f), col(c) {}
public:
    operator Tipo_Elm () const;
}; // class RefC_Elm

/-- Ref_Elm -----
template <typename Tipo>
class Ref_Elm {
private:
    friend class Matriz<Tipo>;
    //

```

```

typedef Tipo                Tipo_Elm;
typedef Matriz<Tipo_Elm>    Matriz_Elm;
//
Matriz_Elm* mat;
unsigned fil;
unsigned col;
//
Ref_Elm(Matriz_Elm* m, unsigned f, unsigned c)
: mat(m), fil(f), col(c) {}
void eliminar_elemento_si_existe();
public:
    Ref_Elm& operator=(const Tipo_Elm& e);
    operator RefC_Elm<Tipo> () const;
}; // class Ref_Elm
//-- Matriz -----
template <typename Tipo>
const typename Matriz<Tipo>::Tipo_RefC_Elm
Matriz<Tipo>::operator() (unsigned f, unsigned c) const throw(OutOfRangeException)
{
    if ((f >= nfil)|| (c >= ncol)) {
        throw OutOfRange();
    }
    return Tipo_RefC_Elm(this, f, c);
}
template <typename Tipo>
typename Matriz<Tipo>::Tipo_Ref_Elm
Matriz<Tipo>::operator() (unsigned f, unsigned c) throw(OutOfRangeException)
{
    if ((f >= nfil)|| (c >= ncol)) {
        throw OutOfRange();
    }
    return Tipo_Ref_Elm(this, f, c);
}

template <typename Tipo>
unsigned Matriz<Tipo>::size() const
{
    unsigned nelm = 0;
    for (ItCFil i = elm.begin(); i != elm.end(); ++i) {
        nelm += i->second.size();
    }
    return nelm;
}
//-- RefC_Elm -----
template <typename Tipo>
RefC_Elm<Tipo>::operator typename RefC_Elm<Tipo>::Tipo_Elm () const
{
    typedef typename Matriz_Elm::ItCElm ItCElm;
    typedef typename Matriz_Elm::ItCFil ItCFil;

    ItCFil itfil = mat->elm.find(fil);
    if (itfil == mat->elm.end()) {
        return Tipo_Elm();
    } else {
        ItCElm itelm = itfil->second.find(col);
        if (itelm == itfil->second.end()) {
            return Tipo_Elm();
        } else {
            return itelm->second;
        }
    }
}

```

```

    }
}
}
//-- Ref_Elm -----
template <typename Tipo>
Ref_Elm<Tipo>::operator RefC_Elm<Tipo> () const
{
    return RefC_Elm<Tipo>(mat, fil, col);
}
template <typename Tipo>
Ref_Elm<Tipo>& Ref_Elm<Tipo>::operator=(const Tipo_Elm& e)
{
    if (e == Tipo_Elm()) {
        eliminar_elemento_si_existe();
    } else {
        typename Matriz_Elm::Lista_Elm& fila = mat->elm[fil];
        fila[col] = e;
    }
    return *this;
}
template <typename Tipo>
void Ref_Elm<Tipo>::eliminar_elemento_si_existe()
{
    typedef typename Matriz_Elm::ItElm      ItElm;
    typedef typename Matriz_Elm::ItFil      ItFil;

    ItFil itfil = mat->elm.find(fil);
    if (itfil != mat->elm.end()) {
        ItElm itelm = itfil->second.find(col);
        if (itelm != itfil->second.end()) {
            itfil->second.erase(itelm);
            if (itfil->second.empty()) {
                mat->elm.erase(itfil);
            }
        }
    }
}

}
}
//-----
} //namespace matriz
#endif
//-- main.cpp -----
#include <iostream>
#include <string>

#include "matriz.hpp"

using namespace std;
using namespace matriz;

typedef Matriz<int> Mat_Int;

const unsigned NFIL = 5U;
const unsigned NCOL = 6U;

void
print(const Mat_Int& mat)
{
    cout << "Número de filas reales: " << mat.getnfilalm() << endl;
    cout << "Número de elementos almacenados: " << mat.size() << endl;
}

```

```

    for (unsigned f = 0; f < mat.getnfil(); ++f) {
        for (unsigned c = 0; c < mat.getncol(); ++c) {
            cout << mat(f,c) << " ";
        }
        cout << endl;
    }
    cout << endl;
}
int
main()
{
    try {
        Mat_Int mat(NFIL,NCOL);

        mat(1U, 3U) = 4;
        mat(2U, 2U) = 5;
        mat(1U, 1U) = 3;

        print(mat);

        Mat_Int aux1(NFIL,NCOL);
        aux1 = mat;
        aux1(2U, 2U) = 0; // elimina un elemento y una fila
        print(aux1);

        const Mat_Int aux2(mat);
        print(aux2);
        //aux2(1U, 1U) = 3; // Error de compilación

        // Fuera de Rango
        mat(NFIL, NCOL) = 5;
    } catch ( OutOfRange ) {
        cerr << "excepción fuera de rango" << endl;
    } catch ( ... ) {
        cerr << "excepción inesperada" << endl;
    }
}
//-- fin -----

```

Capítulo 20

Programación Genérica Avanzada

Las plantillas (“templates” en inglés) son útiles a la hora de realizar programación genérica. Esto es, definir clases y funciones de forma genérica que se instancien a tipos particulares en función de la utilización de éstas.

Los parámetros de las plantillas podrán ser tanto tipos como valores constantes. Veamos un ejemplo de definiciones de funciones genéricas:

20.1. Parámetros Genéricos por Defecto

```
template <typename T, typename C = deque<T> >
class Stack {
    // ...
};

template <typename T, unsigned SIZE=512>
class Array {
    // ...
};
```

20.2. Tipos dentro de Clases Genéricas

```
template <typename Tipo>
class Dato {
    typedef array<Tipo,64> Array;
    // ...
};

template <typename Tipo>
class XXX {
    typedef typename Dato<Tipo>::Array Array;
    // ...
};
```

20.3. Métodos de Clase Genéricos

cuando usar template en la utilización de un método genérico.

```
template <typename Tipo>
class Dato {
    template <typename T>
    T metodo(const Tipo& x) {
```

```

        return T(x);
    }
    // ...
};
int main()
{
    Dato<int> d;
    double z = d.template metodo<double>(3);
}

template <unsigned IDX,
          typename Tipo>
inline const
typename Dato<Tipo>::template Type<IDX>::type &
member(const Dato<Tipo>& m,
        const typename Dato<Tipo>::template Type<IDX>::type& a) {
    return m.template metodo<IDX>(a);
}

```

20.4. Amigos Genéricos

```

//--fichero: subrango.hpp -----
#include <iostream>
#include <stdexcept>
namespace umalcc {
    // Declaración adelantada de la clase Subrango
    template<typename Tipo, Tipo menor, Tipo mayor>
    class Subrango;

    // Prototipo del operador de salida para la clase Subrango
    template<typename Tipo, Tipo menor, Tipo mayor>
    std::ostream& operator <<(std::ostream&, const Subrango<Tipo, menor, mayor>&);
    template<typename Tipo, Tipo menor, Tipo mayor>
    std::istream& operator >>(std::istream&, Subrango<Tipo, menor, mayor>&);

    template<typename Tipo, Tipo menor, Tipo mayor>
    class Subrango {
    public:
        Subrango();
        Subrango(const T_Base& i);
        operator T_Base();
        friend std::ostream& operator << <>(std::ostream&, const Subrango<Tipo, menor, mayor>&);
        friend std::istream& operator >> <>(std::istream&, Subrango<Tipo, menor, mayor>&);
    private:
        typedef Tipo T_Base;
        // -- Atributos --
        T_Base valor;
    };

    template<typename Tipo, Tipo menor, Tipo mayor>
    std::ostream& operator <<(std::ostream& sal, const Subrango<Tipo, menor, mayor>& i)
    {
        return sal << i.valor;
    }

    template<typename Tipo, Tipo menor, Tipo mayor>
    std::istream& operator >>(std::istream& in, Subrango<Tipo, menor, mayor>& i)
    {
        Tipo val;

```



```

        in >> val;
        i = val;
        return in;
    }

    template<typename Tipo, Tipo menor, Tipo mayor>
    Subrango<Tipo,menor,mayor>::Subrango()
        : valor(menor)
    {}

    template<typename Tipo, Tipo menor, Tipo mayor>
    Subrango<Tipo,menor,mayor>::Subrango(const T_Base& i)
        : valor(i)
    {
        if ((valor < menor) || (valor > mayor)) {
            throw std::range_error("Subrango::Subrango range error");
        }
    }

    template<typename Tipo, Tipo menor, Tipo mayor>
    Subrango<Tipo,menor,mayor>::operator Tipo()
    {
        return valor;
    }
}
//--fin: subrango.hpp -----

```

20.5. Restricciones en Programación Genérica

En programación genérica puede ser conveniente especificar un conjunto de restricciones sobre los tipos genéricos que pueden ser instanciados en diferentes clases o funciones:

```

//-- constraint.hpp -----
// Tomado de: B. Stroustrup Technical FAQ
#ifndef _constraint_hpp_
#define _constraint_hpp_
#ifndef _UNUSED_
#ifdef __GNUC__
#define _UNUSED_ __attribute__((unused))
#else
#define _UNUSED_
#endif
#endif
namespace constraint {
    template<typename T, typename B> struct Derived_From {
        static void constraints(T* p) { B* pb = p; }
        Derived_From() { void(*p)(T*) _UNUSED_ = constraints; }
    };
    template<typename T1, typename T2> struct Can_Copy {
        static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
        Can_Copy() { void(*p)(T1,T2) _UNUSED_ = constraints; }
    };
    template<typename T1, typename T2 = T1> struct Can_Compare {
        static void constraints(T1 a, T2 b) { a==b; a!=b; a<b; }
        Can_Compare() { void(*p)(T1,T2) _UNUSED_ = constraints; }
    };
    template<typename T1, typename T2, typename T3 = T1> struct Can_Multiply {
        static void constraints(T1 a, T2 b, T3 c) { c = a*b; }
    };
}

```

```

    Can_Multiply() { void(*p)(T1,T2,T3) _UNUSED_ = constraints; }
};
template<typename T1> struct Is_Vector {
    static void constraints(T1 a) { a.size(); a[0]; }
    Is_Vector() { void(*p)(T1) _UNUSED_ = constraints; }
};
} //namespace constraint
#ifdef _UNUSED_
#undef _UNUSED_
#endif
//-- ~constraint.hpp -----

```

Un ejemplo de su utilización:

```

#include <iostream>
using namespace std;

struct B { };
struct D : B { };
struct DD : D { };
struct X { };

// Los elementos deben ser derivados de 'ClaseBase'
template<typename T>
class Container : Derived_from<T,ClaseBase> {
    // ...
};

template<typename T>
class Nuevo : Is_Vector<T> {
    // ...
};

int main()
{
    Derived_from<D,B>();
    Derived_from<DD,B>();
    Derived_from<X,B>();
    Derived_from<int,B>();
    Derived_from<X,int>();

    Can_compare<int,double>();
    Can_compare<X,B>();
    Can_multiply<int,double>();
    Can_multiply<int,double,double>();
    Can_multiply<B,X>();

    Can_copy<D*,B*>();
    Can_copy<D,B*>();
    Can_copy<int,B*>();
}

```

20.6. Especializaciones

Veamos otro ejemplo de una definición genérica (tomado de GCC 3.3) con especialización:

```

//--fichero: stl_hash_fun.hpp -----
namespace __gnu_cxx {
    using std::size_t;
}

```

```

// definición de la clase generica vacía
template <typename _Key>
struct hash { };

inline size_t __stl_hash_string(const char* __s)
{
    unsigned long __h = 0;
    for ( ; *__s; ++__s) {
        __h = 5*__h + *__s;
    }
    return size_t(__h);
}
// especializaciones
template<> struct hash<char*> {
    size_t operator () (const char* __s) const
    { return __stl_hash_string(__s); }
};
template<> struct hash<const char*> {
    size_t operator () (const char* __s) const
    { return __stl_hash_string(__s); }
};
template<> struct hash<char> {
    size_t operator () (char __x) const
    { return __x; }
};
template<> struct hash<int> {
    size_t operator () (int __x) const
    { return __x; }
};
template <> struct hash<string> {
    size_t operator () (const string& str) const
    { return __stl_hash_string(str.c_str()); }
};

} // namespace __gnu_cxx
// -fin: stl_hash_fun.hpp -----

```

Una posible ventaja de hacer esta especialización sobre un tipo que sobre una función, es que la función estará definida para cualquier tipo, y será en tiempo de ejecución donde rompa si se utiliza sobre un tipo no válido. De esta forma, es en tiempo de compilación cuando falla. Otra ventaja es que puede ser pasada como parámetro a un “template”.

20.7. Meta-programación

```

template <unsigned X, unsigned Y>
struct GcdX {
    static const unsigned val = (Y==0)?X:GcdX<Y, X%Y>::val ;
};
template <unsigned X>
struct GcdX<X,0> {
    static const unsigned val = X;
};
template <unsigned X, unsigned Y>
struct Gcd {
    static const unsigned val = (X>=Y)?GcdX<X,Y>::val:GcdX<Y,X>::val ;
};

```

20.8. SFINAE

```
//-----
#ifndef _sfinae_hpp_
#define _sfinae_hpp_
namespace umalcc {
    //-----
    // SFINAE ("Substitution Failure Is Not An Error" principle)
    //-----
    // has member type: type
    template<typename Tipo>
    class has_type_type {
    public:
        enum { value = (sizeof(test<Tipo>(0)) == sizeof(Yes)) };
    private:
        struct Yes { char a[1]; };
        struct No { char a[2]; };
        template <typename TT> class Helper{};
        template<typename TT> static Yes test(Helper<typename TT::type> const*);
        template<typename TT> static No test(...);
    };
    //-----
    // has member data: int value
    template <typename Tipo>
    class has_data_value {
    public:
        enum { value = (sizeof(test<Tipo>(0)) == sizeof(Yes)) };
    private:
        struct Yes { char a[1]; };
        struct No { char a[2]; };
        template <typename TT, TT t> class Helper{};
        template<typename TT> static Yes test(Helper<int TT::*, &TT::value> const*);
        template<typename TT> static No test(...);
    };
    //-----
    // has static member data: static int value
    template <typename Tipo>
    class has_static_value {
    public:
        enum { value = (sizeof(test<Tipo>(0)) == sizeof(Yes)) };
    private:
        struct Yes { char a[1]; };
        struct No { char a[2]; };
        template <typename TT, TT t> class Helper{};
        template<typename TT> static Yes test(Helper<int *, &TT::value> const*);
        template<typename TT> static No test(...);
    };
    //-----
    // has member function: RT foo()
    template <typename Tipo>
    class has_function_foo {
    public:
        enum { value = (sizeof(test<Tipo>(0)) == sizeof(Yes)) };
    private:
        struct Yes { char a[1]; };
        struct No { char a[2]; };
        template <typename TT, TT t> class Helper{};
        template<typename TT> static Yes test(Helper<void (TT::*)(), &TT::foo> const*);
        template<typename TT> static No test(...);
    };
}
```

```
};  
//-----  
} //namespace umalcc  
#endif
```


Capítulo 21

Buffer y Flujos de Entrada y Salida

Como se vio anteriormente, para realizar operaciones de entrada y salida es necesario incluir las definiciones correspondientes en nuestro programa. para ello realizaremos la siguiente acción al comienzo de nuestro programa:

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

Para declarar simplemente los nombres de tipos:

```
#include <iosfwd>
```

En este capítulo entraremos con más profundidad en las operaciones a realizar en la entrada y salida, cómo controlar dichos flujos y comprobar su estado, así como la entrada y salida a ficheros.

21.1. Operaciones de Salida

Además de las operaciones ya vistas, es posible aplicar las siguientes *operaciones a los flujos de salida*:

```
cout.put('#'); // imprime un carácter
cout.write(char str[], int tam); // imprime 'tam' caracteres del 'string'

int n = cout.pcount(); // devuelve el número de caracteres escritos
cout.flush(); // fuerza la salida del flujo
cout.sync(); // sincroniza el buffer

{
    ostream::sentry centinela(cout); // creación del centinela en <<
    if (!centinela) {
        setstate(failbit);
        return(cout);
    }
    ...
}
```

21.2. Operaciones de Entrada

Además de las vistas anteriormente (no las de salida), se pueden aplicar las siguientes *operaciones a los flujos de entrada*:

Para leer un simple carácter:

```
int n = cin.get(); // lee un carácter de entrada o EOF
cin.get(char& c);  // lee un carácter de entrada
int n = cin.peek(); // devuelve el prox carácter (sin leerlo)
```

Para leer cadenas de caracteres (una línea cada vez):

```
cin.get(char str[], int tam);
// lee un string como máximo hasta 'tam-1' o hasta fin de línea
// añade el terminador '\0' al final de str
// el carácter de fin de línea NO se elimina del flujo

cin.get(char str[], int tam, char delim);
// lee un string como máximo hasta 'tam-1' o hasta que se lea 'delim'
// añade el terminador '\0' al final de str
// el carácter delimitador NO se elimina del flujo

cin.getline(char str[], int tam);
// lee un string como máximo hasta 'tam-1' o hasta fin de línea
// añade el terminador '\0' al final de str
// el carácter de fin de línea SI se elimina del flujo

cin.getline(char str[], int tam, char delim);
// lee un string como máximo hasta 'tam-1' o hasta que se lea 'delim'
// añade el terminador '\0' al final de str
// el carácter delimitador SI se elimina del flujo

cin.read(char str[], int tam);
// lee 'tam' caracteres y los almacena en el string
// NO pone el terminador '\0' al final del string
```

También es posible leer un string mediante las siguientes operaciones.

```
cin >> str; // salta espacios y lee hasta espacios
getline(cin, str); // lee hasta fin de línea ('\n')
getline(cin, str, delimitador); // lee hasta encontrar el delim. especificado
```

Nótese que realizar una operación `getline` después de una operación con `>>` puede tener complicaciones, ya que `>>` dejara los separadores en el buffer, que serán leídos por `getline`. Para evitar este problema (leerá una cadena que sea distinta de la vacía):

```
void leer_linea_no_vacia(istream& ent, string& linea)
{
    ent >> ws; // salta los espacios en blanco y rc's
    getline(ent, linea); // leerá la primera línea no vacía
}
```

También es posible leer un dato y eliminar el resto (separadores) del buffer de entrada.

```
template <typename Tipo>
void leer(istream& ent, Tipo& dato)
{
    ent >> dato; // lee el dato
    ent.ignore(numeric_limits<streamsize>::max(), '\n'); // elimina los caracteres del buffer
}
```

Las siguientes operaciones también están disponibles.

```
int n = cin.gcount(); // devuelve el número de caracteres leídos
cin.ignore([int n][, int delim]); // ignora cars hasta 'n' o 'delim'
cin.unget(); // devuelve al flujo el último carácter leído
```



```

cin.putback(char ch); // devuelve al flujo el carácter 'ch'

{
    istream::sentry centinela(cin); // creación del centinela en >>
    if (!centinela) {
        setstate(failbit);
        return(cin);
    }
    ...
}

std::ios::sync_with_stdio(false);

int n = cin.readsome(char* p, int n); // lee caracteres disponibles (hasta n)
n = cin.rdbuf()->in_avail(); // número de caracteres disponibles en el buffer
cin.ignore(cin.rdbuf()->in_avail()); // elimina los caracteres del buffer

cin.sync(); // sincroniza el buffer

```

Para hacer que lance una *excepción* cuando ocurra algún error:

```

ios::iostate old_state = cin.exceptions();
cin.exceptions(ios::badbit | ios::failbit | ios::eofbit);
cout.exceptions(ios::badbit | ios::failbit | ios::eofbit);

```

la excepción lanzada es del tipo `ios::failure`.

21.3. Buffer

Se puede acceder al buffer de un stream, tanto para obtener su valor como para modificarlo. El buffer es el encargado de almacenar los caracteres (antes de leerlos o escribirlos) y definen además su comportamiento. Es donde reside la inteligencia de los streams. Es posible la redirección de los streams gracias a los buffers (véase [21.4](#)).

```

typedef basic_streambuf<ostream::char_type, ostream::traits_type> streambuf_type;
streambuf* pbuff1 = stream.rdbuf(); // obtener el ptr al buffer
streambuf* pbuff2 = stream.rdbuf(pbuff1); // actualizar y obtener el ptr anterior

```

Ejemplo donde se desactiva el buffer si el nivel de verbosidad no supera un umbral (véase mas ejemplos en Sección [21.4](#)):

```

//-----
//#include <iostream>
#include <iosfwd>
using namespace std;
//-----
namespace umalcc {
    //-----
    inline unsigned verbose_level(unsigned l = -1U) {
        static unsigned verb_level = 1;
        unsigned res = verb_level;
        if (l != -1U) {
            verb_level = l;
        }
        return res;
    }
    //-----
    namespace _verbose_ {
        //-----

```

```

template <class cT, class traits = std::char_traits<cT> >
class basic_nullbuf: public std::basic_streambuf<cT, traits> {
    typename traits::int_type overflow(typename traits::int_type c) {
        return traits::not_eof(c); // indicate success
    }
};

template <class cT, class traits = std::char_traits<cT> >
class basic_ostream: public std::basic_ostream<cT, traits> {
public:
    basic_ostream():
        std::basic_ios<cT, traits>(&m_sbuf),
        std::basic_ostream<cT, traits>(&m_sbuf)
    {
        init(&m_sbuf);
    }
private:
    basic_nullbuf<cT, traits> m_sbuf;
};

typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;
//-----
inline std::ostream& verb(std::ostream& os, unsigned l) {
    static ostream out;
    if (l>verbose_level()) {
        return out;
    } else {
        return os;
    }
}

class VerboseMngr {
    friend std::ostream& operator<< (std::ostream& out,
                                   const class VerboseMngr& vm);

    friend VerboseMngr umalcc::verbose(unsigned l);
    unsigned lv;
    VerboseMngr(unsigned l) : lv(l) {}
};

inline std::ostream& operator<< (std::ostream& out,
                                const VerboseMngr& vm) {
    return verb(out, vm.lv);
}

} //namespace _verbose_
//-----
inline std::ostream& verb(unsigned l) {
    return umalcc::_verbose_::verb(std::cerr, l);
}

inline umalcc::_verbose_::VerboseMngr verbose(unsigned l) {
    return umalcc::_verbose_::VerboseMngr(l);
}
//-----
}

//-----
int main()
{
    verbose_level(2);

    cerr << verbose(0) << "Prueba-1 0" << endl;
    cerr << verbose(1) << "Prueba-1 1" << endl;
    cerr << verbose(2) << "Prueba-1 2" << endl;
    cerr << verbose(3) << "Prueba-1 3" << endl;
}

```

```

cerr << verbose(4) << "Prueba-1 4" << endl;

verb(0) << "Prueba-2 0" << endl;
verb(1) << "Prueba-2 1" << endl;
verb(2) << "Prueba-2 2" << endl;
verb(3) << "Prueba-2 3" << endl;
verb(4) << "Prueba-2 4" << endl;

cerr << verbose(0) << "Prueba-3 0" << endl;
cerr << verbose(1) << "Prueba-3 1" << endl;
cerr << verbose(2) << "Prueba-3 2" << endl;
cerr << verbose(3) << "Prueba-3 3" << endl;
cerr << verbose(4) << "Prueba-3 4" << endl;
}
//-----

```

21.4. Redirección Transparente de la Salida Estándar a un String

Para que de forma transparente la información que se envía a la salida estándar se capture en un string:

```

/-- ostrcap.hpp -----
#ifndef _ostrcap_hpp_
#define _ostrcap_hpp_
#include <iostream>
#include <string>
#include <sstream>
#include <stdexcept>

namespace ostrcap {

    class Capture_Error : public std::runtime_error {
    public:
        Capture_Error() : std::runtime_error ("Capture_Error") {}
        Capture_Error(const std::string& arg) : std::runtime_error (arg) {}
    };

    class Release_Error : public std::runtime_error {
    public:
        Release_Error() : std::runtime_error ("Release_Error") {}
        Release_Error(const std::string& arg) : std::runtime_error (arg) {}
    };

    class OStreamCapture {
    public:
        OStreamCapture()
            : str_salida(), output_stream(), output_streambuf_ptr() {}
        void capture(std::ostream& os) {
            if (output_streambuf_ptr != NULL) {
                throw Capture_Error();
            }
            str_salida.str("");
            output_stream = &os;
            output_streambuf_ptr = output_stream->rdbuf(str_salida.rdbuf());
        }
        std::string release() {
            if (output_streambuf_ptr == NULL) {
                throw Release_Error();
            }
        }
    };
}

```

```

    }
    output_stream->rdbuf(output_streambuf_ptr);
    output_streambuf_ptr = NULL;
    return str_salida.str();
}

private:
    std::ostringstream str_salida;
    std::ostream* output_stream;
    std::streambuf* output_streambuf_ptr;
}; // class OStreamCapture
} // namespace ostrcap
#endif
//-- main.cpp -----
#include <iostream>
#include <string>

#include "ostrcap.hpp"

using namespace std;
using namespace ostrcap;

void salida(int x)
{
    cout << "Información " << x << endl;
}

int main()
{
    OStreamCapture ostrcap;
    ostrcap.capture(cout);
    salida(34);
    cout << "La salida es: " << ostrcap.release();

    ostrcap.capture(cout);
    salida(36);
    cout << "La salida es: " << ostrcap.release();
}
//-- fin -----

```

21.5. Ficheros

Las operaciones vistas anteriormente sobre los flujos de entrada y salida estándares se pueden aplicar también sobre *ficheros* de entrada y de salida, simplemente cambiando las variables `cin` y `cout` por las correspondientes variables que especifican cada fichero. Para ello es necesario incluir la siguiente definición:

```
#include <fstream>
```

21.6. Ficheros de Entrada

Veamos como declaramos dichas variables para *ficheros de entrada*:

```
ifstream fichero_entrada; // declara un fichero de entrada
```

```
fichero_entrada.open(char nombre[] [, int modo]); // abre el fichero
```

donde los modos posibles, combinados con OR de bits (|):

```

ios::in      // entrada
ios::out     // salida
ios::app     // posicionarse al final antes de cada escritura
ios::binary  // fichero binario

ios::ate     // abrir y posicionarse al final
ios::trunc   // truncar el fichero a vacío

```

Otra posibilidad es declarar la variable y abrir el fichero simultáneamente:

```
ifstream fichero_entrada(char nombre[] [, int modo]);
```

Para cerrar el fichero

```
fichero_entrada.close();
```

Nota: si se va a abrir un fichero utilizando la misma variable que ya ha sido utilizada (tras el `close()`), se deberá utilizar el método `clear()` para limpiar los flags de estado.

Para posicionar el puntero de lectura del fichero (puede coincidir o no con el puntero de escritura):

```

n = cin.tellg();    // devuelve la posición de lectura
cin.seekg(pos);     // pone la posición de lectura a 'pos'
cin.seekg(offset, ios::beg|ios::cur|ios::end); // pone la posición de lectura relativa

```

21.7. Ficheros de Salida

```

ofstream fichero_salida; // declara un fichero de salida

fichero_salida.open(char nombre[] [, int modo]); // abre el fichero

```

donde los modos posibles, combinados con OR de bits (`|`) son los anteriormente especificados. Otra posibilidad es declarar la variable y abrir el fichero simultáneamente:

```
ofstream fichero_salida(char nombre[] [, int modo]);
```

Para cerrar el fichero

```
fichero_salida.close();
```

Nota: si se va a abrir un fichero utilizando la misma variable que ya ha sido utilizada (tras el `close()`), se deberá utilizar el método `clear()` para limpiar los flags de estado.

Para posicionar el puntero de escritura del fichero (puede coincidir o no con el puntero de lectura):

```

n = cout.tellp();           // devuelve la posición de escritura
cout.seekp(pos);            // pone la posición de escritura a 'pos'
cout.seekp(offset, ios::beg|ios::cur|ios::end); // pone la posición de escritura relativa

```

21.8. Ejemplo de Ficheros

Ejemplo de un programa que lee caracteres de un fichero y los escribe a otro hasta encontrar el fin de fichero:

```

#include <iostream>
#include <fstream>

void
copiar_fichero_2(const string& salida, const string& entrada, bool& ok)
{
    ok = false;
    ifstream f_ent(entrada.c_str());
    if (f_ent) {
        ofstream f_sal(salida.c_str());
        if (f_sal) {
            f_sal << f_ent.rdbuf();
            ok = f_ent.eof() && f_sal.good();
            f_sal.close(); // no es necesario
        }
        f_ent.close(); // no es necesario
    }
}

inline ostream& operator<< (ostream& out, istream& in)
{
    return out << in.rdbuf();
}

```

21.9. Ficheros de Entrada y Salida

Para *ficheros de entrada y salida* utilizamos el tipo:

```
fstream fich_ent_sal(char nombre[], ios::in | ios::out);
```

21.10. Flujo de Entrada desde una Cadena

Es posible también redirigir la *entrada desde un string de memoria*. Veamos como se realiza:

```

#include <sstream>

// const char* entrada = "123 452";
// const char entrada[10] = "123 452";
const string entrada = "123 452";

istringstream str_entrada(entrada);

str_entrada >> valor1; // valor1 = 123;
str_entrada >> valor2; // valor2 = 452;
str_entrada >> valor3; // EOF

str_entrada.str("nueva entrada")

```

21.11. Flujo de Salida a una Cadena

Así mismo, también es posible redirigir la *salida a un string en memoria*:

```

#include <sstream>

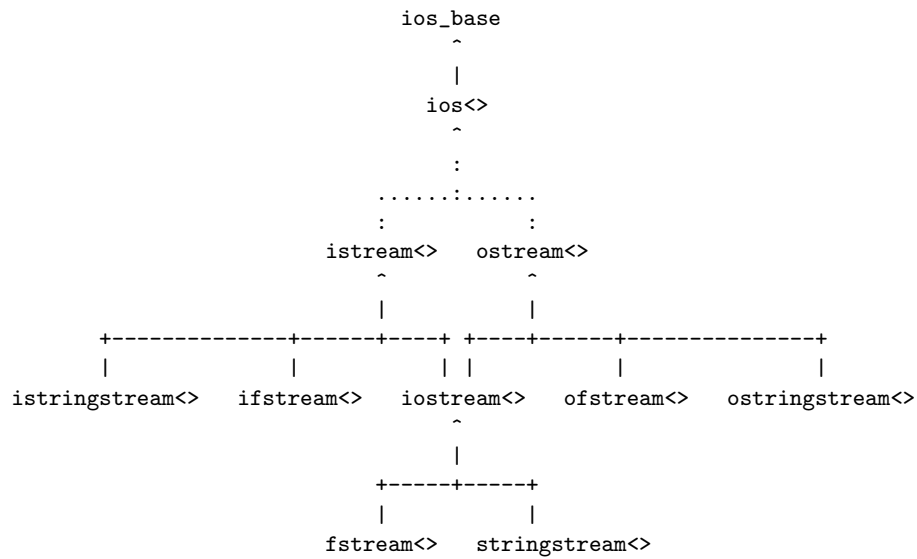
ostringstream str_salida;

str_salida << 123; // salida tendra "123"

```

```
n = str_salida.pcount(); // devuelve la longitud de la cadena
string salida = str_salida.str();
str_salida.str("cadena inicial");
```

21.12. Jerarquía de Clases de Flujo Estándar



Capítulo 22

Técnicas de Programación Usuales en C++

En este capítulo veremos algunas técnicas de programación usualmente utilizadas en C++. Las técnicas mostradas en este capítulo han sido tomadas de diferentes recursos públicos obtenidos de Internet.

22.1. Adquisición de Recursos es Inicialización (RAII)

(DRAFT)

Esta técnica consiste en que las adquisición de recursos se realiza en la inicialización de un objeto que lo gestiona (manager), de tal forma que la destrucción del manager lleva asociado la liberación del recurso. Nótese que este mecanismo permite diseñar mecanismos robustos de gestión de recursos en situaciones complejas, tales como entornos con excepciones.

Se puede diseñar de forma simple, o mediante managers que poseen recursos y permiten la transferencia de propiedad a otros managers.

22.1.1. `auto_ptr` (`unique_ptr`)

Para soportar la técnica “Adquisición de recursos es inicialización” con objeto de implementar clases que se comporten de forma segura ante las excepciones, la biblioteca estándar proporciona la clase `auto_ptr`.

Nota: `auto_ptr` no debe ser utilizado dentro de los contenedores estándares.

El estándar de C++98 define el tipo `auto_ptr<>` para gestionar la memoria dinámica. Sin embargo, el nuevo estándar de C++0x, elimina `auto_ptr<>` y define en su lugar `unique_ptr<>` como mecanismo adecuado para expresar el propietario de un recurso de memoria dinámica.

Esta clase se comporta igual que el tipo puntero (*“smart pointer”*) pero con la salvedad de que cuando se destruye la variable, la zona de memoria apuntada por ella se libera (`delete`) automáticamente, salvo que se indique lo contrario mediante el método `release`. De esta forma, si hay posibilidad de lanzar una excepción, las zonas de memoria dinámica reservada se deberán proteger de esta forma para que sean liberadas automáticamente en caso de que se eleve una excepción. En caso de que haya pasado la zona donde se pueden elevar excepciones, entonces se puede revocar dicha característica. Ejemplo:

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

class Dato {
public:
```

```

    ~Dato() throw() { cout << "Destruye dato " << val << endl; }
    Dato(unsigned v = 0): val(v) { cout << "Construye dato " << val << endl; }
    unsigned get() const { return val; }
private:
    unsigned val;
};

int main()
{
    auto_ptr<Dato> ptr1(new Dato(5));
    auto_ptr<Dato> ptr2;
    cout << "Valor1: " << ptr1->get() << endl;
    ptr2 = ptr1;
    cout << "Valor2: " << (*ptr2).get() << endl;
}

```

Otro ejemplo de utilización

```

#include <memory>

class X {
public:
    ~X();
    X();
    ...
private:
    int* ptr;
};
X::~X() throw() { delete ptr; }
X::X()
: ptr(0)
{
    auto_ptr<int> paux(new int);
    // utilización de paux. zona posible de excepciones
    // *paux    paux->    paux.get()    paux.reset(p)
    ptr = paux.release();
}

```

22.1.2. RAII Simple de Memoria Dinámica

Un gestor de recursos simple, específico para memoria dinámica, sin transferencia de propiedad, donde la adquisición del recurso se realiza durante la inicialización del objeto, y su liberación durante la destrucción del mismo.

```

#include <iostream>
using namespace std;

template <typename PtrBase>
class RaiiPtr {
public:
    ~RaiiPtr() throw() { try { delete ptr; } catch (...) {} }
    explicit RaiiPtr(PtrBase* p) : ptr(p) {}
    void swap(RaiiPtr& o) { std::swap(this->ptr, o.ptr); }
    PtrBase& operator*() const throw() { return *ptr; }
    PtrBase* operator->() const throw() { return ptr; }
    PtrBase* get() const throw() { return ptr; }
private:
    PtrBase* ptr;
    RaiiPtr& operator=(const RaiiPtr& o);
}

```

```

};

class Dato {
public:
    ~Dato() throw() { cout << "Destruye dato " << val << endl; }
    Dato(unsigned v = 0): val(v) { cout << "Construye dato " << val << endl; }
    unsigned get() const { return val; }
private:
    unsigned val;
};

int main()
{
    RaiiPtr<Dato> ptr1(new Dato(5));
    RaiiPtr<Dato> ptr2(new Dato(7));
    cout << "Valor1: " << ptr1->get() << endl;
    cout << "Valor2: " << ptr2->get() << endl;
    ptr1.swap(ptr2);
    cout << "Valor1: " << (*ptr1).get() << endl;
    cout << "Valor2: " << (*ptr2).get() << endl;
}

```

22.1.3. RAII Simple Genérico

```

#include <iostream>
using namespace std;

template <typename Releaser>
class Raii {
public:
    ~Raii() throw() {
        if (owner) { try { rlsr(); } catch (...) {} }
    }
    explicit Raii(const Releaser& r): owner(true), rlsr(r) {}
    Raii(const Raii& o): owner(o.owner), rlsr(o.rlsr) { o.release(); }
    Raii& operator=(const Raii& o) {
        if (this != &o) { Raii(o).swap(*this); } return *this;
    }
    void release() const throw() { owner = false; }
    void swap(Raii& o) {
        std::swap(this->owner, o.owner);
        std::swap(this->rlsr, o.rlsr);
    }
private:
    mutable bool owner;
    const Releaser& rlsr;
};

template <typename PtrBase>
class PtrReleaser {
public:
    ~PtrReleaser() throw() {}
    PtrReleaser(const PtrBase* p) : ptr(p) {}
    void swap(PtrReleaser& o) { std::swap(this->ptr, o.ptr); }
    void operator()() const { delete ptr; }
private:
    const PtrBase* ptr;
};

```

```
int main()
{
    cout << "-----" << endl;
    Dato* ptr1 = new Dato(5);
    Raii<PtrReleaser<Dato> > mgr1(ptr1);
    cout << "Valor1: " << ptr1->get() << endl;
}
```

22.1.4. RAII Genérico

```
template <typename Fun, typename Arg>
class SGuard {
public:
    ~SGuard() throw() {
        if (!_released) { try{ _fun(_arg); } catch(...) {} }
    }
    SGuard(const Fun& f, const Arg& a)
        : _released(false), _fun(f), _arg(a) {}
    void release() const throw() { _released = true; }
private:
    bool _released;
    const Fun& _fun;
    const Arg& _arg;
};

template <typename Fun, typename Arg>
inline SGuard<Fun,Arg> mk_guard(const Fun& f, const Arg& a) {
    return SGuard<Fun,Arg>(f, a);
}

int main()
{
    // adquisición de recurso
    Recurso rec_1 = acquire_resource();
    // liberación automática de recurso si surge una excepción
    SGuard guarda_rec_1 = mk_guard(funcion, arg);
    // ...
    // Acciones que pueden elevar una excepción
    // ...
    guarda_rec_1.release();
    // Anulación de liberación automática del recurso
}
```

22.1.5. RAII Genérico

```
//-----
#ifndef _guard_hpp_
#define _guard_hpp_
/*
 * Generic<Programming>:
 * Change the Way You Write Exception-Safe Code ? Forever
 * Andrei Alexandrescu and Petru Marginean
 * C/C++ User Journal
 * http://www.cuj.com/documents/s=8000/cujcexp1812alexandr/
 */
namespace raii {
    //-----
    class SGuardImplBase {
    public:
```

```

    void release() const throw() { _released = true; }
protected:
    mutable bool _released;
    ~SGuardImplBase() throw() {} // nonvirtual
    SGuardImplBase() throw() : _released(false) {}
    SGuardImplBase(const SGuardImplBase& o) throw()
        : _released(o._released) { o.release(); }
private:
    SGuardImplBase& operator=(const SGuardImplBase& o); //prohibida
}; // class SGuardImplBase
//-----
template <typename Fun>
class SGuard0Arg : public SGuardImplBase {
public:
    ~SGuard0Arg() throw()
        { if (!_released) { try{ _fun(); } catch(...) {} }}
    SGuard0Arg(const Fun& f)
        : _fun(f) {}
private:
    const Fun& _fun;
}; // class SGuard1Arg
//-----
template <typename Fun, typename Arg>
class SGuard1Arg : public SGuardImplBase {
public:
    ~SGuard1Arg() throw()
        { if (!_released) { try{ _fun(_arg); } catch(...) {} }}
    SGuard1Arg(const Fun& f, const Arg& a)
        : _fun(f), _arg(a) {}
private:
    const Fun& _fun;
    const Arg& _arg;
}; // class SGuard1Arg
//-----
template <typename Fun, typename Arg1, typename Arg2>
class SGuard2Arg : public SGuardImplBase {
public:
    ~SGuard2Arg() throw()
        { if (!_released) { try{ _fun(_arg1,_arg2); } catch(...) {} }}
    SGuard2Arg(const Fun& f, const Arg1& a1, const Arg2& a2)
        : _fun(f), _arg1(a1), _arg2(a2) {}
private:
    const Fun& _fun;
    const Arg1& _arg1;
    const Arg2& _arg2;
}; // class SGuard1Arg
//-----
template <typename Fun>
inline SGuard0Arg<Fun>
mk_guard(const Fun& f)
{
    return SGuard0Arg<Fun>(f);
}
//-----
template <typename Fun, typename Arg>
inline SGuard1Arg<Fun,Arg>
mk_guard(const Fun& f, const Arg& a)
{
    return SGuard1Arg<Fun,Arg>(f, a);
}

```

```

    }
    //-----
    template <typename Fun, typename Arg1, typename Arg2>
    inline SGuard2Arg<Fun,Arg1,Arg2>
    mk_guard(const Fun& f, const Arg1& a1, const Arg2& a2)
    {
        return SGuard2Arg<Fun,Arg1,Arg2>(f, a1, a2);
    }
    //-----
    typedef const SGuardImplBase& SGuard;
    //-----
} // namespace raii
#endif
//-----
// _EJEMPLO_DE_USO_
//-----
using namespace raii;
int main()
{
    // adquisición de recurso
    Recurso rec_1 = acquire_resource();
    // liberación automática de recurso si surge una excepción
    SGuard guarda_rec_1 = mk_guard(funcion, arg);
    // ...
    // Acciones que pueden elevar una excepción
    // ...
    guarda_rec_1.release();
    // Anulación de liberación automática del recurso
}
//-----

```

Capítulo 23

Gestión Dinámica de Memoria

23.1. Gestión de Memoria Dinámica

Los operadores `new` y `delete` en sus diferentes modalidades solicitan y liberan memoria dinámica, así como crean y destruyen objetos alojados en ella.

El operador `new Tipo` se encarga de reservar memoria del tamaño necesario para contener un elemento del tipo especificado llamando a

```
void *operator new(std::size_t) throw (std::bad_alloc);
```

y crea un objeto (constructor por defecto) de dicho tipo. Devuelve un puntero al objeto creado. También existe la posibilidad de crear un objeto llamando a otro constructor: `new Tipo(args)`.

El operador `delete ptr` se encarga de destruir el objeto apuntado (llamando al destructor) y de liberar la memoria ocupada llamando a

```
void operator delete(void *) throw();
```

El operador `new Tipo[nelms]` reserva espacio en memoria para contener `nelms` elementos del tipo especificado llamando a

```
void *operator new[](std::size_t) throw (std::bad_alloc);
```

y crea dichos elementos llamando al constructor por defecto. Devuelve un puntero al primer objeto especificado.

El operador `delete [] ptr` se encarga de destruir los objetos apuntados (llamando al destructor) en el orden inverso en el que fueron creados y de liberar la memoria ocupada por ellos llamando a

```
void operator delete[](void *) throw();
```

Los operadores anteriores elevan la excepción `std::bad_alloc` en caso de agotamiento de la memoria. Si queremos que no se lance dicha excepción se pueden utilizar la siguiente modalidad, y en caso de agotamiento de memoria devolverán 0.

```
Tipo* ptr = new (nothrow) Tipo;  
Tipo* ptr = new (nothrow) Tipo(args);  
Tipo* ptr = new (nothrow) Tipo[nelms];
```

Se implementan en base a los siguientes operadores:

```
void *operator new(std::size_t, const std::nothrow_t&) throw();  
void *operator new[](std::size_t, const std::nothrow_t&) throw();  
void operator delete(void *, const std::nothrow_t&) throw();  
void operator delete[](void *, const std::nothrow_t&) throw();
```

El operador `new (ptr) Tipo` (`new` con emplazamiento) no reserva memoria, simplemente crea el objeto del tipo especificado (utilizando el constructor por defecto u otra clase de constructor con la sintaxis `new (ptr) Tipo(args)` en la zona de memoria especificada por la llamada con argumento `ptr` a

```
inline void *operator new(std::size_t, void *place) throw() { return place; }
```

Los operadores encargados de reservar y liberar memoria dinámica y pueden redefinirse para las clases definidas por el usuario [también pueden redefinirse en el ámbito global, aunque esto último no es aconsejable]

```
#include <new>

void *operator new(std::size_t) throw (std::bad_alloc);
void operator delete(void *) throw();
void *operator new[](std::size_t) throw (std::bad_alloc);
void operator delete[](void *) throw();

void *operator new(std::size_t, const std::nothrow_t&) throw();
void operator delete(void *, const std::nothrow_t&) throw();
void *operator new[](std::size_t, const std::nothrow_t&) throw();
void operator delete[](void *, const std::nothrow_t&) throw();

inline void *operator new(std::size_t, void *place) throw() { return place; }
inline void *operator new[](std::size_t, void *place) throw() { return place; }
```

y los comportamientos por defecto podrían ser los siguientes para los operadores `new` y `delete`:

Los operadores `new` y `delete` encargados de alojar y desalojar zonas de memoria pueden ser redefinidos por el programador, tanto en el ámbito global como para clases específicas. Una definición estándar puede ser como se indica a continuación: (fuente: M. Cline, C++ FAQ-Lite, <http://www.parashift.com/c++-faq-lite/>)

```
#include <new>

extern "C" void *malloc(std::size_t);
extern "C" void free(void *);
extern std::new_handler __new_handler;

/*
 * *****
 * Tipo* ptr = new Tipo(lista_val);
 * *****
 *
 * ptr = static_cast<Tipo*>(::operator new(sizeof(Tipo))); // alojar mem
 * try {
 *     new (ptr) Tipo(lista_val);          // llamada al constructor
 * } catch (...) {
 *     ::operator delete(ptr);
 *     throw;
 * }
 */
void*
operator new (std::size_t sz) throw(std::bad_alloc)
{
    void* p;

    /* malloc(0) es impredecible; lo evitamos. */
    if (sz == 0) {
        sz = 1;
    }
}
```



```

    }
    p = static_cast<void*>(malloc(sz));
    while (p == 0) {
        std::new_handler handler = __new_handler;
        if (handler == 0) {
#ifdef __EXCEPTIONS
            throw std::bad_alloc();
#else
            std::abort();
#endif
        }
        handler();
        p = static_cast<void*>(malloc(sz));
    }
    return p;
}
/*
 * *****
 * delete ptr;
 * *****
 *
 * if (ptr != 0) {
 *     ptr->~Tipo();          // llamada al destructor
 *     ::operator delete(ptr); // liberar zona de memoria
 * }
 *
 * *****
 */
void
operator delete (void* ptr) throw()
{
    if (ptr) {
        free(ptr);
    }
}
/*
 * *****
 * Tipo* ptr = new Tipo [NELMS];
 * *****
 *
 * // alojar zona de memoria
 * char* tmp = (char*)::operator new[] (WORDSIZE+NELMS*sizeof(Tipo));
 * ptr = (Tipo*)(tmp+WORDSIZE);
 * *(size_t*)tmp = NELMS;
 * size_t i;
 * try {
 *     for (i = 0; i < NELMS; ++i) {
 *         new (ptr+i) Tipo();          // llamada al constructor
 *     }
 * } catch (...) {
 *     while (i-- != 0) {
 *         (p+i)->~Tipo();
 *     }
 *     ::operator delete((char*)ptr - WORDSIZE);
 *     throw;
 * }
 *
 * *****
 */

```

```

void*
operator new[] (std::size_t sz) throw(std::bad_alloc)
{
    return ::operator new(sz);
}
/*
 * *****
 * delete [] ptr;
 * *****
 *
 *     if (ptr != 0) {
 *         size_t n = *(size_t*)((char*)ptr - WORDSIZE);
 *         while (n-- != 0) {
 *             (ptr+n)->~Tipo();          // llamada al destructor
 *         }
 *         ::operator delete[]((char*)ptr - WORDSIZE);
 *     }
 *
 * *****
 */
void
operator delete[] (void* ptr) throw()
{
    ::operator delete (ptr);
}

```

23.2. Gestión de Memoria Dinámica sin Inicializar

La biblioteca estándar también proporciona una clase para poder trabajar con zonas de memoria sin inicializar, de forma que sea útil para la creación de clases contenedoras, etc. Así, proporciona métodos para reservar zonas de memoria sin inicializar, construir objetos en ella, destruir objetos de ella y liberar dicha zona:

```

#include <memory>

template <typename Tipo>
class allocator {
public:
    Tipo* allocate(std::size_t nelms) throw(std::bad_alloc);
    void construct(Tipo* ptr, const Tipo& val);
    void destroy(Tipo* ptr);
    void deallocate(Tipo* ptr, std::size_t nelms) throw();
};

uninitialized_fill(For_It begin, For_It end, const Tipo& val);
uninitialized_fill_n(For_It begin, std::size_t nelms, const Tipo& val);
uninitialized_copy(In_It begin_org, In_It end_org, For_It begin_dest);

```

23.3. RAII: auto_ptr

Para soportar la técnica “Adquisición de recursos es inicialización” con objeto de implementar clases que se comporten de forma segura ante las excepciones, la biblioteca estándar proporciona la clase `auto_ptr`.

Esta clase se comporta igual que el tipo puntero (*“smart pointer”*) pero con la salvedad de que cuando se destruye la variable, la zona de memoria apuntada por ella se libera (`delete`) automáticamente, salvo que se indique lo contrario mediante el método `release`. De esta forma, si hay posibilidad de lanzar una excepción, las zonas de memoria dinámica reservada se deberán proteger

de esta forma para que sean liberadas automáticamente en caso de que se eleve una excepción. En caso de que haya pasado la zona donde se pueden elevar excepciones, entonces se puede revocar dicha característica. Ejemplo:

```
#include <memory>

class X {
public:
    ~X();
    X();
    ...
private:
    int* ptr;
};
X::~X() throw{} { delete ptr; }
X::X()
    : ptr(0)
{
    auto_ptr<int> paux(new int);
    // utilización de paux. zona posible de excepciones
    // *paux    paux->    paux.get()    paux.reset(p)
    ptr = paux.release();
}
```

23.4. Comprobación de Gestión de Memoria Dinámica

El siguiente código C++ se puede compilar junto con cualquier programa C++ de forma externa *no intrusiva*, y automáticamente realiza en tiempo de ejecución una cuenta del número de nodos alojados y liberados, de tal forma que al finalizar el programa escribe unas estadísticas comprobando si coinciden o no. Nótese que no es necesario modificar el programa principal para que se pueda comprobar automáticamente la cuenta de nodos alojados y liberados, es suficiente con la invocación al compilador.

```
g++ -ansi -Wall -Werror -o main main.cpp newdelcnt.cpp

//-----
//- newdelcnt.cpp -----
//-----
#include <iostream>
#include <cstdlib>
#include <new>
//-----
namespace {
    static unsigned inc_new(unsigned n = 0) throw()
    {
        static unsigned cnt = 0;
        return (cnt += n);
    }
    //-----
    static unsigned inc_delete(unsigned n = 0) throw()
    {
        static unsigned cnt = 0;
        return (cnt += n);
    }
    //-----
    static void check_new_delete() throw()
    {
        if (inc_new() != inc_delete()) {
```

```

        std::cerr << "ERROR, el número de NEW [" << inc_new()
        << "]" es diferente del número de DELETE ["
        << inc_delete() << "]" << std::endl;
#ifdef __WIN32__
        std::system("pause");
#endif
        std::exit(1234);
    } else {
        std::cerr << "OK, el número de NEW [" << inc_new()
        << "]" es igual al número de DELETE ["
        << inc_delete() << "]" << std::endl;
    }
}
//-----
struct ExecAtEnd { ~ExecAtEnd() throw() { check_new_delete(); } };
static ExecAtEnd x;
//-----
} //end namespace
//-----
void* operator new (std::size_t sz) throw (std::bad_alloc)
{
    if (sz == 0) {
        sz = 1;    // malloc(0) es impredecible; lo evitamos.
    }
    void* p = static_cast<void*>(std::malloc(sz));
    if (p == 0) {
        std::cerr << "Error: New: Memory failure" << std::endl;
        throw std::bad_alloc();
    }
    inc_new(1);
    return p;
}
//-----
void operator delete (void* ptr) throw()
{
    if (ptr) {
        inc_delete(1);
        std::free(ptr);
    }
}
//-----

```

Capítulo 24

Biblioteca Estándar de C++. STL

La biblioteca estándar de C++ se define dentro del espacio de nombres `std` y proporciona:

- Soporte para las características del lenguaje, tales como manejo de memoria y la información de tipo en tiempo de ejecución.
- Información sobre aspectos del lenguaje definidos por la implementación, tal como el valor del mayor `double`.
- Funciones que no se pueden implementar de forma óptima en el propio lenguaje para cada sistema, tal como `sqrt` y `memmove`.
- Facilidades no primitivas sobre las cuales un programador se puede basar para portabilidad, tal como `vector`, `list`, `map`, ordenaciones, entrada/salida, etc.
- La base común para otras bibliotecas.

Los *contenedores* de la biblioteca estándar proporcionan un método general para almacenar y acceder a elementos homogéneos, proporcionando cada uno de ellos diferentes características que los hacen adecuados a diferentes necesidades.

Paso de Parámetros de Contenedores

Los contenedores de la biblioteca estándar se pueden pasar como parámetros a subprogramas como cualquier otro tipo compuesto, y por lo tanto se aplican los mecanismos de paso de parámetros para tipos compuestos explicados en la sección 6.1. Es decir, los parámetros de entrada se pasarán por referencia constante, mientras que los parámetros de salida y entrada/salida se pasarán por referencia.

Así mismo, como norma general, salvo excepciones, no es adecuado que las funciones retornen valores de tipos de los contenedores, debido a la sobrecarga que generalmente conlleva dicha operación para el caso de los tipos compuestos. En estos casos suele ser más adecuado que el valor se devuelva como un parámetro por referencia.

24.1. Características Comunes

24.1.1. Ficheros

```
#include <vector>
#include <list>
#include <stack>
#include <queue>
#include <deque>
#include <set>
```

```
#include <map>
#include <iterator>

#include <utility>
#include <functional>
#include <algorithm>
```

24.1.2. Contenedores

```
vector<tipo_base> vec; // vector de tamaño variable
list<tipo_base> ls; // lista doblemente enlazada
deque<tipo_base> dqu; // cola doblemente terminada

stack<tipo_base> st; // pila
queue<tipo_base> qu; // cola
priority_queue<tipo_base> pqu; // cola ordenada

map<clave,valor> mp; // contenedor asociativo de pares de valores
multimap<clave,valor> mmp; // cont asoc con repetición de clave
set<tipo_base> cnj; // conjunto
multiset<tipo_base> mcnj; // conjunto con valores multiples
```

24.1.3. Tipos Definidos

```
t::value_type // tipo del elemento
t::allocator_type // tipo del manejador de memoria
t::size_type // tipo de subíndices, cuenta de elementos, ...
t::difference_type // tipo diferencia entre iteradores
t::iterator // iterador (se comporta como value_type*)
t::const_iterator // iterador const (como const value_type*)
t::reverse_iterator // iterador inverso (como value_type*)
t::const_reverse_iterator // iterador inverso (como const value_type*)
t::reference // value_type&
t::const_reference // const value_type&
t::key_type // tipo de la clave (sólo contenedores asociativos)
t::mapped_type // tipo del valor mapeado (sólo cont asoc)
t::key_compare // tipo de criterio de comparación (sólo cont asoc)
```

24.1.4. Iteradores

```
c.begin() // apunta al primer elemento
c.end() // apunta al (último+1) elemento
c.rbegin() // apunta al primer elemento (orden inverso)
c.rend() // apunta al (último+1) elemento (orden inverso)

bi = back_inserter(contenedor) // insertador al final
fi = front_inserter(contenedor) // insertador al principio
in = inserter(contenedor, iterador) // insertador en posición

it = ri.base(); // convierte de ri a it. it = ri + 1

for (list<tipo_base>::const_iterator i = nombre.begin();
     i != nombre.end(); ++i) {
    cout << *i;
}

for (list<tipo_base>::reverse_iterator i = nombre.rbegin();
     i != nombre.rend(); ++i) {
    *i = valor;
}
```

```

}

contenedor<tipo_base>::iterator i = ri.base(); // ri + 1

```

24.1.5. Acceso

```

c.front() // primer elemento
c.back()  // último elemento
v[i]      // elemento de la posición 'i' (vector, deque y map)
v.at[i]    // elem de la posición 'i' => out_of_range (vector y deque)

```

24.1.6. Operaciones de Pila y Cola

```

c.push_back(e)    // añade al final
c.pop_back()      // elimina el último elemento
c.push_front(e)   // añade al principio (list y deque)
c.pop_front()     // elimina el primer elemento (list y deque)

```

24.1.7. Operaciones de Lista

```

c.insert(p, x)     // añade x antes de p
c.insert(p, n, x)  // añade n copias de x antes de p
c.insert(p, f, l)  // añade elementos [f:l) antes de p
c.erase(p)         // elimina elemento en p
c.erase(f, l)      // elimina elementos [f:l)
c.clear()           // elimina todos los elementos

// sólo para list
l.reverse()        // invierte los elementos
l.remove(v)        // elimina elementos iguales a v
l.remove_if(pred1) // elimina elementos que cumplen pred1
l.splice(pos, lst) // mueve (sin copia) los elementos de lst a pos
l.splice(pos, lst, p) // mueve (sin copia) elemento posición p de lst a pos
l.splice(pos, lst, f, l) // mueve (sin copia) elementos [f:l) de lst a pos
l.sort()           // ordena
l.sort(cmp2)       // ordena según bool cmp2(o1, o2)
l.merge(lst)       // mezcla ambas listas (ordenadas) en l
l.merge(lst, cmp2) // mezcla ambas listas (ordenadas) en l
l.unique()         // elimina duplicados adyacentes
l.unique(pred2)    // elimina duplicados adyacentes que cumplen pred2

```

24.1.8. Operaciones

```

c.size()          // número de elementos
c.empty()         // (c.size() == 0)
c.max_size()      // tamaño del mayor posible contenedor
c.capacity()      // espacio alojado para el vector (sólo vector)
c.reserve(n)      // reservar espacio para expansión (sólo vector)
c.resize(n)       // cambiar el tam del cont (sólo vector, list y deque)
c.swap(y)         // intercambiar
==               // igualdad
!=               // desigualdad
<               // menor

```

24.1.9. Constructores

```

cont()           // contenedor vacío
cont(n)          // n elementos con valor por defecto (no cont asoc)
cont(n, x)       // n copias de x (no cont asoc)

```

```
cont(f, l) // elementos iniciales copiados de [f:l)
cont(x)    // inicialización igual a x
```

24.1.10. Asignación

```
x = y;           // copia los elementos de y a x
c.assign(n, x)    // asigna n copias de x (no cont asoc)
c.assign(f, l)    // asigna de [f:l)
```

24.1.11. Operaciones Asociativas

```
m[k]           // acceder al elemento con clave k (para clave unica)
m.find(k)       // encontrar el elemento con clave k
m.lower_bound(k) // encontrar el primer elemento con clave k
m.upper_bound(k) // encontrar el primer elem con clave mayor que k
m.equal_range(k) // encontrar el lower_bound(k) y upper_bound(k)
m.key_comp()    // copia la clave
m.value_comp()  // copia el valor
```

24.1.12. Resumen

	[]	op.list	op.front	op.back	iter
vector	const	O(n)+		const+	Random
list		const	const	const	Bidir
deque	const	O(n)	const	const	Random
stack				const	
queue			const	const	
prque			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bidir
mmap		O(log(n))+			Bidir
set		O(log(n))+			Bidir
mset		O(log(n))+			Bidir
string	const	O(n)+	O(n)+	const+	Random
array	const				Random
valarray	const				Random
bitset	const				

24.1.13. Operaciones sobre Iteradores

	Salida	Entrada	Avance	Bidir	Random
Leer		=*p	=*p	=*p	=*p
Acceso		->	->	->	-> []
Escribir	*p=		*p=	*p=	*p=
Iteración	++	++	++	++ --	++ -- + - += -=
Comparac		== !=	== !=	== !=	== != < > >= <=

24.2. Contenedores

Los contenedores proporcionan un método estándar para almacenar y acceder a elementos, proporcionando diferentes características.

24.3. Vector

Es una secuencia optimizada para el acceso aleatorio a los elementos. Proporciona, así mismo, iteradores aleatorios.

```
#include <vector>
```

- Construcción:

```
typedef std::vector<int> vect_int;

vect_int v1; // vector de enteros de tamaño inicial vacío
vect_int v2(100); // vector de 100 elementos con valor inicial por defecto
vect_int v3(50, val_inicial); // vector de 50 elementos con el valor especificado
vect_int v4(it_ini, it_fin); // vector de elementos copia de [it_ini:it_fin[
vect_int v5(v4); // vector copia de v4

typedef std::vector<int> Fila;
typedef std::vector<Fila> Matriz;

Matriz m(100, Fila(50)); // Matriz de 100 filas X 50 columnas
```

- Asignación (después de la asignación, el tamaño del vector asignado se adapta al número de elementos asignados):

```
//(se destruye el valor anterior de v1)
v1 = v2; // asignación de los elementos de v2 a v1
v1.assign(it_ini, it_fin); // asignación de los elementos [it_ini:it_fin[
v1.assign(50, val_inicial); // asignación de 50 elementos con val_inicial
```

- Acceso a elementos:

```
cout << v1.front(); // acceso al primer elemento (debe existir)
v1.front() = 1; // acceso al primer elemento (debe existir)

cout << v1.back(); // acceso al último elemento (debe existir)
v1.back() = 1; // acceso al último elemento (debe existir)
```

- Acceso aleatorio a elementos (sólo vector y deque):

```
cout << v1[i]; // acceso al elemento i-ésimo (debe existir)
v1[i] = 3; // acceso al elemento i-ésimo (debe existir)

cout << v1.at(i); // acceso al elemento i-ésimo (lanza out_of_range si no existe)
v1.at(i) = 3; // acceso al elemento i-ésimo (lanza out_of_range si no existe)
```

- Operaciones de Pila

```
v1.push_back(val); // añade val al final de v1 (crece)
v1.pop_back(); // elimina el último elemento (decrece) (no devuelve nada)
```

- Operaciones de Lista

```
it = v1.insert(it_pos, val); // inserta val en posición. dev it al elem (crece)
v1.insert(it_pos, n, val); // inserta n copias de val en posición (crece)
v1.insert(it_pos, it_i, it_f); // inserta [it_i:it_f[ en pos (crece)

it = v1.erase(it_pos); // elimina elem en pos. dev it al sig (decrece)
it = v1.erase(it_i, it_f); // elim elems en [it_i:it_f[. dev it al sig (decrece)

v1.clear(); // elimina todos los elementos. (decrece) (no devuelve memoria)
```

- Número de elementos:

```
n = v1.size();          // número de elementos de v1
bool b = v1.empty();    // (v1.size() == 0)
```

- Tamaño del contenedor:

```
n = v1.max_size();      // tamaño del mayor vector posible

// sólo vector deque list
v1.resize(nuevo_tam);   // redimensiona el vector al nuevo_tam con valor por defecto
v1.resize(nuevo_tam, valor); // redimensiona el vector. utiliza valor.

// sólo vector
v1.reserve(n);          // reservar n elementos (prealojados) sin inicializar valores
n = v1.capacity();       // número de elementos reservados
assert(&v[0] + n == &v[n]); // Se garantiza que la memoria es contigua
// aunque puede ser realojada por necesidades de nuevo espacio
```

- Otras operaciones:

```
v1 == v2 , v1 != v2 , v1 < v2 , v1 <= v2 , v1 > v2 , v1 >= v2
v1.swap(v2);
swap(v1 , v2);
```

```
template <typename Tipo>
void vaciar_vector(std::vector<Tipo>& vect)
{
    std::vector<Tipo> aux;
    vect.swap(aux);
    // se devuelve la memoria anterior de vect
}
```

```
template <typename Tipo>
void reajustar_vector(std::vector<Tipo>& vect)
{
    std::vector<Tipo> aux(vect);
    vect.swap(aux);
    // se devuelve la memoria anterior de vect
}
```

24.4. List

Es una secuencia optimizada para la inserción y eliminación de elementos. Proporciona, así mismo, iteradores bidireccionales.

```
#include <list>
```

- Construcción:

```
typedef std::list<int> list_int;

list_int l1; // lista de enteros de tamaño inicial vacío
list_int l2(100); // lista de 100 elementos con valor inicial por defecto
list_int l3(50, val_inicial); // lista de 50 elementos con el valor especificado
list_int l4(it_ini, it_fin); // lista de elementos copia de [it_ini:it_fin[
list_int l5(14); // lista copia de 14
```

- Asignación (después de la asignación, el tamaño de la lista asignada se adapta al número de elementos asignados):

```

//se destruye el valor anterior de l1
l1 = l2; // asignación de los elementos de l2 a l1
l1.assign(it_ini, it_fin); // asignación de los elementos [it_ini:it_fin[
l1.assign(50, val_inicial); // asignación de 50 elementos con val_inicial

```

- Acceso a elementos:

```

cout << l1.front(); // acceso al primer elemento (debe existir)
l1.front() = 1;      // acceso al primer elemento (debe existir)

cout << l1.back();   // acceso al último elemento (debe existir)
l1.back() = 1;       // acceso al último elemento (debe existir)

```

- Operaciones de Pila

```

l1.push_back(val); // añade val al final de l1 (crece)
l1.pop_back();     // elimina el último elemento (decrece) (no devuelve nada)

```

- Operaciones de Cola (sólo list y deque)

```

l1.push_front(val); // añade val al principio de l1 (crece)
l1.pop_front();     // elimina el primer elemento (decrece) (no devuelve nada)

```

- Operaciones de Lista

```

it = l1.insert(it_pos, val); // inserta val en posición. dev it al elem (crece)
l1.insert(it_pos, n, val);   // inserta n copias de val en posición (crece)
l1.insert(it_pos, it_i, it_f); // inserta [it_i:it_f[ en pos (crece)

it = l1.erase(it_pos);      // elimina elem en pos. dev it al sig (decrece)
it = l1.erase(it_i, it_f);  // elim elems en [it_i:it_f[. dev it al sig (decrece)

l1.clear(); // elimina todos los elementos. (decrece)

```

- Número de elementos:

```

n = l1.size(); // número de elementos de l1
bool b = l1.empty(); // (l1.size() == 0)

```

- Tamaño del contenedor:

```

n = l1.max_size(); // tamaño de la mayor lista posible

// sólo vector deque list
l1.resize(nuevo_tam); // redimensiona la lista al nuevo_tam con valor por defecto
l1.resize(nuevo_tam, valor); // redimensiona la lista. utiliza valor.

```

- Otras operaciones:

```

l1 == l2 , l1 != l2 , l1 < l2 , l1 <= l2 , l1 > l2 , l1 >= l2
l1.swap(l2);
swap(l1 , l2);

```

- Otras operaciones propias del contenedor lista:

```

l1.reverse(); // invierte los elementos de l1

l1.remove(val); // elimina de l1 todos los elementos igual a val
l1.remove_if(pred1); // elimina de l1 todos los elementos que cumplen pred1(elem)

l1.splice(it_pos, l2); // mueve (sin copia) los elementos de l2 a pos

```

```

l1.splice(it_pos, l2, it_el); // mueve (sin copia) (*it_el) de l2 a pos
l1.splice(it_pos, l2, it_i, it_f); // mueve (sin copia) [it_i:it_f[ de l2 a pos

l1.sort(); // ordena l1
l1.sort(cmp2); // ordena l1 según bool cmp2(e1, e2)

l1.merge(l2); // mezcla ambas listas (ordenadas) en l1 (mueve sin copia)
l1.merge(l2, cmp2); // mezcla ambas listas (ordenadas) en l1 según cmp2(e1, e2)

l1.unique(); // elimina elementos duplicados adyacentes
l1.unique(pred2); // elimina elementos adyacentes que cumplen pred2(e1, e2)

```

Tanto los predicados como las funciones de comparación pueden ser funciones (unarias o binarias) que reciben los parámetros (1 o 2) del tipo del elemento del contenedor y devuelven un `bool` resultado de la función, o un objeto de una clase que tenga el operador `()` definido. Ejemplo:

```

bool mayor_que_5(int elem)
{
    return (elem > 5);
}

int main()
{
    . . .
    l1.remove_if(mayor_que_5);
}

bool operator()(const Tipo& arg1, const Tipo& arg2) {}

```

o también de la siguiente forma:

```

class mayor_que : public unary_function<int, bool> {
public:
    mayor_que(int val) : _valor(val) {} // constructor

    bool operator() (int elem) const { return (elem > _valor); }
private:
    int _valor;
};

int main()
{
    std::list<int> l1;
    . . .
    l1.remove_if(mayor_que(5));
    l1.remove_if(mayor_que(3));
}

```

o también de la siguiente forma:

```

int main()
{
    std::list<int> l1;
    . . .
    l1.remove_if(bind2nd(greater<int>(), 5));
    l1.remove_if(bind2nd(greater<int>(), 3));
}

```

Veamos otro ejemplo:

```

bool
mayor1(const string& s1, const string& s2)
{
    return s1 > s2;
}

class mayor2 : public binary_function<string, string, bool> {
public:
    bool operator() (const string& s1, const string& s2) const { return s1 > s2; }
};

int
main()
{
    std::list<string> l1;
    . . .
    l1.sort();                // ordena de menor a mayor
    l1.sort(mayor1);          // ordena de mayor a menor (utiliza mayor1)
    l1.sort(mayor2());        // ordena de mayor a menor (utiliza mayor2())
    l1.sort(greater<string>()); // ordena de mayor a menor (utiliza greater<>())
}

```

24.5. Deque

Secuencia optimizada para que las operaciones de inserción y borrado en *ambos extremos* sean tan eficientes como en una lista, y el acceso aleatorio tan eficiente como un vector. Proporciona iteradores aleatorios.

```
#include <deque>
```

- Construcción:

```

typedef std::deque<int> deq_int;

deq_int d1; // deque de enteros de tamaño inicial vacío
deq_int d2(100); // deque de 100 elementos con valor inicial por defecto
deq_int d3(50, val_inicial); // deque de 50 elementos con el valor especificado
deq_int d4(it_ini, it_fin); // deque de elementos copia de [it_ini:it_fin[
deq_int d5(d4); // deque copia de d4

```

- Asignación (después de la asignación, el tamaño del deque asignado se adapta al número de elementos asignados):

```

//(se destruye el valor anterior de d1)
d1 = d2; // asignación de los elementos de d2 a d1
d1.assign(it_ini, it_fin); // asignación de los elementos [it_ini:it_fin[
d1.assign(50, val_inicial); // asignación de 50 elementos con val_inicial

```

- Acceso a elementos:

```

cout << d1.front(); // acceso al primer elemento (debe existir)
d1.front() = 1;     // acceso al primer elemento (debe existir)

cout << d1.back();  // acceso al último elemento (debe existir)
d1.back() = 1;      // acceso al último elemento (debe existir)

```

- Acceso aleatorio a elementos (sólo vector y deque):

```
cout << d1[i]; // acceso al elemento i-ésimo (debe existir)
d1[i] = 3;      // acceso al elemento i-ésimo (debe existir)

cout << d1.at(i); // acceso al elemento i-ésimo (lanza out_of_range si no existe)
d1.at(i) = 3;     // acceso al elemento i-ésimo (lanza out_of_range si no existe)
```

■ Operaciones de Pila

```
d1.push_back(val); // añade val al final de d1 (crece)
d1.pop_back();     // elimina el último elemento (decrece) (no devuelve nada)
```

■ Operaciones de Cola (sólo list y deque)

```
d1.push_front(val); // añade val al principio de d1 (crece)
d1.pop_front();     // elimina el primer elemento (decrece) (no devuelve nada)
```

■ Operaciones de Lista

```
it = d1.insert(it_pos, val); // inserta val en posición. dev it al elem (crece)
d1.insert(it_pos, n, val);   // inserta n copias de val en posición (crece)
d1.insert(it_pos, it_i, it_f); // inserta [it_i:it_f[ en pos (crece)

it = d1.erase(it_pos);      // elimina elem en pos. dev it al sig (decrece)
it = d1.erase(it_i, it_f);  // elim elems en [it_i:it_f[. dev it al sig (decrece)

d1.clear(); // elimina todos los elementos. (decrece)
```

■ Número de elementos:

```
n = d1.size(); // número de elementos de d1
bool b = d1.empty(); // (d1.size() == 0)
```

■ Tamaño del contenedor:

```
n = d1.max_size(); // tamaño del mayor deque posible

// sólo vector deque list
d1.resize(nuevo_tam); // redimensiona el deque al nuevo_tam con valor por defecto
d1.resize(nuevo_tam, valor); // redimensiona el deque. utiliza valor.
```

■ Otras operaciones:

```
d1 == d2 , d1 != d2 , d1 < d2 , d1 <= d2 , d1 > d2 , d1 >= d2
d1.swap(d2);
swap(d1 , d2);
```

24.6. Stack

Proporciona el “Tipo Abstracto de Datos Pila”, y se implementa sobre un deque. No proporciona iteradores.

```
#include <stack>

typedef std::stack<char> pila_c;

int
main()
{
    pila_c p1; // []
```

```

p1.push('a');           // [a]
p1.push('b');           // [a b]
p1.push('c');           // [a b c]

if (p1.size() != 3) {
    // imposible
}

if (p1.top() != 'c') {
    // imposible
}
p1.top() = 'C';         // [a b C]

if (p1.size() != 3) {
    // imposible
}

p1.pop();               // [a b]
p1.pop();               // [a]
p1.pop();               // []

if (! p1.empty()) {
    // imposible
}
}

```

24.7. Queue

Proporciona el “Tipo Abstracto de Datos Cola”, y se implementa sobre un deque. No proporciona iteradores.

```

#include <queue>

typedef std::queue<char> cola_c;

int
main()
{
    cola_c c1;           // []

    c1.push('a');         // [a]
    c1.push('b');         // [a b]
    c1.push('c');         // [a b c]

    if (c1.size() != 3) {
        // imposible
    }

    if (c1.front() != 'a') {
        // imposible
    }
    c1.front() = 'A';     // [A b c]

    if (c1.back() != 'c') {
        // imposible
    }
}

```

```

    if (c1.size() != 3) {
        // imposible
    }

    c1.pop();           // [b c]
    c1.pop();           // [c]
    c1.pop();           // []

    if (! c1.empty()) {
        // imposible
    }
}

```

24.8. Priority-Queue

Proporciona el “Tipo Abstracto de Datos Cola con Prioridad”, y se implementa sobre un `vector`. No proporciona iteradores.

```

#include <queue>
// #include <functional>

using namespace std;

// typedef std::priority_queue< int, std::vector<int>, std::less<int> > pcola_i;
typedef std::priority_queue<int> pcola_i; // ordenación <

int
main()
{
    pcola_i c1;           // []

    c1.push(5);           // [5]
    c1.push(3);           // [5 3]
    c1.push(7);           // [7 5 3]
    c1.push(4);           // [7 5 4 3]

    if (c1.size() != 4) {
        // imposible
    }

    if (c1.top() != 7) {
        // imposible
    }

    c1.pop();             // [5 4 3]
    c1.pop();             // [4 3]
    c1.pop();             // [3]
    c1.pop();             // []

    if (! c1.empty()) {
        // imposible
    }
}

```

24.9. Map

Secuencia de pares `<clave, valor>` optimizada para el acceso rápido basado en clave. Clave única. Proporciona iteradores bidireccionales.


```

#include <map>
//#include <functional>

//typedef std::map< std::string, int, std::less<std::string> > map_str_int;
typedef std::map<std::string, int> map_str_int;

map_str_int m1;

n = m1.size();
n = m1.max_size();
bool b = m1.empty();
m1.swap(m2);

int x = m1["pepe"]; // crea nueva entrada. devuelve 0.
m1["juan"] = 3;     // crea nueva entrada a 0 y le asigna 3.
int y = m1["juan"]; // devuelve 3
m1["pepe"] = 4;     // asigna nuevo valor

it = m1.find("pepe"); // encontrar el elemento con una determinada clave
int n = m1.count("pepe"); // cuenta el número de elementos con clave "pepe"
it = m1.lower_bound("pepe"); // primer elemento con clave "pepe"
it = m1.upper_bound("pepe"); // primer elemento con clave mayor que "pepe"
pair<it,it> pit = m1.equal_range("pepe"); // rango de elementos con clave "pepe"

clave = it->first;
valor = it->second;

pair<it, bool> p = m1.insert(make_pair(clave, val));
it = m1.insert(it_pos, make_pair(clave, val));
m1.insert(it_ini, it_fin);
m1.erase(it_pos);
int n = m1.erase(clave);
m1.erase(it_ini, it_fin);
m1.clear();

#include <utility>
pair<clave, valor> p = make_pair(f, s);
p.first // clave
p.second // valor

```

24.10. Multimap

Secuencia de pares <clave, valor> optimizada para el acceso rápido basado en clave. Permite claves duplicadas. Proporciona iteradores bidireccionales.

```

#include <map>
//#include <functional>

//typedef std::multimap< std::string, int, std::less<std::string> > mmap_str_int;
typedef std::multimap<std::string, int> mmap_str_int;

mmap_str_int m1;

n = m1.size();
n = m1.max_size();
bool b = m1.empty();
m1.swap(m2);

```

```

it = m1.find("pepe"); // encontrar el elemento con una determinada clave
int n = m1.count("pepe"); // cuenta el número de elementos con clave "pepe"
it = m1.lower_bound("pepe"); // primer elemento con clave "pepe"
it = m1.upper_bound("pepe"); // primer elemento con clave mayor que "pepe"
pair<it,it> pit = m1.equal_range("pepe"); // rango de elementos con clave "pepe"

it = m1.insert(make_pair(clave, val));
it = m1.insert(it_pos, make_pair(clave, val));
m1.insert(it_ini, it_fin);
m1.erase(it_pos);
int n = m1.erase(clave);
m1.erase(it_ini, it_fin);
m1.clear();

#include <utility>
std::pair<clave, valor> p = std::make_pair(f, s);
p.first // clave
p.second // valor

```

24.11. Set

Como un map donde los valores no importan, sólo aparecen las claves. Proporciona iteradores bidireccionales.

```

#include <set>
// #include <functional>

// typedef std::set< std::string, std::less<std::string> > set_str;
typedef std::set<std::string> set_str;

set_str s1;

n = s1.size();
n = s1.max_size();
bool b = s1.empty();
s1.swap(s2);

it = s1.find("pepe"); // encontrar el elemento con una determinada clave
int n = s1.count("pepe"); // cuenta el número de elementos con clave "pepe"
it = s1.lower_bound("pepe"); // primer elemento con clave "pepe"
it = s1.upper_bound("pepe"); // primer elemento con clave mayor que "pepe"
pair<it,it> pit = s1.equal_range("pepe"); // rango de elementos con clave "pepe"

pair<it, bool> p = s1.insert(elem);
it = s1.insert(it_pos, elem);
s1.insert(it_ini, it_fin);
s1.erase(it_pos);
int n = s1.erase(clave);
s1.erase(it_ini, it_fin);
s1.clear();

#include <utility>
std::pair<tipo1, tipo2> p = std::make_pair(f, s);
p.first
p.second

```

24.12. Multiset

Como `set`, pero permite elementos duplicados. Proporciona iteradores bidireccionales.

```
#include <set>
//#include <functional>

//typedef std::multiset< std::string, std::less<std::string> > mset_str;
typedef std::multiset<std::string> mset_str;

mset_str s1;

n = s1.size();
n = s1.max_size();
bool b = s1.empty();
s1.swap(s2);

it = s1.find("pepe"); // encontrar el elemento con una determinada clave
int n = s1.count("pepe"); // cuenta el número de elementos con clave "pepe"
it = s1.lower_bound("pepe"); // primer elemento con clave "pepe"
it = s1.upper_bound("pepe"); // primer elemento con clave mayor que "pepe"
pair<it,it> pit = s1.equal_range("pepe"); // rango de elementos con clave "pepe"

it = s1.insert(elem);
it = s1.insert(it_pos, elem);
s1.insert(it_ini, it_fin);
s1.erase(it_pos);
int n = s1.erase(clave);
s1.erase(it_ini, it_fin);
s1.clear();

#include <utility>
std::pair<tipo1, tipo2> p = std::make_pair(f, s);
p.first
p.second
```

24.13. Bitset

Un `bitset<N>` es un array de N bits.

```
#include <bitset>

typedef std::bitset<16> mask;

mask m1;           // 16 bits todos a 0
mask m2 = 0xaa;    // 0000000010101010
mask m3 = "110100"; // 0000000000110100

m3[2] == 1;

m3 &= m2; // m3 = m3 & m2; // and
m3 |= m2; // m3 = m3 | m2; // or
m3 ^= m2; // m3 = m3 ^ m2; // xor

mask m4 = "0100000000110100";
m4 <= 2; // 0000000011010000
m4 >= 5; // 0000000000000110

m3.set(); // 1111111111111111
```

```

m4.set(5); // 0000000000100110
m4.set(2, 0); // 0000000000100010

m2.reset(); // 0000000000000000
m4.reset(5); // 0000000000000010
m4.flip(); // 111111111111101
m4.flip(3); // 111111111110101

m1 = ~(m4 << 3);

unsigned long val = m1.to_ulong();
std::string str = m1.to_string();

int n = m1.size(); // número de bits
int n = m1.count(); // número de bits a 1
bool b = m1.any(); // (m1.count() > 0)
bool b = m1.none(); // (m1.count() == 0)
bool b = m1.test(3); // (m1[3] == 1)

std::cout << m1 << std::endl;

```

24.14. Iteradores

Proporcionan el nexo de unión entre los contenedores y los algoritmos, proporcionando una visión abstracta de los datos de forma que un determinado algoritmo sea independiente de los detalles concernientes a las estructuras de datos.

Los iteradores proporcionan la visión de los contenedores como secuencias de objetos.

Los iteradores proporcionan, entre otras, las siguientes operaciones:

- Obtener el objeto asociado al iterador

```
*it          it->campo          it->miembro()
```

- Moverse al siguiente elemento de la secuencia

```
++it (-- + - += -=)    advance(it, n)
```

- Comparación entre iteradores

```
== != (< <= > >=)    n = distance(first, last)
```

```
#include <iterator>
```

24.15. Directos

```

typedef std::vector<int> vect_int;
typedef vect_int::iterator vi_it;
typedef vect_int::const_iterator vi_cit;

const int MAX = 10;
vect_int vi(MAX);

vi_it inicio_secuencia = vi.begin();
vi_it fin_secuencia = vi.end();

int n = 0;
for (vi_it i = vi.begin(); i != vi.end(); ++i) {
    *i = n;
}

```

```

    ++n;
}
// vi = { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 }

vi_cit inicio_secuencia_const = vi.begin();
vi_cit fin_secuencia_const = vi.end();

for (vi_cit i = vi.begin(); i != vi.end(); ++i) {
    // *i = n; // error. i es un const_iterator
    cout << *i << " ";
}
// 0 1 2 3 4 5 6 7 8 9

typedef std::map<std::string, unsigned> MapStrUInt;
for (MapStrUInt::const_iterator it = mapa.begin(); it != mapa.end(); ++it) {
    std::cout << "Clave: " << it->first << " Valor: " << it->second << std::endl;
}

```

24.16. Inversos

```

typedef std::vector<int> vect_int;
typedef vect_int::iterator vi_it;
typedef vect_int::reverse_iterator vi_rit;
typedef vect_int::const_reverse_iterator vi_crit;

const int MAX = 10;
vect_int vi(MAX);

vi_rit inicio_secuencia_inversa = vi.rbegin();
vi_rit fin_secuencia_inversa = vi.rend();

int n = 0;
for (vi_rit i = vi.rbegin(); i != vi.rend(); ++i) {
    *i = n;
    ++n;
}
// vi = { 9 , 8 , 7 , 6 , 5 , 4 , 3 , 2 , 1 , 0 }

vi_crit inicio_secuencia_inversa_const = vi.rbegin();
vi_crit fin_secuencia_inversa_const = vi.rend();

for (vi_crit i = vi.rbegin(); i != vi.rend(); ++i) {
    // *i = n; // error. i es un const_iterator
    cout << *i << " ";
}
// 0 1 2 3 4 5 6 7 8 9

vi_rit rit = ...;

template <typename IT, typename RIT>
inline void
reverse_to_direct(IT& it, const RIT& rit)
{
    it = rit.base();
    --it;
}

```

Un puntero a un elemento de un agregado es también un iterador para dicho agregado, y puede

ser utilizado como tal el los algoritmos que lo requieran.

24.17. Inserters

“*Inserters*” producen un iterador que al ser utilizado para asignar objetos, alojan espacio para él en el contenedor, aumentando así su tamaño.

```
typedef std::back_insert_iterator<contenedor> biic;
typedef std::front_insert_iterator<contenedor> fiic;
typedef std::insert_iterator<contenedor> iic;

bit = std::back_inserter(contenedor); // añade elementos al final del contenedor
fiit = std::front_inserter(contenedor); // añade elementos al inicio del contenedor
iit = std::inserter(contenedor, it_pos); // añade elementos en posición
```

24.18. Stream Iterators

```
ostream_iterator<int> os(cout, "delimitador_de_salida");
ostream_iterator<int> os(cout);
*os = 7;
++os;
*os = 79;

istream_iterator<int> is(cin);
istream_iterator<int> fin_entrada;
int i1 = *is;
++is;
int i2 = *is;
copy(is, fin_entrada, back_inserter(v));
```

Ejemplo que imprime el contenido de un vector de dos dimensiones:

```
//-----
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
namespace std{
    template <typename Tipo>
    inline ostream& operator<<(ostream& out, const vector<Tipo>& v) {
        copy(v.begin(), v.end(), ostream_iterator<Tipo>(out, " "));
        return out;
    }
}
using namespace std;
int main()
{
    vector<int> libros(5, 33);
    vector< vector<int> > bib(2, libros);
    cout << bib << ';<< endl;

    bib[1].erase(bib[1].begin());
    cout << bib << ';<< endl;
}
//-----
```

Ejemplo que lee números de un string y los pasa a un vector:

```

#include <iostream>
#include <string>
#include <vector>
#include <iterator>
#include <sstream>

using namespace std;

void str2vect(const string& entrada, vector<unsigned>& salida)
{
    salida.clear();
    istringstream str_entrada(entrada);
    istream_iterator<unsigned> input_string_stream_it(str_entrada);
    istream_iterator<unsigned> fin_input_string_stream_it;
    copy(input_string_stream_it, fin_input_string_stream_it,
        back_inserter(salida));
}

```

Ejemplo que copia el contenido de un fichero a otro:

```

#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>

using namespace std;

void
copiar_fichero(const string& salida, const string& entrada, bool& ok)
{
    typedef istream_iterator<char> Iterador_Entrada;
    typedef ostream_iterator<char> Iterador_Salida;

    ok = false;

    ifstream f_ent(entrada.c_str());
    if (f_ent) {
        ofstream f_sal(salida.c_str());
        if (f_sal) {
            entrada.unsetf(ios::skipws);

            Iterador_Entrada it_ent(f_ent);
            Iterador_Entrada fit_ent;

            Iterador_Salida it_sal(f_sal);

            copy(it_ent, fit_ent, f_sal);

            f_sal.close(); // no es necesario
        }
        f_ent.close(); // no es necesario
    }
}

```

Ejemplo que carga el contenido de un fichero a un contenedor, y salva dicho contenedor a un fichero:

```

#include <iostream>
#include <fstream>
#include <string>

```

```

#include <vector>
#include <iterator>
#include <algorithm>

using namespace std;

//-----
struct FileNotFound : public FileIOError {
    explicit FileNotFound(const char* w) : FileIOError(w) {}
    explicit FileNotFound(const string& w) : FileIOError(w) {}
};
struct FileFormatError : public FileIOError {
    explicit FileFormatError(const char* w) : FileIOError(w) {}
    explicit FileFormatError(const string& w) : FileIOError(w) {}
};
struct FileIOError : public std::runtime_error {
    explicit FileIOError(const char* w) : std::runtime_error(w) {}
    explicit FileIOError(const string& w) : std::runtime_error(w) {}
};
//-----
template <typename Contenedor>
void cargar(const string& filename, Contenedor& c)
    throw(FileNotFound, FileFormatError)
{
    typedef istream_iterator<typename Contenedor::value_type> InputIt;
    ifstream in(filename.c_str());
    if (in.fail()) { throw FileNotFound(filename); }
    InputIt entrada(in);
    InputIt fin_entrada;
    //c.assign(entrada, fin_entrada);
    //c.clear();
    std::copy(entrada, fin_entrada, back_inserter(c));
    if (!in.eof()) { throw FileFormatError(filename); }
}
template <typename Contenedor, typename Predicate1>
void cargar(const string& filename, Contenedor& c, const Predicate1& pred1)
    throw(FileNotFound, FileFormatError)
{
    typedef istream_iterator<typename Contenedor::value_type> InputIt;
    ifstream in(filename.c_str());
    if (in.fail()) { throw FileNotFound(filename); }
    InputIt entrada(in);
    InputIt fin_entrada;
    //c.clear();
    std::remove_copy_if(entrada, fin_entrada, back_inserter(c), not1(pred1));
    if (!in.eof()) { throw FileFormatError(filename); }
}
template <typename Procedure1>
void aplicar(const string& filename, const Procedure1& proc1)
    throw(FileNotFound, FileFormatError)
{
    typedef istream_iterator<typename Procedure1::argument_type> InputIt;
    ifstream in(filename.c_str());
    if (in.fail()) { throw FileNotFound(filename); }
    InputIt entrada(in);
    InputIt fin_entrada;
    std::for_each(entrada, fin_entrada, proc1);
    if (!in.eof()) { throw FileFormatError(filename); }
}

```



```

%-----
template <typename Tipo>
struct AreaEq : public std::unary_function<Tipo, bool> {
    string val;
    explicit AreaEq(const string& a) : val(a) {}
    bool operator()(const Tipo& v) const {
        return val == v.get_area();
    }
};

template <typename TipoB, typename TipoE>
struct Apply : public std::unary_function<TipoE, void> {
    typedef void (TipoB::*PtrFun)(const TipoE&);
    TipoB& obj;
    PtrFun ptr_fun;
    explicit Apply(TipoB& o, PtrFun f) : obj(o), ptr_fun(f) {}
    void operator()(const TipoE& e) const {
        (obj.*ptr_fun)(e);
    }
};

template <typename TipoB, typename TipoE, typename Predicate1>
struct ApplyIf : public std::unary_function<TipoE, void> {
    typedef void (TipoB::*PtrFun)(const TipoE&);
    TipoB& obj;
    PtrFun ptr_fun;
    Predicate1 pred1;
    explicit ApplyIf(TipoB& o, PtrFun f, const Predicate1& p) : obj(o), ptr_fun(f), pred1(p) {}
    void operator()(const TipoE& e) const {
        if (pred1(e)) {
            (obj.*ptr_fun)(e);
        }
    }
};

//-----
template <typename Contenedor>
void salvar(const string& filename, const Contenedor& c)
{
    throw(FileNotFound, FileFormatError)
}

template <typename Contenedor, typename Predicate1>
void salvar(const string& filename, const Contenedor& c, const Predicate1& pred1)
{
    throw(FileNotFound, FileFormatError)
}

template <typename Contenedor, typename Predicate1>
void salvar(const string& filename, const Contenedor& c, const Predicate1& pred1)
{
    typedef ostream_iterator<typename Contenedor::value_type> OutputIt;
    ofstream out(filename.c_str());
    if (out.fail()) { throw FileNotFound(filename); }
    OutputIt salida(out, " ");
    copy(c.begin(), c.end(), salida);
    if (out.fail()) { throw FileFormatError(filename); }
}

template <typename Contenedor, typename Predicate1>
void salvar(const string& filename, const Contenedor& c, const Predicate1& pred1)
{
    typedef ostream_iterator<typename Contenedor::value_type> OutputIt;
    ofstream out(filename.c_str());
    if (out.fail()) { throw FileNotFound(filename); }
    OutputIt salida(out, " ");
    remove_copy_if(c.begin(), c.end(), salida, not1(pred1));
    if (out.fail()) { throw FileFormatError(filename); }
}

template <typename Tipo>
struct MayorQue : public std::unary_function<Tipo, bool> {
    Tipo base;
    explicit MayorQue(const Tipo& x) : base(x) {}
}

```

```

    bool operator()(const Tipo& x) const {
        return x > base;
    }
}
//-----
int main()
{
    vector<int> cnt;
    cargar(cnt, "entrada.txt");
    salvar(cnt, "salida.txt", MayorQue(5));

    cargar(filename, profesores, AreaEq<Profesor>(cfg.area));
    aplicar(filename, ApplyIf<Asignaturas,AsgNAT,AreaEq<AsgNAT> >(asignaturas,&Asignaturas::set_sin_pri

}

```

24.19. Operaciones sobre Iteradores

Operaciones sobre Iteradores

	Salida	Entrada	Avance	Bidir	Random
Leer		=*p	=*p	=*p	=*p
Acceso		->	->	->	-> []
Escribir	*p=		*p=	*p=	*p=
Iteración	++	++	++	++ --	++ -- + - += -=
Comparac		== !=	== !=	== !=	== != < > >= <=

24.20. Objetos Función y Predicados

```

#include <functional>

using namespace std;

bool pred1(obj);
bool pred2(obj1, obj2);
bool cmp2(obj1, obj2); // bool less<Tipo>(o1, o2)

void proc1(obj);
void proc2(obj1, obj2);
tipo func1(obj);
tipo func2(obj1, obj2);

//Objetos función => res operator() (t1 a1, t2 a2) const {}
//base de los objetos
unary_function<argument_type, result_type>
binary_function<first_argument_type, second_argument_type, result_type>

template <typename Tipo>
struct logical_not : public unary_function < Tipo , bool > {
    bool operator() (const Tipo& x) const { return !x; }
};
template <typename Tipo>
struct less : public binary_function < Tipo , Tipo , bool > {
    bool operator() (const Tipo& x, const Tipo& y) const { return x < y; }
};

```

```

pair<it1,it2> p1 = mismatch(vi.begin(), vi.end(), li.begin(), less<int>());

struct persona {
    string nombre;
    . . .
};

class persona_eq : public unary_function< persona , bool > {
public:
    explicit persona_eq(const string& n) : _nm(n) {}
    bool operator() (const persona& p) const { return _nm == p.nombre; }
private:
    const string _nm;
};

it = find_if(lc.begin(), lc.end(), persona_eq("pepe"));

```

Creación de Objetos función a partir de existentes

- Binder: permite que una función de 2 argumentos se utilice como una función de 1 argumento, ligando un argumento a un valor fijo.

```

objeto_funcion_unario = bind1st(operacion_binaria, argumento_fijo_1);
objeto_funcion_unario = bind2nd(operacion_binaria, argumento_fijo_2);

it = find_if(c.begin(), c.end(), bind2nd(less<int>(), 7));

```

- Adapter: convierte la llamada a una función con un objeto como argumento a una llamada a un método de dicho objeto. Ej: `of(obj)` se convierte a `obj->m1()`.

```

objeto_funcion_unario = mem_fun(ptr_funcion_miembro_0_arg);
objeto_funcion_unario = mem_fun_ref(ptr_funcion_miembro_0_arg);

// l es un contenedor<Shape*>
for_each(l.begin(), l.end(), mem_fun(&Shape::draw)); // elem->draw()
// l es un contenedor<Shape>
for_each(l.begin(), l.end(), mem_fun_ref(&Shape::draw)); // elem.draw()
// l es un contenedor<string>
it = find_if(l.begin(), l.end(), mem_fun_ref(&string::empty)); // elem.empty()

objeto_funcion_binario = mem_fun1(ptr_funcion_miembro_1_arg);
objeto_funcion_binario = mem_fun1_ref(ptr_funcion_miembro_1_arg);

// l es un contenedor<Shape*>      elem->rotate(30)
for_each(l.begin(), l.end(), bind2nd(mem_fun1(&Shape::rotate), 30));
// l es un contenedor<Shape>      elem.rotate(30)
for_each(l.begin(), l.end(), bind2nd(mem_fun1_ref(&Shape::rotate), 30));

```

Un adaptador de puntero a función crea un objeto función equivalente a una función. Para utilizar funciones con los “binders” y “adapters”.

```

objeto_funcion_unario = ptr_fun(ptr_funcion_1_arg);
objeto_funcion_binario = ptr_fun(ptr_funcion_2_arg);

it = find_if(l.begin(), l.end(), not1(bind2nd(ptr_fun(strcmp), "pepe")));

```

- Negater: permite expresar la negación de un predicado.

```

objeto_funcion_unario = not1(predicado_unario);
objeto_funcion_binario = not2(predicado_binario);

it = find_if(l.begin(), l.end(), not1(bind2nd(ptr_fun(strcmp), "pepe")));
p1 = mismatch(l.begin(), l.end(), li.begin(), not2(less<int>()));

```

Ejemplo:

```

#include <iostream>
#include <string>
#include <list>
#include <algorithm>

using namespace std;

typedef list<const char*> List_str;

/*
 * Busca la primera ocurrencia de 'cad' en lista
 */
const char*
buscar(const List_str& ls, const char* cad)
{
    typedef List_str::const_iterator LI;
    LI p = find_if(ls.begin(), ls.end(),
        not1(bind2nd(ptr_fun(strcmp), cad)) );
/*
 * LI p = find_if(ls.begin(), ls.end(),
 *               bind2nd(equal_to<string>(), cad) );
 */
    if (p == ls.end()) {
        return NULL;
    } else {
        return *p;
    }
}

int
main()
{
    List_str lista;
    lista.push_back("maría");
    lista.push_back("lola");
    lista.push_back("pepe");
    lista.push_back("juan");
    const char* nm = buscar(lista, "lola");
    if (nm == NULL) {
        cout << "No encontrado" << endl;
    } else {
        cout << nm << endl;
    }
}

```

■ Objetos Predicados

```

equal_to <tipo> (x, y)
not_equal_to <tipo> (x, y)
greater <tipo> (x, y)
less <tipo> (x, y)
greater_equal <tipo> (x, y)
less_equal <tipo> (x, y)

```

```

logical_and <tipo> (b, b)
logical_or <tipo> (b, b)
logical_not <tipo> (b)

```

■ Objetos Operaciones Aritméticas

```

plus <tipo> (x, y)
minus <tipo> (x, y)
multiplies <tipo> (x, y)
divides <tipo> (x, y)
modulus <tipo> (x, y)
negate <tipo> (x)

```

24.21. Algoritmos

```
#include <algorithm>
```

■ Operaciones que no modifican la secuencia

```

for_each(f, l, proc1)           // aplica proc1 a [f:l)
it = find(f, l, x)              // busca x en [f:l)
it = find_if(f, l, pred1)       // busca si pred es true en [f:l)
it = find_first_of(f, l, p, u)  // busc prim [p:u) en [f:l)
it = find_first_of(f, l, p, u, pred2) //
it = adjacent_find(f, l)        // encuentra adyacentes iguales
it = adjacent_find(f, l, pred2) //
n = count(f, l, x)              // cuenta cuantas veces aparece x en [f:l)
n = count_if(f, l, pred1)       //
b = equal(f, l, f2)             // compara 2 secuencias
b = equal(f, l, f2, pred2)      //
p = mismatch(f, l, f2)          // busca la primera diferencia
p = mismatch(f, l, f2, pred2)   //
it = search(f, l, p, u)         // buscan una subsecuencia en otra
it = search(f, l, p, u, pred2)  //
it = find_end(f, l, p, u)       // search hacia atras
it = find_end(f, l, p, u, pred2) //
it = search_n(f, l, n, x)       // busca la sec "x n veces" en [f:l)
it = search_n(f, l, n, x, pred2) //

```

■ Operaciones que modifican la secuencia

```

transform(f, l, d, func1)       // copia [f:l) a d aplicando func1
transform(f, l, ff, d, func2)   // copia [f:l) a d aplicando func2

copy(f, l, d)                   // copia [f:l) a d
copy_backward(f, l, d)           // copia [f:l) a d (hacia atras)

swap(a, b)                      // intercambia los elementos a y b
iter_swap(ita, itb)             // intercambia los elementos *ita y *itb
swap_ranges(f, l, d)            // swap [f:l) por [d:dd)

replace(f, l, v, nv)            // reemplaza v por nv en [f:l)
replace_if(f, l, pred1, nv)     // reemplaza si pred1 por nv en [f:l)
replace_copy(f, l, d, v, nv)    // reemplaza v por nv de [f:l) en d
replace_copy_if(f, l, d, pred1, nv)

fill(f, l, v)                   // pone los valores de [f:l) a v
fill_n(f, n, v)                 // pone n valores a partir de f a v

```

```

generate(f, l, g)           // pone los valores de [f:l) a g()
generate_n(f, n, g)         // pone n valores a partir de f a g()

it = remove(f, l, v)        // elimina elementos de [f:l) iguales a v
it = remove_if(f, l, pred1)
remove_copy(f, l, d, v)
remove_copy_if(f, l, d, pred1)

it = unique(f, l)           // elimina copias adyacentes de [f:l)
it = unique(f, l, pred1)    // elimina copias adyacentes de [f:l)
unique_copy(f, l, d)        // copia sin duplicaciones de [f:l) a d

reverse(f, l)               // invierte el orden de los elementos
reverse_copy(f, l, d)       // invierte el orden de los elementos

rotate(f, m, l)             // rota [f:l) hasta que m sea el primero
rotate_copy(f, m, l, d)

random_shuffle(f, l)        // baraja aleatoriamente
random_shuffle(f, l, g)

donde aparece d => back_inserter(c)

para eliminar en el mismo contenedor:
    sort(c.begin(), c.end());
    it p = unique(c.begin(), c.end());
    c.erase(p, c.end());

```

■ Ordenaciones

```

sort(f, l)                  // ordena [f:l) (O(n*log(n)), O(n*n) peor caso )
sort(f, l, cmp2)            // ordena [f:l) según cmp2
stable_sort(f, l)           // ordena [f:l) (O(n*log(n)*log(n)))
partial_sort(f, m, l)       // ordena [f:m)
partial_sort(f, m, l, cmp2) // ordena [f:m)
partial_sort_copy(f, l, ff, ll) //
partial_sort_copy(f, l, ff, ll, cmp2) //
nth_element(f, n, l)        // pone el n-ésimo elemento en su posición
nth_element(f, n, l, cmp2)  // pone el n-ésimo elemento en su posición
it = lower_bound(f, l, v)   // primer elemento mayor o igual a v
it = upper_bound(f, l, v)   // primer elemento mayor a v
parit = equal_range(f, l, v) // rango de elementos igual a v
b = binary_search(f, l, v)  // búsqueda binaria (si esta)
b = binary_search(f, l, v, cmp2) //
merge(f, l, ff, ll, d)      // mezcla ordenada
inplace_merge(f, m, l)      // mezcla ordenada
partition(f, l, pred1)      // mueve los elementos que satisfacen pred1
stable_partition(f, l, pred1) // mueve los elementos que satisfacen pred1

```

■ Conjuntos

```

b = includes(f, l, ff, ll)
set_union(f, l, ff, ll, d)
set_intersection(f, l, ff, ll, d)
set_difference(f, l, ff, ll, d)
set_symmetric_difference(f, l, ff, ll, d)

```

■ Heap

```

make_heap(f, l)
push_heap(f, l)
pop_heap(f, l)
sort_heap(f, l)

```

■ Comparaciones

```

x = min(a, b)
x = min(a, b, cmp2)
x = max(a, b)
x = max(a, b, cmp2)
x = min_element(f, l)
x = min_element(f, l, cmp2)
x = max_element(f, l)
x = max_element(f, l, cmp2)
b = lexicographical_compare(f, l, ff, ll)      // <
b = lexicographical_compare(f, l, ff, ll, cmp2)

```

■ Permutaciones

```

b = next_permutation(f, l)
b = next_permutation(f, l, cmp2)
b = prev_permutation(f, l)
b = prev_permutation(f, l, cmp2)

```

24.22. Garantías (Excepciones) de Operaciones sobre Contenedores

	vector	deque	list	map
clear()	nothrow (copy)	nothrow (copy)	nothrow	nothrow
erase()	nothrow (copy)	nothrow (copy)	nothrow	nothrow
1-element insert()	strong (copy)	strong (copy)	strong	strong
N-element insert()	strong (copy)	strong (copy)	strong	basic
merge()	-----	-----	nothrow (comparison)	-----
push_back()	strong	strong	strong	-----
push_front()	-----	strong	strong	-----
pop_back()	nothrow	nothrow	nothrow	-----
pop_front()	-----	nothrow	nothrow	-----
remove()	-----	-----	nothrow (comparison)	-----
remove_if()	-----	-----	nothrow (predicate)	-----
reverse()	-----	-----	nothrow	-----
splice()	-----	-----	nothrow	-----
swap()	nothrow	nothrow	nothrow	nothrow (copy-of-comparison)

unique()		-----	-----	nothrow	-----
				(comparison)	
-----+					

Las siguientes garantías se ofrecen bajo la condición de que las operaciones suministradas por el usuario (asignaciones, swap, etc) no dejen a los elementos del contenedor en un estado inválido, que no pierdan recursos, y que los destructores no eleven excepciones.

basic las invariantes básicas de la biblioteca se mantienen y no se producen pérdidas de recursos (como la memoria)

strong además de la básica, la operación, o tiene éxito o no tiene efecto.

nothrow además de la básica, se garantiza que no elevará excepciones.

24.23. Numéricos

```
#include <numeric>

v = accumulate(f, l, vi)           // v = vi + SUM(*it) ParaTodo it in [f, l)
v = accumulate(f, l, vi, func2)    // v = f2(vi, F2(*it)) ParaTodo it in [f, l)

v = inner_product(f, l, ff, vi)     // v = vi + SUM(*it1 * *it2) ...
v = inner_product(f, l, ff, vi, f2_1, f2_2) // v = f2_1(vi, F2_2(*it1,*it2)) ...

adjacent_difference(f, l, d)        // [a, b, c, ...] -> [a, b-a, c-b, ...]
adjacent_difference(f, l, d, func2)

partial_sum(f, l, d)               // [a, b, c, ...] -> [a, a+b, a+b+c, ...]
partial_sum(f, l, d, func2)
```

24.24. Límites

```
#include <limits>

numeric_limits<char>::is_specialized = true;
numeric_limits<char>::digits = 7; // dígitos excluyendo signo
numeric_limits<char>::is_signed = true;
numeric_limits<char>::is_integer = true;
numeric_limits<char>::min() { return -128; }
numeric_limits<char>::max() { return 128; }

numeric_limits<float>::is_specialized = true;
numeric_limits<float>::radix = 2; // base del exponente
numeric_limits<float>::digits = 24; // número de dígitos (radix) en mantisa
numeric_limits<float>::digits10 = 6; // número de dígitos (base10) en mantisa
numeric_limits<float>::is_signed = true;
numeric_limits<float>::is_integer = false;
numeric_limits<float>::is_exact = false;
numeric_limits<float>::min() { return 1.17549435E-38F; }
numeric_limits<float>::max() { return 3.40282347E+38F; }
numeric_limits<float>::epsilon() { return 1.19209290E-07F; } // 1+epsilon-1
numeric_limits<float>::round_error() { return 0.5; }
numeric_limits<float>::infinity() { return xx; }
numeric_limits<float>::quiet_NaN() { return xx; }
numeric_limits<float>::signaling_NaN() { return xx; }
numeric_limits<float>::denorm_min() { return min(); }
```



```

numeric_limits<float>::min_exponent = -125;
numeric_limits<float>::min_exponent10 = -37;
numeric_limits<float>::max_exponent = +128;
numeric_limits<float>::max_exponent10 = +38;
numeric_limits<float>::has_infinity = true;
numeric_limits<float>::has_quiet_NaN = true;
numeric_limits<float>::has_signaling_NaN = true;
numeric_limits<float>::has_denorm = denorm_absent;
numeric_limits<float>::has_denorm_loss = false;
numeric_limits<float>::is_iec559 = true;
numeric_limits<float>::is_bounded = true;
numeric_limits<float>::is_modulo = false;
numeric_limits<float>::traps = true;
numeric_limits<float>::tinyness_before = true;
numeric_limits<float>::round_style = round_to_nearest;

```

24.25. Run Time Type Information (RTTI)

```

#include <typeinfo>

class type_info {
public:
    virtual ~type_info();

    bool operator== (const type_info&) const;
    bool operator!= (const type_info&) const;
    bool before (const type_info&) const;

    const char* name() const;
};

const type_info& typeid(type_name) throw();
const type_info& typeid(expression) throw(bad_typeid);

```


Apéndice A

Precedencia de Operadores en C

Precedencia de Operadores y Asociatividad	
Operador	Asociatividad
() [] -> .	izq. a dch.
! ~ ++ -- - (tipo) * & sizeof	[[[dch. a izq.]]]
* / %	izq. a dch.
+ -	izq. a dch.
<< >>	izq. a dch.
< <= > >=	izq. a dch.
== !=	izq. a dch.
&	izq. a dch.
^	izq. a dch.
	izq. a dch.
&&	izq. a dch.
	izq. a dch.
?:	[[[dch. a izq.]]]
= += -= *= /= %= &= ^= = <<= >>=	[[[dch. a izq.]]]
,	izq. a dch.

Orden de Evaluación
Pag. 58-59 K&R 2 Ed.
Como muchos lenguajes, C no especifica el orden en el cual
Los operandos de un operador seran evaluados (excepciones
son && ?: ,). La coma se evalua de izda a dcha y el valor
que toma es el de la derecha.
Asi mismo, el orden en que se evaluan los argumentos de una
función no esta especificado.

Constantes Carácter:
[\n=NL] [\t=TAB] [\v=VTAB] [\b=BS] [\r=RC] [\f=FF] [\a=BEL]

Apéndice B

Precedencia de Operadores en C++

=====	
nombre_clase::miembro	Resolución de ambito
nombre_esp_nombres::miembro	Resolución de ambito
::nombre	Ambito global
::nombre_calificado	Ambito global

objeto.miembro	Selección de miembro
puntero->miembro	Selección de miembro
puntero[expr]	Indexación
expr(list_expr)	Llamada a función
tipo(list_expr)	Construcción de valor
valor_i++	Post-incremento
valor_i--	Post-decremento
typeid(tipo)	Identificación de tipo
typeid(expr)	Identificación de tipo en tiempo de ejecución
dynamic_cast<tipo>(expr)	Conversión en TEjecución con verificación
static_cast<tipo>(expr)	Conversión en TCompilación con verificación
reinterpret_cast<tipo>(expr)	Conversión sin verificación
const_cast<tipo>(expr)	Conversión const

sizeof expr	Tamaño del objeto
sizeof(tipo)	Tamaño del tipo
++valor_i	Pre-incremento
--valor_i	Pre-decremento
~expr	Complemento
!expr	Negación logica
-expr	Menos unario
+expr	Mas unario
&valor_i	Dirección de
*expr	Desreferencia
new tipo	Creación (asignación de memoria)
new tipo(list_expr)	Creación (asignación de memoria e iniciación)
new tipo [expr]	Creación de array (asignación de memoria)
new (list_expr) tipo	Creación (emplazamiento)
new (list_expr) tipo(list_expr)	Creación (emplazamiento e iniciación)
delete puntero	Destrucción (liberación de memoria)
delete [] puntero	Destrucción de un array
(tipo) expr	Conversión de tipo

objeto.*puntero_a_miembro	Selección de miembro

puntero->puntero_a_miembro	Selección de miembro

expr * expr	Multiplicación
expr / expr	División
expr % expr	Módulo

expr + expr	Suma
expr - expr	Resta

expr << expr	Desplazamiento a izquierda
expr >> expr	Desplazamiento a derecha

expr < expr	Menor que
expr <= expr	Menor o igual que
expr > expr	Mayor que
expr >= expr	Mayor o igual que

expr == expr	Igual
expr != expr	No igual

expr & expr	AND de bits

expr ^ expr	XOR de bits

expr expr	OR de bits

expr && expr	AND logico

expr expr	OR logico

expr ? expr : expr	Expresión condicional

valor_i = expr	Asignación simple
valor_i *= expr	Multiplicación y asignación
valor_i /= expr	División y asignación
valor_i %= expr	Módulo y asignación
valor_i += expr	Suma y asignación
valor_i -= expr	Resta y asignación
valor_i <<= expr	Despl izq y asignación
valor_i >>= expr	Despl dch y asignación
valor_i &= expr	AND bits y asignación
valor_i ^= expr	XOR bits y asignación
valor_i = expr	OR bits y asignación

throw expr	Lanzar una excepción

expr , expr	Secuencia
=====	

Notas

Cada casilla contine operadores con la misma precedencia.

Los operadores de casillas mas altas tienen mayor precedencia que los operadores de casillas mas bajas.

Los operadores unitarios, los operadores de asignación y el operador condicional son asociativos por la derecha.

Todos los demas son asociativos por la izquierda.

Sacado de "El Lenguaje de Programación C++" de B. Stroustrup. pg. 124
=====

Apéndice C

Biblioteca Básica ANSI-C (+ conio)

En este apéndice veremos superficialmente algunas de las funciones más importantes de la biblioteca estándar de ANSI-C y C++.

C.1. `cassert`

```
#include <cassert>

void assert(bool expresion);
    // macro de depuración. Aborta y mensaje si expresión es falsa
    // si NDEBUG esta definido, se ignora la macro
```

C.2. `cctype`

```
#include <cctype>

bool isalnum(char ch); // (isalpha(ch) || isdigit(ch))
bool isalpha(char ch); // (isupper(ch) || islower(ch))
bool iscntrl(char ch); // caracteres de control
bool isdigit(char ch); // dígito decimal
bool isgraph(char ch); // caracteres imprimibles excepto espacio
bool islower(char ch); // letra minúscula
bool isprint(char ch); // caracteres imprimibles incluyendo espacio
bool ispunct(char ch); // carac. impr. excepto espacio, letra o dígito
bool isspace(char ch); // esp, \r, \n, \t, \v, \f
bool isupper(char ch); // letra mayúscula
bool isxdigit(char ch); // dígito hexadecimal

char tolower(char ch); // convierte ch a minúscula
char toupper(char ch); // convierte ch a mayúscula
```

C.3. `cmath`

```
#include <cmath>

double sin(double rad); // seno de rad (en radianes)
double cos(double rad); // coseno de rad (en radianes)
double tan(double rad); // tangente de rad (en radianes)
```

```

double asin(double x);           // arco seno de x, x en [-1,1]
double acos(double x);           // arco coseno de x, x en [-1,1]
double atan(double x);           // arco tangente de x
double atan2(double y, double x); // arco tangente de y/x
double sinh(double rad);         // seno hiperbólico de rad
double cosh(double rad);         // coseno hiperbólico de rad
double tanh(double rad);         // tangente hiperbólica de rad
double exp(double x);            // e elevado a x
double log(double x);            // logaritmo neperiano ln(x), x > 0
double log10(double x);          // logaritmo decimal log10(x), x > 0
double pow(double x, double y);  // x elevado a y
double sqrt(double x);           // raíz cuadrada de x, x >= 0
double ceil(double x);           // menor entero >= x
double floor(double x);          // mayor entero <= x
double fabs(double x);           // valor absoluto de x
double ldexp(double x, int n);   // x * 2 elevado a n
double frexp(double x, int* exp); // inversa de ldexp
double modf(double x, double* ip); // parte entera y fraccionaria
double fmod(double x, double y); // resto de x / y

```

C.4. cstdlib

```

#include <cstdlib>

double atof(const char orig[]); // cadena a double
int atoi(const char orig[]);    // cadena a int
long atol(const char orig[]);   // cadena a long

double strtod(const char orig[], char** endp);
long strtol(const char orig[], char** endp, int base);
unsigned long strtoul(const char orig[], char** endp, int base);

void srand(unsigned semilla); // srand(time(0));
int rand(); // devuelve un aleatorio entre 0 y RAND_MAX (ambos inclusive)

/* Devuelve un número aleatorio entre 0 y max (exclusive) */
inline unsigned aleatorio(int max) { return unsigned(max*doble(rand())/(RAND_MAX+1.0)); }

void abort(); // aborta el programa como error
void exit(int estado); // terminación normal del programa
int atexit(void (*fcn)(void));
    // función a ejecutar cuando el programa termina normalmente

int system(const char orden[]);
    // orden a ejecutar por el sistema operativo

char* getenv(const char nombre[]);
    // devuelve el valor de la variable de entorno 'nombre'

int abs(int n); // valor absoluto
long labs(long n); // valor absoluto

void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*));
void *bsearch(const void *key, const void *base, size_t nitems, size_t size,
    int (*compar)(const void *, const void *));

void *malloc(size_t size);
void *realloc(void *ptr, size_t size);

```

```
void *calloc(size_t nitems, size_t size);
void free(void *ptr);
```

C.5. climits

```
#include <climits>

CHAR_BIT = 8

CHAR_MAX      | SHRT_MAX      | INT_MAX      | LONG_MAX     | LONG_LONG_MAX
CHAR_MIN      | SHRT_MIN      | INT_MIN      | LONG_MIN     | LONG_LONG_MIN
UCHAR_MAX     | USHRT_MAX     | UINT_MAX     | ULONG_MAX    | ULONG_LONG_MAX
SCHAR_MAX     |               |              |              |
SCHAR_MIN     |               |              |              |
```

C.6. cfloat

```
#include <cfloat>

FLT_EPSILON    // menor número float X tal que 1.0+X != 1.0
FLT_MAX        // máximo número de punto flotante
FLT_MIN        // mínimo número normalizado de punto flotante

DBL_EPSILON    // menor número double X tal que 1.0+X != 1.0
DBL_MAX        // máximo número double de punto flotante
DBL_MIN        // mínimo número double normalizado de punto flotante

LDBL_EPSILON   // menor número long double X tal que 1.0+X != 1.0
LDBL_MAX       // máximo número long double de punto flotante
LDBL_MIN       // mínimo número long double normalizado de punto flotante
```

C.7. ctime

```
#include <ctime>

clock_t clock();
    // devuelve el tiempo de procesador empleado por el programa
    // para pasar a segundos: double(clock())/CLOCKS_PER_SEC
    // en un sistema de 32 bits donde CLOCKS_PER_SEC = 1000000
    // tiene un rango de 72 minutos aproximadamente. (c2-c1)
time_t time(time_t* tp);
    // devuelve la fecha y hora actual del calendario
double difftime(time_t t2, time_t t1);
    // devuelve t2 - t1 en segundos
time_t mktime(struct tm* tp);
struct tm* gmtime(const time_t* tp);
```

C.8. cstring

```
#include <cstring>

unsigned strlen(const char s1[]);
    // devuelve la longitud de la cadena s1

char* strcpy(char dest[], const char orig[]);
```

```

char* strncpy(char dest[], const char orig[], unsigned n);
    // Copia la cadena orig a dest (incluyendo el terminador '\0').
    // Si aparece 'n', hasta como máximo 'n' caracteres
    // (si alcanza el límite, no incluye el terminador '\0')

char* strcat(char dest[], const char orig[]);
char* strncat(char dest[], const char orig[], unsigned n);
    // Concatena la cadena orig a la cadena dest.
    // Si aparece 'n', hasta como máximo 'n' caracteres
    // (si alcanza el límite, no incluye el terminador '\0')

int strcmp(const char s1[], const char s2[]);
int strncmp(const char s1[], const char s2[], unsigned n);
    // Compara lexicográficamente las cadenas s1 y s2.
    // Si aparece 'n', hasta como máximo 'n' caracteres
    // devuelve <0 si s1<s2, 0 si s1==s2, >0 si s1>s2

const char* strchr(const char s1[], char ch);
    // devuelve un apuntador a la primera ocurrencia de ch en s1
    // NULL si no se encuentra
const char* strrchr(const char s1[], char ch);
    // devuelve un apuntador a la última ocurrencia de ch en s1
    // NULL si no se encuentra

unsigned strspn(const char s1[], const char s2[]);
    // devuelve la longitud del prefijo de s1 de caracteres
    // que se encuentran en s2
unsigned strcspn(const char s1[], const char s2[]);
    // devuelve la longitud del prefijo de s1 de caracteres
    // que NO se encuentran en s2

const char* strpbrk(const char s1[], const char s2[]);
    // devuelve un apuntador a la primera ocurrencia en s1
    // de cualquier carácter de s2. NULL si no se encuentra

const char* strstr(const char s1[], const char s2[]);
    // devuelve un apuntador a la primera ocurrencia en s1
    // de la cadena s2. NULL si no se encuentra

void* memcpy(void* dest, const void* origen, unsigned n);
    // copia n caracteres de origen a destino. NO Válido si se solapan

void* memmove(void* dest, const void* origen, unsigned n);
    // copia n caracteres de origen a destino. Válido si se solapan

int memcmp(const void* m1, const void* m2, unsigned n);
    // Compara lexicográficamente 'n' caracteres de m1 y m2
    // devuelve <0 si m1<m2, 0 si m1==m2, >0 si m1>m2

const void* memchr(const void* m1, char ch, unsigned n);
    // devuelve un apuntador a la primera ocurrencia de ch
    // en los primeros n caracteres de m1. NULL si no se encuentra
void* memset(void* m1, char ch, unsigned n);
    // coloca el carácter ch en los primeros n caracteres de m1

```

C.9. cstdio

```
#include <cstdio>
```

```

int remove(const char filename[]); // elimina filename. Si fallo -> != 0
int rename(const char oldname[], const char newname[]);

FILE *fopen(const char *filename, const char *mode); // mode: "rwab+"
FILE *freopen(const char *filename, const char *mode, FILE *stream);
int fclose(FILE *stream);

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

int fflush(FILE *stream);

int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);

long ftell(FILE *stream);
int fseek(FILE *stream, long offset, int whence); // SEEK_SET, SEEK_CUR, SEEK_END
void rewind(FILE *stream);

int ferror(FILE *stream);
int feof(FILE *stream);
void clearerr(FILE *stream);

void setbuf(FILE *stream, char *buffer);
int setvbuf(FILE *stream, char *buffer, int mode, size_t size);

FILE *tmpfile(void);
char *tmpnam(char *str);

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int vprintf(const char *format, va_list arg);
int vfprintf(FILE *stream, const char *format, va_list arg);
int vsprintf(char *str, const char *format, va_list arg);

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);

int putchar(int char);
int putc(int char, FILE *stream);
int fputc(int char, FILE *stream);
int fputs(const char *str, FILE *stream);

int getchar(void); // EOF is returned
int getc(FILE *stream); // EOF is returned
int fgetc(FILE *stream); // EOF is returned
char *fgets(char *str, int n, FILE *stream);

int ungetc(int char, FILE *stream);

void perror(const char *str);

```

C.10. cstdarg

```
#include <cstdarg>
```

```

void va_start(va_list ap, last_arg);
type va_arg(va_list ap, type);
void va_end(va_list ap);

int acumular(int num_args, ...)
{
    int i;
    int sum=0;
    va_list ap;
    va_start(ap,num_args);
    for(i = 0; i < num_args; ++i) {
        sum += va_arg(ap, int);
    }
    va_end(ap);
    return sum;
}

```

C.11. conio.h

```

#include <conio.h>  (no es estandar ANSI)

- Salida y entrada de texto
  int cprintf(const char* fmt, ...);
      // envia texto formateado a la pantalla
  int cputs(const char* _str);
      // envia una cadena de caracteres a pantalla
  int putch(int _c);
      // envia un carácter simple a pantalla
  int cscanf(const char* fmt, ...);
      // entrada formateada de datos de consola
  char* cgets(char* _str);
      // entrada de una cadena de caracteres de consola
  int getche(void);
      // lee un carácter (con eco)
  int getch(void);
      // lee un carácter (sin eco)
  int kbhit(void);
      // comprueba si hay pulsaciones de teclado
  int ungetch(int);
      // devuelve el carácter al buffer de teclado

- Manipulación de texto (y cursor) en pantalla
  void clrscr(void);
      // borra e inicializa la pantalla
  void clreol(void);
      // borra la línea desde el cursor al final
  void delline(void);
      // elimina la línea donde se encuentra el cursor
  void gotoxy(int x, int y);
      // posiciona el cursor
  void inpline(void);
      // inserta una línea en blanco
  int movetext(int _left, int _top, int _right, int _bottom,
              int _destleft, int _desttop);
      // copia texto de una zona de pantalla a otra

- Movimientos de bloques

```

```

int gettext(int _left, int _top, int _right, int _bottom, void*_destin);
    // copia texto de pantalla a memoria
int puttext(int _left, int _top, int _right, int _bottom, void*_source);
    // copia texto de memoria a pantalla

- Control de ventanas
void textmode(int _mode);
    // pone el modo texto
void window(int _left, int _top, int _right, int _bottom);
    // define una ventana de texto
void _set_screen_lines(int nlines);
void _setcursortype(int _type); // _NOCURS, _SOLIDCURSOR, _NORMALCURSOR

- Ajuste de color del texto y del fondo
void textcolor(int _color);
    // actualiza el color de texto
void textbackground(int _color);
    // actualiza el color de fondo
void textattr(int _attr);
    // actualiza el color de texto y el de fondo al mismo tiempo

- Control de intensidad
void intensevideo(void);
void highvideo(void);
    // alta intensidad
void lowvideo(void);
    // baja intensidad
void normvideo(void);
    // intensidad normal
void blinkvideo(void);
    // parpadeo

- Información
void gettextinfo(struct text_info *_r);
    // información sobre la ventana actual
int wherex(void);
    // valor de la coordenada x del cursor
int wherey(void);
    // valor de la coordenada y del cursor

struct text_info {
    unsigned char winleft;
    unsigned char wintop;
    unsigned char winright;
    unsigned char winbottom;
    unsigned char attribute;
    unsigned char normattr;
    unsigned char currmode;
    unsigned char screenheight; // Número de líneas de la pantalla
    unsigned char screenwidth;  // Número de columnas de la pantalla
    unsigned char curx;          // Columna donde se encuentra el cursor
    unsigned char cury;          // Fila donde se encuentra el cursor
};

```

La esquina superior izquierda se corresponde con las coordenadas [1,1].

Apéndice D

El Preprocesador

Directivas de pre-procesamiento:

```
#include <system_header>
#include "user_header"

#line xxx
#error mensaje
#pragma ident

#define mkstr_2(xxx)    #xxx
#define mkstr(xxx)     mkstr_2(xxx)
#define concat(a,b)    a##b
#define concatenar(a,b) concat(a,b)
#undef xxx

/-- Simbolos predefinidos estandares -----
NDEBUG
__LINE__
__FILE__
__DATE__
__TIME__
__STDC__
__STDC_VERSION__
__func__
__cplusplus
/-- Simbolos predefinidos en GCC -----
__GNUC__
__GNUC_MINOR__
__GNUC_PATCHLEVEL__
__GNUG__
__STRICT_ANSI__
__BASE_FILE__
__INCLUDE_LEVEL__
__VERSION__
__OPTIMIZE__          [-O -O1 -O2 -O3 -Os]
__OPTIMIZE_SIZE__     [-Os]
__NO_INLINE__
__FUNCTION__
__PRETTY_FUNCTION__
/-------
__linux__
__unix__
__MINGW32__
__WIN32__
```

```
//-----  
  
#ifndef __STDC__  
#ifdef __cplusplus  
#ifndef NDEBUG  
#if ! ( defined(__GNUC__)  
    && ( (__GNUC__ >= 4) || ((__GNUC__ == 3)&&(__GNUC_MINOR__ >= 2))) \  
    && ( defined __linux__ || defined __MINGW32__ ))  
#error "Debe ser GCC versión 3.2 o superior sobre GNU/Linux o MinGW/Windows "  
#error "para usar tipos_control"  
#elif defined __STDC_VERSION__ && __STDC_VERSION__ >= 199901L  
#else  
#endif
```

Apéndice E

Errores Más Comunes

- utilizar = (asignación) en vez de == (igualdad) en comparaciones.

```
if (x = y) {    // error: asignación en vez de comparación
    ...
}
```

- Utilización de operadores relacionales.

```
if (0 <= x <= 99) {    // error:
    ...
}
```

debe ser:

```
if ((0 <= x) && (x <= 99)) {
    ...
}
```

- Olvidar poner la sentencia **break**; tras las acciones de un **case** en un **switch**.

```
switch (x) {
case 0:
case 1:
    cout << "Caso primero" << endl;
    break;
case 2:
    cout << "Caso segundo" << endl;
    // error: no hemos puesto el break
default:
    cout << "Caso por defecto" << endl;
    break;
}
```

debe ser:

```
switch (x) {
case 0:
case 1:
    cout << "Caso primero" << endl;
    break;
case 2:
    cout << "Caso segundo" << endl;
    break;
default:
```

```

        cout << "Caso por defecto" << endl;
        break;
    }

```

- Al alojar agregados dinámicos para contener cadenas de caracteres, no solicitar espacio para contener el terminador '\0'

```

char cadena[] = "Hola Pepe";

char* ptr = new char[strlen(cadena)]; // error. sin espacio para '\0'
strcpy(ptr, cadena); // error, copia sin espacio

```

debe ser:

```

char* ptr = new char[strlen(cadena)+1]; // OK. espacio para '\0'
strcpy(ptr, cadena);

```

- Al alojar con

```

tipo* p = new tipo[30];

```

desalojar con

```

delete p;

```

en vez de con

```

delete [] p;

```

- Overflow en Arrays y en Cadenas de Caracteres (al estilo-C)

Apéndice F

Características no Contempladas

- Paso de parámetros variable

Apéndice G

Bibliografía

- El Lenguaje de Programación C. 2.Ed.
B.Kernighan, D. Ritchie
Prentice Hall 1991
- The C++ Programming Language. Special Edition
B. Stroustrup
Addison Wesley 2000
- C++ FAQ-Lite
M. Cline
<http://www.parashift.com/c++-faq-lite/>

Índice alfabético

- `::`, 137
- ámbito de visibilidad, 34
- `@`, 9
- `OBS`, 9
- agregado, 62, 63
 - acceso, 64
 - multidimensional, 68
 - predefinido, 87
 - acceso, 88
 - multidimensional, 92
 - size, 64
 - tamaño, 64
- array, 62, 63
 - acceso, 64
 - multidimensional, 68
 - predefinido, 87
 - acceso, 88
 - multidimensional, 92
 - parámetros, 95
 - size, 64
 - tamaño, 64
- búsqueda
 - binaria, 78
 - lineal, 77
- biblioteca
 - ansic
 - cassert, 321
 - cctype, 111, 321
 - cfloat, 323
 - climits, 323
 - cmath, 111, 321
 - cstdarg, 325
 - cstdio, 324
 - cstdlib, 112, 322
 - cstring, 323
 - ctime, 323
 - conio.h, 326
 - stl
 - acceso, 287
 - asignación, 288
 - constructores, 287
 - contenedores, 286
 - ficheros, 285
 - garantías, 311
 - iteradores, 286
 - operaciones, 287
 - operaciones asociativas, 288
 - operaciones de lista, 287
 - operaciones de pila y cola, 287
 - operaciones sobre iteradores, 288
 - resumen, 288
 - tipos, 286
- bloque, 33
- buffer, 30, 265
 - de entrada, 30
 - de salida, 30
- cadenas de caracteres estilo-C, 98
 - acceso, 98
 - terminador, 98
- campos de bits, 105
- catch, 148
- cerr, 27
- cin, 29
- comentarios, 15
- compilación separada, 135
- constantes
 - literales, 20
 - simbólicas, 20
 - declaración, 21
- constantes literales, 15
- conversiones aritméticas, 23
- conversiones de tipo
 - automáticas, 23
 - explícitas, 23
- conversiones enteras, 23
- cout, 27
- declaración
 - global, 33
 - ámbito de visibilidad, 33
 - local, 33
 - ámbito de visibilidad, 33
 - vs. definición, 17
- declaracion adelantada, 198
- definición
 - vs. declaración, 17
- delete, 193, 280

- delete [], 241
- delimitadores, 15
- ejecución secuencial, 33
- enlazado, 135
- entrada, 29
- espacios de nombre
 - using, 137
- espacios de nombre, 136, 137
 - ::, 137
 - anónimos, 136
 - using namespace, 137
- espacios en blanco, 15
- estructura, 59
- excepciones, 147
 - catch, 148
 - throw, 148
 - try, 148
- fichero de encabezamiento
 - guardas, 134
- ficheros, 268
 - ejemplo, 269
 - entrada, 268
 - entrada/salida, 130, 270
 - salida, 269
- flujos
 - entrada, 263
 - cadena, 270
 - excepción, 32, 265
 - jerarquía de clases, 271
 - salida, 263
 - cadena, 270
- funciones, 41
 - declaración, 46
 - definición, 42
 - inline, 46
 - return, 43
- guardas, 134
- identificadores, 15
- inline, 46
- listas enlazadas
 - declaracion adelantada, 198
- módulo
 - implementación, 133
 - interfaz, 133
- main, 13
- memoria dinámica, 193
 - abstraccion, 197
 - agregados, 241
 - delete, 193
 - delete [], 241
 - enlaces, 198
 - excepciones, 207
 - listas enlazadas, 200
 - new, 193
 - new T[], 241
- new, 193, 280
- new T[], 241
- operador de dirección, 218
- operadores, 15, 21
 - aritméticos, 22
 - bits, 22
 - condicional, 22
 - lógicos, 22
 - relacionales, 22
- ordenación
 - burbuja, 79
 - inserción, 80
 - inserción binaria, 81
 - intercambio, 79
 - selección, 80
- palabras reservadas, 14
- parámetros de entrada, 43
- parámetros de entrada/salida, 44
- parámetros de salida, 44
- paso por referencia, 44
- paso por referencia constante, 44
- paso por valor, 43
- procedimientos, 41
 - declaración, 46
 - definición, 42
 - inline, 46
- programa C++, 13
- promociones, 23
- prototipo, 46
- put, 30
- registro, 59
- return, 43
- símbolo
 - \mathbb{A} , 9
 - OBS, 9
- salida, 27
- secuencia de sentencias, 33
- sentencia
 - asignación, 34
 - incremento/decremento, 34
 - iteración, 37
 - do while, 39
 - for, 38
 - while, 37
 - selección, 35

- if, 35
 - switch, 36
- throw, 148
- tipo, 17
- tipos
 - cuadro resumen, 19
 - puntero, 192
 - a subprogramas, 242
 - acceso, 194
 - operaciones, 194
 - parámetros, 196
 - punteroagregado
 - parámetros, 241
- tipos compuestos, 17, 51
 - array, 62, 63
 - acceso, 64
 - multidimensional, 68
 - predefinido, 87
 - size, 64
 - cadenas estilo-C, 98
 - campos de bits, 105
 - parámetros, 51
 - struct, 59
 - uniones, 104
- tipos simples, 17
 - enumerado, 19
 - escalares, 18
 - ordinales, 18
 - predefinidos, 17
 - bool, 17
 - char, 17
 - double, 18
 - float, 18
 - int, 18
 - long, 18
 - long long, 18
 - short, 18
 - unsigned, 18
- try, 148
- unión, 104
- using, 137
- using namespace, 137
- variables
 - declaración, 21