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

Programación Elemental en C++

Manual de Referencia Abreviado

Revision : 1,20

Índice general

Prólogo

Este manual pretende ser una guía de referencia abreviada para la utilización del lenguaje C++ en el desarrollo de programas. Está orientada a alumnos de primer curso de programación de Ingeniería Informática.

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++. De hecho no considera ningún aspecto del lenguaje enfocado al paradigma de "Programación Orientada a Objetos".

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 y se encuentra actualmente bajo desarrollo. Se difunde en la creencia de que puede ser útil, aún siendo una versión preliminar.

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 (cap. 10).

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 (vease capítulo 2) y de subprogramas (cap. 4) 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, llamada *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 trás 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 imprime los números menores que uno dado por teclado.

```
//- fichero: numeros.cpp ----------------------------------------------
#include <iostream> // biblioteca de entrada/salida

using namespace std;

/*
 * Imprime los numeros menores a 'n'
 */
void
numeros(int n)
{
   for (int i = 0; i < n; ++i) {
       cout << i << " "; // escribe el valor de 'i'
   }
   cout << endl;   // escribe 'salto de linea'
}

int
main()
{
   int maximo;

   cout << "Introduce un numero: ";
   cin >> maximo;
```

```
   numeros(maximo);

   // return 0;
}
//- fin: numeros.cpp -----------------------------------------------
```

En un programa C++ podemos distinguir los siguientes elementos básicos:

- *Palabras reservadas*

  Son un conjunto de palabras que tienen un significado predeterminado para el compilador, y sólo pueden ser utilizadas con dicho sentido.

- *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 de cualquier longitud, siendo el primer carácter una letra. El _ 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: Sólo se utilizarán letras mayúsculas, dígitos y el caracter _.

  Tipos: Comenzarán por una letra mayúscula seguida por letras mayúsculas, minúsculas, digitos o _. Deberá haber como mínimo una letra minúscula.

  Variables: Sólo se utilizarán letras minúsculas, dígitos y el caracter _.

  Funciones: Sólo se utilizarán letras minúsculas, dígitos y el caracter _.

  Campos de Registros: Sólo se utilizarán letras minúsculas, dígitos y el caracter _.

- *Constantes literales*

  Son valores que aparecen explícitamente en el programa, y podrán ser numéricos, caracteres y cadenas.

- *Operadores*

  Símbolos con significado propio según el contexto en el que se utilicen.

- *Delimitadores*

  Símbolos que indican comienzo o fin de una entidad.

- *Comentarios y espacios en blanco*

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

  Para comentarios cortos en línea con el código de programa utilizaremos // que indica comentario hasta el final de la línea.

  ```
  media =  suma / n; // media de 'n' numeros
  ```

  Para comentarios largos (de varias lineas) utilizaremos /* que indica comentario hasta */

  ```
  /*
   * Ordena por el metodo quicksort
   *  Recibe el array y el numero de elementos que contine
   *  Devuelve el array ordenado
   */
  ```

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

Nota: en C++, las letras minúsculas se consideran diferentes de las mayúsculas.

Capítulo 2

Tipos simples

El *tipo* define las caracteristicas 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.
- La interpretación del valor almacenado.
- El espacio de almacenamiento necesario para almacenar dichos valores.
- El conjunto de operaciones/manipulaciones aplicables a la entidad.

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 mediante operaciones de bits) y los *tipos compuestos* se caracterizan por estar formados como un agregado o composición de otros, 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, solo exista una única definición, aunque puede haber varias declaraciones.

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 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 logicas `true` y `false`. Suele almacenarse en el tamaño de palabra mas pequeño posible direccionable (normalmente 1 byte).

El tipo `char` se utiliza para representar los caracteres, es decir, símbolos alfanuméricos, de puntuación, espacios, control, etc y normalmente utilizan un espacio de almacenamiento de 1 byte (8 bits) y puede representar 256 posibles 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 dia normalmente es de 4 bytes (32 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]).
>
> También puede ser modificado para representar solamente números Naturales 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` tambien puede ser modificado con `long` para representar "cuadruple precisión" (normalmente 12 bytes [96 bits]).

Veamos un cuadro resumen con los tipos predefinidos, su espacio de almacenamiento y el rango de valores para una máquina de 32 bits:

```
-------------------------------------------------------------------------
bool:  1 bytes
char:  1 bytes        Min: -128         Max: 127          UMax: 255
short: 2 bytes        Min: -32768       Max: 32767        UMax: 65535
int:   4 bytes        Min: -2147483648  Max: 2147483647   UMax: 4294967295
long:  4 bytes        Min: -2147483648  Max: 2147483647   UMax: 4294967295
float: 4 bytes        Min: 1.17549e-38  Max: 3.40282e+38
double:8 bytes        Min: 2.22507e-308 Max: 1.79769e+308
long double: 12 bytes Min: 3.3621e-4932 Max: 1.18973e+4932
-------------------------------------------------------------------------
```

2.3. Tipos simples enumerados

Ademas de los tipos simples predefinidos, el programador puede definir nuevos tipos simples que definan 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 llama *tipo enumerado* 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 los diferentes valores enumerados.

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
```

Todos los tipos simples tienen la propiedad de mantener una relación de orden (se les puede aplicar operadores relacionales). Se les conoce también como tipos *Escalares*. Todos, 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*".

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*).

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:

- lógicas (bool)

  ```
  false, true
  ```

- caracter `char` (símbolo entre comillas simples)

  ```
  'a', 'b', ..., 'z',
  'A', 'B', ..., 'Z',
  '0', '1', ..., '9',
  ' ', '.', ',', ':', ';', ...
  '\n', '\r', '\b', '\t', '\a', '\x5F', '\35'...
  ```

- cadenas de caracteres literales (caracteres entre comillas dobles)

  ```
  "Hola Pepe"
  "Hola\nJuan"
  "Hola " "Maria"
  ```

- enteros

  ```
  123, -1520, 30000L, 50000UL, 0x10B3FC23, 0751
  ```

- reales (punto flotante)

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

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

Ejemplos de constantes simbólicas:

```
const char CAMPANA = '\a';
const int MAXIMO = 5000;
const long ULTIMO = 100000L;
const short ELEMENTO = 1000;
const unsigned FILAS = 200U;
const unsigned long COLUMNAS = 200UL;
const float N_E = 2.7182F;
const double N_PI = 3.141592;
const Color COLOR_DEFECTO = ROJO;
const double LOG10E = log(N_E);
```

Las *variables* se definen especificando su tipo e identificador con el que nos referiremos a ella. Se les podrá asignar un valor inicial en la definición.

```
int contador = 0;
float total = 5.0;
```

2.5. Operadores

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

```
sizeof(tipo)
```

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

```
! ~ -          U      asoc. dcha. a izq.
* / %          B      asoc. izq. a dcha.
+ -            B      asoc. izq. a dcha.
<< >>          B      asoc. izq. a dcha.
< <= > >=      B      asoc. izq. a dcha.
== !=          B      asoc. izq. a dcha.
&              B      asoc. izq. a dcha.
^              B      asoc. izq. a dcha.
|              B      asoc. izq. a dcha.
&&             B      asoc. izq. a dcha.
||             B      asoc. izq. a dcha.
?:             T      asoc. dcha. a izq.
```

donde `U`, `B` y `T` significan "operador unario", "operador binario", y "operador ternario" respectivamente.

Significado de los operadores:

- Aritméticos

```
- valor             // menos unario
valor * valor       // producto
valor / valor       // division (entera y real)
valor % valor       // modulo (resto) (no reales)
valor + valor       // suma
valor - valor       // resta
```

- Relacionales (resultado bool)

```
valor <  valor      // comparacion menor
valor <= valor      // comparacion menor o igual
valor >  valor      // comparacion mayor
valor >= valor      // comparacion mayor o igual
valor == valor      // comparacion igualdad
valor != valor      // comparacion desigualdad
```

- Operadores de Bits (sólo ordinales)

```
valor << despl      // desplazamiento de bits a la izq.
valor >> despl      // desplazamiento de bits a la dch.
~ valor             // negacion de bits (complemento)
valor & valor       // and de bits
valor ^ valor       // xor de bits
valor | valor       // or de bits
```

- Lógicos (sólo booleanos)

```
! log               // negacion logica
log && log          // and logico
log || log          // or logico
```

- Condicional

```
log ? valor1 : valor2   // Si log es true => valor1; sino => valor2
```

2.6. Conversiones de tipos

Es posible que nos interese realizar operaciones en las que se mezclen datos de tipos diferentes. C++ realiza *conversiones de tipo automáticas* ("castings"), de tal forma que el resultado de la operación sea del tipo más amplio[1] de los implicados en ella.

No obstante, 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:

```
int('a')        // convierte el caracter 'a' a su valor entero (97)
int(ROJO)       // produce el entero 0
int(AMARILLO)   // produce el entero 2
Color(1)        // produce el Color AZUL
char(65)        // produce el caracter 'A'
double(2)       // produce el real (doble precision) 2.0
int(3.7)        // produce en entero 3
Color(c+1)      // si c es de tipo color, produce el siguiente valor de la enumeracion
```

[1] XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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 AMARILLO
}
```

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

```
int* p_int = static_cast<int*>(p_void);
disp* ptr = reinterpret_cast<disp*>(0xff00);
ave* ptr = dynamic_cast<ave*>(p_animal);
ave& ref = dynamic_cast<ave&>(r_animal);
char* ptr = const_cast<char*>(ptr_const_char)
```

- El operador `static_cast<tipo>(valor)` realiza conversiones entre tipos relacionados como un tipo puntero y otro tipo puntero de la misma jerarquía de clases, un enumerado y un tipo integral, o un tipo en coma flotante y un tipo integral. Para un tipo primitivo, la conversión de tipos expresada de la siguiente forma: `Tipo(valor)` es equivalente a `static_cast<Tipo>(valor)`.
- 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 cógigo no portable, y posiblemente problemática.
- 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.
- 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.
- 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`.

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.

Capítulo 3

Estructuras de control

Las estructuras de control en 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.

3.1. 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" (;).

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:

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

Es posible el anidamiento de bloques.

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

3.2. 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: 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. Se crean

al principio de la ejecución del programa y se destruyen al finalizar éste. Normalmente serán constantes simbólicas y definiciones de tipos.

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. Se crean al comienzo de la ejecución del bloque (realmente en C++ se crean al ejecutarse la definición) y se destruyen al finalizar éste. Normalmente serán constantes simbólicas y variables locales.

```
{
    declaracion_1;
     . . .
    declaracion_n;

    sentencia_1;
     . . .
    sentencia_m;
}
```

En C++ se pueden declarar/definir entidades en cualquier punto del programa, sin embargo nosotros restringiremos dichas declaraciones/definiciones al principio del programa para las entidades globales (constantes simbólicas y tipos) y al comienzo de los bloques para entidades locales (constantes simbólicas y variables).

Respecto al ámbito de visibilidad de una entidad, en caso de declaraciones de diferentes entidades con el mismo identificador en diferentes niveles de anidamiento de bloques, la entidad visible será aquella que se encuentre declarada en el bloque de mayor nivel de anidamiento. 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.

```
{
    int x;
    . . .          // x es vble de tipo int
    {
        float x;
        . . .      // x es vble de tipo float
    }
    . . .          // x es vble de tipo int
}
```

3.3. Sentencias de asignación

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

```
variable = expresion;
```

Ejemplo:

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

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

```
++variable;              // variable = variable + 1;
--variable;              // variable = variable - 1;

variable++;              // variable = variable + 1;
variable--;              // variable = variable - 1;
```

```
variable += expresion;   // variable = variable + expresion;
variable -= expresion;   // variable = variable - expresion;
variable *= expresion;   // variable = variable * expresion;
variable /= expresion;   // variable = variable / expresion;
variable %= expresion;   // variable = variable % expresion;
variable &= expresion;   // variable = variable & expresion;
variable ^= expresion;   // variable = variable ^ expresion;
variable |= expresion;   // variable = variable | expresion;
variable <<= expresion;  // variable = variable << expresion;
variable >>= expresion;  // variable = variable >> expresion;
```

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

3.4. 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 dependiendo de condiciones lógicas.

La más simple de todas es la sentencia de selección condicional `if` cuya sintaxis es la siguiente:

```
if (condicion_logica) {
    secuencia_de_sentencias;
}
```

y cuya semántica consiste en evaluar la `condicion_logica`, y si su resultado es Verdadero (`true`) entonces se ejecuta la secuencia de sentencias entre las llaves.

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

```
if (condicion_logica) {
    secuencia_de_sentencias_v;
} else {
    secuencia_de_sentencias_f;
}
```

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

La sentencia de selección condicional se puede encadenar de la siguiente forma:

```
if (cond_logica_1) {
    secuencia_de_sentencias_1;
} else if (cond_logica_2) {
    secuencia_de_sentencias_2;
} else if (cond_logica_3) {
    secuencia_de_sentencias_3;
} else if ( ... ) {
    . . .
} else {
    secuencia_de_sentencias_f;
}
```

con el flujo de control esperado.

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 condiciones lógicas, sino en función del valor que tome una determinada expresión de tipo ordinal, es decir, una relación de pertenencia entre el valor de una expresión y unos determinados valores de tipo ordinal especificados. Su sintaxis es la siguiente:

```
switch (expresion) {
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_f;
    break;
}
```

en la cual se evalúa la `expresion`, 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).

3.5. 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:

```
while (condicion_logica) {
    secuencia_de_sentencias;
}
```

en la cual primero se evalúa la `condicion_logica`, y si es cierta, se ejecuta la `secuencia_de_sentencias` completamente. Posteriormente se vuelve a evaluar la `condicion_logica` y si vuelve a ser cierta se vuelve a ejecutar la `secuencia_de_sentencias`. 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.

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 palpable los casos en los que el control de la iteración tiene una clara inicialización, y un claro incremento hasta llegar al caso final. La sintaxis es la siguiente:

```
for (inicializacion ; condicion_logica ; incremento) {
    secuencia_de_sentencias;
}
```

y es equivalente a:

```
inicializacion;
while (condicion_logica) {
    secuencia_de_sentencias;
    incremento;
}
```

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

```
for (int i = 0; i < MAXIMO; ++i) {
    // hacer algo
}
// i ya no es visible aqui
```

La sentencia `do while` presenta la siguiente estructura

```
do {
    secuencia_de_sentencias;
} while (condicion_logica);
```

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 `condicion_logica` 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 `condicion_logica`, y si ésta es cierta se repite el proceso.

Capítulo 4

Subprogramas. Funciones y procedimientos

Los subprogramas (procedimientos y funciones) constituyen una unidad de ejecución mayor que el bloque. Proporcionan abstracción operacional al programador y constituyen una unidad fundamental en 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 un método para resolver nuevas operaciones.

4.1. Funciones y procedimientos

Dependiendo de su utilización (llamada) podemos distinguir dos casos:

- *Procedimientos*: encargados de resolver un problema computacional. Se les envia los datos necesarios y produce unos resultados que devuelve al lugar donde ha sido requerido:

```
int
main()
{
    int x = 8;
    int y = 4;

    ordenar(x, y);
    . . .
}
```

- *Funciones*: encargados de realizar un cálculo computacional y generar un resultado (normalmente calculado en función de los datos recibidos) utilizable directamente:

```
int
main()
{
    int x = 8;
    int y = 4;
    int z;

    z = calcular_menor(x, y);
```

```
        . . .
    }
```

4.2. Definición de subprogramas

Su definición podría ser como se indica a continuación:

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

int
calcular_menor(int a, int b)
{
    int menor;

    if (a < b) {
        menor = a;
    } else {
        menor = b;
    }

    return menor;
}
```

La definición de un subprograma comienza con una cabecera 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 y la declaración de sus parámetros.

El cuerpo del subprograma determina la secuencia de acciones necesarias para resolver el problema especificado. En el caso de una función, el valor que toma tras la llamada vendrá dado por el resultado de evaluar la expresión de la construcción `return`. Aunque C++ es más flexible, nosostros sólo permitiremos una única utilización del `return` y deberá ser al final del cuerpo de la función.

Nota: la función `calcular_menor` anterior también podría haber sido definida de la siguiente forma:

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

Normalmente la solución de un subproblema o la ejecución de una operación 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 traves de los parámetros o argumentos del subprograma.

4.3. Parámetros por valor y por referencia

- Llamaremos argumentos o *parámetros de entrada* a aquellos que se utilizan para recibir la información necesaria para realizar la computación. Por ejemplo los argumentos `a` y `b` de la

función `calcular_menor` anterior.

Los argumentos de entrada se reciben *por valor*, que significa que son valores que se copian desde el sitio de la llamada a los argumentos en el momento de la ejecución del subprograma.

Se declaran especificando el tipo y el identificador asociado.

- Llamaremos argumentos o *parámetros de salida* a aquellos que se utilizan para transferir información producida como parte de la computación/solución realizada por el subprograma. Ejemplo:

```
int
main()
{
    int cociente;
    int resto;

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

Su definición podría ser la siguiente:

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

Así, `dividendo` y `divisor` son argumentos de entrada y son pasados *"por valor"* como se vió anteriormente. Sin embargo, tanto `cociente` como `resto` son argumentos de salida (se utilizan para devolver valores al lugar de llamada), por lo que es necesario que se pasen *por referencia* que significa que ambos argumentos serán referencias a las variables que se hayan especificado en la llamada. Es decir, cualquier acción que se haga sobre ellas es equivalente a que se realice sobre las variables referenciadas.

Se declaran especificando el tipo, el símbolo "ampersand" (`&`) y el identificador asociado.

- Llamaremos argumentos o *parámetros de entrada/salida* a aquellos 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 argumentos `a` y `b` del procedimiento `ordenar` anterior.

  Los argumentos de entrada/salida se reciben *por referencia* y se declaran como se especificó anteriormente.

- También es posible recibir los *parámetros de entrada por referencia constante* de tal forma que el parámetro será una referencia al argumento 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.

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

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

En la llamada a un subprograma, una expresión de un tipo compatible es adecuada para un parámetro que se pase por valor, sin embargo, si el parámetro se pasa por referencia es obligatorio que sea pasada como parámetro actual una variable que concuerde en tipo. Si el paso es por referencia constante (`(const int& x)`) es posible que el parámetro actual sea una expresión de tipo compatible.

4.4. Subprogramas "en línea"

La llamada a un subprograma ocasiona cierta pérdida de tiempo de ejecución en el control de la misma (sobrecarga).

Hay situaciones en las que el subprograma es muy pequeño y nos interesa eliminar la sobrecarga que produce su llamada. 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.

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

4.5. Declaración de subprogramas

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 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 habra que especificar el tipo del valor devuelto (o void si es un procedimiento) seguido por el nombre y la declaración de los argumentos igual que en la definición del subprograma, pero sin definir el cuerpo del mismo. En lugar de ello se terminará la declaración con el caracter "punto y coma" (`;`).

```
int calcular_menor(int a, int b);  // declaracion de 'calcular_menor'

int
main()
{

    int x = 8;
    int y = 4;
    int z;

    z = calcular_menor(x, y);
    // ahora 'z' contine el calcular_menor numero de 'x' e 'y'
    ..............
}
```

Ejemplo de un programa que imprime los números primos menores que 100.

```
//- fichero: primos1.cpp ----------------------------------------------

#include <iostream>  // ver siguiente capitulo
```

```
using namespace std;  // ver siguiente capitulo

const int MAXIMO = 100;

inline bool
es_divisible(int x, int y)
{
   return ( x % y == 0 );
}

bool
es_primo(int x)
{
   int i;
   for (i = 2; ((i < x) && ( ! es_divisible(x, i))); ++i) {
       // vacio
   }
   return (i >= x);
}

void
escribir(int x)
{
   cout << x << " "; // ver siguiente capitulo
}

void
primos(int n)
{
   for (int i = 1; i < n; ++i) {
       if (es_primo(i)) {
           escribir(i);
       }
   }
   cout << endl; // ver siguiente capitulo
}

int
main()
{
   primos(MAXIMO);

   // return 0;
}
//- fin: primos1.cpp -----------------------------------------------
```

Capítulo 5

Entrada / Salida básica

Para poder realizar entrada/salida básica de datos es necesario incluir las declaraciones de tipos y operaciones que la realizan. Para ello habrá que incluir la siguiente línea al comienzo del fichero que vaya a realizar la entrada/salida:

```
#include <iostream>
```

Todas las definiciones y declaraciones de la biblioteca estándar se encuentran bajo el espacio de nombres `std` (ver capítulo 10), por lo que, para utilizarlos libremente, habrá que poner la siguiente declaración al comienzo del programa (Nota: ésto sólo es válido en ficheros de implementación y bajo ciertas condiciones):

```
using namespace std;
```

5.1. Salida

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

```
cout << contador ;
```

escribirá en la salida estándar el valor de contador. Tambien es posible escribir varios valores:

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

si queremos escribir un salto de línea:

```
cout << "Contador: " << contador << endl ;
```

5.2. Entrada

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

```
cin >> contador ;
```

incluso es posible leer varios valores simultáneamente:

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

Dicho operador de entrada se comporta de la siguiente forma: salta los espacios en blanco que hubiera al principio, y lee dicha entrada hasta que encuentre algún caracter no válido para dicha entrada (que no será leido y permanece hasta nueva entrada).

En caso de que durante la entrada surja alguna situación de error, dicha entrada se detine y se pondrá en estado erróneo.

Se consideran espacios en blanco los siguientes caracteres: el espacio en blanco, el tabulador, el retorno de carro y la nueva linea (`' '`, `'\t'`, `'\r'`, `'\n'`).

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

```
//- fichero: primos2.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) && ( ! es_divisible(x, i))); ++i) {
       // vacio
   }
   return (i >= x);
}

void
primos(int min, int max)
{
   cout << "Numeros 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);

   // return 0;
}
//- fin: primos2.cpp ------------------------------------------------
```

A las entidades `cin`, `cout` y `cerr` se las denomina flujos de entrada, de salida y de error, y pueden ser pasadas como parámetros (por referencia no constante) a subprogramas:

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

void
leer_int(istream& entrada, int& dato)
{
    entrada >> dato;
}
void
escribir_int(ostream& salida, int dato)
{
    salida << dato << endl;
}
int
main()
{
    int x;
    leer_int(cin, x);
    escribir_int(cout, x);
    escribir_int(cerr, x);
}
```

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".

6.1. Registros o estructuras

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 (campos) que lo componen. Por ejemplo, dado el tipo `Meses` definido en el Capítulo 2, podemos definir un tipo registro para almacenar fechas de la siguiente forma:

```
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;
```

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 trás el operador punto (`.`):

```
f_nac.dia = 18;
f_nac.mes = Octubre;
f_nac.anyo = 2001;
```

6.2. Agregados o "Arrays"

Un tipo *agregado* se especifica declarando el tipo base de los elementos que lo componen y el número de elementos de que consta dicha agregación. Así, por ejemplo, para declarar un agregado de fechas:

```
const int N_CITAS = 20;

typedef Fecha Citas[N_CITAS];
```

estamos definiendo un agregado de 20 elementos, cada uno del tipo `Fecha`, y cuyo identificativo es `Citas`.

Si declaramos una variable de dicho tipo:

```
Citas cit;
```

para acceder a un elemento concreto del agregado, especificaremos entre corchetes (`[` y `]`) la posición que ocupa, teniendo en cuenta que el primer elemento ocupa la posición 0 (cero) y el último elemento ocupa la posición `numero_de_elementos-1`. Por ejemplo `cit[0]` y `cit[N_CITAS-1]` aluden al primer y último elemento del agregado respectivamente.

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

```
cit[0].dia = 18;
cit[0].mes = Octubre;
cit[0].anyo = 2001;

for (int i = 0; i < N_CITAS; ++i) {
    cit[i].dia = 1;
    cit[i].mes = Enero;
    cit[i].anyo = 2002;
}
cit[N_CITAS] = { 1, Enero, 2002 };  // ERROR. Acceso fuera de los limites
```

6.3. Agregados multidimensionales

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:

```
const int N_PROVINCIAS = 8;
const int N_MESES = 12;

typedef float Temp_Meses[N_MESES];
typedef Temp_Meses Temp_Prov[N_PROVINCIAS];
```

o de la siguiente forma:

```
typedef float Temp_Prov[N_PROVINCIAS][N_MESES];
```

y la declaración

```
Temp_Prov tmp_prv;
```

que definiría una variable según el siguiente esquema:

```
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |
    +---+---+---+---+---+---+---+---+---+---+---+---+
  0 |   |   |   |   |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+---+---+---+---+
  1 |   |   |   |   |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+---+---+---+---+
```

```
  2 |@@@|@@@|@@@|@@@|@@@|@@@|@@@|@@@|@@@|@@@|@@@|@@@|
    +---+---+---+---+---+---+---+---+---+---+---+---+
  3 |   |   |   |   |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+---+---+---+---+
  4 |   |   |   |   |   |   |###|   |   |   |   |   |
    +---+---+---+---+---+---+---+---+---+---+---+---+
  5 |   |   |   |   |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+---+---+---+---+
  6 |   |   |   |   |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+---+---+---+---+
  7 |   |   |   |   |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+---+---+---+---+
```

donde

```
tmp_prv[2]
```

se refiere a la variable marcada con el símbolo `@` y

```
tmp_prv[4][6]
```

se refiere a la variable marcada con el símbolo `#`.
De forma similar se declaran agregados de múltiples dimensiones.
Nota: NO ESTÁ PERMITIDA LA ASIGNACIÓN DIRECTA DE AGREGADOS.

6.4. Cadenas de caracteres

Las *cadenas de caracteres* son un tipo de datos fundamental en cualquier lenguaje de programación. En C++ las cadenas de caracteres se representan como un agregado de caracteres del tamaño adecuado, donde la cadena de caracteres en sí está formada por los caracteres comprendidos entre el principio del agregado y un caracter 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:

```
const int MAX_CADENA = 10;
typedef char Cadena[MAX_CADENA];

Cadena nombre = "Pepe";
```

representa el siguiente esquema:

```
          0   1   2   3   4   5   6   7   8   9
        +---+---+---+---+---+---+---+---+---+---+
nombre  | P | e | p | e |\0 | ? | ? | ? | ? | ? |
        +---+---+---+---+---+---+---+---+---+---+
```

Es posible aplicar los operadores de entrada y salida de datos (`>>` y `<<`) a las cadenas de caracteres.
Notas:

- Para evitar sobrepasar el límite del agregado durante la lectura, es conveniente utilizar el manipulador `setw` (ver capítulo 9)

  ```
  cin >> setw(MAX_CADENA) >> nombre;
  ```

- **NO** está permitida la asignación directa de cadenas de caracteres, salvo en la inicialización, como se ha visto en el ejemplo.
- Las cadenas de caracteres pueden tratarse de forma más adecuada mediante el tipo `string`, perteneciente a la biblioteca estándar de C++ que se describe en el capítulo 7.

6.5. Parámetros de tipos compuestos

Respecto al *paso de estructuras como parámetros*, las estructuras se pueden pasar tanto por valor como por referencia. Sin embargo, debido a que el paso por valor exige la copia del valor de su lugar de origen a la nueva localización del parámetro, operación costosa tanto en tiempo de CPU como en consumo de memoria en el caso de tipos compuestos, utilizaremos siempre el *paso por referencia*, que es una operación muy eficiente tanto en tiempo como en memoria. En caso de ser parámetros de entrada, se utilizará el *paso por referencia constante* para garantizar que dicho parámetro no será modificado:

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

Con respecto al *paso de agregados 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 lo tanto, y teniendo en cuenta la discusión anterior sobre la eficiencia (en tiempo y memoria) del paso por referencia, los parámetros de entrada se realizarán mediante *paso por referencia constante* y los parámetros de salida y de entrada/salida se realizarán mediante *paso por referencia*. Ejemplo:

```
void
copiar(Cadena& destino, const Cadena& origen)
{
    int i;
    for (i = 0; origen[i] != '\0'; ++i) {
        destino[i] = origen[i];
    }
    destino[i] = '\0';
}
```

6.6. Parámetros de agregados 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 1 dimensión sin especificar su tamaño. Además, la información sobre el tamaño del agregado se pierde al pasarlo como agregado abierto, por lo que 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:

```
void
imprimir(int n,            // numero de elementos
         const int vct[])  // vector de enteros (paso por referencia constante)
{
    for (int i = 0; i < n; ++i) {
        cout << vct[i] << " " ;
    }
    cout << endl;
}
void
```

```
asignar_valores(int n,      // numero de elementos
                int vct[])  // vector de enteros (paso por referencia)
{
    for (int i = 0; i < n; ++i) {
        vct[i] = i;
    }
}

typedef int Numeros[30];
typedef int Datos[20];

int
main()
{
    Numeros nm;
    Datos dt;

    asignar_valores(30, nm);
    imprimir(30, nm);

    asignar_valores(20, dt);
    imprimir(20, dt);
}
```

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

6.7. Inicialización de variables de tipo compuesto

Es posible *inicializar* tanto las variables como las constantes en el momento de su definición. Para ello utilizamos el operador de asignación seguido por el valor (o valores entre llaves para tipos compuestos) con el que se inicializará dicha entidad. En el caso de cadenas de caracteres el valor a asignar podrá ser también una cadena de caracteres constante literal;

```
int
main()
{
    int x = 20;
    int y = x;
    Fecha f = { 20, Febrero, 1990 };
    Cadena nombre = "pepe";

    Temp_Prov tmp_prv = {
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 }
    }
}
```

Si en la declaración de una variable no se inicializa con ningún valor, dicha variable tendrá un valor inicial inespecificado. Con respecto a las definiciones de constantes, es obligatorio asignarles un valor en el momento de la definición.

Si en la inicialización no se especifican todos los valores de cada elemento, los elementos no especificados tomarán un valor cero.

6.8. Operaciones sobre variables de tipo compuesto

Con respecto a las *operaciones* que se pueden aplicar a las entidades de tipo compuesto, hemos visto que se pueden pasar como parámetros, pero siempre lo realizaremos "por referencia", y en el caso de parámetros de entrada serán "por referencia constante".

Además, en el caso de las estructuras o registros, tambien será posible asignar una entidad a otra utilizando el operador de asignación `=`. Sin embargo, esto no es posible para el caso de agregados, salvo en la inicialización del mismo.

Así mismo, no se aceptará que el tipo del valor devuelto por una función sea un tipo compuesto, en este caso preferiremos que el valor devuelto se realice como un parámetro de salida utilizando el "paso por referencia".

Tampoco están permitidos las comparaciones de ninguna clase entre entidades de tipo compuesto.

6.9. 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, todos 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:

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

struct Vehiculo {
    Tipo vh;
    union {
        float capacidad;// para el caso de tipo COCHE
        int cilindrada; // para el caso de tipo MOTOCICLETA
        int talla;      // para el caso de tipo BICICLETA
    };
};
```

y su utilización:

```
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.

6.10. Campos de bits

```
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
};
```

Capítulo 7

Biblioteca "string" de C++

La biblioteca `string` es un medio mas adecuado para tratar con cadenas de caracteres que los agregados de caracteres vistos en el capítulo anterior.

Para acceder a las facilidades proporcionadas por la biblioteca `string` de C++ debemos incluir el siguiente fichero de definición:

```
#include <string>
```

y por supuesto, se encuentra bajo el espacio de nombres `std`, por lo que habrá que usar la declaración:

```
using namespace std;
```

Un **string** es una secuencia de caracteres, y la biblioteca proporciona tanto el tipo para contenerlos como las operaciones a realizar con ella.

Declaramos un objeto de tipo `string` de la siguiente forma:

```
string s1;
```

Para saber el numero de caracteres (la longitud) del `string`:

```
s1.lenght() // devuelve la longitud del string.
s1.size()   // devuelve la longitud del string.
s1.empty()  // devuelve si el string esta vacio.
```

y accedemos a los caracteres individuales mediante indexación:

```
s1[i]    // accede al caracter que ocupa la posicion i (0 la primera)
s1.at(i) // igual que el anterior, pero comprueba el acceso

s1[0]               // es el primer caracter
s1[s1.lenght()-1] // es el ultimo caracter
```

donde `i` debe estar entre `0` y `s1.lenght()-1`. En caso de acceder mas alla de los límites del string, at(i) lanza una excepción `out_of_range` (cap. 11).

Se puede declarar un string inicialmente vacio, o darle valores iniciales:

```
string s1; // string vacio
string s2 = "pepeluis"; // string que contiene inicialmente "pepeluis"
string s3 = s2; // string que contiene inicialmente el valor de s2
string s4(s2, 2); // copia del valor de s2 desde [2] hasta el final
string s5(s2, 3, 4); // copia del valor de s2 desde [3] hasta [6]
```

```
char nombre[] = "juanlucas";
string s6(nombre, 4); // copia del valor de "juan"
string s7(nombre, 3, 4); // copia del valor de "nluc"

string s8(5, 'a'); // crea un string con 5 'aes' "aaaaa"
```

Otras operaciones que se pueden realizar:
asignación (de strings, agregados de caracteres y caracter):

```
s1 = s3;             | s1.assign(s3, pos, ncar);
s1 = "pepe";         | s1.assign("pepe", ncar);
s1[3] = 'a';         | s1.assign(5, 'a');
```

obtener la cadena como un agregado de caracteres terminado en \0:

```
char* cad = s1.c_str();
s1.copy(dest, ncar, pos = 0);
```

las siguientes operaciones de comparacion entre strings (y strings con agregados de caracteres):

```
== !=  > < >= <=
int c = s1.compare(s2);
int c = s1.compare(pos, ncar, s2);
int c = s1.compare(pos, ncar, s2, p2, n2);
```

añadir strings:

```
s1 += s2;
s1 += "pepe";
s1 += 'a';
s1.append(s2, pos, ncar);
```

insertar strings

```
s1.insert(3, s2);
s1.insert(3, s2, p2, n2);
s1.insert(0, "pepe");
s1.insert(0, "pepe", 2);
```

concatenar strings

```
string s9 = s1 + s2 + "pepe" + 'a';
```

buscar. En las siguientes busquedas, el patron a buscar puede ser un string, array de caracteres o un caracter. Si la posición indicada está fuera de los límites del string, entonces se lanza una excepción `out_of_range`.

```
string s = "accdcde";

unsigned i1 = s.find("cd", pos=0);                //i1=2; s[2]=='c'&&s[3]=='d'
unsigned i3 = s.find_first_of("cd", pos=0);       //i3=1; s[1]=='c'
unsigned i5 = s.find_first_not_of("cd", pos=0);   //i5=0; s[0]!='c'&&s[0]!='d'

unsigned i2 = s.rfind("cd", pos=npos);            //i2=4; s[4]=='c'&&s[5]=='d'
unsigned i4 = s.find_last_of("cd", pos=npos);     //i4=5; s[5]=='d'
unsigned i6 = s.find_last_not_of("cd", pos=npos); //i6=6; s[6]!='c'&&s[6]!='d'

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

borrar una porcion del string

```
string s = "accdcde";
s.erase(2, 3);              // s="acde"
s.erase(pos=0, ncar=npos);  // s=""
```

reemplazar una porcion del string

```
string s = "accdcde";
s.replace(2, 3, "juan");            // s = "acjuande"
s.replace(pos, ncar, s2, p2, n2);
```

obtener subcadenas de un string

```
string s1 = "accdcde";
string s2 = s1.substr(2, 3);             // s2 = "cdc"
string s2 = s1.substr(pos=0, ncar=npos);
```

realizar operaciones de entrada/salida

```
cout << s;                      // imprime el string
cin >> s;                       // salta espacios y lee hasta espacio
getline(cin, s);                // lee hasta salto de linea ('\n')
getline(cin, s, delimitador);   // lee hasta encontrar el delim. especificado
```

reservar capacidad

```
string::size_type mx = str.capacity();
str.reserve(tamano);
```

Capítulo 8

Memoria dinámica. Punteros

8.1. Declaración

El *tipo puntero* es el tipo predefinido que nos da acceso a posiciones de memoria, y más concretamente a la memoria dinámica. El tipo puntero se declara especificando el tipo de la entidad apuntada, un asterisco para indicar que es un **puntero** a un tipo, y el identificador del tipo:

```
typedef int* Ptr_int;

Ptr_int pi; // es una variable puntero a un entero
```

También se puede declarar una variable directamente:

```
int* pi; // es una variable puntero a un entero
```

sin embargo, la siguiente declaración no declara las variables como esperamos:

```
int* p1, p2; // p1 es un puntero a entero y p2 es un entero
```

la declaración debería ser:

```
int *p1, *p2;
```

o

```
int* p1;
int* p2;
```

Tambien se puede declarar un puntero *genérico* a cualquier tipo, aunque para su utilización es necesario después realizar una conversión al tipo correcto.

```
void* Puntero_generico;
```

8.2. Desreferenciación

Para acceder a una variable dinámica, se utiliza el operador unario asterisco (`*`) precediendo al nombre del puntero a través del cual es referenciada:

```
int resultado = *p1 + 2; // obtiene el valor de la vble apuntado por p1
```

La utilización de la "Memoria Dinámica" para entidades de tipo simple no es de gran utilidad, sin embargo, para el caso de entidades de tipos compuestos las posibilidades son muy diversas.

Por ejemplo, si definimos el tipo `Persona` de la siguiente forma:

```
struct Persona {
    string nombre;
    Fecha fec_nacimiento;
    string telefono;
    int edad;
};
```

Podemos definir el tipo puntero a persona de la siguiente forma:

```
typedef Persona* PPersona;
```

8.3. Memoria Dinámica

Podemos *solicitar memoria dinámica* con el operador `new` seguido por el tipo de la entidad a crear:

```
PPersona ppers = new Persona;
```

y acceder a los elementos:

```
(*ppers).nombre
(*ppers).fec_nacimiento
(*ppers).telefono
(*ppers).edad
```

o más comúnmente para el caso que el tipo apuntado sea una estructura:

```
ppers->nombre
ppers->fec_nacimiento
ppers->telefono
ppers->edad
```

Para *liberar la memoria* previamente solicitada:

```
delete ppers;
```

Cuando queramos especificar un puntero que no apunta a ninguna entidad utilizamos el `0` o `NULL`, de tal forma que el *puntero nulo* es el valor `0` o `NULL`.

Si se intenta liberar un puntero nulo, delete no hará nada, por lo tanto no es necesario comprobar si un puntero es o no nulo antes de liberarlo.

8.4. Estructuras autoreferenciadas

Una de las principales aplicaciones de la Memoria Dinámica es el uso de estructuras autoreferenciadas:

```
typedef struct Persona* PPersona;
struct Persona {
    string nombre;
    Fecha fec_nacimiento;
    string telefono;
    int edad;
    PPersona siguiente;  // autoreferencia
};
```

8.5. 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.

8.6. Paso de parámetros de variables de tipo puntero

Las entidades de tipo **puntero** se pueden pasar como *parámetros* "por valor" en el caso de parámetros de entrada, y "por referencia" en caso de parámetros de salida y de entrada/salida. Así mismo, tambien es posible especificar si la entidad apuntada es o no constante:

```
int* p1               // puntero a un entero
const int* p2         // puntero a un entero constante
int* const p3         // puntero constante a un entero
const int* const p4   // puntero constante a un entero constante

int* & r1             // referencia a un puntero a entero
const int* & r2       // referencia a un puntero a un entero constante
int* const & r3       // referencia constante a un puntero a un entero
const int* const & r4 // referencia constante a un puntero a un entero const
```

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 nó, dependiendo de su necesidad el número de elementos del agregado, como se indica a continuación:

```
int ad_enteros[]        // agregado dinamico de enteros
const int adc_enteros[] // agregado dinamico constante de enteros
```

Ejemplo:

```
#include <iostream>
typedef int* AD_Enteros;
```

```
void
leer(int n, int numeros[])
{
    cout << "Introduce " << n << " elementos " << endl;
    for (int i = 0; i < n; ++i) {
        cin >> numeros[i];
    }
}
float
media(int n, const int numeros[])
{
    float suma = 0.0;
    for (int i = 0; i < n; ++i) {
        suma += numeros[i];
    }
    return suma / float(n);
}
int
main()
{
    int nelm;
    AD_Enteros numeros;   // agregado dinamico

    cout << "Introduce el numero de elementos ";
    cin >> nelm;
    numeros = new int[nelm];
    leer(nelm, numeros);
    cout << "Media: " << media(nelm, numeros) << endl;
    delete [] numeros;
}
```

8.7. Operaciones sobre variables de tipo puntero

Las *operaciones* que se pueden aplicar a entidades de tipo puntero son principalmente la asignación de otros punteros del mismo tipo o 0 para indicar el puntero nulo, la asignación de memoria dinámica alojada con el operador `new`, liberar la memoria con el operador `delete`, acceder a la entidad apuntada con el operador `*`, a los campos de una estuctura apuntada con el operador `->` y acceder a un elemento de un agregado dinámico con `[]`. También es posible tanto la comparación de igualdad como de desigualdad. Además, también es posible su paso como parámetros tanto "por valor" como "por referencia".

8.8. Operador de indirección

En C++ un puntero, además de apuntar a variables alojadas en memoria dinámica, también puede apuntar a variables estáticas o 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 unario "ampersand" (`&`). Ejemplo:

```
int x = 3;
int* p = &x;

cout << *p << endl; // escribe 3 en pantalla
```

```
*p = 5;
cout << x << endl; // escribe 5 en pantalla
```

8.9. 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 funcion (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);
}
```

Capítulo 9

Entrada / Salida. Ampliación, control y ficheros

9.1. 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, los datos correspondientes a las teclas pulsadas se almacenan en una zona de memoria intermedia: *el "buffer" de entrada*. Cuando un programa realiza una operación de entrada de datos (`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 seguida por la tecla "enter"). Una vez obtenidos los datos asociados a las teclas pulsadas, se convertirán a un valor del tipo especificado por la operación de entrada, y dicho valor se asignará a la variable especificada.

Cuando se va a mostrar alguna información de salida

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

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

Como se vió anteriormente, para realizar operaciones de entrada/salida es necesario incluir las definiciones correspondientes en nuestro programa. para ello realizaremos la siguiente accion 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/salida, cómo controlar dichos flujos y comprobar su estado, así como la entrada/salida a ficheros.

9.2. Los flujos estándares

Los flujos existentes por defecto son `cout` como flujo de salida estandar, `cin` como flujo de entrada estandar, y `cerr` como flujo de salida de errores estandar.

Los operadores más simples para realizar la entrada y salida son:

- `<<` para la salida de datos sobre un flujo de salida. Sólo es aplicable a los tipos de datos predefinidos y a las cadenas de caracteres.

  ```
  cout << "El valor es " << i << endl;
  ```

  donde `endl` representa el "salto de línea".

- `>>` para la entrada de datos sobre un flujo de entrada. Sólo es aplicable a los tipos de datos predefinidos y a las cadenas de caracteres.

  ```
  cin >> valor1 >> valor2;
  ```

  la entrada se realiza saltando los caracteres en blanco y saltos de línea hasta encontrar datos. Si el dato leido se puede convertir al tipo de la variable que lo almacenará entonces la operación será correcta, en otro caso la operación será incorrecta.

9.3. Control de flujos. Estado

.

Para comprobar si un determinado flujo se encuentra en buen *estado* para seguir trabajando con él o no, se puede realizar una pregunta directamente sobre él:

```
if (cin) { // !cin.fail() |    if (! cin) { // cin.fail()
    // buen estado        |        // mal estado
    cin >> valor;         |        cerr << "Error en el flujo" << endl;
    ...                   |        ...
}                         |    }
```

Así mismo se pueden realizar las siguientes operaciones para comprobar el estado de un flujo:

```
cin >> valor;               |   cin >> valor;
if (cin.good()) {           |   if (cin.fail()) {
    // no error flags set   |       // ocurrio un error y prox op fallara
    // (cin.rdstate() == 0) |       // (cin.rdstate() & (badbit | failbit))!=0
} else {                    |   } else {
    // entrada erronea      |       ....
}                           |   }

if (cin.eof()) {            |   if (cin.bad()) {
    // Fin de fichero       |       // flujo en mal estado
//(cin.rdstate()&eofbit)!=0 |       //(cin.rdstate() & badbit) != 0
} else {                    |   } else {
    ....                    |       ....
}                           |   }

//eofbit:  Indica que una operacion de entrada alcanzo el
//         final de una secuencia de entrada
//badbit:  Indica perdida de integridad en una secuencia de entrada
//         o salida (tal como un error irrecuperable de un fichero)
//failbit: Indica que una operaci\'on de entrada fallo al leer los
//         caracteres esperados, o que una operacion de salida fallo
//         al generar los caracteres deseados
//goodbit: Indica que todo es correcto (goodbit == iostate(0))
```

Para recuperar el estado erroneo de un flujo a un estado correcto (además de otras acciones pertinentes):

```
[ios_base::goodbit | ios_base::failbit | ios_base::eofbit | ios:badbit]
cin.clear(); // pone el estado a ios_base::goodbit
cin.clear(flags); // pone el estado al indicado
ios_base::iostate e = cin.readstate();
cin.setstate(flags); // anade los flags especificados
```

Por ejemplo:

```
if (cin) {                          | if (cin) {
    int valor;                      |     int valor;
    cin >> valor;                   |     cin >> valor;
    if (cin) {                      |     if (cin) {
        cout << valor << endl;      |         cout << valor << endl;
    } else {                        |     } else {
        char entrada[128];          |         cin.clear();
        cin.clear();                |         cin.ignore(10000, '\n');
        cin >> setw(128)>> entrada; |         cout << "Entrada erronea" << endl;
        cout << "Entrada erronea: " |     }
             << entrada << endl;    | }
    }                               |
}                                   |
```

```
int
read_int()
{
    int number;

    cin >> number;
    while (!cin) {
        cout << "Error. Try again" << endl;
        cin.clear();
        cin.ignore(3000, '\n');
        cin >> number;
    }
    return number;
}
```

9.4. Entrada/Salida formateada

Las siguientes operaciones son útiles para alterar el *formato* en que se producen los datos del flujo:

```
cout.fill('#'); // caracter de relleno a '#'. Por defecto ' '.
cout.precision(4); // digitos significativos de numeros reales a 4
cout.width(5); // anchura de campo a 5 proxima salida
cin.width(5); // proxima lectura como maximo 4 caracteres

cout.flags(); // devuelve los flags actuales
cout.flags(ios_base::fmtflags); // pone los flag a los especificados (|)
cout.unsetf(flag); // desactiva el flag especificado
cout.setf(flag);   // activa el flag especificado
cout.setf(flag, mask);   // borra mask y activa el flag especificado
```

Los flags pueden ser:

```
 ios_base::skipws      // saltar los espacios en blanco (en entrada)

ios_base::adjustfield  // mascara de justificacion
 ios_base::left        // justificacion izquierda
 ios_base::right       // justificacion derecha
 ios_base::internal    // justificacion interna

ios_base::basefield    // mascara para indicar la base
 ios_base::dec         // entrada/salida en base decimal
 ios_base::oct         // entrada/salida en base octal
 ios_base::hex         // entrada/salida en base hexadecimal

ios_base::floatfield   // mascara para notacion cientifica
 ios_base::fixed       // no usar notacion cientifica (exponencial)
 ios_base::scientific  // usar notacion cientifica (exponencial)

ios_base::showbase     // mostrar prefijo que indique la base (0 0x)
ios_base::showpoint    // mostrar siempre el punto decimal
ios_base::showpos      // mostrar el signo positivo
ios_base::uppercase    // usar mayusculas en representacion numerica

ios_base::unitbuf      // forzar la salida despues de cada operacion
```

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

Ademas de las anteriores operaciones que afectan al flujo de forma general (salvo width), los siguientes *manipuladores* permiten afectar las operaciones de entrada/salida sólo en el momento que se estan efectúando:

```
cin >> ws;              // salta espacio en blanco
cin >> ws >> entrada;   // salta espacio en blanco antes de la entrada de datos
cout << ... << flush;   // fuerza la salida
cout << ... << endl;    // envia un salto de linea ('\n') y fuerza
cout << ... << ends;    // envia un fin de string ('\0')
```

para utilizar los siguientes manipuladores es necesario incluir el siguiente fichero de definición:

```
#include <iomanip>

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

cout << setw(5) << 234;                 // imprime '  234'
cout << setfill('#') << setw(5) << 234; // imprime '##234'

cin >> setw(10) >> nombre;              // lee como maximo 9 caracteres (+ '\0')

cout << dec << 27;                      // imprime 27
cout << hex << 27;                      // imprime 1b
```

```
cout << oct << 27;                    // imprime 33
cout << boolalpha << true;            // imprime true
```

9.5. Operaciones de salida

Además de las operaciones ya vistas, es posible aplicar las siguientes *operaciones a los flujos de salida*:

```
cout.put('#');                  // imprime un caracter
cout.write(char str[], int tam);// imprime 'tam' caracteres del 'string'

int n = cout.pcount();   // devuelve el numero de caracteres escritos
cout.flush();            // fuerza la salida del flujo
cout.sync();             // sincroniza el buffer

{
    ostream::sentry centinela(cout); // creacion del centinela en <<
    if (!centinela) {
        setstate(failbit);
        return(cout);
    }
    ...
}
```

9.6. 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 caracter:

```
int n = cin.get();  // lee un caracter de entrada o EOF
cin.get(char& c);       // lee un caracter de entrada
int n = cin.peek();     // devuelve el prox caracter (sin leerlo)
```

Para leer cadenas de caracteres (una línea cada vez):

```
cin.get(char str[], int tam);
    // lee un string como maximo hasta 'tam-1' o hasta salto de linea
    // anade el terminador '\0' al final de str
    // el caracter de salto de linea NO se elimina del flujo

cin.get(char str[], int tam, char delim);
    // lee un string como maximo hasta 'tam-1' o hasta que se lea 'delim'
    // anade el terminador '\0' al final de str
    // el caracter delimitador NO se elimina del flujo

cin.getline(char str[], int tam);
    // lee un string como maximo hasta 'tam-1' o hasta salto de linea
    // anade el terminador '\0' al final de str
    // el caracter de salto de linea SI se elimina del flujo

cin.getline(char str[], int tam, char delim);
    // lee un string como maximo hasta 'tam-1' o hasta que se lea 'delim'
    // anade el terminador '\0' al final de str
```

```
    // el caracter 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 salto de linea ('\n')
getline(cin, str, delimitador);  // lee hasta encontrar el delim. especificado
```

Notese que realizar una operacion `getline` despues de una operacion con `>>` puede tener complicaciones, ya que `>>` dejara los separadores en el buffer, que serán leidos por `getline`. Para evitar este problema (leera una cadena que sea distinta de la vacia):

```
void
leer_linea_no_vacia(istream& ent, string& linea)
{
    ent >> ws;           // salta los espacios en blanco y rc's
    getline(ent, linea); // leera la primera linea no vacia
}
```

Las siguientes operaciones también están disponibles.

```
int n = cin.gcount();   // devuelve el numero de caracteres leidos
cin.ignore([int n][, int delim]); // ignora cars hasta 'n' o 'delim'
cin.unget(); // devuelve al flujo el ultimo caracter leido
cin.putback(char ch); // devuelve al flujo el caracter 'ch'

{
    istream::sentry centinela(cin); // creacion 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();  // numero 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_base::iostate old_state = cin.exceptions();
cin.exceptions(ios_base::badbit | ios_base::failbit | ios_base::eofbit);
cout.exceptions(ios_base::badbit | ios_base::failbit | ios_base::eofbit);
```

la excepción lanzada es del tipo `ios_base::failure`.

9.7. 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 redireccion de los streams gracias a los bufferes (vease 17.5).

```
typedef basic_streambuf<ostream::char_type, ostream::traits_type> _x_streambuf_type;
_x_streambuf_type* pbuff1 = stream.rdbuf();       // obtener el ptr al buffer
_x_streambuf_type* pbuff2 = stream.rdbuf(pbuff1); // actualizar y obtener el ptr anterior
```

9.8. Ficheros

Las operaciones vistas anteriormente sobre los flujos de entrada y salida estandares se pueden aplicar tambien 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>
```

9.9. 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_base::in     // entrada
ios_base::out    // salida
ios_base::app    // posicionarse al final antes de cada escritura
ios_base::binary // fichero binario

ios_base::ate    // abrir y posicionarse al final
ios_base::trunc  // truncar el fichero a vacio
```

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 posicion de lectura
cin.seekg(pos);     // pone la posicion de lectura a 'pos'
cin.seekg(offset, ios_base::beg|ios_base::cur|ios_base::end); // pone la posicion de lectura
```

9.10. 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 posicion de escritura
cout.seekp(pos);                // pone la posicion de escritura a 'pos'
cout.seekp(offset, ios_base::beg|ios_base::cur|ios_base::end); // pone la posicion de escritu
```

9.11. 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_ficheros(const Cadena& salida, const Cadena& entrada)
{
    ifstream f_ent(entrada);
    if (!f_ent) {
        // error al abrir el fichero de entrada
    } else {
        ofstream f_sal(salida);
        if (! f_sal) {
            // error al abrir el fichero de salida
        } else {
            char ch;
            f_ent.get(ch);
            while (f_ent && f_sal) {   // mientras no haya un error o EOF
                f_sal.put(ch);
                f_ent.get(ch);
            }
            if (!(f_ent.eof() && f_sal.good())) {
                // error
            }
        }
    }

    f_sal.close();  // no es necesario
    f_ent.close();  // no es necesario
}
```

9.12. Ficheros de entrada/salida

Para *ficheros de entrada/salida* utilizamos el tipo:

```
fstream fich_ent_sal(char nombre[], ios_base::in | ios_base::out);
```

9.13. Flujo de entrada desde una cadena

Es posible también redirigir la *entrada desde un string de memoria*. Veamos como se realiza:

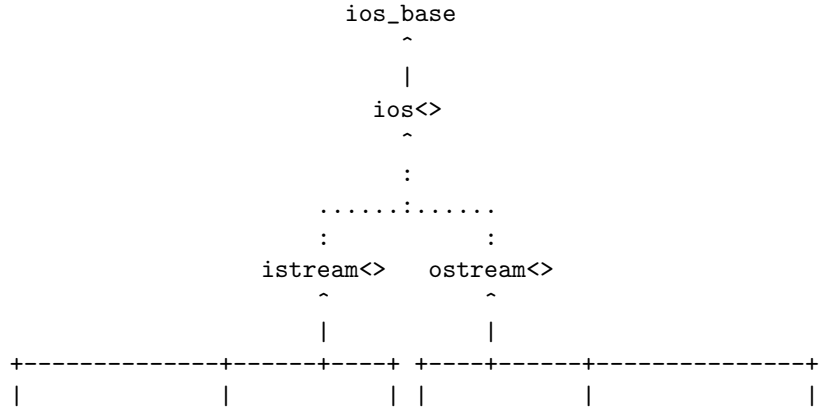
```
#include <sstream>

// 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")
```

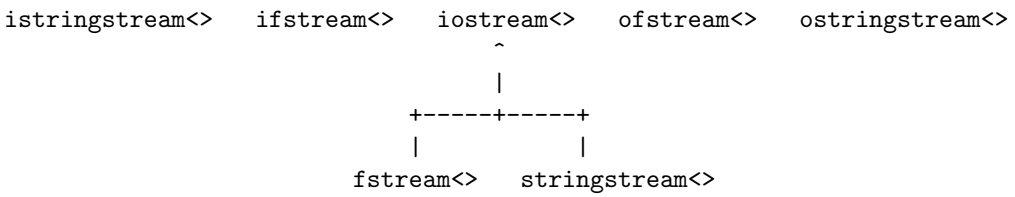
9.14. Flujo de salida a una cadena

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");
```

9.15. Jerarquía de clases de flujo estándar




istringstream<>
ifstream<>
iostream<>
ofstream<>
ostringstream<>
fstream<>
stringstream<>

Capítulo 10

Módulos

Cuando se desarrollan programas en el mundo real, normalmente no se utiliza un único fichero, sino que se distribuyen en varios módulos. Una primera ventaja de la existencia de módulos es el hecho de que si se modifica algo interno en un determinado módulo, sólo será necesario volver a compilar dicho modulo, y no todo el programa completo, lo que se convierte en una ventaja indudable en caso de programas grandes.

Otra ventaja de los módulos es que permiten aumentar la localidad 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.

10.1. Definición e implementación

Normalmente un módulo se compone de dos ficheros: uno donde aparece el código que resuelve un determinado problema o conjunto de problemas (parte privada), y un fichero que contiene las declaraciones de tipos, subprogramas y constantes globales que el módulo ofrece (parte pública).

Llamaremos la *implementación del módulo* al fichero que contiene la parte privada del módulo, y denominaremos *definición del módulo* al fichero que contiene la parte pública del mismo. A este fichero también se le denomina "fichero cabecera".

Así, un programa completo se compone de normalmente de varios módulos, cada uno con su fichero de definición y de implementación, y de un módulo principal donde reside la función principal `main`.

Para que un módulo pueda hacer uso de las utilidades que proporciona otro módulo deberá incluir su definición (su fichero cabecera), de tal forma que tenga acceso a las declaraciones públicas de este.

Normalmente los ficheros de implementación tendrán una extensión ".cpp", ".cxx", ".cc", ..., y los ficheros de definición tendrán una extensión ".hpp", ".hxx", ".hh", ".h", ...

Las declaraciones en los ficheros de definición vendrán dadas entre las siguientes definiciones de compilación condicional. Por ejemplo para el fichero "complejos.hpp"

```
//- fichero: complejos.hpp ----------------------------------------
#ifndef _complejos_     // guarda para evitar inclusion duplicada
#define _complejos_     //

struct Complejo {
   float real;
   float imag;
};

void Crear(Complejo& num, float real, float imag);
```

```
void sumar(Complejo& res, const Complejo& x, const Complejo& y);
void mult(Complejo& res, const Complejo& x, const Complejo& y);

#endif
//- fin: complejos.hpp -----------------------------------------------
```

Para que un módulo pueda utilizar las operaciones aportadas por otro módulo deberá incluir su definición de la siguiente forma:

```
#include "complejos.hpp"
```

así mismo, el módulo de implementación también deberá incluir al módulo de definición con objeto de obtener y contrastar las definiciones alli especificadas.

10.2. Espacios de nombre

Con objeto de evitar posibles problemas de identificadores repetidos cuando se incluyen diferentes definiciones de módulos, surgen los *espacios de nombres*, que es una forma de agrupar bajo una denominación un conjunto de declaraciones y definiciones, de tal forma que dicha denominación será necesaria para identificar y diferenciar cada entidad declarada.

Además, existen *espacios de nombre anónimos* para crear zonas privadas a cada módulo de implementación. Así, cualquier declaración/definición realizada dentro de un espacio de nombres anónimo será visible en el módulo de implementación donde se encuentre, pero no será visible (será privada) fuera del módulo.

```
//- fichero: complejos.hpp -------------------------------------------
#ifndef _complejos_    // guarda para evitar inclusion duplicada
#define _complejos_    //

/*
 * Declaraciones publicas del modulo
 */

namespace complejos {

 struct Complejo {
   float real;
   float imag;
 };

 void Crear(Complejo& num, float real, float imag);
 void sumar(Complejo& res, const Complejo& x, const Complejo& y);
 void mult(Complejo& res, const Complejo& x, const Complejo& y);

} // complejos
#endif
//- fin: complejos.hpp -----------------------------------------------

//- fichero: complejos.cpp -------------------------------------------
#include "complejos.hpp"
#include <cmath>

/*
 * Implementacion privada del modulo
 */
```

```
namespace {

 using namespace complejos;

 struct Polar {
   float rho;
   float theta;
 };

 inline float
 sq(float 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);
 }

}

/*
 * Implementacion correspondiente a la parte publica del modulo
 */

namespace complejos {

 void
 Crear(Complejo& num, float real, float 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);
 }

} // complejos
//- fin: complejos.cpp -----------------------------------------------
```

Para utilizar un módulo definido con "espacios de nombre" hay varias formas de hacerlo:

- Se puede crear un alias para un espacio de nombres para facilitar la cualificación:

  ```
  namespace NC = complejos;
  ```

- La primera forma consiste en utilizar cada identificador cualificado con el espacio de nombres al que pertenece: *(cualificación explícita)*:

  ```
  NC::Complejo num; // si ha sido definido el alias anterior
  complejos::Crear(num, 5.0, 6.0);
  ```

- La siguiente forma consiste en utilizar la directiva `using` para utilizar los identificadores sin cualificar*(cualificación imlícita)*:

  ```
  using complejos::Complejo;

  Complejo num;
  complejos::Crear(num, 5.0, 6.0);
  ```

  utilizo `Complejo` sin cualificar, ya que ha sido declarado mediante la directiva `using`. Sin embargo, debo cualificar `Crear` ya que no ha sido declarado mediante `using`.

- Utilizando la directiva `using namespace` hacemos disponible todos los identificadores de dicho espacio de nombres completo sin necesidad de cualificar.

  ```
  using namespace complejos;
  Complejo num;
  Crear(num, 5.0, 6.0);
  ```

Nota: no es adecuado aplicar la directiva using dentro de ficheros de definición (cabeceras) ya que traería muchos identificadores al espacio de nombres global. Es decir, en ficheros de definición se debería utilizar la cualificación explícita de nombres en vez de la directiva `using`.
Nota: Todas las operaciones de la biblioteca estandar se encuentran dentro del espacio de nombres `std`, y para utilizarlas es necesario aplicar lo anteriormente expuesto en cuanto a la directiva `using`. Ejemplo:

```
using namespace std;
```

Capítulo 11

Manejo de errores. Excepciones

Las excepciones surgen como una forma de manejar situaciones de error **excepcionales** en los programas, por lo tanto no deberían formar parte del flujo de ejecución normal de un programa.

Son muy adecuadas porque permiten diferenciar el código correspondiente a la resolución del problema del código encargado de manejar situaciones anómalas.

La sintaxis es la siguiente:

```
try {
    .....
    ... throw(error);   // lanza una excepcion
    .....
} catch (const tipo1 & var1) {
    .....
} catch (const tipo2 & var2) {
    .....
} catch ( ... ) {   // captura cualquier excepcion
    throw;          // relanza la misma excepcion
}
```

Cuando se lanza una excepción, el control de ejecución salta directamente al manejador que es capaz de capturarla según su tipo, destruyéndose todas las variables que han sido creadas en los ámbitos de los que se salga.

Si la ejecución de un subprograma puede lanzar una excepción al programa llamante, se puede indicar en la declaracion de la funcion indicando los tipos diferentes que se pueden lanzar separados por comas:

```
#include <iostream>

using namespace std;

struct Error_Division_Por_Cero {};

int
division(int dividendo, int divisor) throw(Error_Division_Por_Cero)
{
    int res;

    if (divisor == 0) {
        throw Error_Division_Por_Cero();
    } else {
        res = dividendo / divisor;
```

```
    }
    return res;
}

int
main()
{
    try {
        int x, y;

        cin >> x >> y;
        cout << division(x, y) << endl;
    } catch ( Error_Division_Por_Cero ) {
        cerr << "Division por cero" << endl;
    } catch ( ... ) {
        cerr << "Error inesperado" << endl;
    }
}
```

Excepciones estádares

Las excepciones 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 (basadas en el tipo `exception` anterior), que pueden, a su vez, ser base para nuevas excepciones definidas por el programador (cap. 15):

- `logic_error(str)` son, en teoría, predecibles y por lo tanto evitables mediante chequeos adecuados insertados en lugares estratégicos. (`#include <stdexcept>`)
  - `domain_error(str)`
  - `invalid_argument(str)`
  - `length_error(str)`
  - `out_of_range(str)`
- `runtime_error(str)` son errores impredecibles 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 especificacion de excepciones lanzadas por una función. (`#include <exception>`)
- `ios_base::failure()` lanzada en fallo en operaciones de Entrada/Salida. (`#include <iostream>`)

Nota: para ver una descripción de los requisitos que deben cumplir las clases para comportarse adecuadamente en un entorno de manejo de excepciones vease 13.2

En GNUC, se puede habilitar la característica siguiente para hacer que la terminación por causa de "excepción no capturada" muestre un mensaje de error:

```
#include <exception>

using namespace std;

int main()
{
#ifdef __GNUC__
    set_terminate (__gnu_cxx::__verbose_terminate_handler);
#endif
    ...
}
```

Capítulo 12

Sobrecarga de subprogramas y operadores

12.1. Sobrecarga de subprogramas

En C++ es posible definir diferentes subprogramas que posean el mismo nombre, siempre y cuando los parámetros asociados tengan diferentes tipos, y le sea posible al compilador discenir entre ellos a partir de los parámetros en la llamada. Nota: C++ no discierne entre un subprograma u otro a partir del tipo del valor devuelto.

```
#include <iostream>

using namespace std;

struct Complejo {
   double real;
   double imag;
};

void
imprimir(int x)
{
    cout << "entero: " << x << endl;
}
void
imprimir(double x)
{
    cout << "real: " << x << endl;
}
void
imprimir(char x)
{
    cout << "caracter: " << x << endl;
}
void
imprimir(const Complejo& cmp)
{
    cout << "complejo: (" << cmp.real << ", " << cmp.imag << ")" << endl;
}
void
```

```
imprimir(double real, double imag)
{
    cout << "complejo: (" << real << ", " << imag << ")" << endl;
}

int
main()
{
    Complejo nc;

    nc.real = 3.4;
    nc.imag = 2.3;
    imprimir(nc);
    imprimir(5.4, 7.8);
    imprimir(3);
    imprimir(5.6);
    imprimir('a');
    //return 0;
}
```

12.2. Sobrecarga de operadores

No es posible definir nuevos operadores ni cambiar la sintaxis, aridad, precedencia o asociatividad de los operadores existentes.

Se pueden definir el comportamiento de los siguientes operadores (siempre para tipos definidos por el programador):

`() [] -> ->*`	B	asoc. izq. a dcha.
`++ -- tipo()`	U	asoc. dcha. a izq.
`! ~ + - * &`	U	asoc. dcha. a izq.
`* / %`	B	asoc. izq. a dcha.
`+ -`	B	asoc. izq. a dcha.
`<< >>`	B	asoc. izq. a dcha.
`< <= > >=`	B	asoc. izq. a dcha.
`== !=`	B	asoc. izq. a dcha.
`&`	B	asoc. izq. a dcha.
`^`	B	asoc. izq. a dcha.
`|`	B	asoc. izq. a dcha.
`&&`	B	asoc. izq. a dcha.
`||`	B	asoc. izq. a dcha.
`= += -= *= /= %= &= ^= |= <<= >>=`	B	asoc. dcha. a izq.
`,`	B	asoc. izq. a dcha.
`new new[] delete delete[]`	U	asoc. izq. a dcha.

Los siguientes operadores no podrán ser definidos:

```
:: . .* ?: sizeof typeid
```

Los siguientes operadores sólo podrán definirse como funciones miembros de objetos (vease cap. 13):

```
operator=, operator[], operator(), operator->
```

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 sólo podrá definirse como funciones miembros de objetos (vease cap. 13). Ejemplo:

```
enum Dia {
    lunes, martes, miercoles, jueves, viernes, sabado, domingo
};

Dia&
operator++(Dia& d) // prefijo
{
    d = (d == domingo) ? lunes : Dia(d+1);
    return d;
}
Dia&
operator++(Dia& d, int)  // postfijo
{
    Dia ant = d;
    d = (d == domingo) ? lunes : Dia(d+1);
    return ant;
}

const int MAX_CAD = 30;
struct Cadena {
    int ncar;           // Maximo MAX_CAD-1
    char car[MAX_CAD];
};

bool
operator == (const Cadena& c1, const Cadena c2)
{
    bool res;

    if (c1.ncar != c2.ncar) {
        res = false;
    } else {
        int i;
        for (i = 0; (i < c1.ncar)&&(c1.car[i] == c2.car[i]); ++i) {
            // vacio
        }
        res = (i == c1.ncar);
    }
    return res;
}

Cadena&
copiar (Cadena& dest, const Cadena& orig) // operator=
{
    int i;
    dest.ncar = orig.ncar;
    for (i = 0; i < orig.ncar; ++i) {
        dest.car[i] = orig.car[i];
    }
    return dest;
```

```
}

ostream&
operator << (ostream& sal, const Cadena& cad)
{
    for (int i = 0; i < cad.ncar; ++i) {
        sal << cad.car[i];
    }
    return sal;
}

istream&
operator >> (istream& ent, Cadena& cad)
{
    ent >> setw(MAX_CAD) >> cad.car;
    cad.ncar = strlen(cad.car);
    return ent;
}
```

Capítulo 13

Tipos abstractos de datos

Los tipos abstractos de datos dan la posibilidad al programador de definir nuevos tipos que se comporten de igual manera que los tipos predefinidos, de tal forma que permiten definir su representación interna, la forma en la que se crean y se destruyen, como se asignan y se pasan como parámetros, las operaciones que se pueden aplicar y las conversiones de tipos aplicables. De esta forma se hace el lenguaje extensible.

Para definir un tipo abstracto de datos utilizaremos un nuevo tipo de nuestro lenguaje: la clase (`class`), mediante la cual C++ posibilita la definición de nuevos tipos de datos que tengan asociados una representación interna (atributos miembros) y unas operaciones (funciones miembros) que se le apliquen, además de poder definir la forma en la que se crean, destruyen, copian y asignan.

Vamos a realizar la definición e implementación del *tipo abstracto de datos* "Cadena". Lo definiremos dentro del espacio de nombres `bbl` con el que identificaremos nuestra biblioteca. Además, indicaremos las posibles situaciones de error mediante excepciones, para que puedan ser manejadas correctamente.

```
//- fichero: cadena.hpp -----------------------------------------------
#include <iostream>
#include <iomanip>

namespace bbl {

class Cadena {
private:

    typedef unsigned int uint;
    static const uint MAXIMO = 256;

    uint _longitud;
    char _car[MAXIMO];

public:
    class Fuera_de_Rango {};    // excepcion
    class Overflow {};          // excepcion

    ~Cadena() throw();          // destructor
    Cadena() throw();           // constructor
    Cadena(const Cadena& orig, uint inicio = 0, uint ncar = MAXIMO)
        throw (Fuera_de_Rango, Overflow);// ctor copia
    Cadena(const char orig[], uint inicio = 0, uint ncar = MAXIMO)
        throw (Fuera_de_Rango, Overflow); // ctor
```

```
    Cadena& operator=(const Cadena& orig) throw();  // asignacion
    Cadena& operator=(const char orig[]) throw (Overflow);  // asignacion

    const char& operator[] (uint i) const throw (Fuera_de_Rango);
    char& operator[] (uint i) throw (Fuera_de_Rango);

    uint longitud() const { return _longitud; };
    Cadena subcad(uint inicio = 0, uint ncar = MAXIMO) const;

    // conversion de tipo a natural
    operator uint() const { return longitud(); }

    Cadena& operator+=(const Cadena& cad)   throw (Overflow);
    Cadena& operator+=(const char cad[])    throw (Overflow);
    Cadena& operator+=(char car)            throw (Overflow);

    friend bool operator== (const Cadena& c1, const Cadena& c2);
    friend bool operator<  (const Cadena& c1, const Cadena& c2);
    friend bool operator<= (const Cadena& c1, const Cadena& c2);
    friend bool operator!= (const Cadena& c1, const Cadena& c2)
        { return ! (c1 == c2); }
    friend bool operator>  (const Cadena& c1, const Cadena& c2)
        { return ! (c1 <= c2); }
    friend bool operator>= (const Cadena& c1, const Cadena& c2)
        { return ! (c1 < c2); }

    friend Cadena operator+ (const Cadena& c1, const Cadena& c2);

    friend ostream& operator<< (ostream& s, const Cadena& cad);
    friend istream& operator>> (istream& s, Cadena& cad) throw(Overflow);
};

}// namespace bbl
//- fin: cadena.hpp ----------------------------------------------
```

Así, vemos en la declaración de nuestra clase `Cadena` que hay dos zonas principales, una etiquetada con la palabra reservada `private:` y otra con `public:`. Cada vez que se especifica una de dichas palabras reservadas, las declaraciones que la siguen adquieren un atributo de visibilidad dependiendo de la etiqueta especificada, de tal forma que las declaraciones privadas sólo podrán ser accedidas y utilizadas en las funciones miembros de la clase que se está definiendo y las declaraciones públicas podrán utilizarse por cualquier elemento que utilice alguna instancia de nuestra clase (objeto). A los datos que componen un objeto les llamaremos atributos (miembros en terminología C++).

Dentro de la zona privada definimos un nuevo tipo (`uint`) interno para simplificar la escritura del codigo. Así mismo definimos un atributo constante (`MAXIMO`) que será visible sólo en la implementación del TAD `Cadena`. Dicha constante es necesaria que se declare modificada con la palabra reservada `static`, ya que dicha modificación (tanto para constantes como para variables) indica que dicho atributo va a ser compartido por todas las instancias de la clase (objetos). Es decir, 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 de tipo `static` entonces dicho atributo será único y compartido por todos los objetos de la clase. Así, en nuestro caso de la constante estática `MAXIMO`, será única y compartida por todos los objetos.

A continuación se declaran dos atributos variables para almacenar la longitud de la cadena de un determinado objeto, y la otra para almacenar los caracteres que conforman la cadena.

Estos atributos, al no ser estáticos, serán replicados para cada objeto que se declare de la clase y almacenarán los diferentes valores de cada objeto.

En la zona pública, declaramos dos clases sin atributos ni funciones miembro. Son simplemente dos definiciones de tipos que serán útiles a la hora de indicar los posibles errores mediante excepciones. Posteriormente declaramos el *destructor* de la clase mediante el símbolo `~` seguido del identificador de la clase y una lista de parámetros vacía (`~Cadena();`). Este destructor será llamado automáticamente (y sin parámetros actuales) para una determinada instancia (objeto) de esta clase cuando el flujo de ejecución del programa salga del ámbito de visibilidad de dicho objeto. Nota: un destructor nunca deberá lanzar excepciones.

```
{
    ...
    {
        Cadena c1;  // construccion del objeto 'c1'
        ...         // utilizacion del objeto 'c1'
    }               // destruccion del objeto 'c1'
    ...
}
```

A continuación se declaran los *constructores* de la clase, especificando así las diferentes formas de crear un determinado objeto. De entre todos ellos, hay dos especiales: el constructor por defecto, que no toma ningún argumento y define como se creará un objeto de dicha clase si no se especifica ningún valor; y el constructor de copia que recibe como argumento un objeto de la misma clase y define como crear un nuevo objeto que sea una copia del que se recibe. Dicho constructor se utiliza en las inicializaciones, en el paso por valor y al devolver un objeto una función. Si el constructor de copia no se define, entonces se define uno automáticamente que realizará una copia individual de todos los atributos del objeto. Vemos también que a ciertos paráteros les hemos asignado un valor (`uint inicio = 0`), de tal forma que son parámetros opcionales, es decir, si no se especifican en la llamada, entonces tomarán el valor indicado. Así mismo, también hemos especificado las excepciones que puede lanzar la función, de tal forma que el que utilice dicho constructor esté advertido de las excepiones que lanza. Así mismo, también es útil para que el compilador compruebe la consistencia de dicha indicación y muestre un mensaje de error si se lanzan algún otro tipo de excepciones. Respecto a este último concepto, si no se especifica nada se considera que puede lanzar cualquier excepción, no dando así ninguna información para su utilización. Sin embargo, si queremos indicar que la función no lanza ninguna excepción, se especificará una lista de excepciones vacia. Así mismo, los constructores se utilizan también para conversiones de tipo (tanto implícitas como explícitas) al nuevo tipo definido.

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 formá implícita, haciendo necesario que sea llamado explícitamente. De hecho, es una forma de evitar conversiones de tipo implícitas, y además, de evitar el paso por valor y la devolución de un objeto por parte de una función si se aplica al constructor de copia.

A continuación declaramos el operador de asignación, para definir como se realiza ésta. Si no se define este constructor, entonces el sistema define uno que realiza la asignación de los atributos del objeto.

Posteriormente declaramos el operador de indexación `[]`, que será útil a la hora de acceder a los elementos individuales de la cadena. De hecho, este operador es fundamental a la hora de integrar el nuevo tipo definido dentro de aplicaciones que lo utilicen como si fuera un tipo predefinido. Hemos realizado dos declaraciones de este operador, una para cuando se utiliza sobre un objeto constante y la otra cuando se utiliza sobre uno no constante. Ambos devuelven una referencia a un elemento en vez de una copia de el, este hecho hace posible que el resultado de dicha función pueda aparecer en la parte izquierda de una asignación, por ejemplo `c1[3] = 'U';` además de poder utilizarse para obtener su valor como en `char c = c1[3];`

A continuación declaramos una función para saber el número de caracteres de la cadena. En esta declaración especificamos que es una *función miembro* `const`, es decir, la ejecución de dicha

función no modifica al objeto sobre el que se aplica. Además, hemos definido el comportamiento de la función en la propia declaración. Haciendo esto indicamos al compilador que queremos que dicha función se traduzca *"en línea"*.

Posteriormente definimos como se realiza la conversion del tipo que estamos definiendo a otro tipo. El tipo destino puede ser cualquier tipo, ya sea predefinido o definido por el usuario. A continuación se definen varias operaciones de concatenación y despues se definen una serie de funciones `friend`, que son funciones que no son *funciones miembros* del objeto que estamos definiendo, sino que son funciones que operan sobre los objetos que se especifican en los parámetros, pero que al ser `friend` de la clase, pueden acceder a la representación interna del objeto. Al fina de las declaraciones vemos las declaraciones de los operadores de entrada y salida.

Además de declarar los atributos normales, y de los atributos `static const` que hemos visto, es posible definir atributos con el modificador `mutable`, que indicará que dicho atributo puede ser modificado incluso por una función `const` que indica que no modifica el objeto. Así mismo, dicho atributo también podrá ser modificado en un objeto constante.

También es posible declarar atributos `static` que significa que el valor de dichos atributos es compartido por todos los objetos de la clase, de tal forma que si un objeto modifica su valor, dicho nuevo valor será compartido por todos los objetos. Cuando se declara un atributo `static`, se deberá definir en el módulo de implementación (cualificandolo con el nombre de la clase) y asignarle un valor inicial. Ejemplo:

```
//- fichero: x.hpp --------------------
class X {
    static int cnt;
    // ...
public:
    ~X() {};
    X();
    //...
};

//- fichero: x.cpp --------------------
#include "x.hpp"
int X::cnt = 0;  // definicion e inicializacion del atributo estatico

X::X()
    : //...  // lista de inicializacion
{
    ++cnt;
    //...    // inicializacion
}

// ...
```

También es posible definir una función miembro que sea `static`, en cuyo caso se podrá invocar directamente sobre la clase, es decir, no será necesario que se aplique a ningún objeto especifico.

```
//- fichero: cadena.cpp ------------------------------------------------
#include "cadena.hpp"

namespace {  // anonimo. zona privada al modulo

inline unsigned
minimo(unsigned x, unsigned y)
{
    return (x < y) ? x : y ;
```

```
}

unsigned
long_cstr(unsigned int max, const char cad[])
{
    unsigned i;
    for (i = 0; (i < max) && (cad[i] != '\0'); ++i) {
    }
    return i;
}

}// namespace

namespace bbl {

Cadena::~Cadena()
{}

Cadena::Cadena() throw()
    :
    _longitud(0)
{}

Cadena::Cadena(const Cadena& orig, uint inicio, uint ncar)
    throw (Fuera_de_Rango, Overflow)
    :
    _longitud(0)
{
    if ((inicio >= orig._longitud)||(ncar == 0)) {
        throw Fuera_de_Rango();
    }

    const uint ultimo = minimo(ncar, orig._longitud - inicio);

    if (ultimo >= MAXIMO) {
        throw Overflow();
    }

    for (_longitud = 0; _longitud < ultimo; ++_longitud) {
        _car[_longitud] = orig._car[inicio + _longitud];
    }
}

Cadena::Cadena(const char orig[], uint inicio, uint ncar)
    throw (Fuera_de_Rango, Overflow)
    :
    _longitud(0)
{
    const uint orig_lng = long_cstr(MAXIMO+1, orig);

    if ((inicio >= orig_lng)||(ncar == 0)) {
        throw Fuera_de_Rango();
    }
```

```
    const uint ultimo = minimo(ncar, orig_lng - inicio);

    if (ultimo >= MAXIMO) {
        throw Overflow();
    }

    for (_longitud = 0; _longitud < ultimo; ++_longitud) {
        _car[_longitud] = orig[inicio + _longitud];
    }
}

Cadena&
Cadena::operator= (const Cadena& orig)
    throw()
{
    if (this != &orig) {    // autoasignacion
        // destruir el valor antiguo y copiar el nuevo
        for (_longitud = 0; _longitud < orig._longitud; ++_longitud) {
            _car[_longitud] = orig._car[_longitud];
        }
    }
    return *this;
}
/*
 * Cadena&
 * Cadena::operator= (const Cadena& orig)
 * {
 *     Cadena aux(orig); // copia del valor
 *     this->swap(aux);  // asignacion del valor sin perdida del anterior
 *     return *this;     // destruccion automatica del valor anterior
 * }
 */
/*
 * Cadena&
 * Cadena::operator= (const Cadena& orig)
 * {
 *     Cadena(orig).swap(*this);
 *     return *this;
 * }
 */
/*
 * Referencia: Draft standard 2004/04/11
 *             section: 3.8 Object Lifetime (basic concepts 3-29)
 *
 * Nota en guru of the week [http://www.gotw.ca/publications/advice97.htm]
 *
 * These people mean well: the intent is to improve consistency by
 * implementing copy assignment in terms of copy construction, using
 * explicit destruction followed by placement new. On the surface, this
 * has the benefit of avoiding writing similar code in two places which
 * would then have to be maintained separately and could get out of sync
 * over time. This code even works in many cases.
 *
 * But do these programmers really "know what they write"? Probably not,
```

```
 * since there are subtle pitfalls here, some of them quite serious. Here
 * are just four: 1. This code slices objects whenever *this and other
 * are not the same type. 2. Consider how its correctness could depend on
 * whether operator=() was declared virtual in a base class or
 * not. 3. It's almost always less efficient because construction
 * involves more work than assignment. 4. It makes life a living hell for
 * the authors of derived classes. I'm sometimes tempted to post the
 * above code in the office kitchen with the caption: "Here be dragons."
 *
 * Is there anything wrong with the original goal of improving
 * consistency? No, but in this case "knowing what you write" would lead
 * the programmer to achieve consistency in a better way, for example by
 * having a common private member function that does the work and is
 * called by both copy construction and copy assignment. (Yes, I know
 * that an example like the above actually appears in the draft
 * standard. Pretend it's not there. That was to illustrate something
 * different, and shouldn't encourage imitation.)
 *
 * NO UTILIZAR -> NO FUNCIONA CORRECTAMENTE BAJO DETERMINADAS CIRCUNSTANCIAS
 *
 * Cadena&
 * Cadena::operator= (const Cadena& orig)
 * {
 *     if (this != &orig) {
 *         this->~Cadena();          // destruccion del valor anterior
 *         new (this) Cadena(orig);  // copia del nuevo valor
 *     }
 *     return *this;
 * }
 */

Cadena&
Cadena::operator= (const char orig[])
     throw (Overflow)
{
    // destruir el valor antiguo y copiar el nuevo
    for (_longitud = 0;
         (_longitud < MAXIMO)&&(orig[_longitud] != '\0');
         ++_longitud) {
        _car[_longitud] = orig[_longitud];
    }
    if (orig[_longitud] != '\0') {
        throw Overflow();
    }

    return *this;
}

const char&
Cadena::operator[] (uint i) const throw (Fuera_de_Rango)
{
    if (i >= _longitud) {
        throw Fuera_de_Rango();
    }
```

```
    return _car[i];
}

char&
Cadena::operator[] (uint i) throw (Fuera_de_Rango)
{
    if (i >= _longitud) {
        throw Fuera_de_Rango();
    }
    return _car[i];
}

Cadena
Cadena::subcad(uint inicio, uint ncar) const
{
    return Cadena(*this, inicio, ncar);
}

Cadena&
Cadena::operator+=(const Cadena& cad)
    throw (Overflow)
{
    if (_longitud + cad._longitud > MAXIMO) {
        throw Overflow();
    }
    for (uint i = 0; i < cad._longitud; ++i) {
        _car[_longitud] = cad._car[i];
        ++_longitud;
    }
    return *this;
}

Cadena&
Cadena::operator+=(const char cad[])
    throw (Overflow)
{
    uint i;
    for (i = 0; (_longitud < MAXIMO)&&(cad[i] != '\0'); ++i) {
        _car[_longitud] = cad[i];
        ++_longitud;
    }
    if (cad[i] != '\0') {
        throw Overflow();
    }
    return *this;
}

Cadena&
Cadena::operator+=(char car)
    throw (Overflow)
{
    if (_longitud + 1 > MAXIMO) {
        throw Overflow();
    }
```

```
    _car[_longitud] = car;
    ++_longitud;
    return *this;
}

bool
operator== (const Cadena& c1, const Cadena& c2)
{
    bool res;
    if (c1._longitud != c2._longitud) {
        res = false;
    } else {
        unsigned i;
        for (i = 0; (i < c1._longitud)&&(c1._car[i] == c2._car[i]); ++i) {
            // vacio
        }
        res = (i == c1._longitud);
    }
    return res;
}

bool
operator<= (const Cadena& c1, const Cadena& c2)
{
    unsigned i;
    for (i = 0; (i < c1._longitud)&&(i < c2._longitud)
             &&(c1._car[i] == c2._car[i]); ++i) {
        // vacio
    }
    return (i == c1._longitud);
}

bool
operator< (const Cadena& c1, const Cadena& c2)
{
    unsigned i;
    for (i = 0; (i < c1._longitud)&&(i < c2._longitud)
             &&(c1._car[i] == c2._car[i]); ++i) {
        // vacio
    }
    return (((i == c1._longitud)&&(i < c2._longitud))
            ||((i < c1._longitud)&&(i < c2._longitud)
               &&(c1._car[i] < c2._car[i])));
}

Cadena
operator+ (const Cadena& c1, const Cadena& c2)
{
    Cadena res = c1;
    res += c2;
    return res;
}

ostream&
```

```
operator<< (ostream& sal, const Cadena& cad)
{
    for (unsigned i = 0; i < cad._longitud; ++i) {
        sal << cad._car[i];
    }
    return sal;
}
istream&
operator >> (istream& ent, Cadena& cad)
     throw(Cadena::Overflow)
{
    ent >> setw(Cadena::MAXIMO) >> cad._car;
    cad._longitud = strlen(cad._car);
    return ent;
}

}// namespace bbl
//- fin: cadena.cpp -----------------------------------------------
```

En el módulo de implementación de nuestro tipo abstracto de datos comienza con un *espacio de nombres anónimo* con objeto de definir funciones privadas al módulo.

Posteriormente, dentro del espacio de nombres `bbl`, definimos las funciones miembro de la clase que estamos definiendo. Para ello, cada nombre de función se deberá cualificar con el nombre de la clase a la que pertenece. Así, definimos el destructor de la clase para que no haga nada. Definimos los constructores utilizando la lista de inicialización de tal forma que llamamos individualmente a los constructores de cada atributo de la clase (el orden debe coincidir con el orden en el que estan declarados), y es una lista de inicializaciones separadas por comas que comienza con el símbolo `:`. El cuerpo del constructor se encargará de terminar la construcción del objeto.

A continuación definimos como se realiza la operación de asignación. Al definir dicha operación habrá que tener en cuenta las siguientes consideraciones: Hay que comprobar y evitar que se produzca una autoasignación, ya que si no lo evitamos, podemos destruir el objeto antes de copiarlo. Posteriormente, y una vez que se ha comprobado que estamos asignando un objeto diferente, deberemos destruir el antiguo valor del objeto receptor de la asignación, para posteriormente copiar en nuevo objeto. esta función deberá devolver el objeto actual (`*this`).

Posteriormente implementamos cada función definida en la clase, así como las funciones `friend`. Éstas, al no ser funciones miembro, no se cualifican con el nombre de la clase, ya que no pertenecen a ella.

```
//- fichero: pr_cadena.cpp ----------------------------------------------
#include <iostream>

#include "cadena.hpp"

int
main()
{
    using namespace bbl;

    try {
        Cadena c1;
        const Cadena c2 = "pepeluis";
        Cadena c3 = c2;
        Cadena c4(c3);
        Cadena c5(c3 , 2);
        Cadena c6(c3 , 2, 3);
```

```
        Cadena c7("juanlucas", 3, 5);

        cout << "---------------------" << endl;
        cout << "|" << c1 << "|" << endl;
        cout << "|" << c2 << "|" << endl;
        cout << "|" << c3 << "|" << endl;
        cout << "|" << c4 << "|" << endl;
        cout << "|" << c5 << "|" << endl;
        cout << "|" << c6 << "|" << endl;
        cout << "|" << c7 << "|" << endl;
        cout << "---------------------" << endl;

        c1 = "juanluis";
        c3 = c1.subcad(2, 5);
        // c2 = c3; // ERROR

        cout << "|" << c1 << "|" << endl;
        cout << "|" << c2 << "|" << endl;
        cout << "|" << c3 << "|" << endl;
        cout << "---------------------" << endl;

        c1[5] = 'U';
        //c2[5] = 'U';  // ERROR
        cout << "|" << c1 << "|" << endl;
        cout << "|" << c2 << "|" << endl;
        cout << "|" << c1[5] << "|" << endl;
        cout << "|" << c2[5] << "|" << endl;
        cout << "---------------------" << endl;

        c4 += c1;
        cout << "|" << c4 << "|" << endl;
        cout << "|" << c4.longitud() << "|" << endl;
        cout << "|" << c4 + "pepe" << "|" << endl;

        cout << (c2 == c2) << endl;
        cout << (c2 == c4) << endl;

        Cadena c8(c4, 50);
        Cadena c9(c4, 5, 0);
        cout << c4[50] << endl;
        c4[50] = 'Z';
        for (int i = 0; i < 50; ++i) {
            c7 += "hasta overflow";
        }
    } catch (Cadena::Fuera_de_Rango) {
        cerr << "Excepcion: Indice fuera de rango" << endl;
    } catch (Cadena::Overflow) {
        cerr << "Excepcion: Overflow" << endl;
    } catch ( ... ) {
        cerr << "Excepcion inesperada" << endl;
    }
}
//- fin: pr_cadena.cpp -----------------------------------------------
```

En el programa de prueba vemos una utilización del *Tipo Abstracto de Datos* donde se comprue-

ba el normal funcionamiento de los objetos declarados de dicho tipo. Así, vemos como utilizar las operaciones definidas en la clase sobre los objetos. Los operadores se utilizan con la sintaxis esperada, pero las funciones miembro se utilizan a través de la notación punto como en `c1.subcad(2, 5)` y en `c4.longitud()`. Además, vemos también como se crean los objetos de la clase utilizando diferentes constructores y diferente sintaxis.

13.1. Métodos definidos automáticamente por el compilador

- El constructor por defecto (sin argumentos) será definido automaticamente por el compilador *si el programador no define ningún constructor*. El comportamiento predefinido consistirá en la llamada al contructor por defecto para cada atributo miembro de la clase. El constructor por defecto de los tipos predefinidos es la inicialización a cero.

- El constructor de copia se definirá automaticamente por el compilador en caso de que el programador no lo proporcione. El comportamiento predefinido consistirá en la llamada al contructor 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.

- 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.

- el destructor de la clase se definirá automaticamente 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.

13.2. 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 *"adquisición de recursos es inicialización"*.

- Antes de lanzar una excepción, se deberá asegurar de que cada operando 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 especial en que cuando lanza una excepción, no deja ningún objeto *"creado"* para destruirse posteriormente, por lo que debemos asegurarnos de liberar todos los recursos adquiridos durante la construcción fallida antes de lanzar la excepción (*"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 si alguna excepción es lanzada. (*"adquisición de recursos es inicialización"*).

13.3. 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
}
```

Capítulo 14

Programación Genérica. Plantillas

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:

```
template <typename Tipo>
inline Tipo
maximo(Tipo x, 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);
}
```

Estas definiciones genéricas, por regla general, se encontrarán en los ficheros de definición, y deberán ser incluidos por todos aquellos módulos que las instancien.

Veamos otro ejemplo de una clase genérica:

```
//-fichero: subrango.hpp ------------------------------------------------
#include <iostream>

namespace bbl {
template<typename Tipo, Tipo menor, Tipo mayor>
```

```
class 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>
class Subrango {
    typedef Tipo T_Base;

    T_Base _valor;
public:
    struct Fuera_de_Rango {
        T_Base val;
        Fuera_de_Rango(T_Base i) : val(i) {}
    };  // excepcion

    Subrango() throw();
    Subrango(T_Base i) throw(Fuera_de_Rango);
    Subrango& operator= (T_Base i) throw(Fuera_de_Rango);
    operator T_Base() throw();

    friend std::ostream&
    operator<< <>(std::ostream&, const Subrango<Tipo, menor, mayor>&);
};

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>
Subrango<Tipo,menor,mayor>::Subrango() throw()
    :
    _valor(menor)
{}

template<typename Tipo, Tipo menor, Tipo mayor>
Subrango<Tipo,menor,mayor>::Subrango(T_Base i) throw(Fuera_de_Rango)
{
    if ((i < menor) || (i > mayor)) {
        throw Fuera_de_Rango(i);
    } else {
        _valor = i;
    }
}

template<typename Tipo, Tipo menor, Tipo mayor>
Subrango<Tipo,menor,mayor>&
Subrango<Tipo,menor,mayor>::operator= (T_Base i) throw(Fuera_de_Rango)
{
    if ((i < menor) || (i > mayor)) {
        throw Fuera_de_Rango(i);
    } else {
```

```
        _valor = i;
    }
    return *this;   // return *this = Subrango(i);
}

template<typename Tipo, Tipo menor, Tipo mayor>
Subrango<Tipo,menor,mayor>::operator Tipo() throw()
{
    return _valor;
}

} // namespace bbl
//-fin: subrango.hpp -------------------------------------------------

//-fichero: prueba.cpp -----------------------------------------------
#include "subrango.hpp"
#include <iostream>

using namespace std;
using namespace bbl;

typedef Subrango<int,  3, 20> int_3_20;
typedef Subrango<int, 10, 15> int_10_15;

int
main()
{
    try {
        int_3_20 x;
        int_10_15 y;
        int z;

        x = 17;
        //x = 25;  // fuera de rango

        y = 12;
        //y = 20;  // fuera de rango

        z = x;

        x = x + 5;  // fuera de rango

        y = z;  // fuera de rango

        cout << x << endl;

    } catch (int_3_20::Fuera_de_Rango& fr) {
        cerr << "Fuera de Rango<int, 3, 20>: " << fr.val << endl;
    } catch (int_10_15::Fuera_de_Rango& fr) {
        cerr << "Fuera de Rango<int, 10, 15>: " << fr.val << endl;
    } catch ( ... ) {
        cerr << "Excepcion inesperada" << endl;
    }
}
//-fin: prueba.cpp ---------------------------------------------------
```

Veamos otro ejemplo de una definicion genérica (tomado de GCC 3.3) con especialización:

```
//-fichero: stl_hash_fun.hpp ------------------------------------------------
namespace __gnu_cxx {
    using std::size_t;

    // definicion de la clase generica vacia
    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 estrá definida para cualquier tipo, y será en tiempo de ejecución donde rompa si se utiliza sobre un tipo no valido. De esta forma, es en tiempo de compilación cuando falla. Otra ventaja es que puede ser pasada coomo parámetro a un “template”.

Capítulo 15

Programación orientada a objetos

objetos

La programación Orientada a Objetos en C++ se fundamenta en el concepto de clases visto en el capítulo de Tipos Abstractos de Datos (cap. 13) junto al concepto de herencia y a los mecanismos que la soportan (funciones miembro virtuales y clases base abstractas).

Veamos estos nuevos conceptos que junto con los vistos en el capítulo anterior respecto a las clases nos darán las herramientas que soporten el paradigma de la programación orientada a objetos en C++.

Veamos el siguiente ejemplo de una jerarquía de clases para representar objetos gráficos:

```
//- fichero: figuras.hpp -----------------------------------------------
#include <iostream>

class Posicion {
    int _x;
    int _y;
public:
    //~Posicion() {} // Definido Automaticamente por el compilador
    //Posicion(const Posicion& p) : _x(p._x), _y(p._y) {} // Definido Automatica
    //Posicion& operator=(const Posicion& p) { _x = p._x; _y = p._y; } // D.A.
    Posicion(int x = 0, int y = 0) : _x(x), _y(y) {}
    int get_x() const { return _x; }
    int get_y() const { return _y; }
};

class Figura {
protected:
    const char* _id;  // accesible por las clases derivadas
    Posicion _p;  // accesible por las clases derivadas
private:
    virtual void dibujar(ostream&) const = 0;  // virtual pura
public:
    virtual ~Figura()
            { cout << _id << " en Posicion(" << _p.get_x() << ", "
                   << _p.get_y() << ") destruido" << endl; }
    //Figura(const Figura& f) : _id(f._id), _p(f._p) {} // D.A.
    //Figura& operator= (const Figura& f) { _id = f._id; _p = f._p; } // D.A.
    Figura(const char* id, int x = 0, int y = 0) : _id(id), _p(x, y) {}
    virtual void mover(int x, int y) { _p = Posicion(x, y); }
    virtual Figura* clone() const = 0;  // virtual pura
```

```
    friend ostream& operator<<(ostream& sal, const Figura& fig)
        { fig.dibujar(sal); return sal; }
};

class Rectangulo : public Figura {
    int _base;
    int _altura;
    virtual void dibujar(ostream&) const;
public:
    //virtual ~Rectangulo() {}
    //Rectangulo(const Rectangulo& r)
    //                    : Figura(r), _base(r._base), _altura(r._altura) {}
    //Rectangulo& operator=(const Rectangulo& r)
    //         { Figura::operator=(r); _base = r._base; _altura = r._altura; }
    Rectangulo(int b, int a, int x=0, int y=0)
        : Figura((a==b)?"Cuadrado":"Rectangulo", x, y), _base(b), _altura(a) {}
    virtual Rectangulo* clone() const { return new Rectangulo(*this); }
};

class Cuadrado : public Rectangulo {
public:
    //virtual ~Cuadrado() {}
    //Cuadrado(const Cuadrado& c) : Rectangulo(c) {}
    //Cuadrado& operator=(const Cuadrado& c) { Rectangulo::operator=(c) {}
    Cuadrado(int b, int x=0, int y=0) : Rectangulo(b, b, x, y) {}
    virtual Cuadrado* clone() const { return new Cuadrado(*this); }
};

class Triangulo : public Figura {
    int _altura;
    virtual void dibujar(ostream&) const;
public:
    //virtual ~Triangulo() {}
    //Triangulo(const Triangulo& t)
    //                    : Figura(t), _altura(t._altura) {}
    //Triangulo& operator=(const Triangulo& t)
    //         { Figura::operator=(t); _altura = t._altura; }
    Triangulo(int a, int x=0, int y=0)
        : Figura("Triangulo", x, y), _altura(a) {}
    virtual Triangulo* clone() const { return new Triangulo(*this); }
};
//- fin: figuras.hpp ------------------------------------------------
```

Nuestra definición de clases comienza con la definición de la clase `Posicion` que nos servirá para indicar la posición de una determinada figura en la pantalla. Así, define las coordenadas `_x` e `_y` privadas a la clase, de tal forma que sólo el objeto podrá acceder a ellas. A continuación define los métodos públicos, de los cuales, si el programador no los define, el compilador automáticamente define los siguientes con el comportamiento indicado:

- El constructor por defecto (sin argumentos) será definido automaticamente por el compilador *si el programador no define ningún constructor*. El comportamiento predefinido consistirá en la llamada al contructor por defecto para las clases base y para cada atributo miembro de la clase. El constructor por defecto de los tipos predefinidos es la inicialización a cero.
- El constructor de copia se definirá automaticamente por el compilador en caso de que el programador no lo proporcione. El comportamiento predefinido consistirá en la llamada al

contructor de copia para las clases base y 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.

- 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 las clases base y 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.

- el destructor de la clase se definirá automaticamente por el compilador si no es definido por el programador, y su comportamiento predefinido será llamar a los destructores de las clases base y de los atributos miembros.

Por lo tanto, cuando ese es el comportamiento que queremos que tengan, no es necesario definirlos, ya que lo hace el propio compilador por nosotros, con lo que será mas fácil de mantener y cometeremos menos errores. En caso de necesitar otro comportamiento, deberemos definirlo nosotros. El comportamiento por defecto se encuentra definido simplemente como ejemplo.

A continuación se define el constructor de creación que recibe las coordenadas de la posición (en caso de que no se proporcionen se consideran los valores por defecto especificados). Su definición se realiza en la lista de inicialización inicializando los valores de los atributos `_x` e `_y` a los valores especificados (debe coincidir con el orden de la declaración).

Posteriormente definimos dos métodos para acceder a los valores de la coordenada. Son funciones miembro `const` ya que no modifican en sí el objeto.

A continuación definimos la clase `Figura` que hemos definido que sea una *clase base abstracta*. Es una "clase base" porque de ella derivarán otras clases que hereden sus características y proporciona un conjunto de métodos comúnes a todas ellas. Es "abtracta" porque hemos declarado algunos métodos *abstractos*, lo que le dá esa característica a toda la clase. Un método será abstracto cuando se declare con la siguiente sintaxís `= 0` (además de ser `virtual`). Cuando una clase base es abstracta, no se podrán definir objetos de dicha clase (ya que hay métodos que no se implementan), sino que será necesario que se definan clases derivadas de ella que definan el comportamiento de dichos métodos.

Los métodos virtuales son pieza clave en C++ para el soporte de la orientación a objetos, de forma tal que permiten que las clases derivadas redefinan el comportamiento de tales métodos, y que tras ser accedidos mediante el interfaz de la clase base, dicho comportamiento se vea reflejado correctamente. Es fundamental para el soporte del polimorfismo.

En la definición de la clase `Figura` hemos declarado dos miembros (`_id` y `_p`) como `protected`, con lo que indicamos que dichos atributos no son accesibles (son privados) por los usuarios de la clase, pero sin embargo son accesibles para las clases derivadas.

A continuación declaramos un método privado (sólo será accesible por métodos de la propia clase) virtual y abstracto (es decir, sin cuerpo). Al declararlo virtual estamos especificando que dicho método podrá ser definido por las clases derivadas, y que cuando se ejecuten dichos métodos a través del interfaz proporcionado por la clase base, dicha definición será visible. Además, al ser abstracta (en vez de definir su cuerpo, hemos especificado que no lo tiene con `= 0`) indicamos que las clases derivadas deben definir el comprotamiento de dicho método.

A continuación definimos el destructor de la clase. Cuando hay métodos virtuales en una clase, entonces el destructor siempre deberá ser virtual. Sin embargo, en este caso no es abstracto, sino que hemos definido el comportamiento que tendrá dicho destructor. Cuando se destruye un objeto, automáticamente se llaman a los destructores de su clase, y de todas las clases de las que hereda.

Posteriormente se comenta la definición del constructor de copia (que como es una copia de los campos se prefiere la que define automaticamente el compilador, aunque se muestra como sería si el programador la definiese). Igualmente sucede con el operador de asignación, por lo que está también comentado.

A continuación definimos el constructor, que inicializa los valores de sus atributos a través de la lista de inicialización, llamando a los constructores adecuados (en este caso al constructor de

la clase `Posicion`). Los constructores nunca son virtuales, ya que cada clase debe definir como se construye y no es un concepto redefinible.

Posteriormente definimos el método `mover`, que se define como virtual, pero definimos su comportamiento. De esta forma, las clases derivadas heredarán dicho comportamiento, pero también podrán redefinirlo para adaptar el comportamiento del método.

Para que el sistema de herencia funcione correctamente es necesario que cuando se pasa un objeto de una determinada clase a una función, éste se pase por referencia, o un puntero al objeto, es decir, el mecanismo de soporte del polimorfismo no es soportado por el paso "por valor" de los argumentos a funciones. Así mismo, tampoco una función debe devolver dicho objeto polimórfico, ni emplearse la asignación.

Se declara también el método `clone` como *"virtual pura"* de tal forma que cada objeto derivado sea capaz de copiarse a través de este método.

Después declaramos el operador de salida, y lo definimos de tal forma que llame al método (virtual) `dibujar`, de tal forma que funcione perfectamente dentro de la herencia, y llame a las funciones de redefinan dicho método.

A continuación definimos la clase `Rectangulo` para que sea una `Figura` con características especiales, es decir, definimos un `rectangulo` como una clase derivada de `Figura`. En el definimos como se crea, e inicializa el objeto base en la lista de inicialización. Vemos como se definirían tanto el constructor de copia como el operador de asignación.

Definimos de igual forma la clase `Cuadrado` y la clase `Triangulo`.

```
//- fichero: figuras.cpp ------------------------------------------------
#include "figuras.hpp"
#include <iostream>

void
Rectangulo::dibujar(ostream& sal) const
{
    sal << _id << " en Posicion("
        << _p.get_x() << ", " << _p.get_y() << ");" << endl;
    for (int i = 0; i < _altura; ++i) {
        for (int j = 0; j < _base; ++j) {
            sal << "*" ;
        }
        sal << endl;
    }
}

void
Triangulo::dibujar(ostream& sal) const
{
    sal << _id << " en Posicion("
        << _p.get_x() << ", " << _p.get_y() << ");" << endl;
    for (int i = 0; i < _altura; ++i) {
        for (int j = 0; j < _altura-i; ++j) {
            sal << " " ;
        }
        for (int j = 0; j < 2*i+1; ++j) {
            sal << "*" ;
        }
        sal << endl;
    }
}
//- fin: figuras.cpp ----------------------------------------------------
```

En el módulo de implementación definimos aquellos métodos que no han sido definidos en la propia definición de clase (*"en línea"*).

```
//- fichero: vector.hpp -----------------------------------------------
/*
 *  Vector que aloja punteros a elementos automaticamente.
 *  Requiere que el elemento base disponga del metodo 'clone'
 */
template <typename Tipo, unsigned TAMANO>
class Vector {
    unsigned _nelms;
    Tipo* _elm[TAMANO];

    Vector(const Vector&) {}; // prohibida la copia
    Vector& operator=(const Vector&) {}; // prohibida la asignacion
public:
    class Fuera_de_Rango{}; // excepcion
    class Lleno{};  // excepcion
    ~Vector() throw();
    Vector() throw() : _nelms(0) {}
    unsigned size() const throw() { return _nelms; }
    const Tipo& operator[](unsigned i) const throw(Fuera_de_Rango);
    Tipo& operator[](unsigned i) throw(Fuera_de_Rango);
    void anadir(const Tipo& x, unsigned i = TAMANO) throw(Lleno);
    void borrar(unsigned i) throw(Fuera_de_Rango);
};

template <typename Tipo, unsigned TAMANO>
Vector<Tipo,TAMANO>::~Vector() throw()
{
    for (unsigned i = 0; i < _nelms; ++i) {
        delete _elm[i];
    }
}

template <typename Tipo, unsigned TAMANO>
const Tipo&
Vector<Tipo,TAMANO>::operator[](unsigned i) const throw(Fuera_de_Rango)
{
    if (i >= _nelms) {
        throw Fuera_de_Rango();
    }
    return *_elm[i];
}

template <typename Tipo, unsigned TAMANO>
Tipo&
Vector<Tipo,TAMANO>::operator[](unsigned i) throw(Fuera_de_Rango)
{
    if (i >= _nelms) {
        throw Fuera_de_Rango();
    }
    return *_elm[i];
}
```

```
template <typename Tipo, unsigned TAMANO>
void
Vector<Tipo,TAMANO>::anadir(const Tipo& x, unsigned i) throw(Lleno)
{
    if (_nelms == TAMANO) {
        throw Lleno();
    }
    if (i >= _nelms) {
        _elm[_nelms] = x.clone();
    } else {
        for (unsigned j = _nelms; j > i; --j) {
            _elm[j] = _elm[j-1];
        }
        _elm[i] = x.clone();
    }
    ++_nelms;
}

template <typename Tipo, unsigned TAMANO>
void
Vector<Tipo,TAMANO>::borrar(unsigned i) throw(Fuera_de_Rango)
{
    if (i >= _nelms) {
        throw Fuera_de_Rango();
    }
    delete _elm[i];
    --_nelms;
    for (unsigned j = i; j < _nelms; ++j) {
        _elm[j] = _elm[j+1];
    }
}
//- fin: vector.hpp ----------------------------------------------
```

En este fichero hemos definido un *"contenedor genérico"*, es decir, una clase diseñada para almacenar objetos, y es genérica porque la hemos definido mediante una *plantilla* para que sirva como contenedor para diferentes tipos.

No se debe confundir programación genérica, como la mostrada en la definicón de la clase vector, con polimorfismo, que significa que si tenemos un contenedor de figuras, si un rectángulo es una figura entonces se podrá almacenar en dicho contenedor, si un triángulo es una figura, entonces también podrá ser almacenado en el mismo contenedor, y todos los objetos allí almacenados ofrecerán el mismo interfaz definido para una figura, pero mostrando el comportamiento propio del objeto mostrado.

En este caso, hemos definido un contenedor como un vector de elementos con un tamaño máximo, al que le podemos ir añadiendo elementos, eliminando elementos, y accediendo a ellos. Nuestro `Vector` añade elementos copiándolos de forma transparente a memoria dinámica (mediante el método `clone`), y liberándolos posteriormente cuando son destruidos.

```
//- fichero: prueba.cpp ------------------------------------------------
#include "vector.hpp"
#include "figuras.hpp"
#include <iostream>

enum Opciones {
    FIN,
    Crear_Rectangulo,
```

```
    Crear_Cuadrado,
    Crear_Triangulo,
    Mover_Figura,
    Destruir_Figura
};

Opciones
menu()
{
    int op;

    cout << "0.- Fin" << endl;
    cout << "1.- Crear Rectangulo" << endl;
    cout << "2.- Crear Cuadrado" << endl;
    cout << "3.- Crear Triangulo" << endl;
    cout << "4.- Mover Figura" << endl;
    cout << "5.- Destruir Figura" << endl;
    cout << endl;
    cout << "Opcion ? " ;
    do {
        cin >> op;
        while (! cin) {
            cin.clear();
            cin.ignore(3000, '\n');
            cin >> op;
        }
    } while ((op < FIN)||(op > Destruir_Figura));
    return Opciones(op);
}

typedef Vector<Figura, 32> v_fig_32;

void
dibujar_figuras(const v_fig_32& vect)
{
    for (unsigned i = 0; i < vect.size(); ++i) {
        cout << i << ".- " ;
        cout << vect[i] ; // vect[i].dibujar(cout);
    }
}

int
main()
{
    try {
        Opciones opcion;
        v_fig_32 vect;

        do {
            try {
                int base, altura, x, y;
                opcion = menu();
                switch (opcion) {
                case Crear_Rectangulo:
```

```
                    cout << "Introduce base y altura: " ;
                    cin >> base >> altura ;
                    cout << "Introduce x e y: " ;
                    cin >> x >> y ;
                    vect.anadir(Rectangulo(base, altura, x, y));
                    break;
                case Crear_Cuadrado:
                    cout << "Introduce lado: " ;
                    cin >> base ;
                    cout << "Introduce x e y: " ;
                    cin >> x >> y ;
                    vect.anadir(Cuadrado(base, x, y));
                    break;
                case Crear_Triangulo:
                    cout << "Introduce altura: " ;
                    cin >> altura ;
                    cout << "Introduce x e y: " ;
                    cin >> x >> y ;
                    vect.anadir(Triangulo(altura, x, y));
                    break;
                case Mover_Figura:
                    cout << "Introduce indice: " ;
                    cin >> base ;
                    cout << "Introduce x e y: " ;
                    cin >> x >> y ;
                    vect[base].mover(x, y);
                    break;
                case Destruir_Figura:
                    cout << "Introduce indice: " ;
                    cin >> x ;
                    vect.borrar(x);
                    break;
                default:
                    break;
                }

                dibujar_figuras(vect);
            } catch (v_fig_32::Fuera_de_Rango) {
                cerr << "Indice fuera de rango" << endl;
            } catch (v_fig_32::Lleno) {
                cerr << "Vector lleno" << endl;
            }
        } while (opcion != FIN);
    } catch ( ... ) {
        cerr << "Error inesperado" << endl;
    }
}
//- fin: prueba.cpp -------------------------------------------------
```

El programa de prueba consiste simplemente en un pequeño menú que nos será útil a la hora de crear objetos (rectángulos, cuadrados y triángulos), moverlos, destruirlos, dibujarlos, etc almacenándolos en un contenedor y manejándolos de forma polimórfica. Nótese la gran ventaja que supone éste diseño a la hora de añadir nuevas figuras, de tal forma que pueden añadirse sin prácticamente ninguna modificación.

15.1. Métodos estáticos y virtuales

Los siguientes son unos consejos sobre cuando y como hacer un método virtual o nó (Conversations: Virtually Yours, Jim Hyslop and Herb Sutter. C/C++ Users Journal):

- Un *destructor virtual* indica que la clase se ha diseñado para ser usada como base de un objeto polimórfico.
- Un *método virtual protegido* indica que las clases derivadas deberían (o incluso deben) invocar la implementación de este método de esta clase.
- Un *método virtual privado* indica que las clases derivadas pueden (o no) redefinir el método, pero no pueden invocar esta implementación.
- Un *método virtual publico* debe ser evitado donde sea posible

Herencia public, protected, private, virtual.

Capítulo 16

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.
- Proporciona información sobre aspectos del lenguaje definidos por la implementación, tal como el valor del mayor `float`.
- Proporciona funciones que no se pueden implementar de forma óptima en el propio lenguaje para cada sistema, tal como `sqrt` y `memmove`.
- Proporciona facilidades no primitivas sobre las cuales un programador se puede basar para portabilidad, tal como `list`, `map`, ordenaciones, entrada/salida, etc.
- Proporciona un marco de trabajo para extender las facilidades que proporciona, tales como convenciones, etc.
- Proporciona la base común para otras bibliotecas.

16.1. Características comunes

16.1.1. Ficheros

```
#include <vector>
#include <list>
#include <stack>
#include <queue>
#include <deque>
#include <set>
#include <map>
#include <iterator>

#include <utility>
#include <functional>
#include <algorithm>
```

16.1.2. Contenedores

```
vector<tipo_base> vec;  // vector de tamano 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 repeticion de clave
set<tipo_base> cnj;     // conjunto
multiset<tipo_base> mcnj;// conjunto con valores multiples
```

16.1.3. Tipos definidos

```
t::value_type       // tipo del elemento
t::allocator_type   // tipo del manejador de memoria
t::size_type        // tipo de subindices, 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 (solo contenedores asociativos)
t::mapped_type      // tipo del valor mapeado (solo cont asoc)
t::key_compare      // tipo de criterio de comparacion (solo cont asoc)
```

16.1.4. Iteradores

```
c.begin()     // apunta al primer elemento
c.end()       // apunta al (ultimo+1) elemento
c.rbegin()    // apunta al primer elemento (orden inverso)
c.rend()      // apunta al (ultimo+1) elemento (orden inverso)

bi = back_inserter(contenedor)       // insertador al final
fi = front_inserter(conenedor)       // insertador al principio
in = inserter(contenedor, iterador)  // insertador en posicion

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
```

16.1.5. Acceso

```
c.front() // primer elemento
c.back()  // ultimo elemento
```

```
v[i]      // elemento de la posicion 'i' (vector, deque y map)
v.at[i]   // elem de la posicion 'i' => out_of_range (vector y deque)
```

16.1.6. Operaciones de Pila y Cola

```
c.push_back(e)    // annade al final
c.pop_back()      // elimina el ultimo elemento
c.push_front(e)   // annade al principio (list y deque)
c.pop_front()     // elimina el primer elemento (list y deque)
```

16.1.7. Operaciones de Lista

```
c.insert(p, x)    // annade x antes de p
c.insert(p, n, x) // annade n copias de x antes de p
c.insert(p, f, l) // annade 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

// solo 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 posicion 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 segun 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
```

16.1.8. Operaciones

```
c.size()      // numero de elementos
c.empty()     // (c.size() == 0)
c.max_size()  // tamano del mayor posible contenedor
c.capacity()  // espacio alojado para el vector (solo vector)
c.reserve(n)  // reservar espacio para expansion (solo vector)
c.resize(n)   // cambiar el tam del cont (solo vector, list y deque)
c.swap(y)     // intercambiar
==            // igualdad
!=            // desigualdad
<             // menor
```

16.1.9. Constructores

```
cont()      // contenedor vacio
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)     // inicializacion igual a x
```

16.1.10. 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)
```

16.1.11. 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
```

16.1.12. 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				

16.1.13. Operaciones sobre Iteradores

	|Salida	|Entrada	|Avance	|Bidir	|Random
Leer		=*p	=*p	=*p	=*p
Acceso		->	->	->	-> []
Escribir	*p=		*p=	*p=	*p=
Iteracion	++	++	++	++ --	++ -- + - += -=
Comparac		== !=	== !=	== !=	== != < > >= <=

16.2. Contenedores

Los contenedores proporcionan un método estándar para almacenar y acceder a elementos, proporcionando diferentes características.

16.3. vector

Es una secuencia optimizada para el acceso aleatorio a los elementos. Proporciona, así mismo, iteradores aleatorios.

```
#include <vector>
```

- Construcción:

```
typedef vector<int> vect_int;

vect_int v1;  // vector de enteros de tamano inicial vacio
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 vector<int>  Fila;
typedef 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; // asignacion de los elementos de v2 a v1
v1.assign(it_ini, it_fin); // asignacion de los elementos [it_ini:it_fin[
v1.assign(50, val_inicial); // asignacion 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 ultimo elemento (debe existir)
v1.back() = 1;      // acceso al ultimo elemento (debe existir)
```

- Acceso aleatorio a elementos (sólo `vector` y `deque`):

```
cout << v1[i]; // acceso al elemento i-esimo (debe existir)
v1[i] = 3;     // acceso al elemento i-esimo (debe existir)

cout << v1.at(i); // acceso al elemento i-esimo (lanza out_of_range si no existe)
v1.at(i) = 3;     // acceso al elemento i-esimo (lanza out_of_range si no existe)
```

- Operaciones de Pila

```
v1.push_back(val); // annade val al final de v1 (crece)
v1.pop_back();     // elimina el ultimo elemento (decrece) (no devuelve nada)
```

- Operaciones de Lista

```
it = v1.insert(it_pos, val); // inserta val en posicion. dev it al elem (crece)
v1.insert(it_pos, n, val);   // inserta n copias de val en posicion (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)
```

- Número de elementos:

```
n = v1.size();       // numero de elementos de v1
bool b = v1.empty(); // (v1.size() == 0)
```

- Tamaño del contenedor:

```
n = v1.max_size(); // tamanno del mayor vector posible

// solo 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.

// solo vector
v1.reserve(n); // reservar n elementos (prealojados) sin inicializar valores
n = v1.capacity(); // numero de elementos reservados
```

- Otras operaciones:

```
v1 == v2 , v1 != v2 , v1 < v2 , v1 <= v2 , v1 > v2 , v1 >= v2
v1.swap(v2);
swap(v1 , v2);
```

16.4. list

Es una secuencia optimizada para la inserción y eliminación de elementos. Proporciona, así mismo, iteradores bidireccionales.

```
#include <list>
```

- Construcción:

```
typedef list<int> list_int;

list_int l1;  // lista de enteros de tamano inicial vacio
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(l4); // lista copia de l4
```

- 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; // asignacion de los elementos de l2 a l1
l1.assign(it_ini, it_fin); // asignacion de los elementos [it_ini:it_fin[
l1.assign(50, val_inicial); // asignacion 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 ultimo elemento (debe existir)
l1.back() = 1;      // acceso al ultimo elemento (debe existir)
```

- Operaciones de Pila

```
l1.push_back(val); // annade val al final de l1 (crece)
l1.pop_back();     // elimina el ultimo elemento (decrece) (no devuelve nada)
```

- Operaciones de Cola (solo `list` y `deque`)

```
l1.push_front(val); // annade 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 posicion. dev it al elem (crece)
l1.insert(it_pos, n, val);   // inserta n copias de val en posicion (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();       // numero de elementos de l1
bool b = l1.empty(); // (l1.size() == 0)
```

- Tamaño del contenedor:

```
n = l1.max_size(); // tamanno de la mayor lista posible

// solo 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 segun 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 segun 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 argumentos (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> {
    int _valor;
public:
    mayor_que(int val) : _valor(val) {} // constructor

    bool operator() (int elem) const { return (elem > _valor); }
};

int
main()
{
    list<int> l1;
    . . .
    l1.remove_if(mayor_que(5));
    l1.remove_if(mayor_que(3));
}
```

o también de la siguiente forma:

```
int
main()
{
    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()
{
    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<>())
}
```

16.5. 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 deque<int> deq_int;

deq_int d1;  // deque de enteros de tamano inicial vacio
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; // asignacion de los elementos de d2 a d1
d1.assign(it_ini, it_fin); // asignacion de los elementos [it_ini:it_fin[
d1.assign(50, val_inicial); // asignacion 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 ultimo elemento (debe existir)
d1.back() = 1;      // acceso al ultimo elemento (debe existir)
```

- Acceso aleatorio a elementos (sólo `vector` y `deque`):

```
cout << d1[i]; // acceso al elemento i-esimo (debe existir)
d1[i] = 3;     // acceso al elemento i-esimo (debe existir)

cout << d1.at(i); // acceso al elemento i-esimo (lanza out_of_range si no existe)
d1.at(i) = 3;     // acceso al elemento i-esimo (lanza out_of_range si no existe)
```

- Operaciones de Pila

```
d1.push_back(val); // annade val al final de d1 (crece)
d1.pop_back();     // elimina el ultimo elemento (decrece) (no devuelve nada)
```

- Operaciones de Cola (solo `list` y `deque`)

```
d1.push_front(val); // annade 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 posicion. dev it al elem (crece)
d1.insert(it_pos, n, val);   // inserta n copias de val en posicion (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();       // numero de elementos de d1
bool b = d1.empty(); // (d1.size() == 0)
```

- Tamaño del contenedor:

```
n = d1.max_size(); // tamanno del mayor deque posible

// solo 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);
```

16.6. stack

Proporciona el "Tipo Abstracto de Datos Pila", y se implementa sobre un `deque`. No proporciona iteradores.

```
#include <stack>

typedef 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
    }
}
```

16.7. queue

Proporciona el "Tipo Abstracto de Datos Cola", y se implementa sobre un `deque`. No proporciona iteradores.

```
#include <queue>

typedef 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
    }
}
```

16.8. priority-queue

Proporciona el "Tipo Abstracto de Datos Cola con Prioridad", y se implementa sobre un `vector`. No proporciona iteradores.

```
#include <queue>

//typedef priority_queue< int, vector<int>, less<int> > pcola_i;
typedef priority_queue<int> pcola_i; // ordenacion <

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
    }
}
```

16.9. map

Secuencia de pares `<clave, valor>` optimizada para el acceso rápido basado en clave. Clave única. Proporciona iteradores bidireccionales.

```
#include <map>

//typedef map< string, int, less<string> > map_str_int;
typedef map<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 numero 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
```

16.10. multimap

Secuencia de pares `<clave, valor>` optimizada para el acceso rápido basado en clave. Permite claves duplicadas. Proporciona iteradores bidireccionales.

```
#include <map>

//typedef multimap< string, int, less<string> > mmap_str_int;
typedef multimap<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 numero 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>
pair<clave, valor> p = make_pair(f, s);
p.first  // clave
p.second // valor
```

16.11. set

Como un `map` donde los valores no importan, solo aparecen las claves. Proporciona iteradores bidireccionales.

```
#include <set>

//typedef set< string, less<string> > set_str;
typedef set<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 numero 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>
pair<tipo1, tipo2> p = make_pair(f, s);
p.first
p.second
```

16.12. multiset

Como `set`, pero permite elementos duplicados. Proporciona iteradores bidireccionales.

```
#include <set>

//typedef multiset< string, less<string> > mset_str;
typedef multiset<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 numero 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>
pair<tipo1, tipo2> p = make_pair(f, s);
p.first
```

```
p.second
```

16.13. bitset

Un `bitset<N>` es un array de `N` bits.

```
#include <bitset>

typedef 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();   // 1111111111111101
m4.flip(3);  // 1111111111110101

m1 = ~(m4 << 3);

unsigned long val = m1.to_ulong();
string str = m1.to_string();

int n = m1.size();  // numero de bits
int n = m1.count(); // numero 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)

cout << m1 << endl;
```

16.14. Iteradores

Proporcionan el nexo de unión entre los contenedores y los algoritmos, proporcionando una visión abstracta de los datos de forma que un determinado algortimo 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>
```

16.15. directos

```
typedef 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) {
    cout << "Clave: " << it->first << " Valor: " << it->second << endl;
}
```

16.16. inversos

```
typedef 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.

16.17. inserters

"Inserters" producen un iterador que al ser utilizado para asignar objetos, alojan espacio para él en el contenedor, aumentando así su tamaño.

```
typedef back_insert_iterator<contenedor> biic;
typedef front_insert_iterator<contenedor> fiic;
typedef insert_iterator<contenedor> iic;

bit = back_inserter(contenedor);    // anade elementos al final del contenedor
fiit = front_inserter(contenedor);  // anade elementos al inicio del contenedor
iit = inserter(contenedor, it_pos); // anade elementos en posicion
```

16.18. 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 copia el contenido de un fichero a otro:

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>

using namespace std;

typedef istream_iterator<char> Iterador_Entrada;
typedef ostream_iterator<char> Iterador_Salida;

int
main()
{
    ifstream entrada("entrada.txt");
    ofstream salida("salida.txt");

    entrada.unsetf(ios_base::skipws);

    Iterador_Entrada ent(entrada);
    Iterador_Entrada fin_ent;

    Iterador_Salida sal(salida);

    copy(ent, fin_ent, sal);
}
```

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>

using namespace std;

typedef int Tipo_Base;

typedef vector<Tipo_Base> Contenedor;

typedef istream_iterator<Tipo_Base> Iterador_Entrada;
typedef ostream_iterator<Tipo_Base> Iterador_Salida;
```

```
void
cargar(Contenedor& cnt, const string& filename)
{
    ifstream entrada(filename.c_str());
    Iterador_Entrada ent (entrada);
    Iterador_Entrada fin_ent;

    cnt.assign(ent, fin_ent);
    //cnt.clear();
    //copy(ent, fin_ent, back_inserter(cnt));
}
void
salvar(const Contenedor& cnt, const string& filename)
{
    ofstream salida(filename.c_str());
    Iterador_Salida sal (salida, " ");

    copy(cnt.begin(), cnt.end(), sal);
}
int
main()
{
    Contenedor cnt;
    cargar(cnt, "entrada.txt");
    salvar(cnt, "salida.txt");
}
```

16.19. Operaciones sobre Iteradores

Operaciones sobre Iteradores

```
            |Salida |Entrada|Avance |Bidir  |Random
------------+-------+-------+-------+-------+----------------------
Leer        |       | =*p   | =*p   | =*p   | =*p
Acceso      |       | ->    | ->    | ->    | -> []
Escribir    | *p=   |       | *p=   | *p=   | *p=
Iteracion   | ++    | ++    | ++    | ++ -- | ++ -- + - += -=
Comparac    |       | == != | == != | == != | == != < > >= <=
------------+-------+-------+-------+-------+----------------------
```

16.20. Objetos Función y Predicados

```
#include <functional>

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 funcion => 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 > {
    const string _nm;
public:
    explicit persona_eq(const string& n) : _nm(n) {}
    bool operator() (const persona& p) const { return _nm == p.nombre; }
};

it = find_if(lc.begin(), lc.end(), persona_eq("pepe"));
```

Creacion de Objetos funcion a partir de existentes

- Binder: permite que una funcion 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("maria");
    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)
```

16.21. 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)
copy_if(f, l, d, pred1)       // copia [f:l) a d (que cumplen pred1)

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) peorcaso )
sort(f, l, cmp2)                // ordena [f:l) segun 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-esimo elemento en su posicion
nth_element(f, n , l, cmp2)     // pone el n-esimo elemento en su posicion
it = lower_bound(f, l, v)       // primer elemento igual a v
it = upper_bound(f, l, v)       // primer elemento mayor a v
par = equal_range()             // rango de elementos igual a v
b = binary_search(f, l , v)     // busqueda 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)
```

16.22. 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 perdidas 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.

16.23. Numericos

```
#include <>
```

16.24. Límites

```
#include <limits>
```

```
numeric_limits<char>::is_specialized = true;
numeric_limits<char>::digits = 7; // digitos 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; // numero de digitos (radix) en mantisa
numeric_limits<float>::digits10 = 6; // numero de digitos (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;
```

16.25. 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);
```

Capítulo 17

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.

17.1. Adquisición de recursos es Inicialización

`auto_ptr<>`
`SGuard`

17.2. Creacion y Copia virtual

En un entorno polimórfico es necesario a veces poder efectuar copias (duplicar/clonar) objetos de los cuales no conocemos realmente su tipo, así como crear objetos nuevos del mismo tipo que otro elemento. Para ello, dichos objetos deben proporcionar los métodos virtuales `clone` y `create` de la siguiente forma:

```
class Base {
public:
    virtual ~Base() throw() {}
    virtual Base* create() const =0;
    virtual Base* clone() const =0;
};// class Base
class Obj : public Base {
public:
    virtual ~Obj() throw() {}
    virtual Obj* create() const { return new Obj(); }
    virtual Obj* clone() const { return new Obj(*this); }
    // ...
};// class Obj
```

Para un objeto que no pertenezca a una jerarquía de clases que contengan declarados `clone` y `create`, es posible crear un nuevo objeto que si sea ambos, a través de la herencia múltiple:

```
class Objeto {
public:
    virtual ~Objeto() {}
    virtual Objeto* create() const =0;
```

```
    virtual Objeto* clone() const =0;
};// class Objeto
/*
 *  Annade a una clase ya definida nuevas operaciones virtuales,
 *  y le annade una Clase base comun
 */
template <typename Tipo>
class MkObj : public Tipo , public Objeto {
public:
    virtual ~MkObj() {}
    virtual MkObj* create() const { return new MkObj(); }
    virtual MkObj* clone() const { return new MkObj(*this); }
};// class MkObj
```

Por ejemplo, podemos crear un nuevo objeto `Elemento` que sea a su vez un `Elem` y un `Objeto` utilizando para ello `MkObj`:

```
typedef MkObj<Elem> Elemento;
```

Tambien existe la posibilidad de llamar a una función que llame al método `clone` de un objeto, o lo cree directamente si dicho método no existe. Por defecto supone que todos los objetos tienen el método `clone`, y en caso de no existir para una determinada clase, el compilador emitirá un mensaje de error y el programador deberá especializar un determinado *"tag"* para indicarle a la funcion que dicha clase no tiene el método `clone`.

```
struct Has_Clone_Tag {};
struct Has_No_Clone_Tag {};
// Por defecto, las clases tienen el metodo 'clone'
// si dicho metodo no existe para una clase, el compilador
// avisara, y el programador debera especializar para dicha clase
template<typename Tipo>struct  Clone_Tag { typedef Has_Clone_Tag Tag; };
// Especializacion para clases que no tienen el metodo 'clone'
//template<>struct  Clone_Tag<Tipo> { typedef Has_No_Clone_Tag Tag; };
template <typename Tipo>
inline Tipo* clone(const Tipo* p, const Has_Clone_Tag)
{
    return (p == NULL) ? NULL : p->clone();
}
template <typename Tipo>
inline Tipo* clone(const Tipo* p, const Has_No_Clone_Tag)
{
    return (p == NULL) ? NULL : new Tipo(*p);
}
template <typename Tipo>
inline Tipo* clone(const Tipo* p)
{
    return clone(p,  Clone_Tag<Tipo>::Tag());
}
```

Un ejemplo de su utilización podría ser:

```
class Elem {
    int val;
public:
    Elem() : val() {}
    int get_val() const { return val; }
    void set_val(int v) { val = v; }
```

```
};// class Elem
template<>struct  Clone_Tag<Elem> { typedef Has_No_Clone_Tag Tag; };

int main()
{
    Elem e;
    Elem* pe = clone(&e);
}
```

17.3. Ocultar la implementación

Aunque C++ no soporta directamente tipos opacos, dicha característica se puede simular muy facílmente 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 {
        class ElemImpl;
        ElemImpl* pimpl;
    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);
    };// 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 {
        int val;
    public:
        ~ElemImpl() throw() { cout << "destruccion 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; }
    };// 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)
    {
        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 ------------------------------------------------------------------
```

- 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 {
        typedef elem::Elem _ClaseBase;
        int val;
    public:
        virtual ElemImpl* clone() const { return new ElemImpl(*this); }
        virtual ~ElemImpl() throw()
            { cout << "destruccion 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; }
    };// class ElemImpl
}
namespace elem {
    Elem* Elem::create() { return new ElemImpl(); }
    Elem::~Elem() throw() {}
}// namespace elem
//-- fin -----------------------------------------------------------------
```

17.4. 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
 *
 *      Definicion de tipos
 *
 *          typedef Matriz<int> Mat_Int;
 *
 *      Definicion de Variables
 *
 *          Mat_Int mat(NFIL, NCOL);
 *
 *      Definicion de Constantes
 *
 *          const Mat_Int aux2(mat);
 *
 *      Operaciones
 *
 *          aux1 = mat;                     // asignacion
 *          print(mat);                     // paso de parametros
 *          unsigned nfil = mat.getnfil();  // numero de filas
 *          unsigned ncol = mat.getncol();  // numero de columnas
 *          unsigned nelm = mat.getnelm();  // numero de elementos almacenados
 *          mat(1U, 3U) = 4;                // asignacion 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 {
        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(OutOfRange);
        Tipo_Ref_Elm operator() (unsigned f, unsigned c)
            throw(OutOfRange);
    };// class Matriz
//-- RefC_Elm -----------------------------------
    template <typename Tipo>
    class RefC_Elm {
        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 {
        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(OutOfRange)
    {
        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(OutOfRange)
    {
        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 << "Numero de filas reales: " << mat.getnfilalm() << endl;
    cout << "Numero 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 compilacion

            // Fuera de Rango
            mat(NFIL, NCOL) = 5;
        } catch ( OutOfRange ) {
            cerr << "excepcion fuera de rango" << endl;
        } catch ( ... ) {
            cerr << "excepcion inesperada" << endl;
        }
    }
    //-- fin ------------------------------------------------------------------
```

17.5. 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 {
        std::ostringstream  str_salida;
        std::ostream*       output_stream;
        std::streambuf*     output_streambuf_ptr;
    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();
        }

    };// 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 << "Informacion " << 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 ------------------------------------------------------------------
```

17.6. Eventos

registrando listener que heredan de una base abstracta listener

17.7. Restricciones en programación genérica

En programación genérica puede ser conveniente especificar un conjunto de restricciones sobre los tipos genéricos que pueden ser intanciados 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 utilizacion:

```
#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,float>();
    Can_compare<X,B>();
    Can_multiply<int,float>();
    Can_multiply<int,float,double>();
    Can_multiply<B,X>();

    Can_copy<D*,B*>();
    Can_copy<D,B*>();
    Can_copy<int,B*>();
}
```

Capítulo 18

Gestión Dinámica de Memoria

18.1. 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 contructor: `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 [tambien 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 estandar puede ser como se indica a continuación:

```
#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);
}
```

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);
```

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 {
    int* ptr;
public:
    X();
    ...
};
X::X()
  : ptr(0)
{
    auto_ptr<int> paux(new int);
    // utilizacion de paux. zona posible de excepciones
    // *paux    paux->      paux.get()    paux.reset(p)
    ptr = paux.release();
}
```

Apéndice A

Precedencia de Operadores en C

Precedencia de Operadores y Asociatividad	
Operador	Asociatividad
() [] -> .	izq. a dcha.
! ~ ++ -- - (tipo) * & sizeof	[[[dcha. a izq.]]]
* / %	izq. a dcha.
+ -	izq. a dcha.
<< >>	izq. a dcha.
< <= > >=	izq. a dcha.
== !=	izq. a dcha.
&	izq. a dcha.
^	izq. a dcha.
|	izq. a dcha.
&&	izq. a dcha.
||	izq. a dcha.
?:	[[[dcha. a izq.]]]
= += -= *= /= %= &= ^= |= <<= >>=	[[[dcha. a izq.]]]
,	izq. a dcha.

Orden de Evaluacion
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 funcion no esta especificado.

Constantes Caracter:
[\n=NL] [\t=TAB] [\v=VTAB] [\b=BS] [\r=RC] [\f=FF] [\a=BEL]

Apéndice B

Precedencia de Operadores en C++

```
================================================================================
nombre_clase::miembro           Resolucion de ambito
nombre_esp_nombres::miembro     Resolucion de ambito
::nombre                        Ambito global
::nombre_calificado             Ambito global
--------------------------------------------------------------------------------
objeto.miembro                  Seleccion de miembro
puntero->miembro                Seleccion de miembro
puntero[expr]                   Indexacion
expr(list_expr)                 Llamada a funcion
tipo(list_expr)                 Construccion de valor
valor_i++                       Post-incremento
valor_i--                       Post-decremento
typeid(tipo)                    Identificacion de tipo
typeid(expr)                    Identificacion de tipo en tiempo de ejecucion
dynamic_cast<tipo>(expr)        Conversion en TEjecucion con verificacion
static_cast<tipo>(expr)         Conversion en TCompilacion con verificacion
reinterpret_cast<tipo>(expr)    Conversion sin verificacion
const_cast<tipo>(expr)          Conversion const
--------------------------------------------------------------------------------
sizeof expr                     Tama~no del objeto
sizeof(tipo)                    Tama~no del tipo
++valor_i                       Pre-incremento
--valor_i                       Pre-decremento
~expr                           Complemento
!expr                           Negacion logica
-expr                           Menos unario
+expr                           Mas unario
&valor_i                        Direccion de
*expr                           Desreferencia
new tipo                        Creacion (asignacion de memoria)
new tipo(list_expr)             Creacion (asignacion de memoria e iniciacion)
new tipo [expr]                 Creacion de array (asignacion de memoria)
new (list_expr) tipo            Creacion (emplazamiento)
new (list_expr) tipo(list_expr) Creacion (emplazamiento e inciacion)
delete puntero                  Destruccion (liberacion de memoria)
delete [] puntero               Destruccion de un array
```

```
(tipo) expr                     Conversion de tipo (obsoleto)
-------------------------------------------------------------------------------
objeto.*puntero_a_miembro       Seleccion de miembro
puntero->*puntero_a_miembro     Seleccion de miembro
-------------------------------------------------------------------------------
expr * expr                     Multiplicacion
expr / expr                     Division
expr % expr                     Modulo
-------------------------------------------------------------------------------
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              Expresion condicional
-------------------------------------------------------------------------------
valor_i = expr                  Asignacion simple
valor_i *= expr                 Multiplicacion y asignacion
valor_i /= expr                 Division y asignacion
valor_i %= expr                 Modulo y asignacion
valor_i += expr                 Suma y asignacion
valor_i -= expr                 Resta y asignacion
valor_i <<= expr                Despl izq y asignacion
valor_i >>= expr                Despl dch y asignacion
valor_i &= expr                 AND bits y asignacion
valor_i ^= expr                 XOR bits y asignacion
valor_i |= expr                 OR bits y asignacion
-------------------------------------------------------------------------------
throw expr                      Lanzar una excepcion
-------------------------------------------------------------------------------
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 asignacion y el
    operador condicional son asociativos por la derecha.
    Todos los demas son asociativos por la izquierda.
-------------------------------------------------------------------------------
    Sacado de "El Lenguaje de Programacion 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 estandar de ANSI-C y C++.

C.1. 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);  // digito decimal
bool isgraph(char ch);  // caracteres imprimibles excepto espacio
bool islower(char ch);  // letra minuscula
bool isprint(char ch);  // caracteres imprimibles incluyendo espacio
bool ispunct(char ch);  // carac. impr. excepto espacio, letra o digito
bool isspace(char ch);  // esp, \r, \n, \t, \v, \f
bool isupper(char ch);  // letra mayuscula
bool isxdigit(char ch); // digito hexadecimal

char tolower(char ch);  // convierte ch a minuscula
char toupper(char ch);  // convierte ch a mayuscula
```

C.2. cstring

```
#include <cstring>

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 maximo 'n' caracteres

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 maximo 'n' caracteres
```

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

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

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

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

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

void* memcpy(void* dest, const void* origen, unsigned n);
    // copia n caracteres de origen a destino. NO Valido si se solapan

void* memmove(void* dest, const void* origen, unsigned n);
    // copia n caracteres de origen a destino. Valido si se solapan

int memcmp(const void* m1, const void* m2, unsigned n);
    // Compara lexicograficamente 'n' caracteres de m1 y m2
    // devuelve <0 si m1<m2, 0 si m1==m2, >0 si m1>m2

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 caracter ch en los primeros n caracteres de m1
```

C.3. 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 numero aleatorio entre 0 y max (exclusive) */
unsigned
aleatorio(int max)
{
    return unsigned(max*double(rand())/(RAND_MAX+1.0));
}

void abort();   // aborta el programa como error
void exit(int estado);  // terminacion normal del programa
int atexit(void (*fcn)(void));
        // funcion 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
```

C.4. cassert

```
#include <cassert>

void assert(bool expresion);

    // macro de depuracion. Aborta y mensaje si expresion es falsa
    // si NDEBUG esta definido, se ignora la macro
```

C.5. 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 hiperbolico de rad
double cosh(double rad);            // coseno hiperbolico de rad
double tanh(double rad);            // tangente hiperbolica 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);              // raiz 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.6. 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.7. climits

```
#include <climits>

CHAR_BIT = 8

CHAR_MAX        | SHRT_MAX      | INT_MAX       | LONG_MAX
CHAR_MIN        | SHRT_MIN      | INT_MIN       | LONG_MIN
UCHAR_MAX       | USHRT_MAX     | UINT_MAX      | ULONG_MAX
SCHAR_MAX       |
SCHAR_MIN       |
```

C.8. cfloat

```
#include <cfloat>

FLT_EPSILON     // menor numero float X tal que 1.0+X != 1.0
FLT_MAX         // maximo numero de punto flotante
FLT_MIN         // minimo numero normalizado de punto flotante

DBL_EPSILON     // menor numero double X tal que 1.0+X != 1.0
DBL_MAX         // maximo numero double de punto flotante
DBL_MIN         // minimo numero double normalizado de punto flotante

LDBL_EPSILON    // menor numero long double X tal que 1.0+X != 1.0
```

```
LDBL_MAX        // maximo numero long double de punto flotante
LDBL_MIN        // minimo numero long double normalizado de punto flotante
```

C.9. 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 caracter 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 caracter (con eco)
    int getch(void);
        //  lee un caracter (sin eco)
    int kbhit(void);
        //  comprueba si hay pulsaciones de teclado
    int ungetch(int);
        // devuelve el caracter al buffer de teclado

- Manipulacion de texto (y cursor) en pantalla
    void clrscr(void);
        // borra e inicializa la pantalla
    void clreol(void);
        // borra la linea desde el cursor al final
    void delline(void);
        // elimina la linea donde se encuentra el cursor
    void gotoxy(int x, int y);
        // posiciona el cursor
    void insline(void);
        // inserta una linea 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); // _NOCURSOR, _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

- Informacion
    void gettextinfo(struct text_info *_r);
        // informacion 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; // Numero de lineas de la pantalla
        unsigned char screenwidth;  // Numero 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 preprocesamiento:

```
#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 ------------------------------------
__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__
//-------------------------------------------------------------------------

#ifndef __STDC__
#ifdef __cplusplus
#ifndef NDEBUG
#if ! ( defined(__GNUC__)                                                   \
        && ( (__GNUC__ >= 4) || ((__GNUC__ == 3)&&(__GNUC_MINOR__ >= 2)))   \
        && ( defined __linux__ || defined __MINGW32__ ))
#error "Debe ser GCC version 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 = (asignacion) en vez de == (igualdad) en comparaciones.

```
if (x = y) {    // error: asignacion en vez de comparacion
    ...
}
```

- 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 Programacion 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/