Lenguaje C++

Índice

Capítulo 1 - Introducción a C++ y su biblioteca estándar 2
Capítulo 2 - Tipos y declaraciones 4
Capítulo 3 - Punteros, arrays y estructuras 6
Capítulo 4 - Expresiones y sentencias 7
Capítulo 5 - Funciones 8
Capítulo 6 - Espacios de nombres y excepciones 11
Capítulo 7 - Archivos fuente y programas 12
Capítulo 8 - Clases 12
Capítulo 9 - Plantillas 19
Capítulo 10 - Tratamiento de excepciones 21
Capítulo 11 - Jerarquías de clases 21
Capítulo 12 - Organización de la biblioteca y contenedores 22
Capítulo 13 - Contenedores estandar 25
Capítulo 14 - Algoritmos y objetos función 28
Capítulo 15 - Iteradores y asignadores 28
Capítulo 16 - Cadenas 28
Capítulo 17 - Flujos 31
Capítulo 18 - Números 31

Capítulo 1 - Introducción a C++ y su biblioteca estándar

1) Hola mundo:

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

int main () {
   cout << "¡Hola mundo!" << endl;
   return 0;
}
```

2) El espacio de nombres de la biblioteca estandar:

La biblioteca estandar esta definida dentro del espacio de nombres `std`, si queremos hacer uso de los miembros de dicho espacio sin posibilidad de error tenemos que hacer:

```
#include <iostream> → std::cout << "Hola..." << std::endl;
#include <string>   → std::string c = "¡Jej!";
#include <list>     → std::list<std::string> nombres;
// Nota: endl == '\n'
```

Sin embargo hay una forma de hacer implícito un espacio de nombres en nuestro programa usando using:

```
#include <string>
using namespace std;
string c = "Soy un vago.";
```

3) La salida:

C++ a parte de poder usar `printf`, tiene su propio método para sacar testo por pantalla, con el objeto `cout` que tiene el operador << sobrecargado:

```
cout << "Cadena" << endl;
cout << "Cadena\n";
cout << "Cadena" << '\n';
cout << "Cadena" << "\n";
```

Se puede hacer lo mismo que con `printf`, solo que con un formato distinto al conocido en C.

4) Cadenas:

En C++ el manejo de cadenas es mucho más potente, podemos usar todo tipo de operadores y la clase string trae todo tipo de métodos muy útiles:

```
void main () {
```

```
    string c1 = "Hola", c2 = "mundo";
    string c3 = '¡' + c1 + ", " + c2 + "!\n";
    cout << c3;
}
```

También se puede concatenar texto:

```
c1 = c1 + '\n';
c2 += '\n';
// Añade un salto de linea.
```

O comparar valores:

```
if(cadena == "si") { ... }
```

También podemos realizar operaciones complejas:

```
void main () {
   string nombre = "Niels Stroustrup";
   string c = nombre.substr(6, 10);  //c="Stroustrup";
   nombre.replace(0, 5, "Nicholas"); //nombre="Nicholas Stroustrup";
}
```

Y si alguna vez necesitamos una cadena al estilo C:

```
printf("Nombre: %s\n", nombre.c_str());
```

5) <u>Entrada</u>:

Se utiliza `cin` con el operador >>, para meter datos de entrada via teclado:

```
cin >> cadena;
cin >> numero;
cin >> varchar;
```

Aunque a la hora de meter cadenas existe un problema y es que el espacio en blanco es considerado un caracter de terminación, si metemos "Hola mundo." pasaría:

```
cin >> c1;        // c1 = "Hola";
getline(cin, c2); // c2 = "Hola mundo.";
```

6) <u>Contenedores</u>:

+ <u>Vector</u>: Funciona como un array de elementos.

```
vector<miTipo> miVector(numElementos);
miVector[i].miCampo = valor;
miVector.size(); //= Nº de elementos del vector.
miVector.resize(miVector.size() + 10); // Aumenta el tamaño del
                                       // vector en 10 elementos.
miVector = miVector2; // miVector recibe una copia de todos los
                      // elementos de miVector2.
```

Un punto a considerar es que pasaría si llamamos a un indice fuera de rango. En el caso del vector lanzaría una excepción de tipo `out_of_range`:

```
try {
```

```
    miVector[-1] = dato;
} catch (out_of_range) {
    cerr << "¡¡¡Error!!!" << endl;
}
```

+ List: Dado que para ciertas cosas un array o Vector requeriría un coste brutal, existen las listas dinámicas:

```
list<miTipo> miLista;
```

Pero para recorrer una lista requeriremos de un iterador:

```
list<miTipo>::const_iterator i;
for(i = miLista.begin(); i != miLista.end(); ++i) {
    miTipo & dato = *i;
    cout << dato.campo << endl;
}
```

Y para modificarlo es posible que necesitemos otro:

```
miLista.push_front(dato);   // Mete dato al principio.
miLista.push_back(dato);    // Mete dato al final.
miLista.insert(modo, dato); // Mete dato delante del nodo.
miLista.erase(nodo);        // Borra el nodo indicado.

list<miTipo>::iterator nodo;
```

+ Map: Sirve para casos como el de tener una lista de valores identificados con un código en particular (nombre, datos):

```
map<string, miTipo> miMap;
miMap["pepe"] = datos;
```

+ Listado de contenedores estandar:

`vector<T>`	→ Array de tamaño variable.
`list<T>`	→ Lista doblemente enlazada.
`queue<T>`	→ Cola.
`stack<T>`	→ Pila.
`deque<T>`	→ Cola doblemente enlazada.
`priority_queue<T>`	→ Cola ordenada por valor.
`set<T>`	→ Un conjunto.
`multiset<T>`	→ Un conjunto con repeticiones.
`map<clave, val>`	→ Array asociativo.
`multimap<clave, val>`	→ Un map con claves repetidas.

Capítulo 2 - Tipos y declaraciones

1) Tipos básicos:

`bool`	`true` (= 1) / `false` (= 0)

```
[signed] char              8 bits        -128..127
unsigned char              8 bits        0..255
wchar_t                    Caracteres en unicode
[signed] short [int]       16 bits con signo
unsigned short [int]       16 bits sin signo
[signed] [long] int        32 bits con signo
unsigned [long] int        32 bits sin signo
[signed] short long        32 bits con signo
unsigned short long        32 bits sin signo
[signed] long long         64 bits con signo
unsigned long long         64 bits sin signo

void                       Nada

float                      Reales de 32 bits
double                     Reales de 64 bits
long double                Reales de 80 bits
```

2) Literales:

+ Booleanos = `true`, `false`.
+ Caracteres: ANSI = 'a', 'b', 'c', ...
  Unicode = L'a', L'b', L'c', ...
+ Números:
  - Enteros: `{[+]|-} numero [L|U|UL]`
    - L = long
    - U = unsigned int
    - UL = unsigned long
    - 0Numero = Octal
    - 0xNumero = Hexadecimal
  - Reales: `{[+]|-} numero.numero [F|{e|E}{[+]|-} exponente]`
+ Cadenas: "Cadena ansi"
  L"Cadena unicode"

3) Los tamaños:

`sizeof(tipo)` → Devuelve el tamaño en bytes del tipo.

Luego en la librería "`limits.h`" tenemos más herramientas para este tema:

```
numeric_limits<tipo>::max(); // Da el número máximo del tipo.
numeric_limits<tipo>::is_signed(); // Tipo con signo o no.
```

4) Enumerados:

```
enum nombreDelTipo {
   id01 [= valor],
   ...
};
```

5) Declaraciones:

```
[const] tipo nombre [= inicialización], nombre2 [= val], ...;
```
Ejemplos:
```
int numero;
char * cadena = "Hola pepe.";
const int valores[] = {1, 2, 3, 4, 5};
```

```
typedef definicionTipo nombre;
```
Ejemplos:
```
typedef unsigned int UInt32;
typedef char MiCadena[256];
typedef void (*MiFuncion) ();
typedef void (MiClase::*MiFuncion) ();
```

Capítulo 3 - Punteros, arrays y estructuras

1) Punteros:

```
tipo * nombre; → int * pnum;
  int a = 2;
  int * p = &a;
  int b = *p;

const int NULL = 0; // Valor inicial para los punteros, que hay que
                    // asignarles al principio.
  int * p = NULL;
```

2) Arrays:

```
tipo nombre [val] = {val, ...};
tipo nombre [val][val] = {{val, ...}, ...};
```

3) Punteros y constantes:

```
const tipo * p = varp;
```
No permite → p[0] = 0;
Si permite → p = varp2;

```
tipo * const p = varp;
```
Si permite → p[0] = 0;
No permite → p = varp2;

4) Las referencias:

Sirven para crear un nombre alternativo para otra variable:
```
int a = 1;      int &b = a;
b = 2;          cout << a;     // a = 2
```

5) Estructuras:

```
struct nombreDelTipo {
   tipo nombre;
   ...;
}
```

Capítulo 4 - Expresiones y sentencias

1) Operadores:

+ Resolución de ámbito: claseONamespace::miembro
+ Global: ::nombre

+ Miembro de selección: obj.miembro, obj->miembro
+ Indexación: puntero[expresion]
+ Llamada a función: func(parametros)
+ Llamada al constructor: tipo(parametros)
+ Post-incremento: var++
+ Post-decremento: var--
+ Identificación del tipo: typeid(tipo), typeid(expresion)
+ Conversión de tipos: dynamic_cast<tipo>(expresion), static_cast<tipo>(expresion), reinterpret_cast<tipo>(expresion), const_cast<tipo>(expresion)
+ sizeof expresion, sizeof(tipo), ++var, --var, ~expresion, !expresion, -expresion, +expresion, &expresion, *expresion, new tipo [(parametros)], (tipo) expr, delete [[]] puntero
+ *, /, %, +, -
+ <<, >>
+ <, <=, >, >=
+ ==, !=
+ &
+ ^
+ |
+ &&
+ ||
+ ?:
+ =, *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=
+ throw expresion
+ expresion, expresion

2) Asignación de memoria:

```
char * c = new char[sizeof("Hola mundo.") + 1];
int * i = new int;
```

```
delete [] c;
delete i;
```

3) Sentencias de control:

```
try { ... } [catch(excepcion) { ... } [...]]
if(condicion) { ... } [else { ... }]
switch(expresion) {
    case valor: [...;] [break;]
    [...]
    [default: ...;]
}
while(condicion) { ... }
do { ... } while(condicion);
for(inicializacion; condicion; expresion) { ... }
break;
continue;
return [valor];
```

Capítulo 5 - Funciones

1) Declaración:

```
tipo nombre (parametros) {
   ...;
   [return [valor];]
   ...;
}
```

Parametros: `(tipo nombre [= valorPorDefecto], ...)`
`([void])`

Un caso particular de función son las funciones `inline`, que insertan el código donde son llamadas, para evitar el coste de la llamada a una función:

```
inline tipo nombre (parametros) { ... }
```

Aunque no necesariamente logra el compilador hacer `inline` cualquier función, por ejemplo las recursivas normalmente no son posibles candidatas para un `inline`. Su equivalente en C son las macros.

Una variable local puede ser declarada `static` de la siguiente forma:

```
static int num = 37;
```

Cuando se ejecuta el programa, el primer valor de num será 37, y podremos cambiarlo en la función:

```
void IncNum (int n = 1) {
   static int num = 37;
   num += n;
```

```
}
```

Ha medida que llamamos a `IncNum`, el valor de num cambiará, pero lo interesante es que no se perderán estos cambios al salir de la función.

2) <u>El paso de parámetros</u>:

Normalmente el paso es por valor en casos como:

```
int suma (int a, int b);
```

Pero de vez en cuando deseamos modificar su valor y tenemos que pasar por referencia las variables, y para ello hay dos formas:

+ Mediante punteros:

```
void IncNum (int * num) { (*num)++; }
int miNum = 10;
IncNum(&miNum); // Ahora valdrá 11.
```

+ Mediante una referencia:

```
void IncNum (int & num) { num++; }
int miNum = 10;
IncNum(miNum);
```

El pasar parámetros por referencia tiene otra utilidad, a fin de no saturar la pila con valores de array de miles de posiciones u objetos enormes. Pero para evitar que sean modificados podemos usar const y evitar fallos:

```
void listarLista (const int * listaNum);
void listarLista (const CListaNums & obj);
void listarLista (const CListaNums * obj);
```

3) <u>Sobrecarga de nombres de funciones</u>:

En C++ podemos tener 20 funciones con el mismo nombre, siempre y cuando varíen los tipos de los parámetros:

Bien:

```
void Inc (int & n);
void Inc (int & n, int & m);
void Inc (char & n);
```

Mal:

```
void Inc (int * n, int * m);
void Inc (int * lista, int * tam);
```

4) <u>Apuntes sobre los parámetros</u>:

Los parámetros por defecto hay que respetar lo siguiente, a partir de que uno tenga un valor por defecto, los siguientes deberán tenerlos:

```
int f (int a, int b = 0, int c, char * d = 0);     // Mal
int f (int a, int b = 0, int c = 0, char * d = 0); // Bien
```

Luego podemos crear funciones con parámetros ilimitados como ocurre con `printf`, de la siguiente forma:

```
void miFuncion (miTipo * miVar ...) {
   va_list lp;
   va_start(lp, miVar);
   cout << miVar << ' ';
   do {
      miTipo * v = va_arg(lp, miTipo *);
      if(v) cout << v << ' ';
   } while(v);
   cout << endl;
}
```

Llamada → `miFuncion(var1, var2, var3, NULL);`

5) <u>Punteros a funciones</u>:

```
tipo ([nombreClase::]*nombre) ([parametros]);
nombre = &funcion;
```

Ejemplo función:

```
void error (string c) { ... }

void (*jarl) (string);
jarl = &error;
jarl("error");
(*jarl) ("error");
```

Ejemplo función miembro:

```
void miClase::error (string c) { ... }

void (miClase::*jarl) (string);
miClase * jander = new miClase();
jarl = &miClase::error;
(jander->*jarl) ("error");
```

6) <u>Macros</u>:

```
#define NOMBRE texto

#ifdef {NOMBRE | expresion}
   ...;
#ifndef {NOMBRE | expresion}
   ...;
#elif {NOMBRE | expresion}
   ...;
#else
   ...;
```

```
#endif
```

Capítulo 6 - Espacios de nombres y excepciones

1) Espacioes de nombres:

```
namespace nombre {
    definiciones;
}

tipo nombre::funcion (...) { ... }
// Esto sirve para definir funciones fuera del namespace.

namespace nombre2 = nombre; // Para crear un alias.

// Usos de using
void f () {                        void f () { using nombre::mifun;
   if(nombre::varglob)                if(nombre::varglob)
      nombre::mifun();                   mifun();
}                                  }

namespace nombre {
   ...;
   using jarl::otrafun;     // Añade una definición en este caso
                            // una función.
   using namespace jander; // Añade todo lo contenido en el otro
                            // espacio de nombres.
}
```

2) Excepciones:

```
struct ErrRango {
   int num;
   ErrRango (int n) { num = n; } // Constructor
}

char a_char (int i) {
   if(i < numeric_limits<char>::min() ||
      numeric_limits<char>::max() < i)
      throw ErrRango(i);
   return i;
}

void main () {
   try {
      char a = a_char(0);
      char b = a_char(400); // Error...
      ...;
```

```
    } catch(ErrRango) {
        ...;
    } catch(...) {
        ...;
    }
}
```

Capítulo 7 - Archivos fuente y programas

(Páginas 205-229)

Capítulo 8 - Clases

1) Clases:

Una clase es un tipo definido por el usuario, que está estructurado en propiedades y funciones miembro (o simplemente miembros). En C++ las `struct` también son tratadas como clases, solo que no permiten herencia, y todos sus miembros y propiedades son siempre visibles:

```
class nombre {
    public:    // Definiciones públicas
    protected: // Definiciones protegidas
    private:   // Definiciones privadas
};
```

Visibilidad	public	protected	private
Dentro de la clase	Sí	Sí	Sí
Dentro de una clase hija	Sí	Sí	No
Fuera de la clase o sus hijas	Sí	No	No

+ Definición de propiedades:

```
[static] [const] tipo nombre [= init];
```

Ejemplo:

```
int version;
const char * tipo = "Objeto";
static MiClase * instancia = NULL;
static const int valor = 10;
```

+ Definición de miembros:

```
[static | virtual] tipo nombre (parametros) [const] [= 0];
```

+ Definición del cuerpo de los miembros:

```
[inline] tipo nombreClase::nombre (parametros) [const] {
    ...;
}
```

+ La diferencia de poner static:

Los miembros y propiedades `static` nos asegura, que solo hay en memoria una copia de dicha propiedad. Su uso se diferencia en como llamarlos:

```
objeto.propiedadNoEstatica;
objeto.funcionNoEstatica(...);
nombreClase::propiedadEstatica;
nombreClase::funcionEstatica(...);
```

+ Los constructores:

Dentro de la clase tiene que haber al menos un miembro con el mismo nombre que esta:

```
class Coche {
public:
    Coche ();
};
```

Este sirve con un fin relativamente importante, pues cuando se crea un objeto en memoria (cuando se entra en el programa, en una función o pidiendo memoria con un new), la primera función que es llamada es el constructor. Puede tener parámetros y se suele emplear para "devolver" el objeto que se acaba de crear con sus propiedades inicializadas.

+ La copia de objetos:

Por defecto al hacer inicializaciones con igualdad o si asignamos un objeto a otro, se copia cada propiedad una por una al otro. Esto puede traer trágicas consecuencias si alguna de esas propiedades resulta ser un puntero, pues se copiaría la dirección del puntero y no el contenido al que apunta. Por ello para evitar esto tenemos dos métodos distintos:

- Para las inicializaciones, crear un constructor copia:

```
MiClase obj = obj2;
```

Constructor copia → `MiClase (const MiClase & objeto);`

```
MiClase::MiClase (const MiClase & objeto) { ... }
```

- Para el operador = no hay otra opción que sobrecargarlo:

```
obj = obj2; // Ver el tema de sobrecargas.
```

+ Funciones miembro constantes:

- Dentro de la clase:

```
[static | virtual] tipo nombre (parametros) const [= 0];
```

- Fuera de la clase:

```
[inline] tipo nombreClase::nombre (parametros) const {
    ...;
}
```

Sirven para evitar la modificación involuntaria de alguna propiedad de la clase:

```
void clase::func () const { this->var++; } // error
```

+ La auto-referencia:

Con fin de lograr hacer cosas tan chulas como:

```
matriz.mul(num).sum(num2).div(num3);
```

Existe en C++ la palabra clave `this`, que es un puntero al propio objeto dentro de una función miembro. Con ello podemos hacer:

```
class Matriz {
    ...;
    Matriz & mul (int n);
    Matriz & sum (int n);
    Matriz & div (int n);
};

Matriz & Matriz::sum (int n) {
    ...;          // Sumamos n a la matriz.
    return *this; // Y devolvemos el objeto modificado.
}
```

Con `this` también podemos acceder a los miembros y propiedades del objeto con el operador flecha:

```
this->propiedad = valor;
this->funcion(parametros);
```

+ Mutable:

Hay veces en las que podemos encontrarnos con la rara situación, de tener una función constante, donde tenemos que cambiar un miembro de la clase:

```
class Pepe {
    mutable int numero;
    //...
public:
    //...
    void funcion () const { numero++; }
};
```

De hecho gracias a `mutable`, si declaramos un "`const Pepe pepe;`" y llamamos a "`pepe.funcion();`", la propiedad "`numero`" se modificaría. Otra forma de saltarse la protección del `const` es metiendo las propiedades en una estructura o clase y hacer un puntero desde la clase:

```
struct Datos { int numero; }

class Pepe {
    Datos * d;
    //...
public:
    //...
    void funcion () const { d->numero++; }
};
```

2) Objetos:

+ El destructor:

El proceso inverso al constructor, cuando el objeto es destruido (al salir del programa, de una función o usando delete), si existe, se llama al destructor, donde nos podemos encargar de liberar la memoria dinámica reservada. Solo hay un destructor y no recibe parámetro alguno, se define igual que el constructor solo que con un ~ delante:

```
class Pepe {
public:
   Pepe ();                    // Constructor por defecto.
   Pepe (const Pepe & obj); // Constructor copia.
   ~Pepe ();                   // Destructor.
   Pepe (string nombre);       // Constructor alternativo.
   //...
};
```

+ El constructor por defecto:

Es aquel constructor que o bien no tiene ningún parámetro, o bien no requiere ninguno, porque los que tiene, los tiene con valores por defecto. Ejemplos de situaciones donde se llama al constructor por defecto:

```
string a;
string b ("Hola"); // Esta no llama al constructor por defecto.
string * c = new string;
string * d = new string[10];
string e[10];
```

+ La gestión de la memoria dinámica:

En C++ para estos menesteres se emplean los operadores new y delete. Ejemplos:

```
string * mensaje = new string("Hola mundo.");
string * tablamsg = new string[10];

cout << mensaje->c_str() << endl;
cout << tablamsg[0].c_str() << endl;

delete mensaje;
delete [] tablamsg;
```

+ Objetos como propiedades:

Si en una clase tenemos objetos como propiedades, para poder llamar a sus constructores, lo podemos realizar en el propio constructor de la clase, si queremos usar alguno que no sea el constructor por defecto:

```
class Persona {
   string nombre;
   //...
public:
   Persona (string n);
   //...
};

Persona::Persona (string n) : nombre(n) { ... }
```

```
Persona::Persona () : nombre () { ... }
```

+ Las propiedades constantes:

```
static const tipo nombre = valor;
```

3) Sobrecarga de operadores:

+ Notas iniciales:

La sobrecarga de operadores tiene el fin de hacer más sencillo entender un código cuando se trabaja con objetos. Los operadores que se pueden sobrecargar son:

```
+   -   *   /   %   ^   &   |   ~   !   =   <   >   +=  -=
*=  /=  %=  ^=  &=  |=  <<  >>  >>= <<= ==  !=  <=  >=  &&
||  ++  --  ->* ,   ->  []  ()  new  new[]  delete  delete[]
```

Y estos son redefinidos en funciones del siguiente tipo:

```
tipo operator operador (parametros) { ... }
```

Los parámetros varían de un operador a otros y también varía de estar dentro o fuera de la clase.

+ Operadores unarios: `operador operando`

- Dentro de la clase: `tipo operator operador ();`
- Fuera de la clase: `tipo operator operador (tipo operando);`
- Lista de operadores: `+   -   *   &   ~   !   ++   --`

Ejemplos dentro de la clase:

```
MiClase & operator + ();          MiClase & operator - ();
MiClase & operator * ();          MiClase & operator & ();
MiClase & operator ~ ();          MiClase & operator ! ();
MiClase & operator ++ ([int]);    MiClase & operator -- ([int]);
```

Ejemplos fuera de la clase:

```
int operator + (int op);      int operator - (int op);
int operator * (int op);      int operator & (int op);
int operator ~ (int op);      int operator ! (int op);

float operator ++ (float op[, int aux]);
float operator -- (float op[, int aux]);
```

El tema de que el operador ++ y -- sean un caso particular es por poder ser prefijo y postfijo:

- `prefijo ()      → ++dato;`
- `postfijo (int) → dato++;`

No se usa para nada el `int aux`, es simplemente una necesidad sintáctica para poder hacer esta distinción.

+ Operadores binarios: `opizq operador opder`

- Dentro de la clase: `tipo operator operador (tipo opder);`

- Fuera de la clase: `tipo operator operador (tipo opizq, tipo opder);`
- Lista de operadores: `+ - * / % ^ & | = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= && ||`

Ejemplos:
```
MiClase & operator + (int opder);
friend MiClase & operator + (int opizq, MiClase opder);
```

+ Operadores de memoria: (función miembro)
```
void * operator new [[]] (size_t tam[, void * p]);
void operator delete [[]] (void * p);
```

+ Otros operadores: (funciones miembro)
```
tipo * operator -> ();                    // Página 299-301
void operator () (tipo & obj) const;      // Página 298-299
tipo & operator [] (tipoIndice index); // Página 296-297
```

Ejemplos:
```
double & operator [] (const int i);
float & operator [] (string & c);
```

+ Funciones y clases amigas:
```
friend tipo funcion (parametros);
friend class OtraClase;
```

Declarando como `friend` clases o funciones, nos ganamos el derecho a poder acceder a los miembros y propiedades privadas de la clase. Ejemplo:
```
class Persona {
   string nombre;
   //...
public:
   friend ostream & operator << (ostream &, const Persona &);
   friend istream & operator >> (istream &, Persona &);
};

ostream & operator << (ostream & out, const Persona & p) {
   out << p.nombre;
   return out;
}
```

+ Constructores explicitos:

La palabra `explicit` delante de un constructor:
```
explicit Cadena (int tam);
```

Sirve para impedir conversiones automáticas al llamar al constructor, por ejemplo:
```
Cadena c = 'a'; // Esto da error con explicit
```

Sin `explicit` 'a' sería convertido a int (a menos que tuvieramos un constructor que admitiera un char como parámetro).

+ Sobrecarga de los cast: `(tipo) obj;`

```
operator tipo () const { return valor; }
```

4) Clases derivadas:

Uno de los logros de la POO es la herencia que las clases (y las estructuras dada su "actualización" en C++) soportan.

```
class Persona { ... };
class Frances : public Persona { ... };
```

Con esto Frances obtiene todos los miembros y propiedades contenidos en Persona, luego Frances es un tipo de Persona, equivalente a que Triángulo es un tipo de Figura. La sintaxis completa sería:

```
class Hija [: {public|protected|private} Padre[, ...]] { ... };
```

+ Sobre-escribiendo las funciones miembro:

Es una gran ventaja poder extender nuestros tipos de datos con la herencia, pero no es raro encontrar una situación en la que haya que sobre-escribir una función, como por ejemplo una que liste la información:

```
void Padre::Listar () { ... }
void Hija::Listar () {
   Padre::Listar();
   ...;
}
```

Por suerte en caso de necesitar llamar a una función del padre, podemos hacerlo para evitar posibles confusiones dentro de las funciones hijas.

+ Constructores y destructores:

Al heredar tenemos la posibilidad de usar los constructores y el destructor del padre:

```
Hija::Hija ([parametros]) [: Padre([parametros])[, ...]] { ... }
```

Si no indicamos ningún constructor, se tratará de usar el constructor por defecto del padre, si lo tiene.

```
Hija::~Hija () { ... }
```

Al solo haber un destructor este suele llamarse solo, aunque técnicamente se le puede invocar como: `Padre::~Padre();`

+ Los problemas a la hora de copiar:

No hay que olvidar cuando hemos heredado de una clase con operadores sobrecargados, que las nuevas propiedades de la hija no están atendidas en la sobrecarga de un operador como el =, con lo que podría ser que se copiara solo una parte del nuevo objeto, si no redefinimos los operadores pertinentes en la clase hija.

+ La historia de la palabra virtual:

Dado que C++ permite el polimorfismo, usando punteros de alguna clase padre, podemos tener funciones que puedan trabajar con varios tipos de clase. Por ejemplo:

```
class Figura {
```

```
    //...
public:
    //...
    virtual int Area () const { return 0; }
};

class Triangulo : public Figura {
    //...
public:
    //...
    int Area () const;
};

int DameArea (Figura * fig) {
    return fig->Area();
}
```

Si no hubieramos puesto `virtual` en la función redefinida `Area`, en la clase padre `Figura`, al ejecutar `fig->Area()`, al ser de tipo `Figura`, llamaría siempre a la función `Area` de la clase `Figura`, con ello esto estaría mal:

```
void main () {
    Triangulo a (lista de coordenadas);
    cout << DameArea(&a) << endl;
}
```

Pero con `virtual` sobre una función en una clase cualquiera, indicamos que existe la posibilidad de que sea redefinida en las clases hijas.

+ Clases abstractas:
  Son clases con al menos un método virtual puro, que en el caso anterior de la clase `Figura`, sería abstracta si `Area` fuera definida como:

```
virtual int Area () const = 0;
```

Esto sirve para evitar que el usuario cree un objeto `Figura`, y para que cuando cree una clase hija, esta requiera forzosamente que contenga una redefinición del método virtual puro, para poder realizar una instancia de objeto de dicha clase hija.

Capítulo 9 - Plantillas

(Páginas 339-367)

1) Definición:

```
template<{tipo|class} id[, ...]> <Definición de clase, estructura o
                                  función>
```

Ejemplo:

```
template<class T, int tam> class Buffer {
```

```
    struct tBuffer;
    tBuffer * buffer;
  public:
    Buffer ();
    void PutData (int i, T data);
    T GetData (int i);
  };

  template<class T, int tam> struct Buffer<T, tam>::tBuffer {
    T data[tam];
    int len;
  };

  template<class T, int tam> Buffer<T, tam>::Buffer<T, tam> () {
    buffer = new tBuffer;
    buffer->len = tam;
  }

  template<class T, int tam> Buffer<T, tam>::PutData (int i, T data) {
    buffer->data[i] = data;
  }

  template<class T, int tam> Buffer<T, tam>::GetData (int i) {
    return buffer->data[i];
  }
```

Visto el ejemplo, para darle uso a esto tendríamos que declarar las variables del siguiente modo:

```
  Buffer<char, 10> cadena;
  vector<Buffer<int, 8> > listaNotas;
  Buffer<vector<float>, 16> listaParam;
```

2) Sobre funciones:

```
template<class T1, class T2> void ListarNotas (const vector<T1> &
   nombres, const vector<T2> & notas) { ... }
```

Para llamar a semejante función necesitaríamos:

```
  ListarNotas(l1, l2);              // char *, int
  ListarNotas(l3, l4);              // wchar_t *, float
  ListarNotas<string, long>(l5, l6); // Forma explícita.
```

3) Uso de funciones variables con plantillas:

```
template<class T> class Pintar {
public:
   static void pintar (T var) { cout << var; }
};
```

```
template<class T, class F = Pintar<T> > void PintarDato (T dato) {
   F::pintar(dato);
}

PintarDato<char>('a');
PintarDato<int, Pintar<int> >(10);
```

Capítulo 10 - Tratamiento de excepciones

1) Esquema general:

```
try {
   //...
   if(error) throw MiExcepcion (parametros);
   //...
} [catch(MiExcepcion & e) {
   //...
}] [catch(...) { // Trata cualquier excepción
   //...
}]

class MiExcepcion [: <lista padres>] { ... };
```

2) Re-elevación:

Si dentro de un `catch` consideramos que esa excepción debería ser tratada fuera de la función o bloque, podemos usar "`throw;`" para volverla a lanzar.

3) Excepciones estandar:

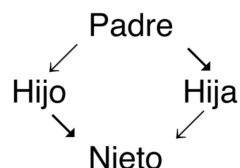
(Páginas 399-401)

Capítulo 11 - Jerarquías de clases

(Páginas 403-440)

1) Los problemas de la herencia múltiple:

C++ nos permite heredar de varias clases padre pudiendose dar el siguiente esquema:



Ahora pensemos en el método `Listar()`, que está redefinida en cada generación y luego el método `FunX()` redefinido en `Hijo` e `Hija`. Con este último al llamar al método, al haber ambigüedad, tenemos que indicar:

```
ObjNieto->Hijo::FunX(); // Versión de la clase Hijo.
ObjNieto->Hija::FunX(); // Versión de la clase Hija.
```

Con el primero encontraríamos un problema bien serio y de difícil resolución:

```
void Nieto::Listar() {
   Hijo::Listar();
   Hija::Listar();
   //Hijo::Padre::Listar();
}
```

Pues suponiendo que en los hijos llamemos al `Listar()` del `Padre`, nos aparecería la información de este repetida. Luego si `Padre` fuera una clase abstracta se podría heredar sin replicación de la siguiente forma:

```
class Hijo : public virtual Padre { ... };
class Hija : public virtual Padre { ... };
```

Porque con una clase base replicada, si `Padre` tuviera nombre como propiedad, nos encontraríamos con un serio problema al hacer referencia a ello desde `Nieto`:

```
this->nombre; // Error de ambigüedad.
this->Hijo::nombre; // Forma válida.
this->Hija::nombre; // Forma válida.
```

2) Temas de acceso en la herencia:

	Clase padre definida como		
Miembros	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	No visible	No visible	No visible

3) El uso del operador typeid:

```
typeid(var) == typeid(tipo)
```

`true` → `var` es del tipo indicado.
`false` → `var` es de otro tipo.

Capítulo 12 - Organización de la biblioteca y contenedores

1) Organización:

+ Contenedores:

<vector> → Array unidimensional de T.
<list> → Lista doblemente enlazada de T.
<deque> → Cola de doble extremo de T.
<queue> → Cola de T (queue y priority_queue).
<stack> → Pila de T.
<map> → Array asociativo de T (map y multimap).
<set> → Conjunto de T (set y multiset).
<bitset> → Array de booleanos.

+ Utilidades generales:
<utility> → Operadores y pares.
<functional> → Objetos función.
<memory> → Asignadores para contenedores (plantilla auto_ptr).
<ctime> → Fecha y hora al estilo de C.

+ Iteradores:
<iterator> → Iteradores y soporte a iteradores.

+ Algoritmos:
<algorithm> → Algoritmos generales.
<cstdlib> → bsearch() y qsort().

+ Diagnósticos:
<exception> → Clase excepción.
<stdexcept> → Excepción estandar.
<cassert> → Macro assert.
<cerrno> → Tratamiento de errores al estilo de C.

+ Cadenas:
<string> → Cadena de T.
<cctype> → Clasificación de caracteres.
<cwctype> → Clasificaciones de caracteres extendidos.
<cstring> → Funciones de cadena estilo C.
<cwchar> → Funciones de cadena de caracteres extendidos estilo C.
<cstdlib> → Funciones de cadena estilo C.

+ Entrada/Salida:
<iosfwd> → Declaraciones adelantadas de utilidades de E/S.
<iostream> → Objetos y operaciones de iostream estandar.
<ios> → Bases de iostream.
<streambuf> → Buferes de flujos.
<istream> → Plantilla de flujo de entrada.
<ostream> → Plantilla de flujo de salida.
<iomanip> → Manipuladores.
<sstream> → Flujos hacia/desde cadenas.
<cstdlib> → Funciones de clasificación de caracteres.

<fstream> → Flujos hacia/desde archivos.
<cstdio> → Familia `printf()` de E/S.
<cwchar> → E/S de caracteres dobles con estilo `printf()`.

+ <u>Localización</u>:
  <locale> → Representa diferencias culturales.
  <clocale> → Representa diferencias culturales al estilo C.

+ <u>Soporte del lenguaje</u>:
  <limits> → Límites numéricos.
  <climits> → Límites numéricos escalares con macros al estilo C.
  <cfloat> → Límites numéricos con coma flotante con macros al estilo C.
  <new> → Gestión de memoria dinámica.
  <typeinfo> → Soporte a la identificación de tipos en tiempo de ejecución.
  <exception> → Soporte al tratamiento de excepciones.
  <cstddef> → Soporte de la biblioteca al lenguaje C.
  <cstdarg> → Listas de parámetros de función de longitud variable.
  <csetjmp> → Rebobinado de la pila al estilo C.
  <cstdlib> → Finalización del programa.
  <ctime> → Reloj del sistema.
  <csignal> → Tratamiento de señales al estilo C.

+ <u>Números</u>:
  <complex> → Números complejos y operaciones.
  <valarray> → Vectores numéricos y operaciones.
  <numeric> → Operaciones numéricas generalizadas.
  <cmath> → Funciones matemáticas estandar.
  <cstdlib> → Números aleatorios al estilo C.

2) <u>Vector</u>: (Páginas 457-474)

+ <u>Iteradores</u>:

```
vector<T>::{iterator|const_iterator|reverse_iterator|
            const_reverse_iterator} nombre;
```

Apunta al primer elemento de la secuencia:

```
iterator               begin ();
const_iterator         begin () const;
reverse_iterator       rbegin ();
const_reverse_iterator rbegin () const;
```

Apunta al siguiente elemento del último de la secuencia:

```
iterator               end ();
const_iterator         end () const;
reverse_iterator       rend ();
const_reverse_iterator rend () const;
```

+ Acceso a los elementos:
```
operator [];
at (posición);
front (); // Primer elemento
back ();  // Último elemento
// Todos devuelven un: T &
```

+ Construcción y copia:
```
vector ([número de elementos]);
vector (tipoIterator primero, tipoIterator último);
vector (const vector & v);
operator =;
void assign (tipoIterador primero, tipoIterador último);
void assign (size_type n, const T & val); // n copias de val
```

+ Operaciones de pila:
```
void push_back (const T & x); // Añadir al final.
void pop_back ();             // Eliminar al final.
```

+ Operaciones de lista:
```
iterator insert (iterator pos, const T & x);
void insert (iterator pos, size_type n, const T & x);
void insert (iterator pos, In primero, In último);
iterator erase (iterator pos);
iterator erase (iterator primero, iterator último);
void clear ();
```

Nota: Los iteradores soportan la suma y la resta.
```
c.erase(c.begin() + 10);
c.erase(c.end() - 10);
```

+ Tamaño y capacidad:
```
size_type size () const;
size_type max_size () const;
size_type capacity () const;
bool empty () const;
void resize (size_type sz, T val = T());
void reserve (size_type n);
```

(max_size == tamaño del vector más grande posible, capacity == tamaño de la memoria en número de elementos.)

+ Otras funciones:
Operadores de comparación: == < > <= >= !=
```
void swap (vector &); // Intercambia los elementos entre dos
                      // vectores rápidamente.
```

Capítulo 13 - Contenedores estandar

(Páginas 477-523)

1) Contenedores estandar:

+ Iteradores: begin(), end(), rbegin(), rend().
+ Acceso a elementos: front(), back(), [] (no es válido para listas), at() (solo sirve para vector y deque).
+ Operaciones de pila y cola: push_back(), pop_back(), push_front() (solo para list y deque), pop_front() (solo para list y deque).
+ Operaciones de lista: insert(), erase(), clear().
+ Otras operaciones: size(), empty(), max_size(), capacity() (vector solamente), reserve() (vector solamente), resize() (vector, list y deque solamente), swap(), get_allocator(), ==, !=, <.
+ Asignación: =, assign().
+ Operaciones asociativas: operator [] (k), find(k), lower_bound(k), upper_bound(k), equal_range(k), key_comp(), value_comp().

2) List:

+ Trasladar: `void splice (iterator pos, list & x[, iterator primero[, iterator último]]);`
+ Mezclar: `void merge (list &[, cmp]);`
+ Ordenación: `void sort ([cmp]);`
+ Operaciones frontales: `void push_front (const T & x);` `void pop_front ();`
+ Otras operaciones:

```
void remove (const T & val);
template<class Pred> void remove_if (Pred p);
void unique (); // Elimina los elementos duplicados, con ==.
template<class BinPred> void unique (BindPred b);
void reverse (); // Invierte el orden de los elementos.
```

3) Deque:

Es más eficiente añadiendo por los extremos que list, pero añadiendo en medio cae frente a list.

4) Stack:

```
T & top ();                // = back();
void push (const T & x);  // = push_back(x);
void pop ();               // = pop_back();
```

5) Queue:

```
void push (const T & x);  // = push_back(x);
void pop ();               // = pop_front();
```

6) Cola de prioridades:

(Páginas 494-496)

7) Map:

+ Temas de los índices:

```
map<string, int> m;
int x = m["Enrique"]; // Crea la nueva entrada y la inicializa a 0.
m["Pedro"] = 7; // La crea, la inicializa a 0 y le asigna 7.
int y = m["Enrique"]; // Devuelve el contenido.
m["Pedro"] = 9; // Asigna 9.
```

+ Operaciones sobre mapas:

```
iterator find (const Key & k);
size_type count (const Key & k);
iterator lower_bound (const Key & k);
iterator upper_bound (const Key & k);
pair<iterator, iterator> equal_range (const Key & k);

pair<iterator, iterator>
```

`.first` → Primer elemento.

`.second` → Último elemento.

+ Operaciones de lista:

```
insert ([iterator pos,] make_pair(Key & k, T & x));
erase (const Key & k);
clear ();
```

8) Set:

La clase `set` sigue un poco el funcionamiento de `map`, donde solo importa la clave y desde luego no hay operador `[]`, pues no se pueden asignar valores.

9) Bitset:

Es un vector de booleanos con las operaciones `[]`, =, ~, `(bool)`, `reference & flip ()`. Se construye:

```
bitset<8> b1;                  // 0x00
bitset<16> b2 = 0xAAAA;        // 0xAAAA
bitset<32> b3 = 0xAAAA;        // 0x0000AAAA
bitset<10> b4 ("0100011100"); // 0x11C
```

Luego las operaciones a nivel de bit son:

Operadores → [], &=, |=, ^=, <<=, >>=, ~, <<, >>.

```
bitset & set   ([size_t pos, int val = 1]); // b[pos] = val
bitset & reset ([size_t pos]);              // b[pos] = 0
bitset & flip  ([size_t pos]);              // b[pos] = ~b[pos]
```

Otras operaciones:

Operadores → ==, !=, &, |, ^.

```
unsigned long to_ulong () const;
string to_string () const;
size_t count () const;        // Nº de bits a 1
size_t size () const;         // Nº de bits
bool test (size_t pos) const; // b[pos] == 1
bool any () const;            // b > 0
bool none () const;           // b == 0
```

Capítulo 14 - Algoritmos y objetos función

(Páginas 525-567)

1) Funciones: (<algorithm>)

+ Busqueda: find, find_if, find_first_of, adjacent_find, count, count_if, mismatch, equal, search, find_end, search_n.
+ Manipulación: for_each, transform, copy, copy_backward, swap, iter_swap, swap_ranges, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n, generate, generate_n, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle.
+ Ordenación: sort, stable_sort, partial_sort, partial_sort_copy, nth_element, lower_bound, upper_bound, equal_range, binary_search, merge, inplace_merge, partition, stable_partition.
+ Conjuntos: includes, set_union, set_intersection, set_difference, set_symetric_difference.
+ Montículo: make_heap, push_heap, pop_heap, sort_heap.
+ Min/Max: min, max, min_element, max_element, lexicographical_compare.
+ Permutaciones: next_permutation, prev_permutation.

2) Objetos función: (<functional>)

+ Predicados: equal_to, not_equal_to, greater, less, greater_equal, less_equal, logical_and, logical_or, logical_not.
+ Aritméticos: plus, minus, multiplies, divides, modulus, negate.

Capítulo 15 - Iteradores y asignadores

(Páginas 569-599)

Capítulo 16 - Cadenas

(Páginas 601-626)

1) Construcción:

```
string c1;           // = "";
string c2 = "Hola";  // c2("Hola");
string c3 = c2;      // c3(c2);
string c4(4, '0');   // = "0000";
string c5(c2, 2, 1); // = "l";
```

2) Asignación:

```
c1 = "Hola";
c1 = c2;
c1 = 'A';
```

3) Conversión al estilo C:

```
const Ch * c_str () const; // Da un char * terminado en '\0'.
const Ch * data () const;  // Da la cadena interna (no termina
                           // en '\0').

size_type copy (Ch * p, size_type numcars, size_type pos = 0) const;
```

4) Comparaciones:

```
int compare (const string & c) const;
int compare (const Ch * c) const;
int compare (size_type pos, size_type n, const string & s[,
             size_type pos2, size_type n2]) const;
int compare (size_type pos, size_type n, const Ch * p,
             size_type n2 = npos) const;
```

5) Inserción:

```
operator +=; // Para string, Ch * y Ch
void push_back (Ch c);

string & append (const string & s[, size_type pos, size_type n]);
string & append (const Ch * p[, size_type n]);
string & append (size_type n, Ch c);
string & append (In primero, In último);

string & insert (size_type pos, const string & s[, size_type pos2,
                 size_type n]);
string & insert (size_type pos, const Ch * p[, size_type n]);
string & insert (size_type pos, size_type n, Ch c);

iterator insert (iterator p, Ch c);
void insert (iterator p, size n, Ch c);
void insert (iterator p, In primero, In último);
```

6) Concatenación:

Operador: +

```
c2 = c3 + " jej " + c4;
```

7) Busqueda:

`size_type pos`	= `find(cadena);`
`find`	= Da la posición donde encontró la cadena.
`rfind`	= Como `find` pero empieza desde el final.
`find_first_of`	= Busca el primer caracter que exista en la cadena.
`find_last_of`	= Como el anterior, pero empezando por el final.
`find_first_not_of`	= Busca el 1er caracter que no exista en la cadena.
`find_last_not_of`	= Como el anterior, pero empezando por el final.

8) Reemplazamiento:

```
string & replace (size_type i, size_type n, const string & s[,
                  size_type i2, size_type n2]);
string & replace (size_type i, size_type n, const Ch * p[,
                  size_type n2]);
string & replace (size_type i, size_type n, size_type n2, Ch c);
string & replace (iterator i, iterator i2, const string & s);
string & replace (iterator i, iterator i2, const Ch * p[,
                  size_type n]);
string & replace (iterator i, iterator i2, size_type n, Ch c);
string & replace (iterator i, iterator i2, In pri, In últ);

string & erase (size_type i = 0, size_type n = npos);
iterator erase (iterator i);
iterator erase (iterator primero, iterator última);
```

9) Subcadenas:

```
string substr (size_type i = 0, size_type n = npos) const;
```

10) Intercambio:

```
void swap (string & a, string & b);
```

11) Tamaño y capacidad:

```
size_type size () const;
size_type max_size () const;
size_type length () const;
bool empty () const;
void resize (size_type n[, Ch c]);
size_type capacity () const;
```

```
void reserve (size_type res_param = 0);
```

Capítulo 17 - Flujos

(Páginas 627-680)

Capítulo 18 - Números

(Páginas 681-712)

1) Funciones matemáticas estandar:

```
double abs   (double d);              // |d|
double fabs  (double d);              // |d|

double ceil  (double d);              // Redondeo al alza de d
double floor (double d);              // Parte entera de d

double sqrt  (double d);              // √d
double pow   (double d, double n);    // d^n
double pow   (double d, int n);       // d^n

double exp   (double d);              // e^d
double log   (double d);              // log_e d
double log10 (double d);              // log_10 d

double <funtrig> (double d);
    funtrig → cos, sin, tan, acos, asin, atan, sinh, cosh, tanh.
double atan2 (double x, double y); // atan(x/y)

double modf (double d, double * p); // Devuelve en p la parte entera y
                                    // la fraccionaria como retorno.

double frexp (double d, int * p); // x en [.5, 1)
                                  // p = y / d = x * 2^y

double fmod (double d, double m); // d % m
double ldexp (double d, int i);   // d * 2^i
```

2) Valarray:

(Páginas 686-703)

3) Complejos:

(Páginas 704-706)

4) Aleatorios:

```
int rand(); // 0..RAND_MAX
void srand (unsigned int semilla);
```