Apuntes de C++

Juan Antonio Romero del Castillo (aromero@uco.es)
Departamento de Informática y Análisis Numérico

Universidad de Córdoba
28 de septiembre de 2021

Índice

1. Introducción

1.1. El paradigma orientado a objetos

- La Programación Orientada a Objetos (POO) es lo que se denomina un "Paradigma de Programación", es decir, un modelo, una propuesta tecnológica adoptada por el sector y por la comunidad de programadores de todo el mundo.
- Intenta simplificar la complejidad de la programación mediante el uso del concepto de **abstracción** y de tipo abstracto de dato (TAD) como base del diseño de programas (más adelante veremos que un TAD es lo que vendría a ser una clase en la POO).
- En los años 60 la protagonista única era la **programación estructurada**: *C, FORTRAN, PASCAL*, etc. Y es a partir de finales de los 80 y en los 90 que surge este nuevo paradigma con los lenguajes *C++, SmallTalk, Eiffel, JAVA, C#,* etc. Hoy lo siguen lenguajes modernos como Python, Ruby, JavaScript, etc.
- Conceptos comunes en los lenguajes de programación orientados a objetos (LPOO) son: abstracción, encapsulamiento, reusabilidad, clases, objetos, polimorfismo y herencia. Todos los LPOO suelen traer integrados todos estos conceptos.
- Los objetivos de la POO es facilitar la labor de programar, conseguir más fácilmente un mayor éxito en la programación, reducir el tiempo de diseño, dotar al software de una mayor reusabilidad y facilitar el mantenimiento. Podríamos resumir diciendo que el objetivo general es aumentar **calidad del software**.

1.2. C++

- En 1986 *Bjarne Stroustrup* define una extensión del Lenguaje *C* que incluye *clases*. El trabajo es publicado por el autor en la obra: *Bjarne Stroustrup*, "The C++ Programming Language" [5].
- C++ es un lenguaje de programación multiparadigma: procedural/imperativa y orientado a objetos. Por ello C++, a veces, no se considera un LPOO *"puro"*, lo cual algunos programadores critican. Aunque su uso es tan extendido que lo convierte en uno de los lenguajes de programación más usados del mundo.
- The current ISO C++ standard is C++17[7].
- In-progress C++20.

2. Algunos aspectos iniciales de C++

C++ es un superconjunto de C, la mayor parte de los programas de C son también programas de C++ de manera implícita.

2.1. Primer ejemplo de un programa en C++

```
#include <cstdio>    //Biblioteca estandar I/O de C (ya no es stdio.h)

#include <iostream> //Biblioteca estandar I/O de C++
```

```
                    /* El comentario de varias
                        líneas del lenguaje  C sigue valiendo */

//using namespace std;  Si no se quiere repetir "std::" cada vez

int main(void)
{
int i;
char cad[50];

std::cout << "\n Hola esto es una prueba";

std::cout << "\n Escribe un entero :  " ;

std::cin >> i;

std::cout << " \n Escribe una cadena : ";

std::cin >> cad;

std::cout << " \n El entero que has escrito es:  " << i << "\n";

std::cout << " y la cadena es :  " << cad << std::endl;;

printf("\n bye bye \n"); // Se puede usar incluyendo <cstdio>
                        // pero durante este curso
                        // usaremos std::cin y std::cout
}
```

2.2. Comentarios al programa de ejemplo

- Los espacios de nombres (*namespace*) se describen más adelante en la sección 11 de este documento. Aquí simplemente diremos que los objetos *cin* y *cout* pertenecen al espacio de nombres *std* y que por ello hay que escribir `std::` delante de ellos para utilizarlos. Como veremos más adelante, la directiva `using namespace std;` nos evita tener que poner `std::` delante de *cin* y *cout* cada vez que los usemos, pero dicha directiva no es recomendable usarla de forma generalizada como veremos más adelante.
- C++ establece un nuevo sistema de incluir archivos sin poner la extensión ".h". Por ejemplo: `#include <iostream>`. Esta es la forma recomendable y la que nosotros usaremos.
- Para usar los includes clásicos de C: `#include <stdio.h>`, `#include <math.h>`, debemos hacerlo anteponiendo la letra "c" y sin poner .h como en: `#include <cstdio>`, `#include <cmath>`, etc.
- A los comentarios clásisos de C con /* ...*/ se añade otra forma con // con la que definimos un comentario hasta el final de esa línea.
- La E/S básica en C++ se hace con *cin* y *cout*, que son respectivamente dos objetos que representan los flujos (*streams*) estandard de entrada (teclado) y salida (pantalla). Estos objetos se usan mediante los operadores de inserción (<<) y extracción

(>>). Veremos más detalles sobre estos operadores cuando veamos la sobrecarga de operadores.

2.3. Compilación

Nosotros usaremos el potente compilador GNU gcc [8], pero lo haremos usando la interfaz g++. Existen otros muchos compiladores de C++ en diversos sistemas operativos (prácticamente en todos).

Algunas normas a seguir:

1. Never, ever, ever put a .h file on a g++ compile line. Only .cpp files. If a .h file is ever compiled accidentally, remove any *.gch files.

2. Never, ever, ever put a .cpp file in an `#include` statement.

3. Introducción a las clases y los objetos

3.1. Declaración de una clase

La característica más importante de C++ es el uso de clases. La base de la programación orientada a objetos son los objetos y en C++ para crear un objeto es preciso definir primero la clase a la que va a pertenecer dicho objeto. Una clase se declara mediante la palabra reservada **class**.

Definamos la clase que representará un punto en el plano.

```
// point.h
// point class small example

#ifndef POINT_H
#define POINT_H

class Point{
private:
  int x_;
  int y_;
public:
  int x();
  int y();
  void set(int x, int y);
};

#endif
```

El cuerpo de esta clase donde se escribe el código de las funciones lo veremos más adelante.

Observar que en todos los ficheros de declaración (ficheros .h o .hpp, etc.) debemos utilizar siempre las denominadas **guardas de inclusión** para que al ser incluídos posteriormente por otros programas (con `#include`) no tengamos problemas de redefinición.

Veamos ahora otro ejemplo de definición de una clase, en este caso una clase que representa una *Pila* de enteros:

```
// stack.h
// An example of a small stack class called Pila

#ifndef STACK_H
#define STACK_H

class Stack{
private:
  int *v_;
  int start_, end_, size_;

public:
  Stack(int initsize=10); //constructor (lo veremos más adelante)
  ~Stack();               // destructor (lo veremos más adelante)
  int push(int i);
  int pop();
  bool full();
  bool empty();
  int size();
};

#endif
```

El cuerpo de esta clase lo veremos más adelante.

3.2. Miembros públicos y privados

Dentro de una clase, en las secciones *public* y *private*, pueden existir tanto declaraciones de datos como de funciones (estas funciones se denominan "métodos" en terminología orientada a objetos). Todo lo que se declare dentro de una clase se dice que es **miembro** de dicha clase. Los miembros de una clase pueden ser datos que almacenen valores, características, el estado de sus objetos, y funciones que gestionen esos datos y representen el **comportamiento** de los objetos de esa clase:

$$\text{CLASE} = \text{DATOS} + \text{MÉTODOS}$$

Los miembros de una clase pueden ser públicos o privados[1]:

- Las declaraciones dentro de la sección (**public:**) se denominan parte pública de la clase. Esta sección también recibe también el nombre de **interfaz** de la clase. Estará compuesta tanto de funciones (también denominados métodos públicos de la clase) como de variables, que forman la interfaz del usuario con un objeto de esa clase. Pueden existir datos públicos, pero se intentan evitar sustituyéndolos por funciones miembro que accedan a ellos, para propiciar así lo que se denomina **encapsulamiento**.

[1] más adelante veremos que también pueden ser *protegidos*

- La sección privada (**private:**) de una clase es la base del encapsulamiento. Está formada por datos y funciones (también denominados métodos privados de la clase) que solo pueden ser accedidos por otros miembros públicos o privados de la clase, o sea, desde las funciones miembro de dicha clase. El usuario de los objetos de esa clase, tiene prohibido el acceso a la parte privada para garantizar el **encapsulamiento**.

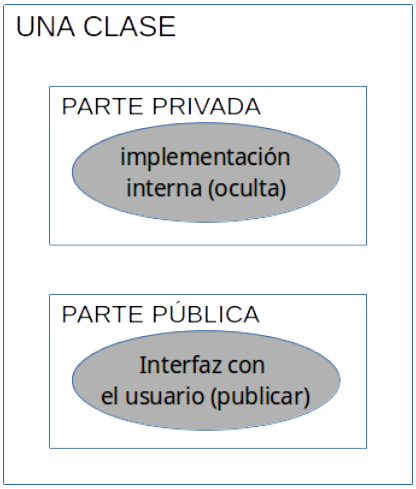

Figura 1: La indiscutible protagonista de la POO

El concepto de **encapsulamiento** se refiere aquí a que las variables internas de una clase, es decir, su implementación, sus estructuras de datos internas (o sea, la parte privada de una clase); no son accesibles directamente desde un objeto, sino a través de sus métodos públicos, de su interfaz. Esto establece una coraza protectora en la parte privada ya que no se puede acceder a ella directamente: encapsula la parte privada. Esto permite que los programas hechos de esta forma sean de mucha más calidad, fáciles de mantener, fáciles de usar, etc. Este concepto se desarrollará en más profundidad y con más extensión en las clases de teoría.

En *C++*, a partir de la declaración de una clase, su nombre se utiliza para la creación de objetos de esa clase de igual manera que en *C* se utiliza el nombre de un identificador de tipo para crear una variable de ese tipo:

```
Point p1, p2;

Stack s;
```

3.3. Estructuras, uniones y clases

- Una estructura es igual que una clase salvo que por defecto todos sus elementos son públicos. Por lo demás son sustituibles, aunque es preferible usar *class*.

- La unión permanece igual que en C, salvo que puede contener funciones miembro. Todos los elementos son públicos; no están permitidas partes privadas en una unión.

3.4. El cuerpo de una clase

La codificación del cuerpo de una clase puede hacerse en el mismo fichero que su definición o en otro. Por ello, al codificarla es necesario indicar a **qué clase pertenece**, puesto que podría haber conflictos, ya que en C++ es posible que funciones de distintas clases tengan el mismo nombre. Posteriormente se verá incluso que dentro de la misma clase **es posible usar el mismo nombre** para distintas funciones.

La clase *Point* podría tener la siguiente codificación:

```
// point.cc
#include "point.h"

int Point::x()
{
  return x_;
}
int Point::y()
{
  return y_;
}
void Point::set(int x, int y)
{
  x_=x;
  y_=y;
}
```

Y la clase *Stack* la siguiente:

```
// stack.cc
#include "stack.h"

Stack::Stack(int inisize=10)
{
  start_ = end_ = 0; //Al comienzo la pila esta vacía,
                      // y la posicion 0 no se usa.
  size_ = initsize;
  v_ = new int [size_];
}
      /* C++ new operator is the default allocation
         for arrays (see manual) */

int Stack::push(int i)
{
  if(start_==size_) return -1; //Pila llena
  else
    {
    end_++;
    v_[end_]=i;
    return 1;
    }
```

```
}

int Stack::pop()
{
  if(start_==end_) // pila vacia
    {
    return(ERROR_EMPTY_STACK);
            // ERROR_EMPTY_STACK podría estar definido a -1
            // o a algún otro valor que no esté previsto en la pila.
            // O bien llamar siempre a empty() antes de llamar a pop().
            // Otra solución es usar excepciones.
    }
  else
    {
    end_--;
    return(v_[end_+1]);
    }
}

bool Stack::full()
{
if (end_==size_) return true;
 else return false;
}

bool Stack::empty()
{
if (end_==start_) return true;
 else return false;
}
```

- Esta es una implementación de la clase *Stack* usando un vector de enteros. Podríamos haberla implementado mediante otra estructura de datos o incluso con memoria dinámica, que sería lo más adecuado.
- Aún implementándola de otra forma, el interfaz de la clase *Stack* (su parte pública) sería idéntico, el interfaz es independiente de la implementación, la cual está protegida mediante encapsulamiento dentro de la sección *private* de la clase.

3.5. Uso de una clase

- Los objetos de una clase se declaran de igual forma que cualquier dato de cualquier tipo: **nombre_clase** `nombre_objeto1, nombre_objeto2, ...;`.
- Al declararse el objeto de una clase **se reserva espacio** para sus datos y éstos se asocian al objeto hasta salir de su ámbito de declaración (observar el parecido con `struct`).
- Para acceder a los miembros de una clase se utiliza el nombre del objeto, el **operador punto** y el nombre del miembro. Además, si se trata de una función se añadirán los

paréntesis propios de las llamadas a función (aunque la lista de parámetros podrá estar vacía).

Veamos en primer lugar el uso de la clase de ejemplo *Point*. Para ello escribiremos el fichero testpoint.cc, definiremos un objeto de la clase *Point* y llamaremos posteriormente a las funciones miembro de dicha clase.

```
// test_point.cc
// small example program using class Point

#include <iostream>
#include "point.h"

int main(void)
{
Point p;
p.set(3,7);

std::cout << "X= " << p.x() << endl;
std::cout << "Y= " << p.y() << endl;
}
```

Ahora veamos el caso de la clase de ejemplo *Stack*.

```
// test_stack.cc
// small example program using class Stack

#include <iostream>
#include "stack.h"

int main(void)
{
Stack s;
s.push(10);
s.push(21);
s.push(32);
s.push(43);

while (!s.empty()) {
    std::cout << s.pop();
    }
}
```

- La función miembro actúa sobre los datos del objeto cuyo identificador aparece delante del operador punto.
- Observar que, por ejemplo, la sentencia `cout << s._v[10];` está prohibida por el escudo que forma un objeto en torno a sus datos privados.
- Las peticiones a un objeto a través del operador punto también reciben el nombre de **mensajes** a dicho objeto y son simples llamadas a funciones.
- El objeto responde a estos mensajes a través del procesamiento que realizan dichas funciones.

3.6. Constructor de una clase

Es muy normal la inicialización de una clase antes del uso de sus objetos. Esto debe realizarse mediante lo que se llama **el constructor** de la clase. Los constructores son opcionales, es decir, una clase puede no tener constructor.

- C++ tiene un mecanismo que evita tener que definir una función (tal como *inicializar()* o *creacion()*) que realice dicha tarea de inicialización. Este mecanismo se denomina: el **constructor**.
- El constructor es una función miembro de la clase que tiene el mismo nombre que la propia clase y que, si existe, se ejecuta automáticamente en el punto del programa donde se declara el objeto. No se puede invocar a este constructor de otra forma.
- El constructor no devuelve nada (ni siquiera void).
- Dicho mecanismo también permite, además, el paso de algún parámetro inicial al objeto en el momento de su definición.
- Un constructor puede recibir varios parámetros. Los parámetros del constructor se definen formalmente en su prototipo y en su cuerpo, y se pasan al objeto en su declaración después del nombre del objeto, entre paréntesis.
- Están permitidos los parámetros por defecto en constructores.
- Se admite la **sobrecarga de constructores**, esto permite inicializar los objetos de formas diferentes según sea necesario. Es una buena idea, al hacer una clase, proporcionar varias formas de inicializar sus objetos, es decir, definir varios constructores, así el usuario tendrá versatilidad en la declaración de objetos de esa clase.
- Se denomina **constructor por defecto** al que se invoca sin suministrar ningún parámetro. El constructor por defecto será aquel que no tenga parámetros o bien aquel que tenga todos los parámetros con valores por defecto.
- La función correspondiente al constructor se invoca cuando se crea el objeto (cuando el flujo de ejecución del programa pasa por el código donde se declara el objeto).

Añadiremos un constructor a la clase *Point*.

```
...
class Point{
private:
  int x_;
  int y_;
public:
  Point(int x, int y);
  int x();
  int y();
  void set(int x, int y);
};
...
```

Otro ejemplo de constructor lo hemos visto en la clase *Stack* en secciones anteriores. También en prácticas en la clase Dados, Fecha, etc.

(**Sobrecargar constructores** es muy interesante por ejemplo para habilitar distintas formas de inicializar un objeto. Por ejemplo, existen distintas formas de escribir una hora o una fecha, y se podría definir un constructor para cada una de ellas. Los constructores serán estudiados más adelante)

3.7. El destructor de una clase

- El complemento al constructor de una clase es su **destructor**. En muchos casos es conveniente realizar algún procesamiento cuando un objeto deja de ser útil y ya no se va a usar más: liberar la memoria utilizada, cerrar conexiones abiertas, etc.
- El destructor tiene igual nombre que el constructor pero anteponiendole el caracter "∼"delante de su nombre.
- El proceso de destrucción de un objeto se activa a la salida de su ámbito local, o sea, al término de la función que lo ha definido.
- Es posible pasar parámetros al constructor, pero el destructor no puede recibir parámetros.

Veamos un ejemplo suponiendo existentes las clases *Point*, *Stack* y *Vector*, y un constructor en cada una que recibe el tamaño de la pila y del vector respectivamente:

```
int main(void)
{
Point(0,0);
Stack c(20);
Vector v(100);

.......
}
```

El sentido de este código podría ser la creación de una cola de 10 elementos y la creación de un vector de 100 elementos.

Los destructores en este caso, si suponemos que la clase *Pila* y *Vector* han usado memoria dinámica, podrían dedicarse a liberar la memoria dinámica usada cuando ya no se usen objetos de dichas clases.

```
// dentro de stack.cc
...

~Stack::Stack()
{
  delete v_;
}
```

Veremos más adelante el uso de "delete" (similar a la función free() en C) para liberar memoria dinámiza en C++.

3.8. LLamando a una función miembro desde otra función miembro

Dentro de una función miembro es posible llamar a otra función miembro de la misma clase. Esto se hace igual que cuando se accede a los datos privados: escribiendo su nombre directamente, o sea, sin usar operador punto u otro mecanismo.

Veamos la codificación de una función miembro (`top()`) que devuelve el valor guardado en la cima de la pila (el primer elemento) sin eliminarlo:

```
int Stack::top()
{
int aux;
if (!empty())
   {
   aux=pop();
   push(aux);
   return aux;
   {
else
  {
  return ERROR_EMPTY_STACK;
  }
}
```

Observar que la función `top()` es una función miembro de la clase *Stack*, al igual que `empty()` y `pop()`. De modo que entre ellas se llaman sin más que poner su nombre.

4. Convención en la notación de identificadores en C++

Existen múltiples convenciones en cuanto a cómo identificar variables, clases, objetos, miembros, etc, en C++ (*C++ naming conventions*). Una de las más extendidas es la siguiente:

1. Los nombres de clases, tipos definidos por el usuario y enumeraciones, llevan la primera letra en mayúscula.

2. Los nombres de métodos, funciones y variables llevan la primera letra en minúscula.

3. Cuando un nombre de los anteriores, está compuesto de varias palabras, la segunda palabra y sucesivas comienza por mayúscula (o separadas por guión bajo).

Otras sugerencias son:

- Los nombres de constantes normalmente llevan todas las letras en mayúscula.

- Se usarán siempre nombres descriptivos y si es necesario de varias palabras. Aunque los nombres excesivamente largos son incómodos de utilizar.

- Intentar escribir identificadores y comentarios en inglés.

- Los identificadores de datos privados terminan con guión bajo.

5. El tipo *bool*

- Los cinco tipos de datos básicos en *C*: char, int, float, double y void, se mantienen en *C++*. Además *C++* añade el tipo **bool**.
- En *C* cualquier valor distinto de 0 es cierto y el propio valor 0 es falso. Esto se mantiene en *C++*. Pero ahora se pueden declarar datos de tipo **bool** que pueden tomar valores *true* y *false*, ambas palabras reservadas de C++.
- *true* está definido a 1, *false* a 0 (incluso ambas pueden usarse en operaciones como enteros con dicho valor si se quiere).

6. Cadenas de caracteres

Ni C ni C++ disponen de un tipo *nativo* (*built-in type*) para el manejo de cadenas. En C teníamos las cadenas a base de vectores de *char* y en C++ lo que tenemos es que la librería estandar de C++ (*C++ Standard Library* parte del *ANSI/ISO Standard C++*), entre otras clases y funciones, proporciona la clase *string* y utilidades para el manejo de cadenas.

A pesar de que aún no hemos visto las clases en C++ veremos el sencillo uso de cadenas en C++.

- `#include <string> //necesario para usar string`
- Es necesario usar *std::string* cada vez, recordar que `using namespace std;` no es recomendable.
- *std::string cad1;* es la definición de una cadena vacía.
- *std::string cad2="Hola Lenguaje C++";* es una definición con inicialización.
- *std::string cad3("Hola otra vez Lenguaje C++");* con el constructor de la clase (más adelante veremos lo que es un constructor).
- *std::string cad4(15,'-');* una cadena con 15 caracteres -.
- Podemos asignar (=), concatenar dos cadenas (+), comparar ($<, >, <=, >=, ==, !=$), buscar subcadena (*find()*), reemplazar subcadena (*replace()*), extraer subcadena (*substr()*), obtener tamaño (*length()*), intercambiar (*swap()*), comprobar si está vacía (*empty()*), escribir en pantalla con *cout*, introducir desde teclado con *cin*, etc.
- C++ adapta el tamaño de la cadena automáticamente en ejecución. El tamaño máximo puede obtenerse con el método *max_size()* (y dicho tamaño máximo es lo suficientemente grande como para no preocuparnos por él) .
- La función *c_str()* devuelve la cadena al estilo de C del objeto de tipo *string* al que se aplique, es decir, devuelve un vector de *char* terminado con el caracter nulo ('\0'). Entonces podemos hacer:

```
std::string cad(''HOLA'');
int len=cad.length();
char *ptr = new char[len+1]; //mas fácil que malloc()
strcpy(ptr,cad.c_str());
```

```
// o bien:

cad.copy(ptr, len,0); // copia y añade caracter nulo
```

6.1. Convertir desde un string

Desde C++ 11 tenemos las siguientes funciones para hacer conversión desde string:

- La función std::stoi(), std::stol(), std::stoll(), convierte un string a un int, long y long long (al menos 64 bits desde C++11).
- std::stof(), std::stod(), std::stold() convierte un string a un float, double y long double (desde C++11).

No olvidar usar la opción de compilación -std=c++11 o bien -std=gnu++11.

6.2. Convertir int, float, etc. a un string

A partir de *C++11* se dispone de la función std::to_string() que permite convertir a string el dato pasado como parámetro sea int, float, etc. Esta función nos será util pare concatenar datos de diferente tipo con cadenas.

6.3. Función getline()

La función std::getline(cin, s, '\n') lees un *string s* desde la entrada estándar *cin* pudiendo contener la cadena varias palabras y espacios en blanco.

6.4. Más sobre la clase string

Existen más ejemplos sobre cadenas (clase string) en C++ a disposición de los alumnos en la página web de la asignatura. Tanto estos como el resto de ejemplos que hay en la página web es muy recomendable que se descarguen, compilen, se prueben y se analicen por parte del alumno. Esto es fundamental en el aprendizaje de cualquier lenguaje de programación.

Más en `https://en.cppreference.com/w/cpp/string/basic_string`

7. typedef

En *C* para definir un nuevo tipo con una estructura era necesario:

```
typedef struct
{
char *name[50];
int age;
} Person;
```

En C++ no es necesario poner `typedef` para definir un tipo nuevo, basta con:

```
struct Person
{
char *name[50];
int age;
};
```

para poder declarar posteriormente datos de tipo `Person`.

```
...
Person p;
...
```

8. Declaración de variables en cualquier lugar

En C++ las declaraciones de las variables y de los objetos pueden situarse en cualquier parte del programa siempre y cuando antecedan su uso.

Así, podríamos hacer:

```
for (int i=0;i<100;i++)
  {
  ...
  }
```

La variable i es declarada en el momento de inicio del bucle y deja de existir cuando termina el bloque en que se ha declarado, es decir, cuando termina el bucle.

9. Constantes

Las constantes deben ser siempre inicializadas en el momento de su declaración (en caso contrario salta un error de compilación) y no se les puede modificar ese valor a lo largo de todo el programa, el compilador daría error.

```
...
const float f=7.9;
const int i=8;
const float PI=3.14159;
...
```

9.1. Constantes y definiciones de preprocesador (#define)

A veces también se usan constantes en vez de constantes definidas en el preprocesador con la orden `#define`, ya que al tener un tipo asignado permiten control de tipos, y además son visibles dentro de los depuradores igual que cualquier otra variable.

9.2. Constantes en la lista de parámetros de una función

Se puede pasar una constante como parámetro a una función, de esta forma no se le podrá asignar un nuevo valor dentro de la función.

```
int f(const int *v)
{
  v[2]=55; // error: asignación a ubicación de sólo lectura
}
```

Si aparece *const* en la declaración del parámetro, ya sabemos que no cambiará dentro de la función.

Es interesante también observar que al explicitar más información sobre los parámetros de una función se mejora lo que denominamos autodocumentación del código, lo cual siempre es conveniente.

10. Funciones en línea (inline)

A veces, continuas y excesivas llamadas a funciones muy simples provocan una penalización en el tiempo de ejecución. C++ proporciona un mecanismo que **solicita** al compilador que, en vez de la llamada, haga una sustitución de la llamada por el código de dicha función. Con esto se evita que se realice la llamada en sí ahorrando tiempo de ejecución y memoria. Estas funciones se denominan funciones en línea.

A tener en cuenta sobre funciones *inline*:

- Debe hacerse solo en funciones cortas, menos de 10 líneas aproximadamente. Típicamente observadores y modificadores sencillos.
- Normalmente debe hacerse siempre en la declaración de la clase (en el fichero .h). La palabra clave *inline* debe escribirse delante de la definición de la función. Ejemplo:

```
// Fichero dados.h
// La clase Dados representa el lanzamiento de 2 dados
#ifndef DADOS_H
#define DADOS_H
class Dados{
  private:
  . . .
  public:
  inline int getDado1() { return d1_; }
};
#endif
```

- En el caso anterior no es necesario preceder la definición con la palabra *inline*, aunque siempre conviene ponerla para que nuestro código sea cuanto más explícito mejor.
- También se puede declarar una función *inline* en el fichero .h pero debajo de la declaración de la clase:

```
// Fichero dados.h
// La clase Dados representa el lanzamiento de 2 dados
#ifndef DADOS_H
#define DADOS_H
class Dados{
  private:
  . . .
  public:
  int getDado1();
};
```

```
inline int Dados::getDado1() { return d1_; }
#endif
```

- También se puede declarar *inline* una función que no pertenezca a ninguna clase. El efecto será análogo.
- *inline* es una solicitud, no una orden. Se deniega cuando la función es algo compleja: existe un bucle, o hay un `switch()`, o un `if..`, o un `goto`, o hay recursividad, etc. El compilador ignora la palabra clave *inline* salvo en funciones muy pequeñas.
- El código de un programa con funciones inline cortas disminuye de tamaño y el tiempo de ejecución, en cambio, el código de un programa con funciones inline grandes aumenta dramáticamente su tamaño.
- No conviene definir una función inline fuera del fichero de cabecera (.h) donde se declara la clase, ya que habría que escribirla entonces en cada unidad de compilación para que el compilador no nos diera error por función indefinida.
- Además, cualquier modificación de la función *inline* requiere una recompilación de todo el código que la usa y generación de nuevos ficheros objeto, ya que la llamada a la función no existe en el código compilado y, por tanto, no se enlazará/linkará con una nueva versión de la función inline. Mucho cuidado con esto.

Las funciones inline también se usan para mejorar el funcionamiento de las macros definidas con las directivas del preprocesador.

En este sentido, es mucho más seguro:

```
inline cuadrado(int x) {return x*x;}
```

que la opción del preprocesador:

```
#define  CUADRADO(x) x * x
```

Puesto que *cuadrado(2+3)* resulta en $5 \times 5 = 25$, y *CUADRADO(2+3)* resulta en $2 + 3 \times 2 + 3 = 11$, por mayor precedencia del producto.

Se podría solucionar el problema de esta macro de la siguiente forma:

```
#define  CUADRADO(x) (x) * (x)
```

Pero aún así se aconseja el uso de funciones en línea en estos casos, ya que, además, existe comprobación de tipos y la expresión pasada se evalúa con anterioridad, como en cualquier función.

11. Espacio de nombres. Namespaces

Un espacio de nombres se declara de la siguiente forma:

```
namespace nombre{
  ...
  //Declaraciones
  ...
}
```

En realidad un espacio de nombres es un ámbito, es una agrupación lógica de declaraciones. Cuando los programas empiezan a ser grandes, los espacios de nombres son muy útiles para separar sus partes. También son espacios de nombres el ámbito local, el ámbito global y las clases.

Dentro del espacio de nombres se pueden definir los identificadores que queramos sin preocupación de que exista conflicto o duplicaciones con otros identificadores fuera de ese espacio de nombres o en otro espacio de nombres.

Si por ejemplo definimos las funciones o las clases de un espacio de nombres en un fichero `.h` de la forma que se ha indicado, al definir el cuerpo de las funciones en el correspondiente fichero `.cc` es necesario anteponer al nombre de la función el nombre del espacio de nombres seguido de dos puntos.

```
...
espacio_de_nombres::funcion()
...
```

Si se trata de una función de una clase definida dentro del espacio de nombres:

```
...
espacio_de_nombres::nombre_de_la_clase::funcion()
...
```

En el fichero `.c` se puede evitar anteponer el nombre del espacio de nombres a cada función metiendo el cuerpo de las funciones dentro de un nuevo bloque:

```
namespace nombre{
  ...
  //Cuerpo de las funciones
  ...
}
```

Si en cualquier lugar de nuestro programa queremos usar un objeto definido dentro de un espacio de nombres determinado, escribiremos:

```
espacio_de_nombres::identificador
```

Si queremos usar (visualizar) un espacio de nombres determinado por defecto, podemos usar:

```
using namespace nombre;
```

Como se hace al usar el espacio de nombres de la librería estándar de C++ 'std':

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

Sin la declaración anterior tendríamos que escribir, por ejemplo:

```
std::cout << "Hola Mundo";
```

Pero al declarar (hacer visible) el espacio de nombres 'std' por defecto, podemos poner:

```
cout << "Hola Mundo";
```

Dentro de un espacio de nombres se pueden declarar variables, constantes, funciones, clases, estructuras, etc.

Es recomendable usar std::cout, std::cin, etc., en vez de visualizar todo el espacio de nombres ya que puede generar otros conflictos de nombres con otros espacios de nombres, por lo que no es una buena práctica y, por tanto, no se recomienda el uso generalizado de la cláusula `using namespace ...`.

11.1. *using::* un objeto concreto

Para visualizar un objeto concreto se puede usar la directiva *using* únicamente con ese objeto sin visualizar el resto de nombres del espacio de nombres. Ejemplo:

```
using std::string;

using std::cout;

using std::cin;
```

Después de estas declaraciones no será necesario escribir *std::* delante de estos objetos. Y es mejor práctica, mas conveniente, que visualizar todo el espacio de nombres con *using namespace std;*

12. Parámetros por defecto

C++ permite que los parámetros de las funciones tengan valores por defecto. El valor por defecto se especifica en la lista de parámetros de la declaración de la función, igual que se inicializa una variable.

Importante: los parámetros por defecto se establecen **solo en la declaración de la función**. Cuando se escriba el cuerpo de la función ya no se puede poner su valor por defecto pues nos dará un error el compilador.

```
void velocidad(int v=25);
```

La función *velocidad* puede llamarse: `velocidad(70)`, con lo que *v* toma el valor pasado; o bien, `velocidad()`, con lo que *v* toma el valor 25.

- Los parámetros que admitan valores por defecto deben ir a la **derecha** de los que no, ya que cuando se invoca a una función, todos sus argumentos se hacen corresponder con los parámetros respectivos en estricto orden, de izquierda a derecha. A los que queden sin correspondencia, se les asigna el valor por defecto.
- Por lo expuesto anteriormente, no puede haber un parámetro por defecto anterior a otro sin valor por defecto.
- Los valores por defecto de los parámetros deben representar la forma en que se ejecuta la función la mayor parte de las veces, para proporcionar facilidad de uso. En otros casos puede perjudicar la comprensión del código.
- Puede haber conflictos cuando dos funciones con el mismo nombre tengan parámetros por defecto y no haya forma de resolver el conflicto. Por ejemplo si tienen un parámetro por defecto, en una un *int* y en otra un *float*, y se llama a la función sin argumentos, el compilador no puede decidir. La solución pasa por que cambiemos la situación que ha provocado el conflicto.

Veamos el siguiente ejemplo:

```
void carga(float peso, float volumen, int tipo=0, int subtipo=2);
void descarga(float peso, int tipo=0, float volumen, int subtipo=2);
```

La función *carga()* está correctamente definida. La función *descarga()* está definida incorrectamente según los puntos anteriores.

13. Referencias

Las referencias en C++ pueden verse como alias o nombres alternativos que pueden darse a una variable u objeto. Al definirse siempre debe indicarse a qué variable hace referencia, digamos que no pueden definirse sin hacer referencia a una variable o a un objeto (no se puede declarar haciendo referencia a un valor literal). Esto las hace muy seguras, ya que evitamos que una referencia apunte a una zona errónea.

```
int i;
int &p=i;
```

'p' es una referencia a la variable de tipo int 'i'. Si cambio el valor de 'p' cambia 'i'. Ambos comparten el mismo espacio de almacenamiento en memoria.

Para modificar 'i' solo tengo que hacer, por ejemplo:

```
p=77;
```

- Las referencias se usan sobre todo en paso de parámetros a funciones como veremos a continuación. También los veremos en las secciones constructores de copia y en otras versiones de la sobrecarga de operadores.
- Las referencias son parecidas a los punteros clásicos del lenguaje C (no exáctamente, pero para su mejor comprensión ayuda mucho verlas así). Lo que las diferencia de ellos es que nos ahorran tener que usar la nomenclatura tediosa de esos punteros (asteriscos (*), indirecciones (&), etc.), y que son más seguras, como se ha mencionado antes. También son menos dadas a errores. Los punteros suelen ser dados a errores y quebraderos de cabeza por su notación, sintaxis, etc.
- Por ello hay que usar referencias siempre que sea posible en vez de punteros.

13.1. Paso de parámetros usando referencias en C++

Los parámetros de una función es el lugar más común donde se verán referencias. En C y C++ el paso por valor de una variable hace que la función cree una nueva copia de dicha variable y sea la que se use dentro de la función. Con lo cual, al modificar la copia no se modifica la variable u objeto original.

Para que una función pueda modificar una variable que se pasa como parámetro debe recibirla en un puntero y en la llamada poner la dirección de la variable con el operador &.

Pero C++ aporta una nueva forma de pasar parámetros por referencia más limpia y cómoda: usando **referencias**.

Si una función recibe una variable mediante una referencia la función crea una referencia a la variable (y nó una copia) y si se modifica la referencia se modificará la variable original.

Veamos el clásico ejemplo de la función que intercambia el valor de dos variables, en este caso de tipo *int*, pero utilizando referencias:

```
void intercambia(int &a, int &b)
{
int aux;
aux=a;
a=b;
b=aux;
}
```

La llamada a la función puede ser:

```
int main(void)
{
int i,j;
.....

intercambia(i,j);
......
}
```

Comentarios:

- Dentro de la función intercambia() a y b son referencias a las variables i y j de la llamada. Al modificar a y b se modifican i y j.
- Al preceder con un & el nombre del parámetro, C++ envía a la función la referencia de la variable directamente sin tener que usar el operador & en la llamada.
- No es necesario usar dentro de la función punteros para referirse a la/s variable/s.

13.2. Paso de referencias a funciones. Paso de referencias constantes

Si vamos a modificar una variable dentro de una función siempre conviene pasar usando referencias. Es más cómomo, sencillo y rápido.

Pero hay casos en los que aun sin modificar la variable dentro de la función conviene pasar referencias, y es cuando la variable contiene gran cantidad de datos. Si contiene gran cantidad de datos, el paso por valor obliga a hacer una copia de dichos datos con el consiguiente gasto en memoria y tiempo de ejecución. Si en vez de pasar por valor se pasa usando referencias, no se realiza la copia por lo que se ahorra tiempo de ejecución y memoria.

Si se pasa la referencia, la función podrá modificar la variable, pero esto se puede evitar usando el calificador **const** que se usa para indicar que un objeto es constante y, por tanto, no puede ser modificado.

```
void Tabla::inserta(const Persona &p)

Persona p;
Tabla t;
...

t.inserta(p) // se envía al método una referencia constante a p
```

Por ello, si la variable u objeto contienen muchos datos siempre debe pasarse usando referencias ya que ahorramos memoria y tiempo de CPU. En general, aunque para los tipos básicos (int, char, float, etc.) no se gana mucho pasando referencias constantes, **esta forma será la que usemos de forma predeterminada** (ESTO ES MUY IMPORTANTE).

13.3. Funciones que devuelven referencias

No se recomienda devolver una referencia a menos que estemos seguros de que lo que se devuelve apunta a algo seguro.

Una función puede devolver una referencia por dos motivos:

1. Para evitar que se cree una copia del objeto que se devuelve ahorrando así memoria y la llamada al constructor de copia (tiempo de ejecución).

2. Para poder modificar el valor que devuelve la función, por ejemplo, cuando se quiere modificar usando este tipo de expresiones:

   ```
   f()=7;
   ```

   Esto ocurre cuando se quiere sobrecargar el operador [], como en el siguiente ejemplo (este ejemplo se entenderá mejor cuando estudiemos la sobrecarga de operadores, volver aquí cuado se haya hecho):

   ```
   class Matriz
   {
   private:
   int array[10];
   public:
   Matrix() {memset(array,0,sizeof(array));}
   int& operator[](int index) {return array[index];}

   };

   int main()
   {

   Matrix m;
   m[0] = 1;
   m[1] = 2;
   int x = m[2];
   cout << m[0] << endl;
   cout << m[1] << endl;

   return 0;
   }
   ```

   (Ejemplo tomado de: http://www.daniweb.com/forums/thread40348.html)

Una función que devuelve una referencia se procesa como si devolviera un valor regular, no hay que recibirla con punteros ni referencias ni nada de eso.

```
double& GetWeeklyHours()
{
    double h = 46.50;

    double &hours = h;

    return hours;
}

int main()
{
    double hours = GetWeeklyHours();
    //se hace la asignación y el objeto hours almacena el valor
    cout << "Weekly Hours: " << hours << endl;

    return 0;
}
```

(Ejemplo tomado de: http://www.functionx.com/cpp/examples/returnreference.htm)

Veremos también la utilidad de funciones que devuelven referencias cuando estudiemos el apartado 34 insertadores y extractores propios.

13.4. Otros aspectos de las referencias en C++

- El simple nombre de un objeto es una referencia a su comienzo en memoria.
- No se puede hacer por ejemplo `int &a=NULL`, ya que NULL es un objeto literal temporal que se usa solo para la asignación y deja de existir en la siguiente línea. El compilador lanza un error en este caso, ya que si intentaramos modificar la referencia el resultado sería erróneo e imprevisible (probablemente un *segmentation fault*).
- Por el mismo motivo del punto anterior no se puede hacer `int &a=9`, por ejemplo.
- Si indicamos que la referencia no se va a modificar posteriormente (anteponiendo *const* en su declaración), entonces si se podrían hacer estas asignaciones, pero aún así, la utilidad de ello es muy relativa (el calificador *const* lo veremos en una sección posterior).

14. Sobrecarga de funciones

Una de las formas de **polimorfismo** en C++ es a través de la sobrecarga de funciones: *una o más funciones puede compartir nombre siempre y cuando las declaraciones de sus parámetros sean diferentes.*

```
void intercambia(int &x, int &y);
void intercambia(float &x, float &y);

void intercambia(int &x, int &y)
{
int aux;
aux=x;
```

```
x=y;
y=aux;
}
void intercambia(float &x, float &y)
{
float aux;
aux=x;
x=y;
y=aux;
}
```

- El **compilador sabe qué función debe usar** en cada caso por la lista de parámetros.
- La sobrecarga de funciones permite crear un **nombre genérico** para una operación.
- No sobrecargar funciones no relacionadas.
- La sobrecarga de funciones que difieren solo en el tipo devuelto es ambigua: debe evitarse.

15. Herencia

La herencia es uno de los aspectos más importantes en los lenguajes de programación orientados a objetos y la base de la reutilización del código. Básicamente consiste en definir una clase a partir de otra previamente definida.

Veamos un ejemplo de herencia:

```
class Vehiculo{
public:
  void asignaRuedas(int r);
  int infoRuedas();
  void asignaPasajeros(int p);
  int infoPasajeros();
private:
  int ruedas_;
  int pasajeros_;
};

class Camion : public Vehiculo{
public:
  void setCarga(float c);
  float getCarga();
private:
  float carga_;
};

class Moto : public Vehiculo{
public:
  void setSidecar(bool s);
```

```
  bool getSidecar();
private:
  bool sidecar_;
};
```

- Terminología C++: *Vehiculo* es la **clase base**, o padre; *Camion* y *Moto* son **clases derivadas**, o hijas, de la clase *Vehiculo*.
- Los miembros de *Camion* y de *Moto* tienen acceso a la parte pública de la clase *Vehiculo* exactamente igual que si se hubieran declarado dentro de la parte pública de sus respectivas clases.
- Pero las funciones miembro de *Camion* y de *Moto* no tienen acceso a la parte privada de la clase *Vehiculo*, solo pueden acceder a la parte pública. Esto protege la parte privada de la clase base.
- Al escribir: `class Camion : public Vehiculo`, se está indicando que la clase *Vehiculo* es un elemento público de la clase *Camion*, y por tanto, el usuario de la clase *Camión* podrá llamar a las funciones públicas de *Vehiculo* como si estuvieran en la parte pública de *Camión*.
- Las clases *Camion* y *Moto* pueden tener definidas funciones miembro con el mismo nombre. Por ejemplo una función *mostrar()* que en cada clase será diferente puesto que deben mostrar datos diferentes, pero el compilador no tendría problemas en distinguirlas.
- Si la clase base tiene constructor, éste se ejecuta automáticamente antes de empezar a ejecutarse el constructor de la clase derivada.

La herencia es una relación jerárquica. Permite crear una **jerarquía de clases** de varios niveles. Al definir nuevas clases derivadas, éstas ya tienen incorporadas las características de sus clases base. Con lo que se reutiliza el código, se organiza el código, y se ahorra tiempo de desarrollo, mantenimiento, etc.

Cuando construyo una clase derivándola de otra clase base, estoy en realidad reutilizando todo el código de la clase base. De ahí que la herencia sea considerada una herramienta básica en la **reutilización** del código.

```
#include "camion.h" // las clases Vehiculo, Camion y Moto
#include "moto.h"  // pueden estar definidas en el mismo fichero
                   // o en ficheros diferentes.
                   //Supondremos aquí en ficheros diferentes.
int main()
{
Camion c;
Moto m;
...
cout << "ruedas del camión = " << c.infoRuedas();
cout << "ruedas de la moto = " << m.infoRuedas();
...
cout << "carga del camión = " << c.getCarga();
...
```

```
cout << "tiene sidecar la moto? " << m.getSidecar();
...
}
```

En el programa anterior observar que no hay que declarar ningún objeto de la clase *Vehículo*; no es necesario, al declarar los objetos *c* y *m* ya se crea todo lo necesario.

15.1. Iniciadores de la clase base

Si una clase base dispone de un constructor que recibe parámetros obligatorios, el constructor de la clase derivada debe enviarle esos parámetros.

Supongamos el siguiente código:

```
class Vehiculo{
public:
  Vehiculo(int r) {ruedas_=r;};
...
private:
  int ruedas_;
  int pasajeros_;
};
```

La clase derivada deberá proporcionar el parámetro al constructor de la siguiente manera:

```
class Camion : public Vehiculo{
public:
  Camion(int n): Vehiculo(n) { ... }
};
```

El constructor de la clase derivada, pasará como parámetro a la clase base bien un valor literal o bien un parámetro que él mismo reciba como es el caso del ejemplo. Puede haber más de un parámetro. Esto se denomina iniciadores de la clase base.

Importante: los iniciadores se consideran ya código que ejecuta las llamadas a la clase base y solo deben ponerse donde se vaya a poner el cuerpo del constructor de la clase derivada. Si dicho constructor es inline, en la declaración de la clase; pero si no es inline solo debe ponerse en el cuerpo del constructor y no en su declaración.

16. Objetos constantes y funciones miembro constantes

Un objeto constante se declara mediante el uso del calificador *const* delante del nombre de la clase en su declaración.

```
const Fecha fechaNacimiento(10,1,2001);
```

Normalmente se inicializan pasando parámetros al constructor y se prohibe el uso de **ninguna de sus funciones miembro**, salvo las calificadas mediante *const*, que serán funciones que no modifiquen los datos del objeto.

Importante: *const* se escribe después de la lista de parámetros en la declaración de la clase y también es obligatorio en el cuerpo de la función.

```
int Fecha::verDia() const {return dia};
```

- Las funciones miembro sin *const* en su declaración no pueden ser invocadas con objetos constantes. Generalmente éstas serán modificadores.
- Sí se podrán invocar las funciones miembro declaradas como *const*. Generalmente éstas serán observadores.
- Si una función declarada con *const* intenta modificar el estado del objeto, el compilador producirá un error.
- Aunque los constructores modifiquen el objeto, son los únicos que pueden invocarse sin especificar explícitamente que son *const*. Esto se permite para realizar la inicialización, como es obligatoria en todo objeto constante.
- Si la declaración del objeto no se hace con *const*, entonces se podrán invocar las funciones miembro normalmente, independientemente de si las funciones se han declarado con *const* o no.
- Lo dicho es igual dentro de las funciones que reciben parámetros const, es decir, si una función recibe como parámetro un objeto const, entonces dentro de la función solo se podrá invocar a métodos *const* de ese objeto.

Importante: a partir de ahora, cada método de una de nuestras clases que no modifique al objeto, lo codificaremos como *const*. De este modo no tendremos problemas al pasar como parámetros const nuestros objetos.

17. Constructores de copia

Hacer una copia de un objeto es algo frecuente. Por ejemplo si hacemos la siguiente declaración:

```
MiClase obj1;
```

podemos declarar otro objeto a partir de obj1 de dos formas, así:

```
MiClase obj2(obj1);
```

o bien:

```
MiClase obj2 = obj1;
```

es decir se declara obj2 haciendo una copia de obj1.

También se hacen copias de objetos automáticamente cuando se pasan por valor a funciones o cuando se devuelven en el 'return' de una función. **Importante:** cuando se pasan por referencia o cuando una función devuelve una referencia **no se ejecuta el constructor de copia**.

La copia por defecto que hace C++ es asignando variable a variable entre los objetos (este es el llamado **constructor de copia por defecto**). Es igual que si se igualan dos datos de tipo *struct* en Lenguaje C.

Pero el constructor de copia por defecto no conviene en muchos casos. Por ejemplo cuando la parte privada de la clase haga uso de memoria dinámica. En este caso habrá que definir un **constructor de copia** específico que efectúe bien la copia.

Un ejemplo de definición de un constructor de copia en este caso sería:

```
class MiClase{
.....
MiClase(const MiClase &e);
.....
};
```

- La sintaxis del constructor de copia obliga a pasar una referencia. Pasar una referencia al constructor indica que es el constructor de copia. El hecho de que el objeto que recibe sea constante no es obligatorio pero sí conveniente, ya que en una copia de un objeto éste no debe modificarse. Y así además el código es más autodocumentado.
- Dentro de la función constructor de copia, como el objeto que se recibe es const, para invocar a sus funciones miembro, éstas deben ser const.
- Es importante observar que cuando pasamos referencias como parámetro a una función o devolvemos una referencia en una función no se invoca al constructor de copia ya que no se crea una copia del objeto sino que se usa el objeto original.

17.0.1. Constructor de copia de clases derivadas

Cuando usamos herencia (sección 14), tendremos que tener en cuenta que si hacemos un constructor de una clase derivada y el constructor de la clase base tiene parámetros obligatorios será necesario pasárselos con iniciadores base. Esto también hay que hacerlo en el caso del constructor de copia (en la sección 15 se estudian los iniciadores base).

18. *Containers* en la STL

En la *Standard Template Library* de C++ existen una serie de clases y plantillas definidas que podemos usar fácilmente en nuestros programas. A muchos de estos tipos se les da el nombre de "contenedores" porque de forma genérica pueden "contener" internamente cualquier tipo base.

Están los contenedores clásicos: *vector, list* y *deque*. Y también existen otros contenedores como los asociativos: *multiset, set, map, multimap*.

18.1. `vector` container class

(Parte en inglés extraída de http://www.cplusplus.com/reference/vector/vector/)

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for

growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with amortized constant time complexity (see push_back).

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and forward_lists.

La clase vector (en realidad es una plantilla como veremos más adelante) permite una forma muy sencilla y potente de manipular vectores de cualquier tipo base.

Dispone de métodos que la hacen totalmente funcional y el concepto de **iterador** que permite recorrer y acceder a cada elemento del vector de forma sencilla. Este iterador es del tipo *Random Access Iterator* y funciona como un puntero y permite todas las operaciones de la aritmética de punteros. Al ser posiciones contiguas de almacenamiento se permite el incremento de varias posiciones, indexación mediante corchetes, resta, comparación de posiciones con operadores <, >, etc.

En la web de la asignatura se han preparado algunos ejemplos para estudiar su funcionaminto (código de los ejemplos: ejemplo1Vector.cpp, ejemplo2Vector.cpp, ejemplo3Vector.cpp, ejemplo4Vector.cpp). Es importante que se analicen estos ejemplos, se hagan pequeñas modificaciones, se compilen y se prueben para entender el uso de esta clase.

Por tanto, el/la estudiante debe descargar estos archivos, compilarlos, ejecutarlos y analizar su código hasta su comprensión como ejercicio adicional de prácticas de la asignatura.

Importante: No es necesario fijar el tamaño de un vector inicialmente ya que puede ir cambiando en tiempo de ejecución, pero si se quiere un vector de un tamaño fijo hay disponible un constructor que recibe como primer parámetro el tamaño y como segundo el valor inicial de los elementos. Entonces, si se declara un vector dentro de una clase, la llamada a este constructor debe hacerse en el constructor de la clase y no en su declaración. En C++ los constructores de las clases base y de las clases que se usan como miembros deben invocarse en el constructor de la clase que se está declarando y no antes.

18.2. `list` container class

La clase *list*, o mejor dicho y como veremos más adelante, la plantilla (template) *list* es una secuencia de datos doblemente enlazados de cualquier tipo base que permite insertar y borrar en cualquier lugar dentro de la secuencia. Usa memoria dinámica no necesariamente contigua.

Dispone también de métodos que facilitan todas las operaciones sobre listas y el concepto de **iterador** que permite recorrer y acceder a cada elemento de la lista de forma sencilla. Este iterador es un *Bidireccional Iterator* y, como para acceder al siguiente elemento o al anterior se usan los punteros de cada elemento, no es posible sino la operación de incremento y decremento para moverse por la lista, no permitíendose accesos directos, ni incrementos o decrementos de varias posiciones ni operadores de comparación como < o > (para comparar solo está permitido ! =).

En la web de la asignatura se han dejado algunos ejemplos que el alumno deberá descargar, compilar, ejecutar, analizar su comprensión y hacer pequeñas modificaciones para entender bien su funcionamiento.

18.3. Adaptadores de contenedores

Existen adaptadores de los contenedores. Para adaptar a una pila los contenedores *vector*, *list* y *deque* está el adaptador **stack**. Proporcionan las operaciones: push, pop, top, empty y size.

Para adaptar a una cola los contenedores *list* y *deque* está el adaptador **queue**. Proporcionan las operaciones: push, pop, front, back, empty y size.

Para adaptar a una cola de prioridades los contenedores *vector* y *deque* está el adaptador **priority_queue** (igual que la cola pero permitiendo inserciones ordenadas). Proporcionan las operaciones: push, pop, top, empty y size. so La STL dispone de una gran variedad de clases útiles muy utilizadas entre programadores. Conviene su estudio para poder utilizarlas cuando sea necesario, lo cual es muy conveniente por el ahorro de tiempo de desarrollo.

19. Miembros estáticos

19.1. Variables miembro `static`

Cada vez que se declara un objeto se reserva espacio en memoria para sus variables miembro, es decir, cada objeto tiene una zona independiente de memoria en la que se almacenan sus propias copias de dichas variables.

Las variables miembro de un objeto son diferentes a las de otro. Pero una variable miembro de una clase declarada como `static` es común a todos los objetos de esa clase, es decir, todos los objetos comparten la misma variable. Solo hay una copia de dicha variable.

Algunas características de las variables `static` son:

- Es recomendable no usar el nombre de un objeto para acceder a ella mediante el operador punto, ya que la variable pertenece a todos los objetos de esa clase (por eso también recibe el nombre de variable de clase). Por eso es siempre recomendable usar la notación: `nombre_de_la_clase::variable_static` (aunque no sea obligatorio).
- No se pueden inicializar en el constructor, ya que se inicializarían muchas veces, cada vez que se definiera un nuevo objeto. Tampoco en la declaración (esto solo pueden hacerlo los datos `static const`). Tampoco en el iniciador del constructor (esto se hace solo con los miembros constantes). Se inicializa exclusivamente en un punto del ámbito global del programa de la forma:
- Las variables miembro estáticas se deben declarar dentro de la definición de la clase. No es posible hacerlo fuera de ella, ni usar una variable estática sin que esté previamente definida allí.

```
class A{
.....
static int i;
.....
};


int A::i=4;  // sin poner static de nuevo. int A::i; inicializa a cero

int main(void)
{
```

```
.....
}
```

OJO: si no se inicializa explícitamente la variable estática, es automáticamente inicializada a 0 al crearse el primer objeto.

- En el momento en que la ejecución del programa llega a la inicialización de la variable estática, es cuando se reserva espacio en memoria para ella y se inicializa (se inicializa automáticamente a cero por defecto). Se crea al inicializarse incluso si no hay ningún objeto creado de su clase.
- Es necesario poner el tipo delante en la inicialización.
- Puede ser pública o privada, dependiendo del uso que queramos darle en el programa.

Un ejemplo típico del uso de variables miembro estáticas es para controlar el número de objetos de una clase que crea nuestro programa. O también, almacenar un valor útil para todos los objetos o que evoluciona en tiempo de ejecución dependiendo de todos los objetos.

19.2. Constantes miembro `static`

La misma idea que las variables estáticas es válida para las constantes estáticas salvo que su valor una vez establecido no puede modificarse nunca en la ejecución del programa.

Estas constantes si pueden inicializarse en la definición de la clase como en el siguiente ejemplo:

```
class A
{
public:

static const int i = 4;

...
};
```

- Aunque tampoco se pueden inicializar en el constructor de la clase.
- Tampoco se pueden inicializar con iniciadores de miembros (veremos lo que son más adelante).
- Si no se inicializa en la definición (lo más conveniente) puede inicializarse en un punto global (en este caso no es necesario poner `static` en la inicialización) como en el siguiente ejemplo:

```
class A
{

public:

static const int i;
```

```
...
};

....


const int A::i = 4;

....
```

19.3. Funciones miembro `static`

Una función miembro estática es aquella que se declara con la palabra reservada `static` delante de su definición (solo se pone `static` en la definición, en el .h) y que puede invocarse sin necesidad de declarar un objeto de dicha clase.

Las funciones estáticas solo pueden acceder a variables estáticas de la clase. No pueden manejar variables que no sean estáticas, y si se desea que manipulen variables no estáticas del objeto, debe pasarse el objeto de forma explícita como parámetro. Además, las funciones estáticas no tienen puntero *this* (veremos el puntero *this* más adelante).

Las funciones estáticas, lógicamente, no pueden invocar funciones de la clase que no sean a su vez estáticas.

Para invocar una función estática debe hacerse de la forma:

```
nombre_de_la_clase::funcion_static()
```

Claro que también pueden invocarse con el correspondiente objeto como cualquier otra función miembro, pero esto no es conveniente.

Estas funciones también se suelen usar cuando se quieren agrupar operaciones relacionadas entre sí que tienen sentido aun sin haber objetos declarados de esa clase.

20. Manejo de excepciones

Las excepciones son un mecanismo adicional para el control de errores en la ejecución del programa. Cuando un segmento de código dentro del bloque *try* lanza una excepción, ésta puede ser capturada en el bloque *catch* correspondiente y no abortar el programa de forma descontrolada, sino que se ejecutará el código dentro del bloque *catch*. Si dentro de una de nuestras funciones (incluso un constructor) queremos lanzar una excepción usaremos *throw*. Esto nos permite lanzar un error en cualquier momento y no solo como valor devuelto o parámetro dato-resultado.

Las excepciones suelen usarse para el control de errores en tiempo de ejecución inevitables o que no pueden controlarse de otra forma, aunque el manejo de excepciones es cada vez más utilizado en todos los lenguajes de programación modernos.

El manejo de excepciones es una parte de C++ (y de otros lenguajes) dedicada específicamente a la gestión de errores.

20.1. Manejo básico

```
try {
    ... // código que se quiere monitorizar
```

```
    }
catch (int &i)
    {
    ...
    }
catch (float &i)
    {
    ...
    }
catch (double &i)
    {
    ...
    }
catch(...)
    {
    ...
    }
```

El código que se monitoriza puede lanzar excepciones de varios tipos, para cada uno se puede habilitar un *catch()* con el tipo correspondiente. Cuando haya una excepción de ese tipo, se ejecutará el código dentro del catch correspondiente y no se ejecutarán el resto si los hay.

catch(...) captura la excepción, sea del tipo que sea, si no ha sido capturada por algún catch anterior. Es decir, captura todas las excepciones sean del tipo que sean. Si no se captura la excepcion el programa aborta la ejecucion inmediatamente.

Cada función puede lanzar una serie de excepciones que viene dada por la **especificación de excepciones**. Es conveniente conocer la especificación de excepciones de cada función que se utilice.

Veamos un ejemplo sencillo:

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

int main(void)
{
int i(1);
float f(1.2);
double d(2.3);


cout << "Ejemplo 1 de Excepciones en C++ \n";
// En la siguiente línea cambio f por i, d
cout << "con una excepcion de tipo: " << typeid(f).name() << endl;

try {
    cout << "Estoy dentro del bloque try \n";
    throw f;
    cout << "Estoy despues del throw \n";
    }
```

```
catch (int i)
    {
    cout << "Error capturado, su valor entero es " << i << endl;
    }
catch (float i)
    {
    cout << "Error capturado, su valor float es " << i << endl;
    }
catch (double i)
    {
    cout << "Error capturado, su valor double es " << i << endl;
    }
catch(...)
    {
    cout << "Error capturado por defecto \n";
    }
//Si no se captura la excepcion el programa aborta la ejecucion inmediatamente
cout << "Fin \n";
}
```

Este y otros ejemplos que usan excepciones se encuentran en la web de la asignatura para su estudio por parte de los alumnos.

20.2. Ejemplo básico de excepciones

Ver código del fichero: ejemplo1Ex.cpp

20.3. Excepciones con *enum*

Ver código del fichero: ejemplo2Ex.cpp

20.4. La clase *exception*

Ver código del fichero: ejemplo3Ex.cpp

20.5. Aspectos relacionados

Podemos controlar los errores dentro de nuestros programas de tres formas diferentes:

- Mediante los valores devueltos en las funciones. Útil y la forma más rápida y cómoda, sobre todo cuando la función es equivamente a una función matemática (max(), min(), media(), suma(), etc.) que devuelve un valor correspondiente a un cálculo.
- Mediante parámetros (dato-resultado, por referencia o usando referencias) que se modifican en la función con un código de error. En ciertas situaciones es necesario, por ejemplo en funciones que devuelven resultados de operaciones o cálculos y no pueden devolver además el código de error.
- Y ahora hemos estudiado que mediante el uso de excepciones. Gestión de errores graves que no se pueden controlar de otra forma, como los errores devueltos por el SO, o errores que consideramos graves. También se usan las excepciones cada vez más para el control de todo tipo de errores.

Cuando estudiamos un tipo de dato o una función en un lenguaje de programación tendremos que ver si el manual nos describe las posibilidades de manejo de excepciones de ese tipo o función para analizar su uso en nuestros programas.

21. El puntero *this*

Cuando se invoca a una función miembro se le pasa automáticamente el puntero del objeto que la ha invocado (esta es la forma interna en que la función miembro puede acceder a los datos de ese objeto concreto). Ese puntero se denomina en el interior de dicha función como el puntero *this*.

El puntero *this* es un **parámetro implícito** a todas las funciones miembro de una clase, es decir, que todas las funciones miembro de una clase lo reciben automáticamente (a excepción, como hemos visto anteriormente, de las funciones estáticas).

Teniendo en cuenta esto, se puede acceder a los miembros de una clase escribiendo directamente su nombre, o bien, antecediendo `this->`
Veamos el siguiente ejemplo:

```
class X
{
private:
  int i_;
public:
  void fun(int a);
....
};

void X::fun(int a)
{

//accedo al dato 'i_'  mediante:

i_=a;

// o bien

this->i_=a;
}
```

En el ejemplo ambos accesos a *_i* son correctos, lo que ocurre que es más breve el primer uso por lo que es el habitual. Esta es una introducción a lo que significa el puntero *this*. Posteriormente en este curso veremos otros usos de *this* más interesantes.

22. Sobrecarga de operadores

La forma general de una función *operator* es:

tipo nombre-clase:: `operator#`(*lista argumentos*)

`#` se sustituirá por el símbolo del operador a sobrecargar (+, -, *, =, etc.).

Veamos un ejemplo de una clase para un punto en el plano a la que se le definen varios operadores.

```
class Punto{
private:
  int coordx_, coordy_;
public:
  Punto(int x, int y){coordx_=x; coordy_=y;};
  int getX(){return coordx_;};
  int getY(){return coordy_;};
  Punto operator=(const Punto &p);
  Punto operator+(const Punto &p);
  Punto operator++(void); // para ++p
  Punto operator++(int);  // para p++
};

Punto Punto::operator=(const Punto &p)
{
coordx_ = p.coordx_;
coordy_ = p.coordy_;
return *this;
}

Punto Punto::operator+(const Punto &p)
{
Punto aux(0,0);
aux.coordx_ = coordx_ + p.coordx_;
aux.coordy_ = coordy_ + p.coordy_;
return aux;
}

Punto Punto::operator++(void) //prefijo ++p
{
++coordx_;
++coordy_;
return *this;
}

Punto Punto::operator++(int) //postfijo p++
{
Punto temp=*this;
++coordx_;
++coordy_;
return temp;
}
.......

int main(void)
{
Punto a(1,2),b(10,10),c(0,0);
c=a+b;
cout << "c.x= " << c.getX() << "c.y= " << c.getY() << endl << endl;
```

```
c=a++;
cout << "a.x= " << a.getX() << "a.y= " << a.getY() << endl;
cout << "c.x= " << c.getX() << "c.y= " << c.getY() << endl << endl;
c=++a;
cout << "a.x= " << a.getX() << "a.y= " << a.getY() << endl;
cout << "c.x= " << c.getX() << "c.y= " << c.getY() << endl << endl;
}

$ g++ clase_punto.cc
$ /a.out
c.x= 11 c.y= 12

a.x= 2 a.y= 3
c.x= 1 c.y= 2

a.x= 3 a.y= 4
c.x= 3 c.y= 4
```

Aspectos a considerar en este ejemplo:

- En el operador + no es posible devolver una referencia a una variable local, ya que la variable local deja de existir al salir de la función y la referencia apuntaría a una variable inexistente con el consiguiente error al acceso, etc..
- El objeto que llama a la función es siempre el operando izquierdo, el operando derecho se pasa como parámetro.
- El operador unario sin parámetros es para el operador prefijo (++b) y el que recibe un entero es para el postfijo (b++). Observar que cada versión devuelve una cosa diferente según el comportamieno que ya conocemos que este operador, que no es igual prefijo que postfijo.
- El operador ternario `?:` no se puede sobrecargar.
- Observar que en el operador + no se modifican los operandos manteniendo así la semántica propia del operador +.
- Los operadores devuelven un objeto de la propia clase *Punto*. Así se pueden construir expresiones complejas como las tres sumas o las tres asignaciones del ejemplo. (El operador = también devuelve un objeto). Se podría hacer usando una referencia también para ahorrar tiempo de ejecución y memoria (recordar las ventajas del uso de referencias).
- La prioridad de los operadores es la establecida por C/C++ y no se puede modificar.
- No se puede modificar el número de operandos (parámetros) de los operadores.
- No se pueden sobrecargar: . :: .* ?

Todos los objetos de cualquier clase tienen un operador de asignación por defecto que copia las variables de un objeto en otro. Cuando los objetos manejan memoria dinámica, el operador por defecto no funciona correctamente ya que al copiar las variables, el objeto

original y el copia apuntarían a la misma memoria dinámica. Es necesario definir un operador específico que haga la copia y la reserva de memoria dinámica para el objeto copiado, cosa que se estudia en la la sección **constructores de copia**.

22.0.1. El operador = y el constructor de copia

Cuando una clase tiene constructores de copia, es mejor que el operador = reciba y devuelva referencias para evitar la invocación adicional del constructor de copia, lo cual puede dar lugar a retardos y situaciones imprevistas.

Recordar que si una función recibe objetos por valor (o los devuelve) se hace una copia del objeto pasado o devuelto y por tanto se invoca al constructor de copia. Si se usan referencias no ocurre esto.

Por ello, el operador = debería en estos casos declararse siempre usando parámetros por referencia y devolviendo referencias:

```
.....
MiClase& MiClase::operator=(const MiClase &e);
.....
```

- Esta función será la que se ejecute cuando se use el operador de asignación en la clase *MiClase*.
- Recibe una referencia y por tanto no se invoca al constructor de copia. Además es más eficiente al no hacer copias adicionales e innecesarias de los objetos.
- Es constante (opcional pero conveniente) porque el operando de la derecha no se modifica y así se hace mayor énfasis autodocumental en este hecho. También puede ser constante el valor devuelto porque se usará como operando en la siguiente invocación (si hay asignación múltiple) y no se modificará.
- El objeto/operando de la izquierda del igual (al que apunta *this*) se modifica pero la función devuelve una referencia para que en caso de asignación múltiple no se dupliquen objetos innecesariamente y sea más eficiente (tampoco se invoca el constructor de copia). Es decir, para que pueda hacerse eficientemente lo siguiente:

  ```
  obj1 = obj2 = obj3;
  ```

- Hay que tener cuidado con ciertas cosas aquí. Por ejemplo, si el objeto de la izquierda ya está usando memoria dinámica, habrá que liberarla antes de hacer la copia. Para ello se puede comprobar si los punteros a la memoria dinámica que usa el objeto de la izquierda son distintos de NULL.
- También habrá que evitar situaciones como esta:

  ```
  obj1 = obj1;
  ```

  en la que, si aplicamos el punto anterior, primero se liberaría la memoria del objeto de la izquierda con lo cual se destruiría obj1, y el resultado podría ser imprevisible. Para evitar esto es bueno comprobar antes de nada en la función operador de asignación si el objeto de la izquierda es el mismo que el de la derecha para, en ese caso, no hacer nada.

Dicho esto, a veces nos encontraremos con operadores sobrecargados que reciben y devuelven referencias a gusto del programador, que puede decidir hacer el código más eficiente o permitir que se modifique el objeto de la izquierda como se dijo en la sección 13.3.

23. Variables miembro constantes. Iniciación necesaria

Si una clase tiene miembros constantes deben ser inicializados mediante iniciadores. No se pueden inicializar en su declaración (esto sólo sucede al iniciar constantes estáticas, ver apartado 19), ni siquiera dentro del constructor de la clase se pueden inicializar. Por tanto se inician **solo mediante iniciadores**.

- Si se trata de un **miembro que es dato constante**, éste se debe iniciar después de la lista de parámetros del constructor (si los tiene), como en este ejemplo:

```
class A{
private:
  const int i_;
public:
  A(int edad, float peso): i_(7);
};
```

- Los dos puntos y el iniciador se pueden poner como en el ejemplo, o bien en la cabecera del cuerpo de la función.
- Si se trata de **miembros objetos de otras clases** que necesitan ser inicializados mediante el paso de parámetros a su constructor, se debe proceder de manera análoga. No se pueden inicializar en su declaración, sino mediante iniciadores en el constructor de la clase en la que están definidos, como en el siguiente ejemplo:

```
class B{
public:
  .....
  B(int x){_n=x;};
  .....

private:
  int n_;
  .....
};

class A{
private:
  const int i_;
  const B obj1_;
  B obj2_;
public:
  A(): i_(7), obj1_(8), obj2_(6);
};
```

- El iniciador se puede usar tanto para inicializar miembros constantes, como *i* o *obj1*, como para pasar al constructor del objeto los parámetros necesarios, como en *obj2*, aspecto este último que se estudiará específicamente en el apartado siguiente (apartado 24).
- Existen otros miembros constantes que son las **constantes estáticas** que se estudian en el apartado 19.

24. Iniciadores de miembros. Objetos como miembros

Una clase puede contener otros objetos como miembros de ella. Si estos objetos necesitan ser inicializados, ésto debe hacerse con **iniciadores de miembros**.

Esto ocurre con miembros objetos sin constructores por defecto y miembros constantes. En el siguiente ejemplo supongamos que ni la clase `Pila` ni la clase `Vector` tienen constructor por defecto y siempre hay que pasarles el tamaño al crear los objetos. La clase `X` tiene la siguiente declaración:

```
class X{
  private:
    const int i;
    Pila c;
    Vector v;
  public:
    X(int a, int b):i(7),c(a),v(b){ ... };
  ...
};
```

- No hay ninguna otra forma de iniciar dichos miembros, ya que: en cuanto a las constantes, una constante debe iniciarse al declararse y no en una línea del código del constructor, por tanto el iniciador es la única opción prevista por C++. Por otro lado, un objeto no puede iniciarse en el constructor porque podría darse el caso no deseable de que se usara en el constructor antes de ser iniciado por su propio constructor.
- Las constantes que se pueden inicializar en la declaración de una clase son solo las constantes estáticas (`static const`) que se estudian en el apartado 19.2.
- Otros miembros no constantes de la clase también pueden ser inicializados de esta forma en el constructor. Esto también se llama *iniciadores de atributos de la clase*.
- El código de los iniciadores se ejecuta antes que el del constructor.
- En el apartado 23 (Miembros Constantes) también se trata este aspecto y se describe un nuevo ejemplo en el marco del calificador *const*.

25. Funciones *friend*

Una función *friend* es aquella que, **no siendo función miembro** de la clase, puede acceder a su parte privada.

Este mecanismo, que puede parecer una contradicción a la filosofía del encapsulamiento, se permite para que dos clases puedan compartir funciones, de utilidad para ambas, que acceden a sus datos privados en orden a mejorar la eficiencia al programar.

```
class Circulo;

class Linea{
private:
  int x1_, y1_, x2_, y2_;
```

```
  int color_;
public:
  friend int mismoColor(Linea l, Circulo c);
  void dibuja();
  void oculta();
  void longitud();
};

class Circulo{
private:
  int centrox_, centroy_, radio_;
  int color_;
public:
  friend int mismoColor(Linea l, Circulo c);
  void dibuja();
  void oculta();
  void longitud();
};


//verdadero si linea y circulo son del mismo color
int mismoColor(Linea l, Circulo c)
{
if (l.color_==c.color_)
   return 1;
else
   return 0;
}
```

- Es necesaria la **declaración anticipada** de la clase *circulo*. Esto se aprecia en la primera línea del ejemplo.
- Se pone 'friend' solo dentro de la declaración de la clase. Si se pone en el cuerpo de las funciones el compilador lanzará un error.
- El acceso a los datos privados se hace a través de los objetos que recibe la función y el operador punto.
- La amistad es concedida, no tomada; para que una función pueda acceder a la parte privada de una clase, esa clase debe concederle ese privilegio declarándola función amiga.
- Se podrían haber declarado funciones *public* que acceden al color y luego compararlo, pero provoca llamadas adicionales, y la función final de comparación estaría fuera de las clases y no dentro, como debe ser.

26. Sobrecarga de operadores con funciones *friend*

En muchos casos (la mayoría) no es necesario ni beneficioso utilizar una función *friend* para sobrecargar operadores. Pero **existen casos en los que es necesario sobrecargar operadores con funciones** *friend*, como veremos a continuación:

Suponer que existe una clase sencilla denominada *Contador* que mantiene un entero a modo de contador y solo admite modificaciones con el operador ++ y el operador - -. Pero al ser entero el contador, podrá insertarse en operaciones con enteros.

Si tenemos:

```
int Contador::operator+(int x) {
  return get()+x;
}
```

podríamos hacer:

```
Contador c; // Se inicializa a 0
c++;
int i;

...
i = c + 10;
```

pero la expresión

```
i = 10 + c;
```

sería incorrecta porque "10" es un entero y los enteros no tienen definida una operación "+" que reciba como parámetro un objeto de tipo *Contador* como es "c".

La forma de solucionar este problema es con dos funciones *friend*:

```
class Contador{
public:
  friend int operator+(Contador ob, int i); // esta podría no ser friend
  friend int operator+(int i, Contador ob);
.....
private:
  int valor_;
};
//La primera funcion friend
int operator+(Contador ob, int i) {
  return ob.get() + i;
}
//La segunda funcion friend
int operator+(int i, Contador ob) {
  return i+ob.get();
}
......
```

Recordar que las funciones *friend*, al no ser miembros, no reciben el puntero `this`.

Ahora el siguiente código es correcto y no plantea conflictos porque el compilador los resuelve utilizando el mecanismo de la sobrecarga de funciones:

```
int main(void)
{
Contador a;
```

```
a=10; //suponiendo también sobrecargado el operador =
a = 10 + a;
a = a + 12;
a.get();
}
```

27. Control de acceso. Elementos *public:*, *private:* y *protected:*

En la declaración de una clase pueden existir miembros *public, private*, o *protected*. De esta forma se controla el acceso a dichos miembros por parte de los usuarios de esa clase y por parte de las clases derivadas de esa clase.

- Un miembro **public** de una clase puede ser accedido desde el exterior de la clase. El cliente de la clase puede acceder a él.
- Un miembro **private** solo puede ser accedido desde el interior de la clase, es decir desde las funciones miembro de la propia clase (y también desde dentro de una función *friend* de dicha clase).
- Igual que los miembros private, un miembro **protected** solo puede ser accedido desde el interior de la clase, es decir desde las funciones miembro de la propia clase (y también desde dentro de una función *friend* de dicha clase).

27.1. Miembros protegidos

La sección *protected* de una clase se usa para que los desarrolladores de las clases derivadas puedan acceder a dicha sección de modo que puedan sacar mayor rendimiento/beneficio con dicho acceso. Por ejemplo cuando una clase base manipula un vector, puede dejar que dicho vector sea también manipulado por los desarrolladores de las clases derivadas para que accedan también a sus datos.

Las secciones public y private no son suficientes en muchos casos, sobre todo si se usan clases derivadas. En este caso existen dos tipos de usuarios de una clase: el público en general y los desarrolladores de clases derivadas. Los miembros protegidos de una clase tratan de abastecer específicamente a los desarrolladores de clases derivadas con miembros que solo ellos puedan utilizar.

Se pueden declarar funciones miembro protegidas en previsión de que puedan ser usadas por clases derivadas.

28. Tipos de herencia

Existen tres formas de heredar de una clase según el siguiente formato:

`class` *nombre-clase-derivada* `:` *acceso nombre-clase-base*

Independientemente del *acceso* debemos tener en cuenta lo siguiente:

1. Una clase derivada puede acceder a la parte *public* y *protected* de su clase base.

2. La parte privada de una clase no puede ser accedida por nadie fuera de la propia clase (excepto funciones *friend*).

Dependiendo de si el tipo de *acceso* es *public*, *private* o *protected* tenemos las siguientes clases de herencia:

1. **public:** los elementos *public* de la clase base siguen siendo *public* en la clase derivada, y los *protected* siguen siendo *protected* en la clase derivada. Los usuarios de la clase derivada podrán acceder únicamente a la parte pública de la clase base y derivada.
2. **protected:** los elementos públicos de la clase base y los elementos *protected* pasan a ser *protected* en la clase derivada.
3. **private:** los elementos públicos y *protected* de la clase base pasan a ser privados en la clase derivada y pueden ser accedidos desde la clase derivada pero no se heredan a otras clases ni pueden ser accedidos por los usuarios de la clase derivada. Es el modo por defecto.

La herencia *public* es la más habitual y la que se utiliza en la mayoría de los casos. La herencia *private* es la herencia por defecto si no se indica ninguna en la declaración.
Ejemplo:

```
class Hija: X // la clase hija deriva en modo privado de la clase X

class Hija:public X,Y // hija deriva en modo público de X y privado de Y
```

28.1. Herencia *public*: ES-UN, ES-UNA

Una clase derivada de otra utilizando este mecanismo mantiene una relación tal que un objeto de la clase derivada **ES-UN** objeto de la clase base al mismo tiempo.

- Por ejemplo, si la clase *bola* hereda de la clase *esfera*, podemos decir que una bola es una esfera, y todas las operaciones aplicables a esferas son aplicables a bolas.
- Si una función espera como argumento una esfera, se le puede enviar una bola.
- La clase *bola*, además, tendrá un conjunto de operaciones específicas, no aplicables a la clase *esfera*, como por ejemplo, *nombre-juego* (en el que usar la bola: billar, tenis, etc).
- La bola, al ser una esfera, puede acceder a su parte publica y protected, por si necesita acceso a datos privados, pues *se lo ha ganado*, ya que es como una esfera y tiene el mismo privilegio.

En la figura 2 puede verse un esquema con los accesos permitidos a los objetos de las clases A, B y C en el caso de herencia pública.

28.2. Herencia POR-MEDIO-DE protegida

Cuando heredemos de forma protegida, la clase derivada recibe la parte pública de la clase base en su zona protegida. Este mecanismo puede apreciarse mejor en la siguiente figura.

En la figura 3 puede verse un esquema con los accesos permitidos a los objetos de las clases A, B y C en el caso de herencia protegida.

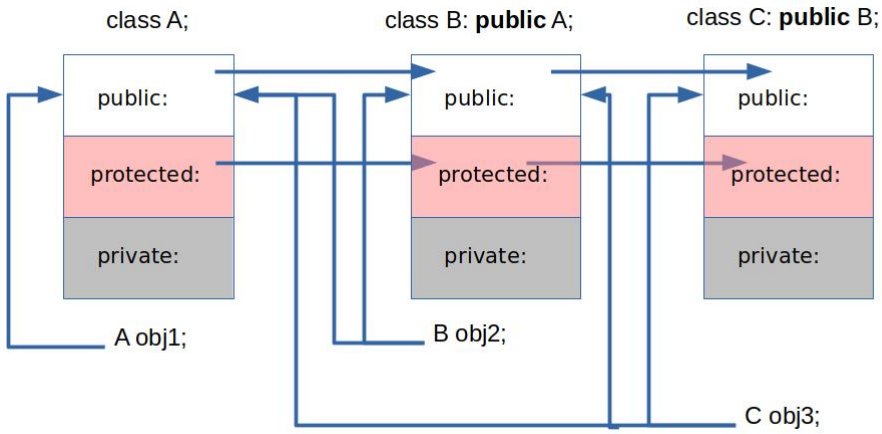

Figura 2: Herencia public

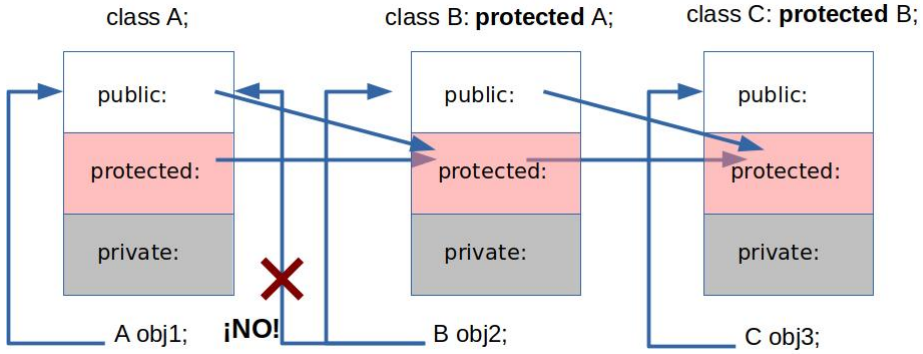

Figura 3: Herencia protected

28.3. Herencia POR-MEDIO-DE privada

Un ejemplo típico de herencia privada es la relación entre una hipotética clase *Lista* y su derivada la clase *Pila*.

La clase *pila* puede construirse POR-MEDIO-DE la clase *lista*. La clase *lista* dispone de operaciones y datos que permiten hacer cualquier operación sobre listas, y un grupo de ellas permiten montar una pila. Pero no queremos que los usuarios de las pilas sepan esto, o quieran modificar la pila mediante el uso de funciones de la lista (y posiblemente violar su integridad). Por eso decimos que todos los elementos de *lista* son privados en *pila*, y solo se puede usar en la clase *pila* y no en clases derivadas de *pila*. Para usar esta pila especial, solo se podrá usar el interfaz público de la clase *pila*.

Todos los elementos de *lista*, ya sean *public* o *protected* pasan a ser privados en *pila*.

En la figura 4 puede verse un esquema con los accesos permitidos a los objetos de las clases A, B y C en el caso de herencia privada.

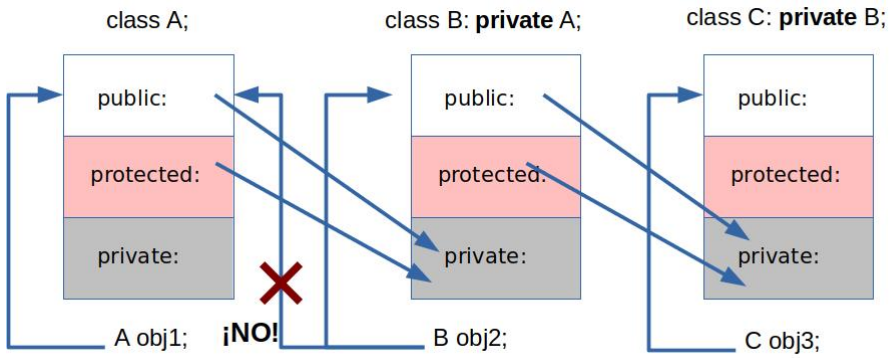

Figura 4: Herencia private

28.4. La relación CONTIENE-UN, CONTIENE-UNA

Si dentro de una clase se declara un dato de cualquier tipo o de cualquier clase, aquí no hay una relación de herencia. Lo que ocurre es que alguno de los componentes de esa clase es un objeto de otra clase.

A veces esto se confunde con la herencia cuando no lo es.

29. Herencia múltiple

Una clase puede heredar de varias clases a la vez:

```
class Z: public X, public Y{

.....

};
```

- Los objetos de Z pueden llamar a funciones de X e Y.
- Los constructores se ejecutan en el mismo orden que aparecen en la declaración de Z.
- Si los constructores reciben parámetros, debe haber un constructor en Z donde se hagan corresponder los argumentos del constructor de Z con los de los constructores de las clases base.

```
Z::Z(int x, int y) : X(x), Y(y)
{
.......
}
```

En el ejemplo x se pasa al constructor de la clase X e y se pasa al constructor de la clase Y.

29.1. Aspectos relacionados

Si tenemos el código:

```
class hija:public X, public Y
```

hay ambiguedad si existen los métodos X::f1() y Y::f1(). La solución es hacer las llamadas siguientes, suponiendo que tenemos un objeto *h* de la clase *hija*: `h.X::f1()` o bien `h.Y::f1()`.

30. Un objeto es una variable más

Un objeto puede pasarse a una función por valor o por referencia. La forma de hacerlo y el resultado es igual que para cualquier variable de C/C++.
Ejemplo:

```
class X
{
private:
  int a_;
public:
  X(){a_=5;};  // tres funciones en linea
  int set(int i){a_=i;};
  int get(){return a_;};
};

void f(X obj){obj.set(8);}

int main(void)
{
  X x;
  cout << x.get();
  f(x);
  cout << x.get();
}
```

El resultado de la compilación y ejecución de este programa, suponiendo que se guarda en `prueba.cpp`, es:

```
$ g++ prueba.cc -o prueba
$ ./prueba
5
5
```

Si la función se hubiera definido de esta forma:

```
void f(X &obj){obj.set(8);}
```

el resultado hubiera sido:

```
$ g++ prueba.cc -o prueba
$ ./prueba
5
8
```

30.1. Vectores o matrices de objetos

Pueden construirse vectores o matrices de objetos de igual forma que de cualquier otro tipo. El acceso es como a un vector de *struct*.

30.2. Punteros a objetos

Pueden declararse punteros a objetos y se usan exactamente igual que el resto de punteros en C o C++. Por ejemplo:

```
Fecha f, *p;
p=&f;
cout << p->getDay();
```

30.3. Referencias de objetos

Los objetos pueden tener referencias a ellos exáctamente igual que cualquier otra variable de cualquier tipo.

31. Polimorfismo

El concepto de polimorfismo es uno de los más importantes de la POO. Podríamos definirlo como: *proceso mediante el cual se puede acceder a diferentes implementaciones de una función utilizando el mismo nombre*. Se dice que este proceso atiende al esquema: **una interfaz, múltiples métodos**.

Poder mantener el mismo nombre para una operación a lo largo de un proyecto software, y enlazar ese nombre con la función concreta que debe ejecutarse en cada momento, es algo que simplifica y facilita mucho la programación. De ahí que el polimorfismo sea interesante.

Existen dos tipos de polimorfismo: estático y dinámico

31.1. Polimorfismo estático

El polimorfismo **estático** o "en tiempo de compilación" es aquel cuyos parámetros quedan fijados y determinados por el compilador en tiempo de compilación.

En este caso nos referimos a la sobrecarga de funciones y a la sobrecarga de operadores que ya hemos visto en apartados anteriores.

También las plantillas de función y las plantillas de clase, que veremos más adelante, son polimorfismo estático, aunque en este caso también se denomina "polimorfismo paramétrico".

31.2. Polimorfismo dinámico

El polimorfismo dinámico o "en tiempo de ejecución" es aquel cuyos parámetros se fijan y se concretan durante la ejecución del programa y no antes.

En este caso nos referimos al uso conjunto de clases derivadas y funciones virtuales, como veremos en esta sección.

Este tipo de polimorfismo también se denomina "vinculación dinámica" ya que la vinculación entre parámetro y el tipo o la clase a la cual se vincula el parámetro se hace en tiempo de ejecución de forma dinámica y, en una misma ejecución, se podrá vincular unas veces a un tipo y otras a otro.

31.3. Funciones virtuales: el interfaz genérico

En un sistema software es habitual definir algunas clases a partir de las cuales derivar otras formando una jerarquía de clases que faciliten el desarrollo posterior del sistema y su reutilización y extensión.

Veamos también como se puede imponer una interfaz a las clases derivadas de forma que todas ellas compartan una misma interfaz aun cuando algunas de las funciones que forman esa interfaz no se puedan concretar hasta que las clases derivadas no estén totalmente definidas. Para ello se usan las funciones virtuales.

Veamos un ejemplo:

```
class Figura{
protected:
  double x_, y_;
public:
  void setDim(double i, double j){x_=i;y_=j;};
  virtual double area(){cout << "funcion no definida aqui";return 0.0;};
};

class Triangulo : public Figura{
public:
  double area() override {
  // x_= base, y_= altura
  return(x_*y_/2);
  }
};

class Cuadrado : public Figura{
public:
  double area() override {
  // x_,y_ lados
  return(x_*y_);
  }
};

int main(void)
{
int opcion;
Figura *p;
cout << "elige figura \n"
     << "1.- triangulo \n"
     << "2.- cuadrado \n";
cin >> opcion;
if(opcion==1)
  {
  Triangulo t;
  p=&t;
  }
else
  {
  Cuadrado c;
```

```
  p=&c;
  }
p->setDim(3.0,4.0);
cout << "\n Area = " << p->area() << endl;
}
```

Comentarios:

- *p− >setDim(3.0,4.0)*, llama al método de la clase base.
- *p− >area()*, llama unas veces al método de la clase *Triangulo* y otras al método de la clase *Cuadrado*. Es una llamada a una función polimórfica. La llamada se resuelve en tiempo de ejecución (vinculación dinámica).
- Este mecanismo permite que una clase base dicte el interfaz que va a tener cualquier clase que se derive de ella.
- Y también permite que la clase derivada pueda *redefinir* un método por sí misma adaptándolo a sus particularidades. De hecho, desde C++11, en la clase derivada es preferible el uso de `override` dejando `virtual` para la clase base.
- La clase base implementa las funciones virtuales a un nivel básico, o no las implementa, y las clases derivadas las redefinen adaptándolas a sus particularidades. Si no se redefinen en la clase derivada, mientras tanto, se usan las versiones de la clase base.
- Se proporciona una clase base genérica y de ahí se derivan las particulares y todas tienen el mismo interfaz, lo cual mejora mucho: entendibilidad, facilidad de uso, modificabilidad, etc.
- El **polimorfismo en tiempo de ejecución** se consigue mediante el uso de clases derivadas y funciones virtuales.

31.4. Clases abstractas

Las clases definidas con funciones virtuales se denominan **clases abstractas** (también clases diferidas) porque no se usan para definir objetos de ella, sino que se usan solo para definir un interfaz que deberán cumplir las clases derivadas.

Las clases abstractas están parcialmente definidas, ya que algunas de sus funciones no están implementadas. Su implementación se hará con posterioridad en las clases derivadas de ella. Por eso estas clases también reciben el nombre de **clases diferidas** en contraste con el resto de clases que es este aspecto son **clases efectivas**.

31.5. Otros aspectos de las funciones virtuales

- **Una función virtual es** aquella que se define como `virtual` en la clase base y que se redefine en la clase derivada.
- Cuando se accede a funciones virtuales con un puntero a la clase base que apunta a un objeto de una clase derivada, C++ resuelve esta situación **en tiempo de ejecución** y determina qué función llamar basándose en el objeto al cual apunta.
- Cuando se apunta desde el puntero base a diferentes objetos, **se ejecutan versiones diferentes** de la función virtual.

- Para declarar funciones virtuales en la clase base, se antecede la palabra `virtual`. En la clase derivada es opcional pero conveniente.
- Si los prototipos no son idénticos, no se consideran funciones virtuales, solo sobrecarga de funciones. Aunque lo lógico es que los prototipos sean iguales.
- Funciones de igual prototipo en clases derivadas se supondrán virtuales aunque no este la palabra `virtual` en la clase derivada (y si está en la clase base donde es obligatorio indicarlo).
- Si una clase derivada no redefine (invalida) una función virtual de la clase base, se ejecutará la de la clase base.

NOTA: Accesos desde un puntero a la clase base a otras funciones, a parte de las virtuales, de la clase derivada no tiene sentido porque si interesaran en la clase base estarían definidas allí y declaradas como virtuales.

31.6. Funciones virtuales puras

El formato de una función virtual *pura* es el siguiente:

`virtual` *tipo nombre_funcion(lista_parametros)*`=0;`

- La función *area()* no tiene sentido en la clase base *figura* porque no sabemos de qué figura concreta se trata y no podemos, por tanto, calcular su área. Pero si declaramos `area()` como virtual pura en *figura* obligamos a que las clases derivadas la redefinan, porque, por ejemplo, consideremos que esta función es fundamental para todas las clases derivadas.
- Las clases derivadas no pueden usar la función virtual pura de la base: deben **redefinirla** siempre.
- Para las clases bases así definidas no podemos definir objetos y usarlos porque tienen funciones no proporcionadas. Pero si se pueden definir punteros a ellas para hacer polimorfismo en tiempo de ejecución.
- También se dice que `area()` es un método diferido. Diferido porque en la clase base solo se especifica el nombre; su código, y por tanto lo detalles de lo que hace, son diferidos a la clase derivada.
- A veces, en C++ se denomina clase abstracta únicamente a la clase que tiene al menos una función virtual pura.

Ejemplo:

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

class Animal
{
private:
    string nombre_;
```

```
public:
    Animal(string nombre): nombre_(nombre){}
    string getNombre() { return nombre_; }
    virtual string habla() = 0;    //#####Función virtual pura
};

class Gato: public Animal
{
public:
    Gato(string nombre): Animal(nombre){}
    virtual string habla() { return "Miau"; }
};

class Perro: public Animal
{
public:
    Perro(string nombre): Animal(nombre){}
    virtual string habla() { return "guau"; }
};

int main()
{
Gato g("milu");
Perro p("boby");

cout << g.habla() << endl;
cout << p.habla() << endl;
}
```

Si no se declara la función habla() dentro de alguna de las clases derivadas obtenemos al compilar:

```
animal.cc: En la función ‘int main()’:
animal.cc:44:7: error: no se puede declarar que la variable ‘p’
     sea del tipo abstracto ‘Perro’
animal.cc:29:7: nota:   porque las siguientes funciones virtual
     son pure dentro de ‘Perro’:
animal.cc:15:25: nota:  virtual const char* Animal::habla()
```

31.7. Funciones virtuales inline

Mediante un puntero a la clase base, las funciones virtuales de clases derivadas se enlazan en tiempo de ejecución. Como las funciones inline son sustituidas por su código en tiempo de ejecución, dificilmente una función virtual puede ser inline.

Solo cuando una función virtual se invoca a través de un objeto de esa clase y no desde un puntero a su clase base se puede mantener inline ya que desde el objeto siempre va a llamarse a la misma función, en cambio, desde el puntero se puede enlazar unas veces con una función y otras con otra, pero en tiempo de ejecución y no en tiempo de compilación.

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

```
class B {
   public:
      virtual void s() {
         cout<<" In Base \n";
      }
};

class D: public B {
   public:
      void s() {
         cout<<"In Derived \n";
      }
};

int main(void) {
   B b;
   D d;
   B *bptr = &d;  // A pointer of type B* pointing to d
   b.s();         // B::s() called, inlined, called through object of class
   d.s();         // D::s() called, inlined, called through object of class
   bptr->s();     // D::s() called, not inlined, virtual function
                  //       is called through pointer.
   return 0;
}
```

31.8. Destructor virtual en clases abstractas. Declaración necesaria

Cuando tenemos una clase abstracta con funciones virtuales normalmente se declara un puntero a la clase abstracta y con ese puntero accedemos a objetos de clases derivadas de ella. Cuando hagamos delete de ese puntero, si queremos que se ejecute el destructor de la clase derivada que implementa la interfaz de la clase abstracta, la clase abstracta debe haber declarado un destructor virtual, o de otra forma no se ejecutará el de la clase derivada con un posible comportamiento erróneo del programa. Por eso siempre conviene declarar un destructor virtual en la clase abstracta por si acaso la clase derivada necesitara uno.

```
class Interface
{
   virtual void doSomething() = 0;
};

class Derived : public Interface
{
   Derived();
   ~Derived()
   {
      // Do some important cleanup...
   }
};

void myFunc(void)
```

```
{
   Interface* p = new Derived();
   // The behaviour of the next line is undefined.
   // default Interface::~Interface destructor,
   // not Derived::~Derived
   delete p;
}
```

Por tanto lo correcto sería:

```
class Interface
{
   virtual void doSomething() = 0;
   virtual ~Interface(){};
};
```

32. Plantillas

Las plantillas en C++ se utilizan para definir funciones y clases genéricas dando lugar a lo que dentro de la POO se llama "genericidad".

32.1. Plantillas de función o funciones genéricas (*function templates*)

Cuando existen operaciones con el mismo nombre suele utilizarse la sobrecarga de funciones para implementarlas y facilitar así la programación. Pero si además son idénticas en cuanto al código que ejecutan, las plantillas de función son mucho más útiles. Veamoslo en un ejemplo.

```
template <class T>
void print_vector(T *v, const int n)
{
for(int i=0;i<n;i++)
   cout << v[i] << " , ";
}

......

int main(void)
{
int a[5]={1,3,5,7,9};
float b[4]={5.6, 7.8, 3.9, 1.2};
char c[5]="hola";

cout << "vector de enteros";
print_vector(v,5);
cout << "vector de floats";
print_vector(b,4);
cout << "vector de char";
print_vector(c,4);
```

```
}
```

- *print_vector()* es una función genérica (o *template function*). Cuando el compilador crea una versión de esta función para un determinado tipo, se dice que es una especialización o instanciación de la función genérica o de la plantilla.
- T es el parámetro formal que indica el tipo genérico que manipula la función. Puede haber varios separados por coma: <class T1, class T2>. Puede tener cualquier nombre escogido por el usuario.
- Habitualmente se utiliza la palabra clave *class*, pero también se puede usar la palabra clave *typename*.
- Puede instanciarse a cualquier tipo del lenguaje o del usuario, con la condición de que permita las operaciones realizadas dentro de la función, en este caso: la indexación mediante corchetes y el insertador propio.
- Si existe una función que no sea una plantilla, que se llame igual que la plantilla y que tenga unos tipos específicos, la función que no es plantilla será la que se invoque cuando se usen esos tipos específicos.

32.2. Plantillas de clases o clases genéricas (*class templates*)

Una clase genérica define los algoritmos que se usarán en ella, pero el tipo o tipos básicos que se manipularán serán especificados como parámetros cuando se declaren objetos de esa clase.

Muchas veces la misma definición de una clase que manipula una lista (o una pila, cola, etc.) de enteros puede servir para una lista de datos de otro tipo (float, char, etc.).

Podemos definir una clase genérica, con un tipo genérico y el compilador generará automáticamente el código correcto para el tipo de dato concreto que usemos en cada momento.

Una plantilla de clase es a una clase lo que una clase es a un objeto. Es una relación parecida.

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

template <class T> class MiClase{
  private:
    T x_, y_;
  public:
    MiClase (T a, T b){ x_=a; y_=b;};
    T div(){return x_/y_;};
};

int main()
{
MiClase <int> iobj(10,3);
MiClase <double> dobj(3.3, 5.5);

cout << "división entera = " << iobj.div() << endl;
cout << "división real = " << dobj.div() << endl;
}
```

la salida de la ejecución sería:

```
$ ./a.out
división entera = 3
división real = 0.6
```

- Puede haber varios tipos genéricos separados por coma.
- Lo más habitual y aconsejable es que el cuerpo de las funciones de las clases genéricas esté dentro de la propia definición de la clase. Se podría poner el cuerpo fuera de la definición pero habría que indicar que cada función es una plantilla y la sintaxis queda muy engorrosa. La inmensa mayoría de programadores prefieren hacerlo dentro.
- El cuerpo de las funciones debe estar totalmente definido antes de la instanciación de la clase genérica, ya que el compilador al ver la instanciación genera el código de una nueva clase pero instanciada al tipo indicado como parámetro, y para ello debe estar todo el código de las funciones miembro definido. Esto hace que aumente el tamaño del código objeto ejecutable.

33. Biblioteca de E/S de C++

Veremos algunas clases y funciones de la biblioteca estándar de C++ para E/S.

33.1. Streams

Los *streams* controlan el flujo de datos en diversas situaciones durante la ejecución de un programa en C++. Un *stream* se puede considerar como un sitio donde se envian bytes en secuencia. Los más importantes son los de la biblioteca *iostream* y los de la biblioteca *fstream*. La pantalla, el teclado, los ficheros, etc. son streams de datos.

33.2. E/S estándar en C++

Uno de los más importantes es el de la biblioteca **iostream**, que para usar debemos hacer: `#include` $<$ *iostream* $>$. En él existen tres clases que los controlan: `istream` para flujos de entrada, `ostream` para flujos de salida e `iostream` para flujos de entrada y salida.

- `cin` es un objeto de la clase `istream` y está asociado al dispositivo de entrada estándar, que normalmente es el teclado.
- `cout` es un objeto de la clase `ostream` y está asociado al dispositivo de salida estándar, que normalmente es la pantalla.
- `cerr` es un objeto de la clase `ostream` y está asociado al dispositivo de error estándar, que normalmente es el dispositivo de salida estándar, que normalmente es la pantalla.
- `clog` es un objeto de la clase `ostream` y está asociado al dispositivo de *log* estándar, que normalmente es el dispositivo de salida estándar, que normalmente es la pantalla.

Algunas características destacables de estos objetos son las siguientes:

- `cout` $<<$ `endl;` es equivalente a `cout` $<<$ `"`$\backslash n$`";`.
- `cout` $<<$ `flush;` vacía el buffer de salida.

- `cout.put('A');` muestra carácter en pantalla.
- `cout.put(65);` muestra el ASCII de 65 en pantalla.
- `cin.get(c);` Pone un carácter en *c*.
- La función std::getline(cin, s, '\n') lee un *string s* desde la entrada estándar *cin* pudiendo contener la cadena varias palabras y espacios en blanco.

33.3. E/S: ficheros

Para llevar a cabo la E/S con ficheros en C++ es necesario crear flujos y vincularlos a archivos (o dispositivos). Los flujos para ficheros se encuentran definidos en <`fstream`>, por tanto tendremos que incluir siempre:

```
#include <fstream>
```

En *C++* un fichero es simplemente un *flujo externo*: una secuencia de bytes almacenados en disco. Puede ser un flujo de salida, de entrada o de entrada/salida.

Existen varias clases de importancia en el manejo de ficheros:

- **ifstream**, derivada de `istream`, para los ficheros de entrada.
- **ofstream**, derivada de `ostream`, para los ficheros de salida.
- **fstream**, derivada de `iostream`, para los ficheros de entrada y salida.

33.3.1. Declaración de un fichero

Para declarar ficheros tenemos los constructores:

```
ifstream(const char*, int modo = ios::in); //Fichero de entrada
ofstream(const char*, int modo =ios::out); //Fichero de salida
fstream(const char*, int modo =ios::in|ios:out); //Fichero de entrada/salida
```

Ejemplo:

```
#include <fstream>
ifstream archivo_entrada("datos1.txt");
ofstream archivo_salida("datos2.txt");
fstream archivo_entrada_salida("datos3.txt")
```

Los objetos declarados son ficheros (flujos) de entrada, salida y entrada/salida respectivamente. Se pasa al constructor el nombre que tendrá el archivo con el que se vincula el flujo. Si no le pasamos el nombre del fichero podemos hacerlo posteriormente con la función open(), pero si lo hacemos aquí, esta declaración implica tambien la apertura del fichero.

El modo del fichero es un parámetro opcional que se puede indicar para declarar el fichero para distintas cosas:

- `ios::in` modo entrada.
- `ios::out` modo salida. Si ya existe, se pierden los datos anteriores, si no existe, lo crea.

- `ios::binary` modo binario.
- `ios::app` para añadir, si no existe, se crea.
- `ios::ate` Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
- `ios:trunc` If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

33.3.2. Apertura y cierre de un fichero

Si se vincula un nombre externo en la declaración de un fichero, éste además se abre. Pero si se declara el fichero de la siguiente forma:

```
ofstream fich1;
```

donde no se vincula a ningún nombre externo, se debe asociar dicho nombre en la llamada posterior al método `open()`. En cada una de las tres clases mencionadas existe la función miembro `open()`, cuyo formato es:

```
//sobre ifstream
void open(char *, int modo =ios::in, int prot = filebuf::openprot)

//sobre ofstream
void open(char *, int modo =ios::out, int prot = filebuf::openprot)

//sobre fstream
void open(char *, int modo =ios::in | ios::out, int prot = filebuf::openprot)
```

En realidad solo es obligatorio proporcionar el nombre externo del fichero ya que el resto de parámetros tienen valores por defecto y el fichero se abrirá en modo ios::in si es *ifstream*, en modo ios::out si es *ofstream*, y en modo ios::in | ios::out si es *fstream*. Los modos (parámetro *modo*) de apertura son los mismos vistos en el apartado anterior.

Los permisos de acceso del fichero (parámetro *prot*) varían de una implementación a otra. En UNIX por ejemplo podemos indicar los permisos que tendrá el fichero en formato de 4 dígitos en octal (por defecto, en muchos casos, 0664).

El **cierre** de un fichero se realiza con la función miembro *close()*.

Veamos un ejemplo en el que se abre un fichero con las opciones por defecto y se comprueba si ha existido error en la apertura:

```
#include <fstream>
ofstream f;

f.open("datos.txt");

if(!f) cout << "Error de apertura";
```

33.3.3. Operaciones sobre ficheros

- La función miembro *eof()* devuelve un valor 'true' o distinto de cero si el flujo ha alcanzado el final del fichero. Tener en cuenta que debe leerse el final del fichero para que eof() se active a 'true'.

33.3.4. Lectura y escritura en un fichero de texto

Un fichero de texto (text file) almacena datos de forma que el fichero será *human-readable*, es decir, podremos hacer un `cat` de ese fichero o abrirlo con un editor de texto plano y ver su contenido.

Los ficheros binarios (binary file) guardan datos de forma similar a como se guardan en la memoria del ordenador, en binario y en una representación que no es *human-readable*. En muchos casos, para los mismos datos, ocupan menos espacio que los ficheros de texto.

Para decidir si usar usar ficheros de texto o binarios es necesario analizar el problema en cuestión y su contexto para escoger el más adecuado en cada caso.

Ejemplo de utilización del operador << para escritura en un fichero texto:

```
#include <fstream>
ofstream salida("prueba");

if(!salida) cout << "ERROR de apertura";

salida << 10 << 47.28 << "Esto es un ejemplo";

salida.close();
```

Utilización del operador >> para lectura:

```
int a;
float b;
char cad[80];

ifstream entrada("prueba");

if(!entrada) cout << "ERROR de apertura";

entrada >> a >> b >> cad;

entrada.close();
```

El operador << escribe en el fichero de salida en modo texto, es decir, escribe un caracter ASCII por cada carácter alfanumérico que tienen los datos que se escriben, sean datos numéricos (int, float, etc.) o alfanuméricos. Entonces si tenemos `char cadena[20]=''hola''; int i=4; float f=5.7;`, y ejecutamos `salida << cadena << i << f;`, se escribirán en el fichero de salida 4 bytes con las letras de la palabra "hola", seguidos de 1 byte para el dígito "4" y de tres bytes para los ASCII de la cadena "5.7".

Estos ficheros en modo texto tienen sentido si posteriormente se utiliza el texto que hay en ellos para visualizarlo en pantalla, o realizar sobre ellos un procesamiento sencillo, ya que la recuperación de los datos y su acceso mediante un programa es muy tedioso y lento, además de ocupar demasiado espacio en disco en la mayoría de los casos comparado con los ficheros binarios.

Ambos tipos de ficheros, texto y binarios, son útiles y son muy utilizados en multitud de aplicaciones.

33.3.5. Lectura de datos separados por comas de un fichero texto

Se puede usar el método getline() de la clase ifstream que tiene el siguiente formato (tener en cuenta que el primer parámetro es un char* donde se almacenará la lectura):

```
istream getline(char* s, std::streamsize n, char delim );
    s, es una cadena en C que alojará la cadena leída.
    n, es el número máximo de 'char' a leer
    delim, es el delimitador que indica donde termina la lectura
           (si no se indica delimitador, se usa \n por defecto)

// Ejemplo de lectura de un campo con getline() usando
// una longitud máxima de lectura de 256 o hasta el separador ','

char campo1[10];

while(entrada.getline(campo1,256,','))
{
  ...
}
```

El método getline() devuelve false si se llega al final del fichero.

También existe otra versión diferente de la función getline() que recibe el fichero como primer parámetro y trabaja con datos de tipo string:

```
// Ejemplo de lectura de un campo con getline() usando
// el separador ','

string campo1;

while(getline(entrada, campo1,','))
{
  ...
}
```

33.3.6. Lectura y escritura en un fichero binario

(Ver diferencias entre ficheros texto y binarios en el apartado 33.3.4)

Supongamos la siguiente declaración:

```
ifstream entrada;
ofstream salida;
entrada.open ("entrada.bin", ios::binary );
salida.open ("salida.bin", ios::binary );
```

Podemos hacer las siguientes operaciones para manipular ficheros binarios:

- `entrada.get(char &c)` lee en *c* un byte del fichero. Devuelve 0 cuando se llega al final del fichero.

- `salida.put(c)` escribe el byte *c* en el fichero.
- `entrada.read(char *buf, int n)` Lee en 'buf' 'n' bytes del fichero binario (igual que fread() en C).
- `salida.write(char *buf, int n)` Escribe en el fichero binario 'n' bytes almacenados en 'buf'(igual que fwrite() en C).

33.3.7. Acceso aleatorio a un fichero

Se utilizan las funciones miembro *seekg()* para mover el cursor de lectura y *seekp()* para el de escritura:

- Ambas funciones reciben como primer parámetro un entero que indica el **desplazamiento** en bytes del cursor de lectura (*seekg()*), o el de escritura (*seekp()*).
- El segundo parámetro es la posición relativa de dicho desplazamiento, que puede ser:
  - `ios::beg` desde el principio del fichero.
  - `ios::cur` desde la posición actual.
  - `ios::end` desde el final del fichero.
- Para obtener las posiciones de cada cursor se usan las operaciones: *tellg()* y *tellp()*.

34. Sobrecarga del operador << y del operador >>

34.1. Sobrecarga del operador << (insertador)

Es posible sobrecargar el operador << para enviar a *cout* directamente objetos de cualquier clase. El operador << suele llamarse insertador, y si lo sobrecargamos para nuestra propia clase también suele llamarse insertador propio.

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

class Punto{
private:
  int coordx_;
  int coordy_;
public:
  Punto(){coordx_=coordy_=1;};
  friend ostream &operator<<(ostream &stream, const Punto &p); // insertador
  friend istream &operator>>(istream &stream, Punto &p); // extractor
  ...
  ...
};
ostream &operator<<(ostream &stream, const Punto &p)
{
stream << "(";
stream << p.coordx_;
stream << ", ";
```

```
stream << p.coordy_;
stream << ")";
return stream;
}
```

El insertador no es una función como el operador +. El operador + sumaría dos objetos de tipo *Punto* y es, por tanto, una función de la clase *Punto*. Pero el insertador es una función aparte que inserta en un objeto de tipo *ostream* un contenido para sacarlo en pantalla. Es decir, gestiona flujos de salida y se relaciona con la clase *Punto* porque recibe un parámetro de tipo *Punto*.

No se trata de hacer:

```
Punto p1, p2;
p1 << p2;
```

En este caso << sería una función parecida a la del operador +, pero no se trata de esto. De lo que se trata es de hacer:

```
Punto p1, p2;
cout << p1 << p2;
```

NOTA: `cout` es un objeto de la clase *ostream*, y es el objeto `cout` el que hace la llamada al operador <<.

- Para hacer un operador<< en la clase *Punto* e indicar que el parámetro de la izquierda es un objeto de tipo *ostream*, no hay otra manera que hacer esa función *friend* y declarar que el operando de la izquierda es *ostream*.
- No puede estar fuera de la clase Punto porque entonces sería una función independiente y no un operador de la clase Punto. O tendríamos que haberla añadido a la clase ostream, cosa que no podemos.
- Dicho de otra manera, el operador << no puede ser *miembro normal* de la clase porque recibiría forzosamente como primer parámetro el objeto de la izquierda en el puntero `this` (es decir, el de la clase *Punto* al ser miembro de ella). En el caso de un operador << debe recibirse primero un objeto de tipo *ostream*, y esto sólo puede indicarse mediante una función friend.
- Además, el primer parámetro debe ser una referencia de tipo *ostream* porque se ha de modificar dentro de la función.
- Además, el objeto de tipo *ostream* modificado se devuelve como una referencia para que, si se encadena otra llamada al operador <<, se pase correctamente como parámetro en expresiones complejas. Al devolver una referencia, el valor que se devuelve puede modificarse por otras llamadas que lo pudieran recibir.
- El segundo parámetro conviene indicarlo como constante explicitando que no se modificará dentro. Para ahorrar tiempo y memoria pasa como una referencia constante.
- No debe aparecer la palabra friend en el cuerpo de la función, solo en la declaración de la clase, ya que es ésta quien otorga a mistad al operador.

Ahora podemos hacer:

```
Punto p;
cout << p;
```

34.2. Sobrecarga del operador >> (extractor)

Es posible sobrecargar el operador >> para realizar operaciones de entrada de tipos creados por el usuario. Este operador suele denominarse extractor (o extractor propio).

```
istream &operator>>(istream &stream, Punto &p)
{
cout << "Introduce x ";
stream >> p._coordx;
cout << "Introduce y ";
stream >> p._coordy;

return stream;
}
```

Notas sobre el ejemplo:

- Se utilizan los mismos criterios al pasar el objeto *istream* como una referencia y devolverlo.
- Ahora también el objeto `p` pasa como una referencia ya que se cambia dentro de la función.
- De nuevo, no debe aparecer la palabra friend en el cuerpo de la función, solo en la declaración de la clase, ya que es ésta quien otorga a mistad al operador.

El uso de estos operadores podría ser:

```
int main(void)
{
Punto a,b;
cin >> a;
cin >> b;

cout << a << b << endl;
}
```

35. Asignación dinámica con *new* y *delete*

C++ presenta una forma mejorada de reservar y liberar memoria dinámicamente: `new` y `delete`.

`new` es una llamada a la reserva de memoria en el *montón* (zona de la memoria utilizada para la reserva de elementos de memoria dinámica en tiempo de ejecución), y `delete` permite liberar la memoria previamente reservada con `new`.

El formatos de estas instrucciones es el siguiente:

puntero `=` `new` *tipo*;
`delete` *puntero*;

Ejemplo con un vector:

```
int *v;
float *f;
v = new int [10]   // vector de 10 enteros
f = new float [5]; // vector de 5 reales
delete [] v; // conviene siempre usar [] para invocar los destructores de
delete [] f; // cada elemento del vector y no solo el del primero.
```

Ejemplo con una matriz de 2 dimensiones:

```
#include <iostream>
using namespace std;
int main(void)
{
  //Hay otras formas, pero mejor de la forma clásica con bucle for.
  // Una matriz de f filas y c columnas:
  int f,c;
  cout <<''Introduce las filas'';
  cin >> f;
  cout <<''Introduce las columnas'';
  cin >> c;
  int **matriz;
  matriz = new int* [f]; // Espacio para un vector de f punteros a entero
  for(int i=0;i<f;i++) matriz[i] = new int [c]; //A cada puntero le asigno
                                                // espacio para un vector c enteros

  // Ahora usemos la matriz:
  matriz[0][0]=1;
  cout << "matriz[0][0]= " << matriz[0][0] << endl;

  //Liberando espacio
  for(int i=0;i<10;i++) delete [] matriz[i]; //Libero los vectores
  delete [] matriz; // y el vector inicial con los punteros

}
```

Ejemplo con un objeto de la clase *"A"*:

```
A *obj;
obj = new A(parametros del constructor);

delete obj;
```

Referencias

[1] Schildt, Herbert. *C++: A Beginner's Guide. Second Edition.* McGraw-Hill/Osborne 2004.

[2] Harvey M. Deitel, Paul J. Deitel. *Cómo Programar en C++.* Cuarta edición. Pearson Educación, México, 2003. ISBN:970-26-0254-8. Páginas:1376.

[3] Herbert Schildt. *C++ : manual de referencia.* Osborne MacGraw-Hill, 1995. (Disponible biblioteca)

[4] Herbert Schildt. *Aplique Turbo C++*. Osborne McGraw-Hill, 1991.

[5] Bjarne Stroustrup. *The C++ Programming Language*. Third Edition. Addison-Wesley, 1997. (Disponible biblioteca)

[6] Bjarne Stroustrup. *El Lenguaje de Programación C++*. Edición Especial. Addison-Wesley, 2002. (mejora de la "Third Edition" y más recomendable).

[7] Standard C++ Foundation. `http://isocpp.org`

[8] GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`

[9] cppreference.com provides programmers with a complete online reference for the C and C++. `https://en.cppreference.com`

NOTA: el autor agradece el envío de correcciones/sugerencias a *aromero@uco.es*