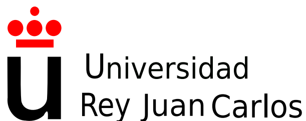
ÍNDICE

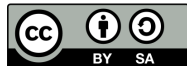
1. Introducción a C++	2
2. Introducción a la POO	43
3. Clases I. Generalidades	56
4. Clases II. Uso avanzado	84
5. Sobrecarga de operadores	114
6. Herencia y polimorfismo	129
7. Plantillas. Librería STL	162
8. Manejo de excepciones	201
9. Depuración y documentación	216
10. Manejo de ficheros	246
11. Patrones de diseño	273

1. Introducción a C++

Julio Vega

julio.vega@urjc.es





©2024 **Julio Vega Pérez**
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. `Hello world`
2. Códigos básicos con C++
3. Operadores
4. Instrucciones de control
5. Instrucciones de repetición
6. Instrucciones `break` y `continue`
7. Tipos enumerados

```cpp
/* -------------------------
   File: helloworld.cpp
   Author: Julio Vega
   Date: 22/07/14
   Goal: this program shows the basics about C++
------------------------- */
#include <iostream> // library to output data onto the screen

int main() { // function main begins program execution
   std::cout << "Hello world in C++!\n"; // show message
   return 0; // indicate that program ended successfully
} // end function main
```

- Editor recomendable: *Gedit*. Muy extendido, sencillo pero potente.
- Instalar compilador: `sudo apt-get install build-essential`
  - GCC = GNU project C and C++ compiler.
- Compilación: `g++ -o helloworld helloworld.cpp`
  - `-o`: *out*, permite indicar el nombre del ejecutable de salida.
    - Si no se indica, se crea ejecutable por defecto: *a.out*.
- Ejecución: `./helloworld`

- Todo fichero de código debe incluir una cabecera descriptiva.
- La línea `#include <iostream>` es una directiva del preprocesador.
  - Toda línea que comienza por # son procesadas por el preprocesador.
    - El preprocesador es aquel que se ejecuta antes de compilar el programa.
  - En este caso, se indica que debe incluir el contenido de `iostream`.
    - Esta es una librería que permite mostrar/recibir datos del I/O estándar.
    - E.g. para usar `cout` (salida por pantalla).
- Los comentarios de una línea comienzan con //
  - Si tienen más líneas, empiezan con /* y terminan con */
- La función `main` comienza la ejecución del programa.
  - Un programa solo puede contener una función `main`.
  - Esta función devuelve un entero (`int`).
- Todo bloque de código va entre llaves ({ y }).
- Toda instrucción acaba en ; excepto las directivas del preprocesador.

- La línea de `std::cout` imprime el mensaje entrecomillado.
  - Está mandando (`<<`, *pipe*) el mensaje a la salida estándar (`std::cout`).
  - `std::` indica que `cout` pertenece al espacio de nombres de *std*.
    - Otros flujos estándar son: `cin` (entrada) y `cerr` (error).
  - \n = \(carácter de escape) + `n`(*new line*) = salto de línea.
    - Otros c. escape: \t (tab), \r (retorno carro), \a (alert sound), \', \".
- La última línea, `return 0;`, indica que el programa termina con éxito.
- Sangría no necesaria (Python sí) pero sí obligatoria en la asignatura.
  - Tamaño uniforme, obligatorio insertar espacios, no tab (config. editor).
- Idioma recomendable: inglés. Reduce tamaños y facilita reutilización.

```
#include <iostream>

int main() {
  std::cout << "Welcome back ";
  std::cout << "to C++!\n";

  std::cout << "Welcome back\nto\n\nC++!\n";
  return 0;
}
```

```
#include <iostream>

int main() {
  int num1;
  int num2;
  int total;

  std::cout << "Write the first integer: ";
  std::cin >> num1;

  std::cout << "Write the second one: ";
  std::cin >> num2;

  total = num1 + num2;
  std::cout << "The total is " << total << std::endl;

  return 0;
}
```

```cpp
#include <iostream>

int getTotal (int a, int b){
  return a + b;
}

int main() {
  int num1;
  int num2;
  int total;

  std::cout << "Write the first integer: ";
  std::cin >> num1;

  std::cout << "Write the second one: ";
  std::cin >> num2;

  std::cout << "Total: " << getTotal(num1, num2) << std::endl;
  return 0;
}
```

library.h

```
int getTotal (int,int);
```

library.cpp

```
#include "library.h"

int getTotal (int a, int b){
  return a + b;
}
```

main.cpp

```
#include <iostream>
#include "library.h"

int main() {
  int num1;
  int num2;
  int total;

  std::cout << "Write the first integer: ";
  std::cin >> num1;

  std::cout << "Write the second one: ";
  std::cin >> num2;

  std::cout << "Total: " << getTotal(num1, num2) << std::endl;
  return 0;
}
```

- Este es un fichero de texto plano que contiene **órdenes**.
  - Las órdenes son ejecutadas por la utilidad `make` siguiendo reglas.
    - Al ejecutar `make` desde el Terminal, busca el fichero *Makefile*.
  - **Explícitas**: dan instrucciones a `make` para construir fich. indicados.
    - `main.o: main.cpp library.h`
    - `>$(CC) $(CFLAGS) -c main.cpp`
      Se lee: para construir el fichero main.o se requiere main.cpp y library.h. Y se le da la orden para que make pueda construir ese fichero...
  - **Implícitas**: instrucciones generales a seguir si $\nexists$ regla explícita.
- Las instrucciones de las reglas han de estar debidamente indentadas.
  - Lo habitual para la indentación es usar el tabulador.
  - Si se usa un *indentador* diferente, hay que especificarlo.
    - En este caso se ha establecido el caracter `>` para indentar.
    - Para ello se establece la variable de entorno: `.RECIPEPREFIX = >`
- También puede contener **comentarios**, precedidos por el símbolo `#`.

```
#========================================
# Example of Makefile
#========================================

CC = g++
CFLAGS = -Wall -g
.RECIPEPREFIX = >

main: main.o library.o
>$(CC) $(CFLAGS) -o main main.o library.o

main.o: main.cpp library.h
>$(CC) $(CFLAGS) -c main.cpp

library.o: library.h

clean:
>rm *.o main
```

- Las tres variables se podrían declarar en una sola línea.
  - E.g.: `int num1, num2, total;`
  - Esto reduce la legibilidad e impide poner comentarios individuales.
- Recomendable poner espacio tras coma para aumentar legibilidad.
- Todos los identificadores van en minúscula y notación camello.
  - Excepto las constantes, que se escriben todo en mayúscula.
  - C++ es *case sensitive*, por lo que: `num1` ≠ `Num1`.
- La notación camello consiste en poner mayúscula cada palabra anexa.
  - En lugar de separarlas por guiones bajos, algo desaconsejable.
  - E.g.: `getTotal()`, son dos palabras (*get* + *total*).
    - Otro ejemplo: `getMaxGradeOfSubject()`.
- Las funciones, además, deben incluir verbo: `int getTotal ();`

- IDs cualquier longitud;recomendable 31 o menos (↑ compatibilidad).
- Nombres de variables que sean significativos (e.g. `counter` VS. `c`).
- Evita IDs que comiencen por _; GCC puede usar nombres así.
- Evita usar palabras muy simbólicas; C++ podría integrarlo en futuro.
  - Hoy podrías usar `object`; mañana C++ podría usarla como *keyword*.
- Deja línea en blanco entre la declaración de variables e instrucciones.
- Coloca espacios a ambos lados de un operador binario (e.g. `a = 2;`)

- Uso de `cin >> myValue;`
  - Para leer enteros: `cin >> myInt;`).
  - Para leer caracteres: `cin >> myChar;`). Si +1 char $\Longrightarrow$ *overflow*.
- Uso de `myValue = cin.get()`
  - Para leer enteros: `myInt = cin.get()`. Vierte su código ASCII.
    - Para obtener el ASCII de un char: `static_cast<int>('2')`.
  - Para leer caracteres: `myChar = cin.get()`. Vierte el mismo char.
- Leer cadenas de caracteres (sin usar `string`, de librería `string`).
  - Se puede usar `cin >>`, pero divide la cadena por espacios.
  - O mediante array de chars, con: `cin.getline (myString, long);`

```
char myChar; int myInt; char myString[200];

cout << "Write a char: " << endl; // READING WITH "cin >>"
cin >> myChar; // supposed to be read '2'
cout << "Read: " << myChar << endl << endl; // shows '2'
cout << "Write an int: " << endl;
cin >> myInt; // supposed to be read 2
cout << "Read: " << myInt << endl << endl; // shows 2

cout << "Write another char: " << endl; // READING "cin.get()"
myChar = cin.get (); // supposed to be read '2'
cout << "Read: " << myChar << endl; // shows '2'
cout << "Write another int: " << endl;
myInt = cin.get (); // supposed to be read '2'
cout << "Read: " << myInt << endl << endl; // shows 50

cin.getline (myString, 200); // READING A CHAR ARRAY
cout << "cadena: " << myString << endl;
```

- Algunos caracteres especiales a leer son EOF (End Of File).
  - Este caracter, en sistemas UNIX/Linux, se escribe con *CTRL+d*.
    - En UNIX/Linux, *CTRL+c* cancela proceso; *CTRL+z* lo deja zombie.
  - En Windows, EOF se introduce mediante *CTRL+z*.
- Otro caracter especial es el caracter nulo ('\0').
  - Antes había que ponerlo manual/ al final de cada cadena de char.
  - Ahora ya no es necesario porque lo pone el compilador automática/.
    - E.g. `char greeting[6] = 'H', 'e', 'l', 'l', 'o', '\0';`

```
#include <iostream>

int main () {
  int number;
  bool exit = false;
  do {
    std::cout << "Write a number (EOF to exit): " << std::endl;
    if ((number = std::cin.get()) == EOF) exit = true;
    std::cin.ignore(); // try with this line commented... errors!
  } while (!exit);   // due to how cin works (splits on spaces)
  return 0;
}
```

Comunes de C y C++

`auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while`

De C++

`and and_eq asm bitand bitor bool catch class compl const_cast delete dynamic_cast explicit export false friend inline mutable namespace new not not_eq operator or or_eq private protected public reinterpret_cast static_cast template this throw true try typeid typename using virtual wchar_t xor xor_eq`

- Básicos: $+$, $-$, $*$, $/$
- Módulo: %. Vierte el resto de una división entera.
- Potencia: función `pow`. C++ da errores con operadores como $**$ o ˆ.
- Paréntesis: sirven para agrupar subexpresiones (e.g. `a * (b + c)`).
- Orden precedencia: 1.º paréntesis, 2.º $*$, $/$ o %, 3.º $+$ o $-$.
- El uso redundante de paréntesis $\uparrow$legibilidad expresiones complejas.

- Op. relacionales: $>$, $<$, $>=$, $<=$
- Op. de igualdad: $==$, $!=$
- En operadores que contengan dos símbolos, estos han de estar juntos.
- Invertir los símbolos de un op. supone un error sintáctico.
  - Y, en el peor de los casos, un error lógico. E.g. $=!$ en lugar de $!=$.
- Confundir $==$ por $=$ genera error lógico ($\uparrow\uparrow$difícil detectar).

- Op. básico de asignación (=): `i = i + 3;`
- Op de asignación de suma (+=): `i += 3;`
  - E igualmente: `-=`, `*=`, `/=`, `%=`.
- Operadores unarios ofrecidos por C++ para sumar y restar.
  - Postincremento y postdecremento (usa valor actual y luego modifica).
    - `i++;` o `i--;`
  - Preincremento y predecremento (modifica y luego usa nuevo valor).
    - `++i;` o `--i;`

```
int i;
i = 5;
cout << i << endl; // shows 5
cout << i++ << endl; // shows 5 and increment 1
cout << i << endl; // shows 6
// -----------------------------------------
i = 5;
cout << i << endl; // shows 5
cout << ++i << endl; // increment 1 => shows 6
cout << i << endl; // shows 6
```

- Necesarios para evaluar varias condiciones y tomar una decisión.
- Operadores: && (AND), || (OR), ! (NOT).

```
if (woman && age >= 65) oldWomen++;

if ((practicalExercises < 4) || (practicalFinalExam < 4))
  failSubject = true;

std::cout << std::boolalpha << "Fail subject = " << failSubject;
```

- Buenas prácticas: aprovecha los cortocircuitos por eficiencia.
  - En AND, la condición con más prob. ser `false`, mejor a la izquierda.
  - En OR, la condición con más prob. ser `true`, mejor a la izquierda.
- `boolalpha` es un manipulador de flujo *pegajoso*.
  - Hace que los valores `bool` se impriman como `true` o `false`.
  - *Pegajoso*: se queda activado para el resto del programa.

- Todo problema puede resolverse ejecutando ciertas acciones en orden.
- Algotirmo: acciones a ejecutar + orden en que se ejecutan estas.
- Ejecución secuencial: instrucciones se ejecutan una después de otra.
  - VS. *goto* (1960): permite transferencia de control a cualquier destino.
- Programación estructurada: secuencia + selección + repetición.
  - Supuso todo un reto cambiar a este nuevo paradigma (fin del *goto*).
- Instrucciones selección: `if`, `if...else` y `switch`.
- Instrucciones repetición: `while`, `do...while` y `for`.
- Todo programa C++ se puede hacer con estas 6 inst. + secuencias.
  - Combinando inst. control en dos formas: apilamiento + anidamiento.

Instrucción de selección doble if...else

```
if (studentGrade >= 5)
  cout << "Passing grade";
else
  cout << "Failing grade";
```

- C++ ofrece tipo de datos `bool`, con posibles valores: `true` o `false`.
  - También se pueden usar valores enteros: 1 <=> `true` y 0 <=> `false`.
  - Usar valores enteros ↑ compatibilidad con versiones anteriores.
- Se han omitido las llaves de apertura y cierre en bloques `if` y `else`.
  - Esto es posible porque ambos bloques solo contienen una instrucción.
  - Las llaves son solo obligatorias si bloque contiene varias instrucciones.
  - Pero úsalas en caso de ambigüedad o para evitar errores lógicos.

Operador condicional ? . . . :

```
studentGrade >= 5 ? cout << "Passing grade" : cout << "Failing
    grade";
```

- Se lee: *Si calificación >= 5, entonces aprobado, si no, suspenso.*

Instrucciones if...else anidadas

```
if (studentGrade >= 9)
  cout << "A grade";
else
  if (studentGrade >= 8)
    cout << "B grade";
  else
    if (studentGrade >= 7)
      cout << "C grade";
    else
      if (studentGrade >= 6)
        cout << "D grade";
      else
        cout << "F grade";
```

- Una instr. `if...else` anidada es más rápido que varios `if` simples.
  - Con anidamiento se puede salir antes de tiempo al cumplirse condición.

Instrucciones if...else anidadas (forma popular)

```
if (studentGrade >= 9)
  cout << "A grade";
else if (studentGrade >= 8)
  cout << "B grade";
else if (studentGrade >= 7)
  cout << "C grade";
else if (studentGrade >= 6)
  cout << "D grade";
else cout << "F grade";
```

- Esta sintaxis es idéntica a la anterior, excepto por espaciado y sangría.
- Esta forma es más usada; evita usar mucha sangría y ↑ legibilidad.

El problema del `if` ignorado

```
if (studentGrade >= 9);
  cout << "A grade";
```

- Lo que parece: si grade $>=$ 9 $\Longrightarrow$ *A grade*.
  - Pero en realidad siempre veremos por pantalla *A grade*.
- Al poner ; al final del `if` ¡la hemos convertido en una instrucción!
  - Se ejecuta, se evalúa la condición, y se acaba su *influencia*.
  - A continuación se ejecuta la siguiente instrucción ¡siempre!
- Esto es un error de lógica, un fallo en nuestra implementación.
  - No generar error de compilación, salvo que después apareciese un `else`.

```
if (finalGrade = 10)
  cout << "A grade with honors";
```

- Otro error lógico: confundir comparación $==$ con asignación $=$.
- Lo que parece: si grade $==$ 10 $\Longrightarrow$ *Matrícula de honor*.
  - Pero en realidad ¡ponemos matrículas a todos los alumnos!

El problema del else suelto (1/2)

```
if (studentGrade >= 9)
  if (practicalExercises >= 9)
    cout << "A grade with honors";
else
    cout << "studentGrade < 9";
```

- Lo que parece: si grade $>=$ 9 y exerc $>=$ 9 $\Longrightarrow$ *A with honors*.
  - Si no (grade $<$ 9) $\Longrightarrow$ *studentGrade* $<$ *9*.

El problema del `else` suelto (2/2)

- ¡Pero tal anidamiento no se ejecuta como parece!
- Es un problema de indentación por nuestra parte.
  - Recuerda siempre: el `else` siempre va anidado al último `if`.
- A continuación vemos cómo deberíamos haberlo indentado...
  - ...que es como lo interpreta el compilador.

```
if (studentGrade >= 9)
  if (practicalExercises >= 9)
    cout << "A grade with honors";
  else
    cout << "studentGrade < 9";
```

- Y vemos que lo implementado no tiene sentido $\Longrightarrow$ error lógico.
  - Si el `if` externo es `true` y el `if` interno es `false`.
    - Tendremos como salida que *studentGrade* < 9.
  - Y partíamos de que el `if` externo era `true` (*studentGrade* $>= 9$).
- Estos son los errores más difíciles de detectar, ¡los de lógica!

- Para realizar distintas acciones según valor el valor de una variable.
  - Esta variable o expresión puede ser tipo carácter o entera.
- Olvidar `break` cuando es necesario es un error lógico.

```
switch (grade) { // goal: get number of A's, B's, C's...
case 'A': // A grade (upper case)
case 'a': // or a (lower case)
  aCounter++; // increment number of As
  break; // break the switch
case 'B': // B grade (upper case)
case 'b': // or b (lower case)
  bCounter++; // increment number of Bs
  break; // break the switch
case 'C': // C grade (upper case)
case 'c': // or c (lower case)
  cCounter++; // increment number of Cs
  break; // break the switch

// [...] It is continued next slide
```

```
case 'D': // D grade (upper case)
case 'd': // or d (lower case)
  dCounter++; // increment number of Ds
  break; // break the switch
case 'F': // F grade (upper case)
case 'f': // or f (lower case)
  fCounter++; // increment number of Fs
  break; // break the switch

case '\n': // ignores \n, \t and blank spaces
case '\t':
case ' ':
  break; // break the switch

default: // catch the rest of characters
  cout << "Grade is wrong!" << endl;
  break; // optional (since it is the last sentence)
} // end of switch
```

- Sirven para repetir una acción o conjunto de estas bajo una condición.
- El resultado de los siguientes tres bucles es exactamente el mismo.

```
int i = 1;
while (i <= 100) {
  cout << "Step n. " << i << endl;
  i += 1;
}
// ------------------------------------
for (i = 1; i <= 100; i += 1)
  cout << "Step n. " << i << endl;
// ------------------------------------
i = 1;
do {
  cout << "Step n. " << i << endl;
  i += 1;
} while (i < = 100);
```

- `while`: es el más polivalente, pero tiene problema de bucle infinito.
  - Error de lógica frecuente: olvido de acción que torne la condición falsa.
- `for`: si el objetivo es recorrer alguna colección (sin otra condición).
  - E.g. recorrer un array, una imagen, etc.
  - Por sintaxis, impide olvidar inicializar contador o acción de condición.
- `do...while`: si se necesita entrar la primera vez sin condición.
  - E.g. pedir valores al usuario hasta que introduzca un valor adecuado.
    - Como mínimo voy a necesitar pedírselo la primera vez, y quizás más.
- Usa adecuadamente todos los tipos; ¡no siempre el bucle `while`!
- Evita estos errores lógicos; recuerda que son muy difíciles de detectar.
  - No olvides revisar si la condición de terminación va a ocurrir.
  - No olvides inicialiar los contadores (normalmente a 0 o a 1).
  - Error dpzmto. en 1: revisa inicializar 0 o 1 y condición con $<$ o $<=$.

- Alteran flujo de control (en instr. de selección y repetición).
- Ya hemos visto el uso de `break` dentro del `switch`.
  - Al igual que en este, dentro de bucle ocasiona salida inmediata.
- Un uso adecuado de `break` y `continue` aumenta eficiencia.
  - Pero, ¿se siguen los ppios. de la programación estructurada?

```
#include <iostream>

int main() {
  int counter;
  for (counter = 1; counter <= 10; counter++) { // 10 steps?
    if (counter == 5) break; // if counter == 5 => out!
    std::cout << counter << " ";
  } // end for
  std::cout << "\nEnd for with counter = " << counter;
  return 0;
}
```

- Al usar `continue` en bucle, salta a la siguiente iteración.
    - Omitiendo las instrucciones restantes de la iteración actual.
- En `for`, se ejecuta incremento y después se evalúa `continue`.

```
#include <iostream>

int main() {
  int counter;
  for (counter = 1; counter <= 10; counter++) { // 10 steps?
    if (counter == 5) continue; // counter == 5 => jump next it.!
    std::cout << counter << " "; // ignored when continue
  } // end for
  std::cout << "\nContinue avoided showing number 5\n";
  return 0;
} // It shows 1 2 3 4 6 7 8 9 10
```

- Son aquellos tipos de datos definidos por el propio usuario.
- Facilitan la legibilidad y el mantenimiento del software.
- Se le pueden asignar diferentes valores limitados.
  - Estos valores se deben definir al declarar el tipo.
- Para declarar un tipo enumerado se usa la *keyword* `enum`.
  - Y luego, entre llaves, se definen los valores posibles.
  - E.g. `enum gender {male, female};`
- Una vez definido, se pueden crear variables de tal tipo enum.
  - Como si de cualquier otro tipo básico se tratara.
  - E.g. `gender myGender;`
- Los valores se almacenan en un array, según orden declarado.
  - Así que, como cualquier array, estos comienzan por el índice 0.
  - E.g. para acceder al valor `female` sería `myGender(1)`.

```
enum gender {male, female};
enum weekendday {Saturday, Sunday};
enum errorFlag {ioError, success, working};

int main() {
  gender field;
  weekendday day = Saturday;
  errorFlag flag;
  // weekendday anotherDay = Wednesday; // syntax error
  field = male; // it must be used without quotation marks

  // << are overloaded to show the enum value instead of index
  cout << "The gender is " << field << endl; // it shows male
  cout << "The day is " << day << endl; // it shows Saturday

  if (day == Saturday) flag = success;
  else flag = errorFlag(2); // index 2 of errorFlag is working

  cout << "And the flag shows " << flag << endl; // shows success
}
```

1. Introducción a C++

Julio Vega

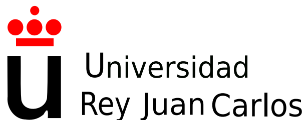
julio.vega@urjc.es

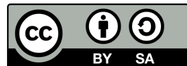
2. Introducción a la POO

Julio Vega

julio.vega@urjc.es

GSyC





©2024 **Julio Vega Pérez**
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. Introducción
2. La orientación a objetos (OO)
3. El diseño orientado a objetos (DOO)
4. La programación orientada a objetos (POO)

- Hasta ahora habéis aprendido a programar de forma estructurada.
  - Pero hay más; eso solo han sido los primeros pasos en programación.
- Programación estructurada: forma habitual de programar (vs. *GoTo*).
  - Todo parte del *main()*, escribiendo las líneas en orden.
  - Existen funciones que encapsulan varias líneas y hacen *algo*.
  - Cuando hay que usar esas funciones, se invocan.
  - Cuando estas acaban, retornan al punto desde el que se les ha llamado.

- La programación nace para facilitarnos la vida.
- Con ella somos capaces de plasmar y resolver problemas reales.
- Pero si miramos el mundo real, ¡vemos objetos por todas partes!
  - Gente, animales, plantas, coches, aviones, edificios, ordenadores.
- En conclusión, los humanos pensamos en términos de objetos.
  - E.g. manejamos una silla; no un conjunto de patas y un respaldo.
- Dividimos los objetos en dos categorías: animados e inanimados.
  - Los animados se mueven, hacen cosas; los inanimados no por sí solos.
  - Pero todos tienen atributos y muestran un comportamiento.
    - E.g. atributos: tamaño, forma, color, peso.
    - E.g. comportamientos: una pelota rueda, rebota, se desinfla, etc.

- Los humanos aprendemos sobre los objetos existentes.
  - Estudiando sus atributos y observando sus comportamientos.
  - Distintos objetos pueden tener atributos y comportamientos similares.
- El diseño orientado a objetos modela el SW y lo acerca al mundo real.
  - Este diseño aprovecha la relaciones entre los objetos.
    - E.g. los objetos de la clase vehículo tienen las mismas caract.
  - Este DOO también aprovecha las relaciones de herencia.
    - Las nuevas clases se derivan de las existentes, refinándolas.
  - El DOO también modela la comunicación entre los objetos.
    - Los objetos se comunican, al igual que se comunican las personas.

- La idea del DOO es: objeto = atributos + operaciones.
- Además, los objetos pueden/deben ocultar información.
  - Los objetos deben poder comunicarse entre sí.
    - A través de interfaces bien definidas.
  - Pero no tienen que saber cómo están implementados otros objetos.
    - Los detalles de la implementación se ocultan dentro de los objetos.
- Podemos conducir un coche sin saber cómo funciona la transmisión.
  - Siempre que sepamos usar los pedales, el volante, etc.
- Ocultar información es imprescindible para un buen diseño software.
  - Ya lo veremos más adelante...

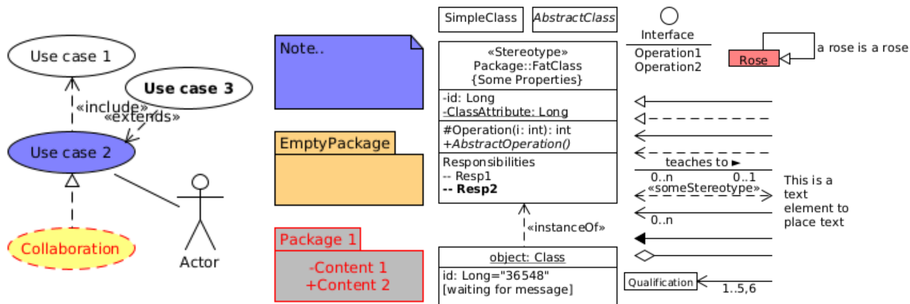
- Lenguajes como Java o C++ son orientados a objetos.
- La programación en estos lenguajes se llama POO.
    - Permiten a los programadores implementar un DOO.
- Otros lenguajes, como C, son por procedimientos.
- La unidad en C es la función; en C++, la clase.
    - Y a partir de la clase se instancian (crean) los objetos.
- Las clases contienen funciones que implementan operaciones.
    - Y datos que implementan atributos.

- Una clase es un tipo (enriquecido) definido por el usuario.
  - Contiene datos y también las funciones que manipulan estos.
  - También proporciona servicios a clientes (clases que usan clases).
  - Atributos: componentes de datos de la clase (miembros de datos).
  - Métodos: componentes de funciones de la clase (funciones miembro).
- Las clases son a los objetos lo que los planos de obra para las casas.
  - Una clase es un plano para construir un objeto de esa clase.
  - (O un molde con el que obtener figuras con esa forma.)
  - Podemos obtener (instanciar) varios objetos de una clase.
- Las clases pueden tener relaciones (asociaciones) con otras clases.
  - Es una bondad de la POO, la reutilización (de clases).
  - En DS las clases relacionadas se empaquetan en componentes.
  - (Un mantra del DS es reutilización, reutilización, reutilización.)

- *Picar código* (teclear sin más) puede valer para pequeños programas.
- Trabajar en equipo y/o en un gran sistema sw requiere mucho más.
    - Seguir *proceso* detallado para **analizar** los requerimientos del sistema.
        - *Qué* es lo que se supone que debe hacer el sistema.
    - Desarrollar un **diseño** que cumpla con esos requerimientos.
        - *Cómo* debe hacerlo el sistema.
- Puede implicar analizar y diseñar el sistema desde punto de vista OO.
    - Se denomina proceso de análisis y diseño orientado a objetos (A/DOO).
- Os repetiré este mantra: *hacer un buen A/D ahorra mucho tiempo*.
    - Primero ten claro *qué* y *cómo* y luego finalmente *impleméntalo*.

- Cuando problemas y n.º programadores $\uparrow \implies$ unificar proceso.
- $\exists$ muchos A/DOO $\implies$ un lenguaje gráfico muy popular es UML.
  - Unified Modelling Language o Lenguaje Unificado de Modelado.
- Surge en los 90s, cuando muchas empresas comienzan a usar POO.
  - E.g. HP, IBM, Microsoft, Oracle y Rational Software.
    - Estas y otras se unieron como socias de UML (*UML Partners*).
  - Lo desarrolla la *Rational Software Corporation*[1] (ahora de IBM).
    - Gracias en gran medida a las proposiciones del *OMG.org*.
    - La *Object Management Group* es una org. sin ánimo de lucro.
    - Promueve la estandarización de las tecnologías orientadas a objetos.
- UML 1.1 (1997) es mantenido por OMG hasta ahora, con UML 2[2].

[1] *https://www.ibm.com/docs/en/rational-soft-arch/9.5?topic=diagrams-creating-uml-m*
[2] *https://www.uml.org*

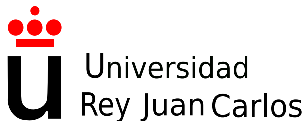


- Esquema de representación gráfica más usado para modelar sist. OO.
- Es un lenguaje gráfico complejo, con muchas características.
  - Nosotros usaremos un subconjunto conciso y simplificado de estas.
- Un analista puede diseñar sistemas usando varios procesos.
  - Pero todos se expresan mediante esta notación gráfica estándar.

2. Introducción a la POO

Julio Vega

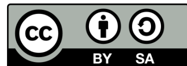
julio.vega@urjc.es



3. Clases I. Generalidades

Julio Vega

julio.vega@urjc.es



©2024 **Julio Vega Pérez**
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. Creación de una clase
2. Alcance y uso de una clase
3. Constructor y destructor de clase
4. Bonus: Ejemplo de una mala práctica de programación

```
#ifndef TIME_H
#define TIME_H

class Time {
public:
   Time(); // constructor
   void setTime (int, int, int); // set hour, minute and second
   void printUniversal () const; // print time in univ. format
   void printStandard () const; // print time in stand. format
private:
   unsigned int hour; // 0 - 23 (24-hour clock format)
   unsigned int minute; // 0 - 59
   unsigned int second; // 0 - 59
}; // end class Time

#endif
```

- El nombre de una clase (= fichero) siempre con mayúscula inicial.
- Uso de las directivas del preprocesador: `#ifndef`, `#define` y `#endif`.
  - Es la denominada envoltura del preprocesador.
  - Evita incluir los archivos de encabezado más de una vez por programa.
- Uso del archivo de encabezado en mayúsculas.
  - Sustituyendo el punto por un guión bajo.
- Por claridad, usa el especificador de acceso solo una vez.
  - Por legibilidad, los miembros `public` primero, fáciles de localizar.
  - Por seguridad, los atributos de la clase deben ser `private`.
    - A menos que se demuestre la necesidad de que sea `public`.

```
Time::Time() {} // constructor (special method)

void Time::setTime (int h, int m, int s) { // set new Time value
   if ((h >= 0 && h < 24) && (m >= 0 && m < 60) &&
      (s >= 0 && s < 60)) { // validate hour, minute and second
      hour = h; minute = m; second = s;
   } else throw invalid_argument ("H, M or S was out of range" );
} // end function setTime

void Time::printUniversal() const { // print Time in (HH:MM:SS)
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

void Time::printStandard() const { // print Time in AM or PM
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
      << setfill('0') << setw(2) << minute << ":" << setw(2)
      << second << (hour < 12 ? " AM" : " PM");
} // end function printStandard
```

- `setfill`: manipulador de flujo parametrizado.
  - Carácter de relleno al imprimir un entero en un campo más ancho.
    - Ejemplo: si el valor de minuto es 2, se mostrará como 02.
  - *Función pegajosa*: tal relleno se aplicará ya en todos los valores.
- `setw`: indica el número de digitos que se mostrarán. No es pegajosa.
- Cuando no se use una función pegajosa hay que restaurarla.
  - Ya lo veremos más adelante, en el tema de entrada/salida de flujos.
- `?:` es un operador condicional (*si* $A \Longrightarrow B$*, sino* C).

- Las definiciones de clase y sus métodos pueden estar en un fichero.
- Pero lo ideal es separar ambos códigos en dos archivos.
  - La definición de la clase en el archivo de encabezado (.h, *header*).
  - La definición de sus miembros en el archivo de código fuente (.cpp).
- Esta separación facilita la modificación de los programas.
  - Modificar la implementación de una clase no afecta a los clientes.
  - Siempre que se respete el interfaz definido en el *header*.
- El .h incluye comentarios para cliente; .cpp, para desarrollador.
- Los *Independent Sofware Vendor* o ISVs (software privativo).
  - Proporcionan solo las bibliotecas de clase para su venta.
    - Por contra está el movimiento *open source*.
  - Pero un cliente necesita poder enlazarse al código objeto de la clase.
  - Por ello se entregan los .h y los .o (pero no los .cpp).
  - Ya veremos (Cap. 3) que un .h debe incluir las funciones `inline`.
  - Pero hasta los atributos `private` pueden ocultarse (clase *proxy*).

- Los métodos y atributos pertenecen al alcance de la clase.
- Cualquier miembro de clase puede usar otro miembro de clase.
- Solo los miembros `public` pueden ser accedidos desde fuera.
  - Mediante el nombre, la referencia o el apuntador a un objeto.
- Los métodos se pueden sobrecargar por otros métodos.
  - Con el mismo identificador pero distintos parámetros.
    - `int getName (int index);` ≠ `int getName (float index);`
- Las variables declaradas en un método solo son visibles en este.
  - ¿Y si se declara una variable con el mismo nombre que un atributo?
    - El atributo se *oculta*, y para acceder a él hay que usar `::`

```
int main() {
   Time t; // instantiate object t of class Time

   cout << "The initial universal time is ";
   t.printUniversal(); // 00:00:00
   cout << "\nThe initial standard time is ";
   t.printStandard(); // 12:00:00 AM
   t.setTime( 13, 27, 6 ); // change time
   cout << "\n\nUniversal time after setTime is ";
   t.printUniversal(); // 13:27:06
   cout << "\nStandard time after setTime is ";
   t.printStandard(); // 1:27:06 PM
   // attempt to set the time with invalid values
   try {
      t.setTime( 99, 99, 99 ); // all values out of range
   } catch ( invalid_argument &e ) {
      cout << "\n\nException: " << e.what() << endl;
   } // end catch
}
```

- Una vez definida la clase se puede usar como cualquier tipo de datos.
  - Es como un `typedef struct` pero enriquecido con métodos.
- Lo primero es *instanciar* un objeto de la clase: `Time t;`
  - La clase es el *molde* del que salen copias/instancias.
- Lo siguiente ya es usar esa instancia `t`.
- Podremos invocar a todos sus miembros `public` mediante el `.`
  - Ya veremos más detenidamente esto de la visibilidad...
  - El operador `.` se usa tras una referencia a objeto.
    - Mientras que el operador `->` se usa tras un puntero a objeto.

Creación de la clase Count

```
class Count {
public: // public data is dangerous
  void setX (int value) {
    x = value;
  } // end function setX

  void print() {
    cout << x << endl;
  } // end function print

private:
   int x;
}; // end class Count
```

Usos de la clase Count con diferentes manejadores

```
int main() {
  Count counter; // create counter object
  Count *counterPtr = &counter; // create pointer to counter
  Count &counterRef = counter; // create reference to counter

  cout << "Set x to 1 and print using the object's name: ";
  counter.setX (1); // set data member x to 1
  counter.print (); // call member function print

  cout << "Set x to 2 and print using a reference to an object:";
  counterRef.setX (2); // set data member x to 2
  counterRef.print (); // call member function print

  cout << "Set x to 3 and print using a pointer to an object: ";
  counterPtr->setX (3); // set data member x to 3
  counterPtr->print (); // call member function print
} // end main
```

- Constructor y destructor son métodos especiales de la clase.
- Su función es la de crear/destruir el objeto cuando sea oportuno.
  - Invocados implícitamente por el compilador: instanciar/fin de ámbito.
    - Cuando se crea la instancia del objeto se está invocando al constructor.
- El nombre de ambos métodos ha de ser igual al de la clase.
  - Así, el compilador sabe a qué método buscar cuando ha de invocarlos.
  - En el caso del destructor, precedido por la virgulilla ($\sim$).
    - No es casualidad: $\sim$ es el operador de complemento a nivel de bits.
    - Y el destructor es el complemento del constructor...
- Ya veremos el gran protagonismo de constructor y destructor en C++.

```cpp
#ifndef TIME_H
#define TIME_H

class Time {
public:
  Time(); // (a) basic constructor
  Time (int = 0, int = 0, int = 0); // (b) with defaults
  // [...]
private:
  // [...]
}; // end class Time

#endif
```

```
Time::Time() {} // (a) with empty implementation, not recommended

Time::Time() : hour(0), minute(0), second(0) {} // (a) init to 0

Time::Time (int h, int m, int s) { // (b) with defaults
  setTime (h, m, s);
}
```

- Lo ideal es que el constructor inicialice los miembros de datos con 0.
  - Recuerda: al crear un objeto se hace una llamada al constructor.
  - Así pues, esto asegura que el objeto empieza en un estado consistente.
- Podemos definir varios constructores (sobrecargados) para una clase.
  - Ya veremos sobrecarga $\iff$ polisemia: misma palabra $\neq$ significados.
- Un constructor puede llamar a otras funciones miembro de la clase.
  - Pero cuidado; es el encargado de inicializar el objeto.
  - Si usa un atributo antes de inicializarlo $\implies$ error lógico.

```
int main() {
  Time t; // default values are set in all members (h=0,m=0,s=0)
  Time t2 (2); // h = 2, m = s = 0
  Time t3 (21, 34); // h = 21, m = 34, s = 0
  Time t4 (12, 25, 42); // values for h, m, s
  Time t5 (27, 74, 99); // wrong values (checked by setTime)
}
```

- Corolario: un constructor con *defaults* garantiza consistencia.
  - Tanto en caso de no poner valores como de ponerlos incorrectos.

- Un destructor es `public`, no recibe parámetros ni devuelve nada.
  - Si no se especifica explicitamente, el compilador crea uno vacío.
- $\nexists$ sobrecarga de destructores $\iff$ solo puede haber uno por clase.
- El destructor se invoca implícitamente cuando se destruye un objeto.
  - E.g. cuando se sale de un ámbito en el que se creó un objeto.
- El destructor no libera la memoria del objeto.
  - Prepara esa zona de memoria para ser reusada cuando se reclame.

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

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {
public:
   CreateAndDestroy( int, string ); // constructor
   ~CreateAndDestroy(); // destructor
private:
   int objectID; // ID number for object
   string message; // message describing object
}; // end class CreateAndDestroy

#endif
```

```
#include <iostream>
#include "CreateAndDestroy.h" // include class definition
using namespace std;

CreateAndDestroy::CreateAndDestroy (int ID, string msg) {
   objectID = ID; // set object's ID number
   message = msg; // set object's descriptive message

   cout << "Object " << objectID << " constructor runs "
      << message << endl;
} // end CreateAndDestroy constructor

CreateAndDestroy::~CreateAndDestroy () { // destructor
   // output newline for certain objects; helps readability
   cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );

   cout << "Object " << objectID << " destructor runs "
      << message << endl;
} // end ~CreateAndDestroy destructor
```

- El compilador llama implícitamente a constructores y destructores.
- El constructor de un obj. global se llama 1.º, antes incluso que `main`.
- Los destructores se invocan en el orden inverso a los constructores.
  - El último invocado será el de un obj. global (tras finalizar `main`).
- La ejecución *normal* del programa se puede interrumpir.
  - No es una buena práctica, porque no se ejecutarán los destructores.

```
#include <iostream>
#include "CreateAndDestroy.h" // include class definition
using namespace std;

void create (void); // prototype

CreateAndDestroy first (1, "(global before main)"); // global ob.

int main() {
   cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy second (2, "(local automatic in main)");
   static CreateAndDestroy third (3, "(local static in main)");

   create(); // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
   CreateAndDestroy fourth (4, "(local automatic in main)");
   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
} // end main
```

```
// function to create objects
void create (void) {
   cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy fifth (5, "(local automatic in create)");
   static CreateAndDestroy sixth (6, "(local static in create)");
   CreateAndDestroy seventh (7, "(local automatic in create)");
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
} // end function create
```

- Funciones: `exit(int code)`, `atexit()`, `abort()`, `return code`
- La función `exit(code)` obliga al programa a terminar de inmediato.
  - Es la forma menos brusca de terminar.
    - Ejecuta el destructor de los objetos `static` y los globales.
    - Cierra los ficheros que pudieran estar abiertos en operación I/O.
    - Fuerza la escritura de datos que quedasen en los buffers de I/O (*flush*).
    - Elimina los ficheros temporales que se estuviesen usando.
    - `code = 0` (o `EXIT_SUCCESS`) o `code = 1` (o `EXIT_FAILURE` o $\neq 0$).
- `atexit()` permite customizar `exit()` para realizar acciones extra.
- `abort()` es muy brusco: no hace nada de lo que hace `exit()`.
  - Levanta una excepción `SIGABRT`, pudiéndose —eso sí— manejar esta.
    - En tal manejador nosotros podemos cerrar las *cosas*.
- `return` o `return code` termina la ejecución del ámbito actual.
  - Devolviendo el control al ámbito desde el que fue llamado.

- Una referencia a objeto es un alias para el nombre del objeto.
  - Por tanto, puede usarse desde el lado izquierdo de una asignación.
- Se puede hacer método public que devuelva ref. a atributo private.

```
public:
  // [...]
  int &badSetHour (int); // (.h) DANGEROUS reference return
private:
   int hour;
  // [...]
```

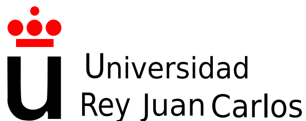
```
// [...]
// (.cpp) POOR PRACTICE: returning reference to a private member!
int &Time::badSetHour (int hh) {
  hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
  return hour; // DANGEROUS reference return
} // end function badSetHour
```

```
int main() {
  Time t; // create Time object

  // initialize hourRef with the reference returned by badSetHour
  int &hourRef = t.badSetHour (20); // 20 is a valid hour

  cout << "Valid hour before modification: " << hourRef;
  hourRef = 30; // use hourRef to set invalid value in object t
  cout << "\nInvalid hour after modification: " << t.getHour();

  // Dangerous: Function call that returns a reference
  //  can be used as an lvalue (left value modificable)!
  t.badSetHour(12) = 74; // assign another invalid value to hour

  cout << "\n\n****************************\n"
    << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
    << "t.badSetHour(12) as an lvalue, invalid hour: "
    << t.getHour() << "\n*****************************" << endl;
} // end main
```

3. Clases I. Generalidades

Julio Vega

julio.vega@urjc.es

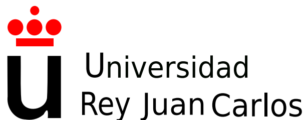


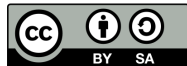
4. Clases II. Uso avanzado

Julio Vega

julio.vega@urjc.es

GSyC





©2024 **Julio Vega Pérez**
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. El modificador `inline`
2. El espacio de nombres o *namespaces*
3. El modificador `const`
4. Composición: objetos como atributos de clases
5. Funciones y clases `friend`
6. Uso del apuntador `this`
7. Gestión dinámica de memoria con `new` y `delete`
8. Miembros `static`

- Todos sabemos el mecanismo de la llamada a una función *normal*.
  - 1. Al llamarla, la CPU almacena la dirección de la siguiente instrucción.
  - 2. La CPU ejecuta la función, almacena el valor de retorno y *vuelve*.
- Puede suponer un sobrecoste comput. si $t_{ejec_funcion} < t_{cambio_contexto}$.
  - En función compleja compensa, ya que $t_{ejec_funcion} \ggg t_{cambio_contexto}$.
  - En función simple, puede ocurrir que $t_{ejec_funcion} < t_{cambio_contexto}$.
- C++ ofrece el tipo de función `inline` para paliar este problema.
  - Es una solicitud, no una orden; el compilador puede no ponerla en línea.
  - Una función declarada como `inline` no conlleva cambio de contexto.
  - El compilador *copia* el código de una función `inline`.
    - Y lo *pega* en cada sitio desde donde se llame, en t.^{o} de compilación.
  - Pros: $t_{cambio_contexto} \approx 0$. Contras: tamaño de ejecutable $\uparrow\uparrow$.
  - ¿Cuándo usar explícitamente `inline`? Con funciones muy simples.

- Un método de la clase se puede definir fuera de la definición de clase.
  - Y *enlazarse* a la clase mediante el operador de resolución binario.
- Lo normal es definir una función miembro en el cuerpo de la clase.
  - Porque el compilador intenta poner en línea las llamadas a estas.
  - Si están fuera se pueden poner en línea con la palabra `inline`.
- Otro beneficio de la POO es la simplificación en las llamadas.
  - El número de parámetros de los métodos de clase es reducido.
  - Los atributos y métodos están encapsulados en el objeto.
  - Los métodos tienen el derecho a acceder directamente a los atributos.

Uso redundante e inapropiado de inline dentro de una clase

```
class A {
public:
   inline int methodA () { // redundant use of inline
      // [...] this function is automatically inline
   }
}; // end class A
```

Uso redundante aunque apropiado de inline en la definición del método

```
class B {
public:
   int methodB (); // declare the function
};

inline int B::methodB () { // inline in the definition
  // redundante, pero "sugerencia fuerte" al compilador?
}
```

```
#include <iostream>
using namespace std; // to be analyzed
inline void smallFunction () { // inline function out of class
  cout << "Me gusta usar inline" << endl;
}

class C {
private:
  int x;
public:
  C (int x0) : x (x0) {}
  inline int getX () const { // to be analyzed
    return x; }
}; // end of class C

int main() {
  smallFunction();
  C c(7); cout << c.getX () << endl; return 0;
}
```

Uso generalizado y recomendado de `inline`

- La función *inline* no pertenece a la clase.
  - Por tanto, `inline` en este caso sí que tiene *efecto*.
- La función *inline* es muy simple; tiene poco código.
  - Se va a copiar su contenido en todos lo sitios desde los que se llame.
  - Pero el tamaño resultante del ejecutable no supondrá un problema.
- Las funciones *inline* deben ir directamente en el encabezado (.h).
  - No hay peligro de multi-definición.
    - La función no *inline* es implementada en un encabezado, y este es...
    - incluido en varios lugares $\Longrightarrow$ error compilación - definición múltiple.
  - El símbolo de la función *inline* no aparece explícit. en el enlazado.
    - Pues todas las llamadas a esta función han sido reemplazadas.
  - Al compilar un .h de ISV, la def. de la función pueda ir en su sitio.
    - ISV = *Independent Software Vendor* (o software privativo).

- Un *namespace* (ns) o espacio de nombre es un bloque o ámbito.
- Se usa para limitar el alcance de variables y funciones.
- Surge en C++ ante la limitación de no poder repetir identificadores.
  - En un mismo ámbito no se pueden repetir variables, funciones o clases.
- Problema: conflicto al importar librería con un identif. que ya usamos.
  - E.g. al importar OpenGL, que tiene función `print`, y nosotros también.
  - Solución: poner prefijo al identificador (e.g. `print` $\Longrightarrow$ `gl_print`).
- Para poder usar las variables o funciones que están en un `namespace`.
  - Hay que poner su *prefijo* con el operador de resolución de alcance (`::`).
  - Para no tener que usar el *prefijo*, se resuelve con `using`.
    - E.g. `using namespace std;` para usar la librería estándar.
    - Así, podemos usar `cout` en lugar de `std::cout`.
- Pros: $\uparrow$ legibilidad $\downarrow$ tamaño código. Contras: conflictos con otros *ns*.
  - E.g. uso dos ns *opencv* y *opengl* y uso una función común a ambos...
    - ¿A qué función se llamaría? ¡El resultado sería impredecible!

```
namespace saludo {
  void print () { std::cout << "hola";
}

void print () {
  std::cout << "adios";
}

int main() {
  print ();
  saludo::print ();
  return 0;
}
```

- *Algo* `const` indica al compilador que ha de prevenir su modificación.
  - E.g. si intentamos modificar una variable *const*, el compilador nos avisa.
- Una función miembro declarada como *const* es una *read-only func.*
  - E.g. en las *getters*, que solo consultan un valor, pero no lo modifican.
- Para usar una variable *const* desde otro módulo se le pone `extern`.
  - Tanto en la declaración de esta como en el módulo donde se usa.
    - E.g. (.h): `extern const int i = 2;` y (.cpp): `extern const int i;`
  - Esto no ocurre en C: `const int i = 2;` y `extern const int i;`

```
int main() { // Example A: it shows compile errors
  const int i = 5;
  i = 10; // C3892: assignment of read-only variable 'i'
  i++; // C2105: increment of read-only variable 'i'
}

int main() { // Example B: const value to specify size of array
  const int maxArray = 128;
  char storeChar [maxArray]; // allowed in C++, but not in C
}

int main() { // const objects: only const functions can be called
    const C c{}; // calls default constructor
    //c.x = 5; // (with x public) compiler error: violates const
    //c.setX(5); // compiler error: violates const
    c.getX(); // it's the only function that works
    return 0;
}
```

- Definir como `const` un método que modifica atributos del objeto.
- Definir como `const` un método que llama a un método no `const`.
- Invocar a un método no `const` en un objeto const.
- Declarar un constructor o destructor `const`.

Clase Increment con count (no const) e increment (const)

```
class Increment { // Definition of class Increment
public:
   Increment( int c = 0, int i = 1 ); // default constructor

   // function addIncrement definition
   void addIncrement() {
      count += increment;
   } // end function addIncrement

   void print() const; // prints count and increment

private:
   int count;
   const int increment; // const data member
}; // end class Increment
```

Inicializador de miembros utilizado para inicializar un atributo const

```
Increment::Increment (int c, int i) // constructor
   : count( c ), // initializer for non-const member
     increment( i ) { // required initializer for const member
   // empty body
} // end constructor Increment

void Increment::print() const { // print count & increment values
   cout << "count = "<<count<<", increment = "<<increment<<endl;
} // end function print
```

- Todos los atributos se pueden inicializar mediante inicializador.
- Los miembros `const` y miembros que sean referencias **deben**.
  - Para así proporcionar al constructor el valor inicial de los atributos.
- Ya veremos que los objetos miembro también **deben**.
  - Y con herencia, los fragmentos heredados por las clases hija también.
- Un objeto `const` no se puede modificar mediante la asignación.
  - Por lo que debe inicializarse.
- *Tip*: declara `const` todas los métodos que no modifiquen el objeto.
  - Aunque no haya intención de:
    - Crear objetos `const` de esa clase.
    - Acceder a objetos de esa clase a través de referencias `const`.
  - Así, si por error el método modifica el objeto $\Longrightarrow$ error de compilación.

- Pensemos en un objeto `Empleado` (de *Julio Veganos e Hijos*).
- Este empleado tendrá un objeto fecha (*date*) de nacimiento.
- ¿Por qué no incluir un objeto `Date` como miembro de `Employee`?
  - De hecho, un empleado tendría dos fechas: f. nac. y f. contrato.
- Esta capacidad se conoce como composición o relación *tiene un*.
  - Es decir, una clase puede tener objetos de otras clases como atributos.
- Una forma común de reutilización de software es la composición.

```
#ifndef DATE_H
#define DATE_H

class Date {
public:
   static const unsigned int monthsPerYear = 12; // in a year
   explicit Date( int = 1, int = 1, int = 1900 ); // def. const.
   void print() const; // print date in month/day/year format
   ~Date(); // provided to confirm destruction order
private:
   unsigned int month; // 1-12 (January-December)
   unsigned int day; // 1-31 based on month
   unsigned int year; // any year

   // utility funct. to check if day is proper for month and year
   unsigned int checkDay( int ) const;
}; // end class Date

#endif
```

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
#include "Date.h" // include Date class definition

class Employee {
public:
   Employee (const std::string &, const std::string &,
      const Date &, const Date &);
   void print() const;
   ~Employee(); // provided to confirm destruction order
private:
   std::string firstName; // composition: member object
   std::string lastName; // composition: member object
   const Date birthDate; // composition: member object
   const Date hireDate; // composition: member object
}; // end class Employee

#endif
```

- La clase `Date` y la clase `Employee` incluyen su destructor.
- Los objetos se construyen de dentro a afuera y se destruyen inversa/.
  - Si ponemos trazas a los const./destructores se puede apreciar.
- Un objeto miembro no necesita inicializarse de manera explícita.
  - A través de un inicializador de miembros.
  - Si no existe, se llama implícita/ a su constructor predeterminado.
- Si objeto `A` es atributo `public` de una clase `B`, no se viola...
  - ...encapsulamiento/ocultamiento de los atributos private del objeto `A`.
  - Pero sí se viola ocultamiento de la clase `B` que lo contiene.
  - Corolario: los atributos objeto deben ser `private` como los demás.

- Una función `friend` de una clase es la que se define fuera de esta.
  - Pero que tiene derecho a acceder a todos sus miembros (pub. y priv.).
- Se pueden declarar funciones independientes y clases `friend`.
  - Func. útiles para sobrecargar operadores, y tb. para clases iteradoras.
- Para declarar una función como `friend` de una clase.
  - Se pone `friend` al prototipo de la función en definición de clase.
- Para que todos los métodos de la clase `B` sean `friend` de clase `A`.
  - Declarar `friend class B;` en la definición de la clase `A`.
- Todo miembro puede ser `friend`: `private`, `protected` y `public`.
  - Todos los `friend` van al ppio. de clase sin especificar visibilidad.
- La relación `friend` no es simétrica ni transitiva.
  - Si A `friend` de B & B `friend` de C:
    - $\not\Rightarrow$ B `friend` de A (la amistad no es simétrica).
    - $\not\Rightarrow$ C `friend` de B (la amistad no es simétrica).
    - $\not\Rightarrow$ A `friend` de C (la amistad no es transitiva).
- `friend` puede corromper ocultamiento y debilitar POO. Ejemplos...

```
class A {
   friend void setX (A &, int); // friend declaration

public:
   A () : x (0) {};
   void print() const { cout << x << endl; }

private:
   int x;
};

void setX (A &a, int value) {
   a.x = value; // it is possible due to setX is A's friend
}
```

- Lo ideal sería definir la función `setX` como miembro de `A`.
- Archivo de encabezado (`A.h`) y archivo de implementación (`A.cpp`).
- Y en otro archivo separado, el programa de prueba (`main.cpp`).

- Hemos visto que los métodos pueden manipular los atributos de clase.
  - ¿Cómo saben los métodos qué atributos del objeto pueden manipular?
- ¿Recuerdas lo de clase = plano para construir un objeto de esa clase?
  - ¿Al ejecutar código de clase para objeto concreto, cómo sabe dónde?
- Cada objeto tiene acceso a su propia direc. mediante apuntador `this`.
  - Este apuntador se pasa implícitamente como un argumento más.

```
class A {
public:
  void print() const { // includes argument "this" (hidden)
    cout << x << endl; // implicit use
    cout << this->x << endl; // explicit use
    cout << (*this).x << endl; // explicit unreferenced use
// () is mandatory, otherwise: *this.x = *(this.x) => comp. error
  }

private:
  int x;
};
```

- C++ permite controlar la (des)asignación de memoria del programa.
- Uso: e.g. en la práctica, ¿cómo almacenar el nombre de un empleado?
  - Uso fijo: aprovecha mal la memoria (¿y si nombre más largo/corto?).
  - Uso dinámico: usar exactamente lo necesario según la necesidad.
- Toda colección dinámica creada se almacena en el *heap* o cúmulo.
  - Es una de las regiones de la memoria (RAM) asignada al programa.
  - Se accede a la colección mediante un apuntador al primer elemento.

```
Date *datePtr;
datePtr = new Date; // "new" calls the default constructor
delete datePtr; // "delete" calls the destructor & frees memory
// --------------------------
int *grades = new int [10]; // dynamic array with a static size
delete [] *grades; // "delete []" calls every element's destr.
                   // "delete" for a collection is a logic error!
```

- En general, un objeto tiene copia de todos los miembros de la clase.
- En ciertos casos, solo se necesita de algunos miembros; no de todos.
- Un miembro `static` es compartido por todas las instancias de clase.
  - Es decir, no es una propiedad de un objeto específico de la clase.
    - Es como una variable global pero con acceso a miembros de clase.
  - E.g. atributo `count` para llevar cuenta de instancias de la clase.
    - Si atributo es no `static`, cada objeto llevaría su propia cuenta.
    - Si atributo es `static`, este llevaría la cuenta *global* de la clase.
- Corolario: se usa `static` cuando solo se necesite una copia.

```
class A { // --------- A.h -----------------
public:
  A ();
  static int getCounter (); // return no. objects
private:
  static int counter; // no. objects
};
//--------------- A.cpp --------------------
int A::counter = 0; // can't include static keyword
int A::getCounter () { return counter; }
A::A () { counter++; }
//-------------- main.cpp ------------------
A *a = new A (); // constructor is called
cout << A::getCounter() << endl; // shows 1 (obj.)
delete a;
a = 0;
```

- En llamadas a func. miembro `static` se antepone nombre de clase.
- Usar `this` en func. miembro `static` es un error de compilación.
  - Una función miembro `static` no tiene apuntador `this`.
  - Un miembro `static` es independiente de cualquier obj. de la clase.
- Declarar `const` una func. miembro `static` es error de compilación.
  - `const` indica que la func. no puede modificar el contenido del objeto.
  - Pero func. miembro `static` opera independiente/ de los obj. de clase.
- Tras eliminar la memoria asignada de forma dinámica (`delete a;`).
  - Este espacio podría seguir teniendo información de lo que sea.
    - Y el programa podría acceder, obteniendo errores lógicos.
  - Una buena costumbre es establecer el puntero en 0 (`a = 0;`).
    - Así se desconecta el puntero del espacio que le había sido asignado.
    - Apuntando a un espacio al cual el programa no tiene acceso.

4. Clases II. Uso avanzado

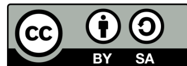
Julio Vega

julio.vega@urjc.es

5. Sobrecarga de operadores

Julio Vega

julio.vega@urjc.es



©2024 **Julio Vega Pérez**
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. Introducción
2. Ejemplos de sobrecarga de operadores
3. Función de operador de conversión (`cast`) sobrecargada
4. Sobrecarga de `++` y `--`
5. La clase `string` de la Bibl. estándar

- Comunicación entre objetos mediante sus funciones es incómoda.
  - E.g. clase `Matemática`, pensemos en las llamadas a sus funciones...
  - La sobrecarga logra un código más claro que con llamadas a funciones.
- Muchas manipulaciones comunes se realizan con operadores (`<<`).
  - Sobrecargar operadores es hacer que estos puedan manipular objetos.
- *La sobrecarga es la polisemia de los operadores.*
  - Polisemia: cuando una misma palabra tiene varios significados.
  - E.g. Banco: dinero, asiento. `<<`: op. flujo, *shift* bits.
- C++ no permite crear, pero sí sobrecargar muchos de sus operadores.
  - El compilador genera el código apropiado según el contexto.

Operadores que se pueden sobrecargar

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

Operadores que no se pueden sobrecargar

```
. .* :: ?:
```

- Mediante método no `static` o con función global.
- Nombre de función: `operator` seguida del símbolo a sobrecargar.

Def. `Phone.h` con operadores flujo sobrecargados como func. friend

```
#include <iostream>
#include <string>

using namespace std;

class Phone {
  friend ostream &operator<< (ostream &, const Phone &);
  friend istream &operator>> (istream &, Phone &);
private:
  string countryCode; // 2-digit country code (e.g. "34")
  string areaCode; // 3-digit area code (e.g. "924")
  string number; // 6-digit number
}; // end class Phone
```

Implementación de la clase Phone (Phone.cpp)

```
#include <iomanip>
#include "Phone.h"
using namespace std;
ostream &operator<< (ostream &output, const Phone &phone) {
  output << "(" << phone.countryCode << ") "
    << phone.areaCode << "-" << phone.number;
  return output; // enables cout << a << b << c;
} // end function operator<< [e.g. format: (+34) 924-736515]

istream &operator>> (istream &input, Phone &phone) {
  input.ignore (2); // skip ( and +
  input >> setw (2) >> phone.countryCode; // input country code
  input.ignore (2); // skip ) and space
  input >> setw (3) >> phone.areaCode; // input area code
  input.ignore (); // skip dash (-)
  input >> setw (6) >> phone.number; // input number
  return input; // enables cin >> a >> b >> c;
} // end function operator>>
```

Ejemplo de uso de la clase Phone, con operadores sobrecargados

```
#include <iostream>
#include "Phone.h"
using namespace std;

int main() {
  Phone phone; // create object phone
  cout << "Enter phone number (format (+34) 924-736515):\n";

  // cin >> phone invokes operator>> by implicitly issuing
  // the global function call operator>> (cin, phone)
  cin >> phone; // >> overloaded!

  cout << "The phone number entered was: ";

  // cout << phone invokes operator<< by implicitly issuing
  // the global function call operator<< (cout, phone)
  cout << phone << endl; // << overloaded!
} // end main
```

- (A) Método no `static` (sin args) o (B) func. global (con un arg.).

(A) Def. `Cadena.h` con operador ! sobrecargado como método

```
class Cadena { // String could be confused with C++ class
  bool operator!() const; // return if string is empty
}; // end class Cadena
```

(B) Declaraciones operador ! sobrecargado como funciones globales

```
bool operator! (const Cadena); // Op. 1: with an object as arg.
bool operator! (const Cadena &); // Op. 2: ref. to object as arg.
```

- Op. 1: se crea copia de obj.; la función no altera obj. original.
- Op. 2: no se hace copia de obj.; la función altera obj. original.

Uso del operador ! sobrecargado para devolver si cadena vacía

```
Cadena miCadena;
if (!miCadena) cout << "Empty string" << endl;
```

- Todo lo ya visto es aplicable a este tipo de operadores.
- (A) Método no `static` (un arg.) o (B) func. global (dos args).

(A) Def. `Cadena.h` con operador $<$ sobrecargado como método

```
class Cadena { // String could be confused with C++ class
  bool operator<() const; // return if left string is smaller
}; // end class Cadena
```

(B) Declaraciones operador $<$ sobrecargado como función global

```
bool operator< (const Cadena &, const Cadena &);
// could it be defined with two objects as args (as seen above)
```

- Los programas procesan información de muchos tipos.
- A veces, todas las operaciones *permanecen* dentro de un tipo.
  - E.g. al sumar un `int` con un `int` se produce un `int`.
- Pero muchas veces se necesita convertir datos de un tipo a otro.
  - E.g. cálculos, paso de valores a funciones, devolución de valores.
- Podemos usar operadores de conversión de tipos para forzarlas:

```
int main () {
   char c;
   int i = 34;
   float f = 1.5;
   double d;

   c = static_cast<char>(i); // int to char
   d = static_cast<double>(f); // float to double
}
```

- Pero, ¿y los tipos definidos por el usuario? El compilador no sabe.

- Para ello son necesarios los **constructores de conversión**.
  - Son const. con un arg. que convierte objetos de un tipo a otro.
  - `A::operator int () const; // obj. tipo A -> tipo int`
    - Esta función convierte obj. `A` (de usuario) en tipo `int`.
  - Y así, cuando usemos el operador `cast` de conversión...
    - `static_cast<int> (a) // Habiendo declarado A a;`
  - ...el compilador genera la llamada:
    - `a.operator int()`
- Estas func. deben ser un método no `static` (de la clase A).
- Y deben ser `const` porque no modifican el objeto original.
- Se pueden definir varias funciones de operador de conversión.
  - `A::operator char* () const; // obj. tipo A -> tipo char*`
  - `A::operator B () const; // obj. tipo A -> tipo B`

- Tanto en prefijo como postfijo, se pueden sobrecargar.
- Las funciones son distintas, así el compilador distingue versiones:

```
A &operator++ (); // pre-increment (++a) as a class method
A &operator++ (A &); // pre-increment (++a) as a global function
A operator++ (int); // post-increment (a++) as a class method
A operator++ (A &, int); // post-incr. (a++) as a global function
```

- Todo lo anterior es igualmente aplicable al decremento.

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

int main() {
  string s1 ("Hello");
  string s2 (" world!");
  string s3;

  cout << "s1 = \"" << s1 << "\"; s2 = \"" << s2
    << "\n(s1 == s2): " << (s1 == s2 ? "true" : "false")
    << "\n(s1 < s2): " << (s1 < s2 ? "true" : "false")
    << "\n(s1 >= s2): " << (s2 >= s1 ? "true" : "false");

  if (s3.empty()) {
    s1 += s2;
    s3 = s1;
  }
  cout << "\ns3 = \"" << s3 << endl;
}
```

5. Sobrecarga de operadores

Julio Vega

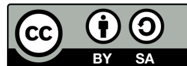
julio.vega@urjc.es

6. Herencia y polimorfismo

Julio Vega

julio.vega@urjc.es

GSyC



©2024 **Julio Vega Pérez**
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. Introducción
2. Clase base vs. clase derivada
3. Miembros `protected`
4. Ejemplo: `CommissionProgrammer` y `SalariedProgrammer`
5. Análisis del ejemplo
6. Polimorfismo
7. Clase abstractas y funciones virtuales puras

- Una de las principales virtudes de la POO es la herencia.
- Es una forma de reutilización de software.
- Para crear la nueva clase, se *heredan* datos/comportamientos de otra.
  - Clase base = clase existente VS. clase derivada = clase nueva.
  - Métodos de la derivada no acceden a atributos `private` de la base.
  - Y las funciones `friend` no se heredan (no eran de la clase...).
- La clase derivada contiene lo del padre + funcionalidad adicional.
  - Ya depende de si es herencia `public`, `protected` o `private`.
  - De momento, nos centraremos en la herencia de tipo `public`.
    - Todo objeto de clase derivada es objeto de la clase base.
- C++ soporta herencia múltiple (heredar de varias).
  - No así en Java, con buen criterio; evita complejidad y errores.
  - Para ello, en Java, manejan el concepto de `interfaces`.

- Las relaciones de herencia forman estructuras jerárquicas.
  - Una clase se convierte en base a otras, derivada de otras, o en ambas.
- E.g. `Vehiculo` (clase base) y `Moto` (clase derivada):
  - Todas las motos son vehículos, pero no todos los vehículos son motos.
- Para evitar confusión, léase: una moto **es un** vehículo $\Longrightarrow$ ¡herencia!
  - VS. *tiene un* $\Longrightarrow$ composición. E.g. una moto *tiene un* manillar.
- Otro ejemplo, con la clase `PersonaUniversidad`.
  - Definición clase: `class Profesor : public PersonaUniversidad`
  - La relación de herencia, en UML, se representa por esa flecha *hueca*.
  - La lectura siempre en el sentido de la flecha.

Figura: Un profesor *es una* persona/figura de la universidad

Figura: Una persona de la comunidad universitaria *es una* persona

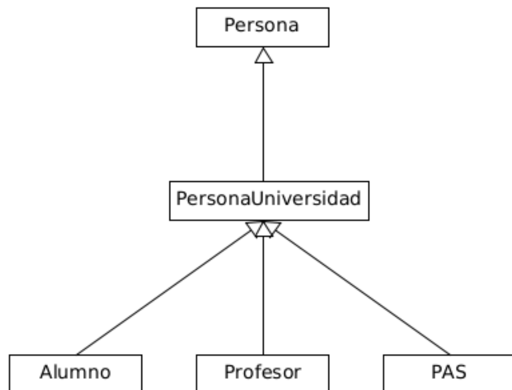


Figura: Alumno y PAS *son* personas de la universidad

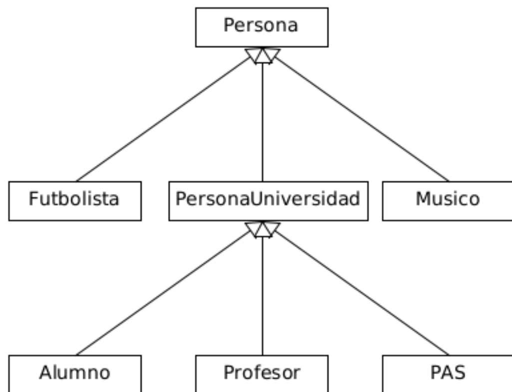


Figura: Un futbolista o un músico también *son* personas

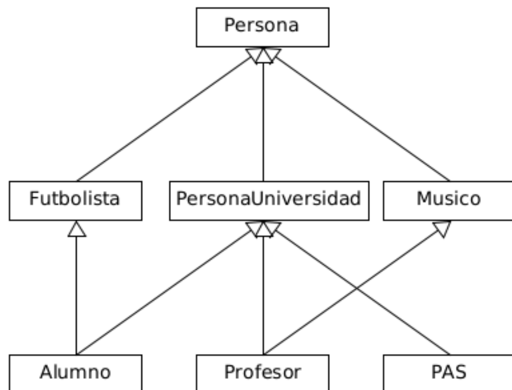


Figura: Un alumno *es*, además, futbolista. Un profesor *es*, además, músico

- Miembros `public` de clase base son accesibles desde esta.
- También desde las funciones `friend` de esta clase base.
  - Y desde cualquier parte donde haya un manejador (objeto, puntero).
- Miembros `private` de clase base solo accesibles desde dentro.
  - Y también desde las funciones `friend` de esta clase base.
- Miembros `protected` son como `public` para las clases derivadas.
  - Se accede desde dentro, mediante `friend`, e igual para las derivadas.
  - Si derivada redefine método, accede a este con `nombreBase::metodo`

```
class CommissionProgrammer {
public:
  CommissionProgrammer (const string &, const string &, double =
      0.0, double = 0.0);
  void setName (const string &);
  string getName () const;
  void setSSN (const string &);
  string getSSN () const;
  void setNumberOfProjects (double);
  double getNumberOfProjects () const;
  void setCommissionRate (double);
  double getCommissionRate () const;
  double earnings() const; // calculate earnings
  void print() const; // print CommissionProgrammer object
private:
  string name;
  string ssn; // Social Security Number
  double numberOfProjects;
  double commissionRate;
}; // end class CommissionProgrammer
```

```
CommissionProgrammer::CommissionProgrammer (const string &name,
    const string &ssn,
  double projects, double rate) : name (name), ssn (ssn) {
  setNumberOfProjects (projects); // validate/store no. projects
  setCommissionRate (rate); // validate and store commission rate
}

void CommissionProgrammer::setName (const string &name) {
  this->name = name;
}
string CommissionProgrammer::getName () const {
  return name;
}

void CommissionProgrammer::setSSN (const string &ssn) {
  this->ssn = ssn;
}
string CommissionProgrammer::getSSN () const {
  return ssn;
}
```

```
void CommissionProgrammer::setNumberOfProjects (double projects)
    {
  if (projects >= 0.0)
    numberOfProjects = projects;
  else
    throw invalid_argument ("No. of projects must be >= 0.0" );
}

double CommissionProgrammer::getNumberOfProjects () const {
  return numberOfProjects;
}

void CommissionProgrammer::setCommissionRate (double rate) {
  if (rate > 0.0 && rate < 100.0)
    commissionRate = rate;
  else
    throw invalid_argument ("Commission rate must be > 0.0 and <
        100.0");
}
```

```
double CommissionProgrammer::getCommissionRate () const {
   return commissionRate;
}

double CommissionProgrammer::earnings() const {
  return getCommissionRate() * getNumberOfProjects();
}

void CommissionProgrammer::print() const {
  cout << "commission programmer: " << getName()
    << "\nsocial security number: " << getSSN()
    << "\nnumber of projects: " << getNumberOfProjects()
    << "\ncommission rate: " << getCommissionRate();
}
```

```
class SalariedProgrammer : public CommissionProgrammer {
public:
  SalariedProgrammer (const std::string &, const std::string &,
    double = 0.0, double = 0.0, double = 0.0);

  void setSalary (double);
  double getSalary() const;

  double earnings() const; // calculate earnings
  void print() const; // print SalariedProgrammer object
private:
  double salary; // base salary
}; // end class SalariedProgrammer
```

```
SalariedProgrammer::SalariedProgrammer(const string &name, const
    string &ssn,
   double projects, double rate, double salary)
   // explicitly call base-class constructor
   : CommissionProgrammer (name, ssn, projects, rate) {
   setSalary (salary); // validate and store base salary
}

void SalariedProgrammer::setSalary (double salary) {
  if (salary >= 0.0)
    this->salary = salary;
  else
    throw invalid_argument ("Salary must be >= 0.0");
}

double SalariedProgrammer::getSalary () const {
  return salary;
}
```

```
double SalariedProgrammer::earnings () const {
  // call base-class earnings function:
  return getSalary() + CommissionProgrammer::earnings();
}

void SalariedProgrammer::print () const {
  cout << "salaried ";

  // call base-class print function:
  CommissionProgrammer::print();

  cout << "\nbase salary: " << getSalary();
}
```

- Podemos observar la limpieza y el ahorro de código al heredar.
  - Si en vez de ello duplicáramos código, los errores se multiplicarían.
- Con herencia, atrib. y métodos comunes se declaran solo en la base.
  - Si hubiera que modificarlos, se modifican solo una vez, en la base.
  - Y la clase derivada automáticamente se beneficia de esos cambios.
- En el ejemplo, la derivada solo ha de añadir la cualidad de asalariado.
  - Este es el ejemplo típico de cuándo y cómo usar herencia.
- La herencia, en la clase derivada, se representa por `: nombreBase`
- Se usa `this` para evitar confusión entre parámetros y atributos.
  - E.g. `this->salary = salary;`

- En la clase derivada se incluye el `.h` de la clase base debido a que:
  - La clase derivada utiliza el nombre de la clase base.
  - El compilador usa la def. de clase (.h) para saber el tamaño del objeto.
    - Con herencia, tal tamaño depende de los atrib. de su clase y de la base.
- Cuando se redefine una función, esta puede llamar a la función base.
  - `SalariedProgr::print()` $\Longrightarrow$ `CommissionProgr::print()`
  - La llamada de func. derivada a func. base es `claseBase::funcion()`.
    - Si no, ¡la función derivada se estaría llamando a sí misma en bucle!

- La derivada no hereda const., dest. ni op. asignación sobrecargados.
  - Pero todo ello de la derivada puede llamar a sus homónimos de la base.
- Al crear ob. de clase derivada se genera cadena de llamadas a const.
  - El constructor de la derivada, antes de sus tareas, invoca al de la base.
    - Explícita/ (ejemplo), o implícita/ (se llama al const. pred. de la base).
  - Si base deriva de otra clase, se invoca a const. hacia arriba sucesiva/.
  - El último const. llamado será el de la *cima* de la jerarquía.
    - Y el cuerpo del const. de la derivada termina de ejecutarse el último.
- En ejemplo, el const. de la clase base inicializa atrib. de la base.
  - Una mejora a esto sería validar tanto el nombre como el NSS.
- Al destruir objeto de clase derivada, se llama a su destructor.
  - Este realiza su tarea y después invoca al destr. de la base.
  - La cascada de llamadas de destr. se ejecutan inversa/ a los constr.

- Al compilar, hemos de tener en cuenta la jerarquía de la herencia.
- Por comodidad, lo habitual es implementar un archivo Makefile:

```
CC = g++            # specified the compiler g++
CFLAGS = -Wall -g # shows all Warnings and adds debugging symbols
.RECIPEPREFIX = > # uses '>' as prefix of my recipes since all...
# actions of rules are identified by tabs and I do not use them

main: main.o CommissionProgrammer.o SalariedProgrammer.o
>$(CC) $(CFLAGS) -o main main.o CommissionProgrammer.o
    SalariedProgrammer.o
main.o: main.cpp CommissionProgrammer.h SalariedProgrammer.h
>$(CC) $(CFLAGS) -c main.cpp
CommissionProgrammer.o: CommissionProgrammer.h
SalariedProgrammer.o: SalariedProgrammer.h CommissionProgrammer.h
```

- El polimorfismo permite programar en general en lugar de específica/.
- El polimorfismo es un concepto que va muy unido a la herencia.
  - Permite procesar obj. de clases que formen parte de una jerarquía...
  - ...como si todos fueran objetos de la clase base de dicha jerarquía.
- E.g.: clase base `Robot` y derivadas: `Nao`, `Pepper` y `Whiz`.
  - Son los tres robots de la compañía *SoftBank Robotics*.
  - Tenemos una colección (vector) de punteros a varios de estos robots.
  - Para moverlos hacia adelante hay una función `moveForward()`.
  - Programa *los mueve* iterativa/ llamando siempre a la misma función.
  - Pero cada objeto sabe cómo *moverse* según su morfología.
  - ¡Esto es el polimorfismo: 1 función $= X$ formas de resultados!
- El polimorfismo facilita la reutilización de código.
  - Cuando empresa cree nuevo robot, solo necesita crear la nueva clase.
  - Pero el programa podrá seguir manteniendo su estructuras.
    - E.g.: el vector podrá albergar al nuevo robot y todo funcionará igual.

- **En tiempo de compilación**. Lo que habíamos tratado hasta ahora.
    - Sobrecargando las funciones o los operadores.
    - Recuerda: mismo nombre o símbolo y distintos comportamientos.
        - Para ello, debían tener diferentes *n.º* y/o tipos de parámetros.
        - O redefiniendo comportamiento (en herencia): e.g. `print` Sec. 17.
- **En tiempo de ejecución**. Este tipo solo se puede dar en herencia.
    - Sobrescribiendo las funciones de tipo `virtual` en las clases derivadas.
        - Es la idea que se introducía con el ejemplo de los robots.
    - Ahora veremos ejemplo con funciones `virtual` y no virtual.

Clase base con dos funciones: virtual y no virtual

```
class Base {
public:
  virtual void print () {
    cout << "print base class" << endl;
  }

  void show () {
    cout << "show base class" << endl;
  }
};
```

Clase derivada sobrecargando las funciones

```
class Derived : public Base {
public:
  void print () { // print () is already virtual function in
      derived class. We could also declared as virtual void
      print () explicitly
    cout << "print derived class" << endl;
  }

  void show () {
    cout << "show derived class" << endl;
  }
};
```

Uso del polimorfismo con funciones virtual vs. *no-virtual*

```
int main() {
  Base *bptr;
  Derived d;
  bptr = &d;

  // virtual function, binded at runtime (runtime polymorphism)
  bptr->print();

  // non-virtual function, binded at compile time
  bptr->show();

  return 0;
}
// ******** output:
// print derived class
// show base class
```

- Asignado direc. de obj. de clase derivada a puntero de la clase base.
  - `Base *bptr; Derived d; bptr = &d;`
  - C++ lo permite: obj. de clase derivada *es un* obj. de la clase base.
- Lo contrario, con toda lógica, genera error de compilación en C++.
  - Por ello, tpc se pueden invocar métodos derivados desde obj. base.

- Las funciones virtuales son las que hacen brillar al polimorfismo.
  - Permiten que el programa determine dinámica/ (al usar puntero `->`).
  - ...de qué clase derivada se va a ejecutar una determinada función.
    - Proceso denominado **vinculación dinámica o en tiempo de ejecución**.
  - También se pueden invocar usando el operador punto `.`
    - La invocación se resuelve en t. de compilación: **vinculación estática**.
    - En el ejemplo, sería: `d.print();`
- Si función `virtual` $\Longrightarrow$ lo será siempre (hacia abajo de jerarquía).
  - Aunque, para evitar confusión, al heredar, se puede poner `virtual`.
- Diferencia entre **redefinir** y **sobrescribir** al heredar un método:
  - Si el método es no virtual, podemos redefinirlo.
  - Si el método es virtual, podemos sobrescribirlo.

- Al definir una clase, se supone que se van a crear objetos de ese tipo.
- Pero a veces es útil definir clases de las que no se van a crear objetos.
    - A estas clases se las conoce como **clases abstractas**.
- Lo habitual es crear *clases base abstractas*, que estarán incompletas.
    - Y son las clases derivadas las que deben definir *lo que falta*.
- ¿Qué sentido tienen las clases abstractas?
    - Establecer un agrupamiento para aquellas clases que sean *hermanas*.
- Vemos ejemplo claro en la clasificación jerarquizada de los seres vivos.
    - Por ejemplo, todos los mamíferos tienen características comunes.
        - Pero no veremos a un mamífero por la calle $\Longrightarrow$ concepto abstracto.
        - Veremos alguno de sus tipos derivados: persona, perro, etc.
    - `Mamifero` = clase base abstracta y `Persona`/`Perro` = derivadas.

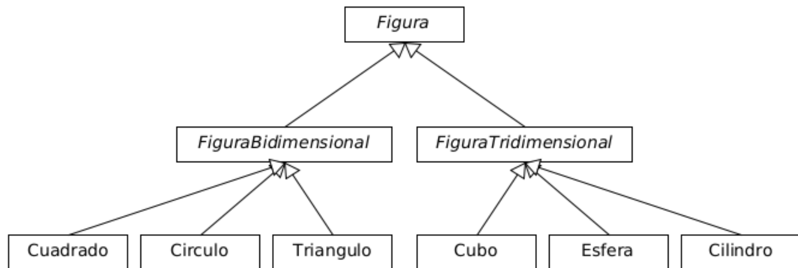


Figura: Abstractas: `Figura`, `FiguraBidimensional` y `FiguraTridimensional`

- Nótese que las clases abstractas se escriben en cursiva en UML.
- No podemos dibujar figura bidimensional $\Longrightarrow$ concepto abstracto.
  - Podríamos dibujar un cuadrado, o un círculo o un triángulo.

- Clase abstracta se da cuando 1 o más funciones virtuales son puras.
- Función virtual pura: `virtual void dibujar() const = 0;`
  - El `=0` se conoce como un **especificador puro**.
- Las funciones virtuales puras no proporcionan implementación.
  - Cada clase derivada concreta **debe** sobrescribirlas todas.
  - Esta es la diferencia entre virtual *pura* y *no pura*.
    - Recuerda que en función virtual no pura *se puede* sobrescribir.
- Instanciar un objeto de clase abstracta produce error de compilación.
  - Pero la clase base abstracta sí se puede usar para declarar punteros.
    - Recuerda que esta es la magia del polimorfismo.

- Hasta hemos visto los destructores *normales* (no virtuales).
- En polimorfismo, al destruir obj. derivado con `delete basePtr;`
  - Se genera comportamiento indefinido según el estándar de C++.
- Solución: implementar un destructor virtual en la clase base.
  - Así, al hacer `delete basePtr;` se invoca al destr. derivado concreto.
  - Y después, recuerda, se ejecuta el destructor de la base automática/.

6. Herencia y polimorfismo

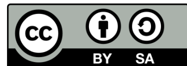
Julio Vega

julio.vega@urjc.es

7. Plantillas. Librería STL

Julio Vega

julio.vega@urjc.es



©2024 **Julio Vega Pérez**
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. Introducción
2. Plantillas de funciones
3. Plantillas de clases
4. Biblioteca de plantillas estándar (STL)

- Las plantillas persiguen el objetivo de siempre: reutilizar código.
- Existen **plantillas de funciones** y **plantillas de clases**.
- Permiten especificar, con un código, varias func./clases relacionadas.
  - A esta técnica se le conoce como **programación genérica**.
  - Las variedades generadas son las **especializaciones de las plantillas**.
- Son como las plantillas que usábamos de pequeño para trazar figuras.
  - Una especialización sería dibujar la figura en azul; otra, en verde, etc.
- E.g. plantilla de clase tipo pila que valga para cualquier tipo de elem.
  - C++ genera las especializaciones de plantillas de clases separadas.
    - Tamaño código = implementación func. sobrecargadas por separado.
    - E.g. *clase pila de* `int`, otra de `float`, otra de `string`, etc.
- Normal/ las plantillas se definen en el `.h` y se incluyen en los fuentes.

- Las funciones sobrecargadas realizan operaciones similares.
- Las plantillas de funciones realizan operaciones idénticas.
  - Lo único que cambian son los tipos de datos sobre los que operan.
- La definición de plantilla de función comienza por `template`.
  - Seguida por los parámetros de plantilla entre `<` y `>`.
  - A cada parámetro se le antepone la palabra `class` o `typename`.
    - Significan *cualquier tipo integrado o definido por el usuario*.

```
template <typename T>

template <class Mamifero>

template <typename Alumno, typename Profesor>
```

- La función se define después y su sintaxis es como siempre.

```
template <typename T>
void printArray (const T * const array, int numElems) {
  // Si T es un tipo definido por el usuario, el operador <<
     debe estar sobrecargado para ese tipo; si no, no compila
  for (int i = 0; i < numElems; i++) cout << array [i] << " ";
}

int main() {
  const int sizeA = 6, sizeB = 7, sizeC = 8; // arrays size
  int arrayA [sizeA] = {1, 2, 3, 4, 5, 6};
  double arrayB [sizeB] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
  char arrayC [sizeC] = "WELCOME"; // 8th position for null
  cout << "Array A: " << endl;
  printArray (arrayA, sizeA); // receives an int-array
  cout << "Array B: " << endl;
  printArray (arrayB, sizeB); // receives a double-array
  cout << "Array C: " << endl;
  printArray (arrayC, sizeC); // receives a char-array
}
```

- Pensemos en una funcionalidad independiente de los datos a manejar.
  - E.g. una pila, cuyo comportamiento es independiente de los elementos.
  - Podemos crear esa *plantilla* de comportamiento = gran reutilización sw.
- Se escribe solo una definición de plantilla de clase.
  - Y por cada especialización de instancia, se indica el tipo a usar.

```
// Class-template definition:
template <typename T>
class Stack { ... };
// [...]


// Use of Stack class:
Stack <double> miDoubleStack (5);
Stack <int> miIntStack (12);
// [...]
```

```
template <typename T>
class Stack {
public:
  Stack (int = 20); // (default Stack size = 20)
  ~Stack() { delete [] stackPtr; }
  bool push (const T &); // push an element onto the Stack
  bool pop (T &); // pop an element off the Stack
  bool isEmpty() const { return top == -1; }
  bool isFull() const { return top == size - 1; }
private:
  int size; // number of elements in the stack
  int top; // location of the top element (-1 means empty)
  T *stackPtr; // pointer to internal representation of the Stack
}; // end class template Stack

template <typename T>
Stack <T>::Stack (int s): size (s > 0 ? s : 20), // validation
  top (-1), // initially is empty
  stackPtr (new T [size]) { } // allocate memory for elements
```

```
template <typename T>
bool Stack <T>::push (const T &elem) {
  if (!isFull()) {
    stackPtr [++top] = elem; // place elem on Stack
    return true; // push successful
  } // otherwise:
  return false; // push unsuccessful
}

template <typename T>
bool Stack <T>::pop (T &elem) {
  if (!isEmpty()) {
    elem = stackPtr [top--]; // remove elem from Stack
    return true; // pop successful
  } // otherwise:
  return false; // pop unsuccessful
}
```

```
int main() {
  Stack <double> dStack (5);
  double dValue = 1.0;
  while (dStack.push (dValue)) { // pushing 5 doubles
    cout << dValue << ' ';
    dValue += 1.0;
  }
  while (dStack.pop (dValue)) // poping elements
    cout << dValue << ' ';

  Stack <int> iStack; // default size 20
  int iValue = 1;
  while (iStack.push (iValue)) { // pushing 20 integers
    cout << intValue++ << ' ';
    iValue += 1.1;
  }
  while (intStack.pop (intValue)) // poping elements
    cout << intValue << ' ';
} // end main
```

- Programadores usan normal/ muchas estruct. de datos y algoritmos.
  - Por ello, el comité del estándar de C++ agregó la STL a este.
- La STL está formada por contenedores, iteradores y algoritmos.
  - Permite escribir códigos que no dependen del contenedor subyacente.
    - Este estilo de programación se conoce como **programación genérica**.
- Los **containers** son estr. datos que almacenan todo tipo de datos.
  - Cada contenedor tiene asociadas varias funciones miembro.
- Los **iter.** ($\sim$ punteros) son para manipular los elem. de los *containers*.
  - Se pueden usar punteros pero iteradores quedan mucho más *elegantes*.
- Los **algoritmos** son funciones que hacen manipulaciones comunes.
  - E.g. búsqueda, ordenación y comparación de elementos.
  - STL proporciona unos 70 algoritmos aprox., la mayoría con iteradores.

- Pueden ser de secuencia, asociativos o adaptadores de contenedores.
  - Los adapt. de cont. tb se llaman de 2.ª clase; y, los otros, de 1.ª.
- **De secuencia**: representan estruc. datos lineales (vector, lista enl.).
  - `vector`: con acceso directo e inserción/eliminación rápida al final.
  - `deque`: ídem al anterior, pero ins./elim. rápida en inicio y final.
  - `list`: lista con enlace doble, ins./elim. rápida en cualquier parte.
- **Asociativos**: son no lineales, y pueden encontrar elem. rápidamente.
  - `set`: búsqueda rápida, no se permiten duplicados.
  - `multiset`: ídem al anterior, pero sí se permiten duplicados.
  - `map`: asignación uno-uno, $\nexists$ duplicados, búsqueda rápida por claves.
  - `multimap`: ídem al anterior, pero asignación uno-varios.
- **Adapt. de cont.**: versiones restringidas de los cont. anteriores.
  - `stack`: último en entrar, primero en salir (LIFO).
  - `queue`: primero en entrar, primero en salir (FIFO).
  - `priority_queue`: elem. de más prioridad es el primero en salir.

- Constructor predeterminado, const. de copia y destructor.
- `empty`: devuelve `true` si no hay elem. en el cont.
- `insert`: inserta un elemento en el contenedor.
- `size`: devuelve el n.º de elementos del contenedor.
- `capacity`: n.º elem. que caben antes de cambiar tamaño dinámica/.
- `operator=`: asigna un contenedor a otro.
- `operator<`: devuelve `true` si el 1.er cont. es menor que el 2.º.
  - Y siguiendo la misma dinámica, los comparadores `<=`, `>`, `>=`, `==`, `!=`.
- `swap`: intercambia los elementos de los contenedores.

- `max_size`: devuelve el n.º máx. de elementos para un cont.
- `begin`: devuelve iterador que apunta al primer elemento del cont.
    - ∃ dos versiones según devuelva un `iterator` o un `const_iterator`.
- `end`: devuelve iterador a la siguiente posición del último elem.
    - ∃ dos versiones según devuelva un `iterator` o un `const_iterator`.
- `rbegin` y `rend`: consideran la estructura a la inversa (*reverse*).
    - `rbegin`: devuelve iterador que apunta al último elemento del cont.
    - `rend`: devuelve iterador a la posición de antes del primer elem.
    - ∃ dos versiones, con `reverse_iterator` o `const_reverse_iterator`.
- `erase`: elimina uno o varios elementos del contenedor.
- `clear`: elimina todos los elementos del contenedor.

- `<vector>`
- `<list>`
- `<deque>`
- `<queue>` Contiene tanto a `queue` como a `priority_queue`.
- `<stack>`
- `<map>` Contiene tanto a `map` como a `multimap`.
- `<set>` Contiene tanto a `set` como a `multiset`.
- `<valarray>`
- `<bitset>`

Todo el contenido de estos archivos cabecera está en el `namespace std`.

- `allocator_type`: tipo de obj. usado para asignar memoria del cont.
- `value_type`: tipo de elem. almacenado en el contenedor.
- `reference`: ref. al tipo de elem. almacenado en el contenedor.
- `const_reference`: ref. cte. al tipo de elem. almacenado en el cont.
    - Esta solo se puede usar para leer elem. e insertarlos en el cont.
    - También para realizar operaciones tipo `const`.
- `pointer`: puntero al tipo de elem. almacenado en el contenedor.
- `const_pointer`: puntero cte. al tipo elem. almacenado en el cont.
- `iterator`: iterador que apunta al tipo elem. almacenado en el cont.
- `const_iterator`: iterador similar al `const_reference`.
- `reverse_iterator`: iterador para iterar en sentido inverso.
- `const_reverse_iterator`: como `const_iterator` en sent. inverso.
- `difference_type`: tipo resultado al restar dos iter. del mismo cont.
- `size_type`: tipo usado para contar elem. en un contenedor.

- Los iteradores son muy similares a los punteros, pero más sofisticados.
- Si `i` apunta a elem., `++i` apunta al sig. y `*i` es el contenido de `i`.
- El it. que vierte `end` se usa normal/ en comparación para saber si fin.
- Tipo `iterator` se usa para referenciar elem. que puede modificarse.
  - Y el `const_iterator` para ref. elemento que no puede modificarse.
- Los contenedores `vector` y `deque` admiten it. de acceso aleatorio.
  - `list`, `set`, `multiset`, `map` y `multimap`, solo bidireccional.
  - `stack`, `queue` y `priority_queue` no soportan iteradores.
- Para operar con flujos: `istream_iterator` y `ostream_iterator`.
  - Estos flujos pueden estar en contenedores o pueden ser flujos de I/O.

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

int main() {
  cout << "Enter a pair of integers: ";
  istream_iterator <int> inInt (cin); // read integers << cin

  int n1 = *inInt; // first value from standard input
  ++inInt; // shift iterator to next value
  int n2 = *inInt; // second value from standard input

  ostream_iterator <int> outInt (cout); // write integers >> cout

  cout << "Adding numbers: ";
  *outInt = n1 + n2;
  cout << endl;
}
```

- Se pueden utilizar genéricamente mediante diferentes contenedores.
    - Tb. pueden ser extendidos sin modificar los contenedores de la STL.
- Los alg. operan sobre los elementos indirectamente, con iteradores.
    - Por contra de un diseño puro de clases donde métodos = algoritmos.
    - Para facilitar tener alg. genéricos aplicables a muchos contenedores.
- ∃ diferentes tipos de algoritmos, según actúen sobre los contenedores.
    - Algoritmos de secuencia cambiantes: modifican los contenedores.
    - Algoritmos de secuencia no cambiantes: no producen modificaciones.
        - Contenedores de secuencia (cambiante o no): `vector`, `list` y `deque`.
    - Algoritmos numéricos (librería `<numeric>`): op. numéricas.

De secuencia cambiantes

`copy` `remove` `reverse_copy` `copy_backward` `remove_copy` `rotate` `fill` `remove_copy_if` `rotate_copy` `fill_n` `remove_if` `stable_partition` `generate` `replace` `swap` `generate_n` `replace_copy` `swap_ranges` `iter_swap` `replace_copy_if` `transform` `partition` `replace_if` `unique` `random_shuffle` `reverse` `unique_copy`

De secuencia no cambiantes

`adjacent_find` `find` `find_if` `count` `find_each` `mismatch` `count_if` `find_endsearch` `equal` `find_first_of` `search_n`

Numéricos (`<numeric>`)

`accumulate` `partial_sum` `innerproduct` `adjacent_difference`

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

template <typename T> void printV (const vector <T> &intV);

int main() {
  const int V_SIZE = 8;
  int array [V_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8};
  vector <int> integers; // vector of integers

  cout << "Content before adding elements:"
    << "\n-integers size: " << integers.size()
    << "\n-integers capacity: " << integers.capacity();

  // function push_back adds a new element at the end
  integers.push_back (31);
  integers.push_back (66);
  integers.push_back (11);
```

```
  cout << "\nContent after using push_back:"
    << "\n-integers size: " << integers.size()
    << "\n-integers capacity: " << integers.capacity();

  cout << "\n\nShowing array content using pointers: ";
  for (int *ptr = array; ptr != array + V_SIZE; ptr++)
    cout << *ptr << ' ';

  cout << "\nShowing vector content using iterators: ";
  printV (integers);

  cout << "\nShowing vector content in reverse mode: ";
  vector <int>::const_reverse_iterator rIt;
  vector <int>::const_reverse_iterator tempIt = integers.rend();
  for (rIt = integers.rbegin(); rIt != tempIt; ++rIt)
    cout << *rIt << ' ';

  cout << endl;
} // end main
```

```
// function template for displaying vector elements
template <typename T> void printV (const vector <T> &intV) {
  typename vector <T>::const_iterator constIt;

  // showing array content using a const_iterator
  for (constIt = intV.begin(); constIt != intV.end(); ++constIt)
    cout << *constIt << ' ';
}
```

- Nótese que, tras usar `push_back` tres veces, `capacity = 4`.
  - El comportamiento de `push_back` depende de implementación de STL.
  - En este caso, el vector está aumentando su tamaño al doble.
    - `p_b` → `capacity=1`, `p_b` → `capacity=2`, `p_b` → `capacity=4`.
    - Esta implementación puede provocar gran desperdicio de memoria.
  - Para controlar uso de memoria ⟹ usar funciones `resize` y `reserve`.

```
#include <iostream>
#include <vector>
#include <algorithm> // includes copy algorithm
#include <iterator>
#include <stdexcept>
using namespace std;

int main() {
  const int V_SIZE = 8;
  int array [V_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8};
  vector <int> integers (array, array + V_SIZE);
  ostream_iterator <int> outIt (cout, " ");

  cout << "Showing vector content using output stream: ";
  copy (integers.begin(), integers.end(), outIt);

  cout << "\nThe first element is: " << integers.front()
    << "\nAnd the last one is: " << integers.back();
```

```
integers [0] = 31; // 0 is the first indexed position
integers.at (2) = 23; // another way to set an elem.
integers.insert (integers.begin() + 1, 77); // to 2nd pos.

cout << "\n\nVector content (after changes): ";
copy (integers.begin(), integers.end(), outIt);

try { // trying to access an out-of-range element
  integers.at (100) = 777;
} // an exception will be raised!
catch (out_of_range &outOfRange) { // out_of_range exception
  cout << "\nException: " << outOfRange.what();
} // end catch
```

- Ya veremos excepciones en profundidad. En STL existen estos tipos:
  - `out_of_range`: indica cuando el subíndice está fuera de rango.
  - `invalid_argument`: si se pasa un argumento inválido a una función.
  - `length_error`: si se intenta crear un contenedor demasiado largo.
  - `bad_alloc`: al intentar asignar memoria con `new` y $\nexists$ memoria.

```
  integers.erase (integers.begin()); // erase first element
  cout << "\n\nVector content after erasing the first elem.: ";
  copy (integers.begin(), integers.end(), outIt);

  integers.erase (integers.begin(), integers.end()); // all
  cout << "\nVector content after erasing remaining elem. "
    << (integers.empty() ? "is" : "is not") << " empty";

  // insert elements from array
  integers.insert (integers.begin(), array, array + V_SIZE);
  cout << "\nVector content before clear(): ";
  copy (integers.begin(), integers.end(), outIt);

  integers.clear(); // it calls erase to empty a collection
  cout << "\nVector content after clear() "
    << (integers.empty() ? "is" : "is not") << " empty" << endl;
} // end main
```

```
#include <iostream>
#include <list>
#include <algorithm>
#include <iterator>
using namespace std;

// prototype for function template printL
template <typename T> void printL (const list< T > &lRefs);

int main() {
  const int V_SIZE = 4;
  int array [V_SIZE] = {1, 2, 3, 4};
  list <int> integers;
  list <int> integers2;

  integers.push_front (2);
  integers.push_front (4);
  integers.push_back (8); // remember: reverse mode!
  integers.push_back (6);
```

```
cout << "integers content: ";
printL (integers);

integers.sort(); // sorting list in ascendent order
cout << "\nintegers content after sorting: ";
printL (integers);

// insert elements of array into integers2
integers2.insert (integers2.begin(), array, array + V_SIZE);
cout << "\nintegers2 content after insert: ";
printL (integers2);

// remove integers2 elements and insert at end of integers
integers.splice (integers.end(), integers2);
cout << "\nintegers content after splice: ";
printL (integers);
cout << "And integers2 content after splice: ";
printL (integers2);
```

```
integers.sort();
cout << "\nintegers content after sorting: ";
printL (integers);

integers2.insert (integers2.begin(), array, array + V_SIZE);
integers2.sort();
cout << "\nintegers2 content after inserting and sorting: ";
printL (integers2);

integers.merge (integers2); // =splice, but insert in order
cout << "\nintegers content after merging integers2: ";
printL (integers);
cout << "And integers2 content: ";
printL (integers2);

integers.pop_front(); // remove front element
integers.pop_back(); // remove back element
cout << "\nintegers content after poping front&back elem.: ";
printL (integers);
```

```
integers.unique(); // remove duplicate elements
cout << "\nintegers content after unique operation: ";
printL (integers);

integers.swap (integers2); // swap elements
cout << "\nintegers content after swapping : ";
printL (integers);
cout << "And integers2 content: ";
printL (integers2);

// integers content is replaced with elements of integers2
integers.assign (integers2.begin(), integers2.end());
cout << "\nintegers content after assign operation: ";
printL (integers);

integers.merge(integers2);
cout << "\nintegers content after merging integers2: ";
printL (integers);
```

```
  integers.remove (2); // remove all 2s
  cout << "\nintegers content after removing all 2s: ";
  printL (integers);
  cout << endl;
} // end main

template <typename T> void printL (const list <T> &lRefs) {
  if (lRefs.empty())
    cout << "List empty!";
  else {
    ostream_iterator <T> output (cout, " ");
    copy (lRefs.begin(), lRefs.end(), output);
  }
}
```

```
#include <iostream>
#include <deque>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
  deque <double> doubles; // deque of doubles
  ostream_iterator <double> outIt (cout, " ");

  doubles.push_front (3.1);
  doubles.push_front (4.3);
  doubles.push_back (1.5);
```

```
  cout << "doubles content: "; // output: 4.3 3.1 1.5
  for (unsigned int i = 0; i < doubles.size(); i++ )
    cout << doubles [i] << ' '; // subscript op. to get elem.

  doubles.pop_front(); // rm 1st. elem., output: 3.1 1.5
  cout << "\ndoubles content after pop_front: ";
  copy (doubles.begin(), doubles.end(), outIt);

  doubles [1] = 6.7; // subscript op. to set value at pos. 1
  cout << "\ndoubles content after doubles [1] = 6.7: ";
  copy (doubles.begin(), doubles.end(), outIt); // out: 3.1 6.7
  cout << endl;
} // end main
```

```
#include<iostream>
#include<set>
#include<string>
class MyObject {
public:
  MyObject(std::string name_, std::string surname_) :
      name (name_), surname (surname_) {}
  bool operator< (const MyObject & object) const {
      std::string right = object.name + object.surname;
      std::string left = this->name + this->surname;
      return (left < right); }
  friend std::ostream& operator<<(std::ostream& os, const
      MyObject& obj);
private:
  std::string name;
  std::string surname;
}; // end class MyObject
std::ostream& operator<< (std::ostream& os, const MyObject& o) {
    os << "Name: " << o.name << " & " << "Surname " << o.surname
       << std::endl; return os; }
```

```
int main() {
    std::set<MyObject> mySetOfObjects;
    MyObject obj1("name1", "surname1");
    MyObject obj2("name2", "surname2");
    MyObject obj3("name3", "surname3");
    MyObject obj4("name1", "surname1"); // duplicate object
    mySetOfObjects.insert(obj1);
    mySetOfObjects.insert(obj2);
    mySetOfObjects.insert(obj3);
    mySetOfObjects.insert(obj4); // it will not be inserted
        because its duplicate

    // iteration through all the elements in the set
    for (std::set<MyObject>::iterator it=mySetOfObjects.begin();
        it!=mySetOfObjects.end(); ++it)
      std::cout << *it;

    return 0;
}
```

```
#include <iostream>
#include <stack>
#include <vector>
#include <list>
using namespace std;
template <typename T> void pushElems (T &stackRef);
template <typename T> void popElems (T &stackRef);

int main() {
  stack <int> dequeStack; // stack with a deque container
  stack <int, vector <int>> vectorStack; // with a vector cont.
  stack< int, list< int > > listStack; // with a list cont.
  cout << "Pushing elements on the dequeStack: ";
  pushElems (dequeStack);
  cout << "\nPushing elements on the vectorStack: ";
  pushElems (vectorStack);
  cout << "\nPushing elements on the listStack: ";
  pushElems (listStack);
  cout << "\n\n";
```

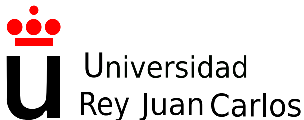
```
  cout << "Popping elements from the dequeStack: ";
  popElems (dequeStack);
  cout << "\nPopping elements from the vectorStack: ";
  popElems (vectorStack);
  cout << "\nPopping elements from the listStack: ";
  popElems (listStack);
  cout << endl;
} // end main
template <typename T> void pushElems (T &stackRef) {
  for (int i = 1; i <= 10; i++) {
    stackRef.push (i);
    cout << stackRef.top() << ' '; // show the top element
  }
}
template <typename T> void popElems (T &stackRef) {
  while (!stackRef.empty()) {
    cout << stackRef.top() << ' '; // show the top element
    stackRef.pop(); // it removes the top element
  }
}
```

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

int main() {
  queue <double> doubles;

  doubles.push (3.1);
  doubles.push (4.3);
  doubles.push (1.5);

  cout << "Popping from doubles: ";
  while (!doubles.empty()) {
    cout << doubles.front() << ' '; // show front element
    doubles.pop(); // it removes the front element
  }
  cout << endl;
}
```

7. Plantillas. Librería STL

Julio Vega

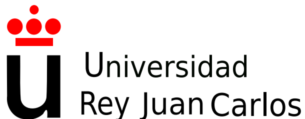
julio.vega@urjc.es

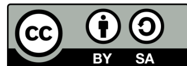


8. Manejo de excepciones

Julio Vega

julio.vega@urjc.es





©2024 **Julio Vega Pérez**
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. Introducción
2. Manejo básico de una excepción
3. Ejemplo básico de manejo de excepción: la división por cero
4. Ejemplo de relanzamiento de una excepción
5. Ejemplo de excepción `bad_alloc` al usar `new`

- Una excepción es la indicación de un problema dado en ejecución.
- El nombre ya implica que el problema ocurre con poca frecuencia.
- El manejo de excepciones permite crear rutinas para resolverlas.
  - En muchos casos permite continuar la ejecución del programa.
- El manejo de excepciones es ideal para procesar errores síncronos.
  - Aquellos que ocurren cuando se ejecuta una determinada instrucción.
  - E.g. arrays fuera de rango, desbordamiento aritmético, división por 0.
- El manejo de excepciones no es para trabajar con eventos asíncronos.
  - Aquellos que ocurren en paralelo a la ejecución del programa.
  - E.g. completar I/O disco, recepción mensajes red, I/O teclado/ratón.

- La clase `exception` es la clase base estándar de C++.
  - Recoge todo tipo de exceptiones y está definida en `<exception>`.
  - Ofrece la función virtual `what`, que devuelve mensaje de error.
- Una clase derivada de `exception` es `runtime_error`.
  - Clase base estándar de C++ para errores en tiempo de ejecución.
  - Esta clase está definida en el *header* `<stdexcept>`.
- Para definir excepción propia, lo más cómodo es mediante herencia.
  - De este modo, tenemos acceso a la función `what`.
  - E.g. heredando de la clase `runtime_error`.
  - Lo más sencillo es definir un constructor con el mensaje de error.

- La excepción se puede lanzar manualmente mediante `throw`.
  - E.g. cuando al intentar hacer división, el denominador $= 0$.
    - `if (den == 0) throw DivisionException();`
- La instrucción `throw` (o la función que la contiene) deberá.
  - Estar contenida en un bloque de *posible excepción*: bloque `try`.
  - Este va seguido de otro bloque, el `catch`, para su tratamiento.
- Un programa puede seguir su ejecución tras tratar la excepción.
- Al terminar manejador `catch` el programa sigue tras último `catch`.
  - No se vuelve al punto en que ocurrió la excepción (punto lanzamiento).
  - También se seguiría en tal punto si no hubiese excepción en el `try`.

```
try {
  result = getDivision (n1, n2);
  cout << "The getDivision result is: " << result << endl;
} catch (DivisionException &except) {
  cout << "Exception: " << except.what() << endl;
}
```

- El bloque `try` incluye instrucciones que podrían causar excepción.
  - Además de las instrucciones que se omitirían si hubiera tal excepción.
- El bloque `catch` es el encargado de atrapar y manejar la excepción.
  - Manejará la excepción del tipo según parámetro dado entre paréntesis.
    - O si la excepción es de un tipo derivado del especificado por parámetro.
  - Es más eficiente atrapar un objeto excepción por referencia.
    - Se evita la sobrecarga de copiar el objeto de la excepción lanzada.
  - Puede haber varios `catch`; entonces solo se ejecuta el del tipo dado.
    - Si no hay un `catch` que coincida, la ejecución se termina inmediata/.
  - Es un error sintáctico poner más de un parámetro a un bloque `catch`.
- Sería un error sintáctico implementar código entre ambos bloques.

```
#include <stdexcept> // contains runtime_error

class DivisionException : public std::runtime_error {
public:
  DivisionException()
    : std::runtime_error ("trying to divide by zero!") {}
};
```

```
#include <iostream>
#include "DivisionException.h"
using namespace std;

// get division but throw divisionException if divide by zero
double getDivision (int num, int den) {
  if (den == 0) throw DivisionException(); // function is ended

  // otherwise:
  return static_cast<double> (num) / den;
} // end function getDivision
```

```
int main() {
  int n1, n2;
  double result;

  cout << "Please, type two integers:\n";

  while (cin >> n1 >> n2) {
    try {
      result = getDivision (n1, n2);
      cout << "The getDivision result is: " << result << endl;
    } catch (DivisionException &except) {
      cout << "Exception: " << except.what() << endl;
    }

    cout << "Please, type two integers:\n";
  }
  cout << endl;
}
```

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

void rethrowException() {
  try {
    cout << "rethrowException(): Throwing an exception\n";
    throw exception();
  } catch (exception &except) {
    cout << "rethrowException(): Exception caught & rethrown\n";
    throw; // exception is raised!
  }
  cout << "rethrowException(): After caught & rethrown\n";
}
```

```
int main() {
  try {
    cout << "main(): Calling rethrowException()\n";
    rethrowException();
    cout << "main(): After calling rethrowException()\n";
  } catch (exception &except) {
    cout << "main(): Exception caught\n";
  }
  cout << "main(): After caught exception()\n";
}
```

- Estándar de C++: si `new` falla $\rightarrow$ lanzar excep. `bad_alloc`.
  - Esta excepción está definida en el *header* `<new>`.
- Algunos compiladores no están conformes con esta especificación.
  - E.g. *Microsoft Visual C++ 6.0* devuelve 0 al fallar el `new`.
  - Otros, lanzan `bad_alloc` con o sin incluir la cabecera `<new>`.

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

int main() {
  double *p[100];

  try {
    // size_t is widely used for counts
    // it represents size of any objet in bytes
    for (size_t i = 0; i < 100; ++i) { // each p->(huge memory)
      p[i] = new double[999999999]; // may cause exception!
      cout << "p[" << i << "] points to 999.999.999 doubles\n";
    }
  } catch (bad_alloc &except) {
    cerr << "Exception!: " << except.what() << endl;
  }
}
```

8. Manejo de excepciones

Julio Vega

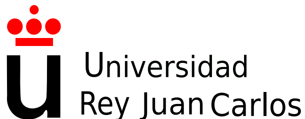
julio.vega@urjc.es

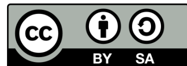
9. Depuración y documentación

Julio Vega

julio.vega@urjc.es

GSyC





©2024 Julio Vega Pérez
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. El depurador GDB
2. GDB. Ejemplo 1: división con tipo `double`
3. GDB. Ejemplo 2: división con tipo `int`
4. El depurador Valgrind
5. Documentación con Doxygen

- Exisen dos tipos de errores: de compilación y lógicos (*bugs*).
- El compilador nos ayuda con los primeros, pero no con los segundos.
- Para ayudarnos con los errores lógicos está el depurador o *debugger*.
- Para sistemas GNU tenemos el GNU Debugger (GDB), *open-source*.
  - Si no lo tenemos instalado, instalar con: `sudo apt install gdb`.
- Existen IDEs con sus propios depuradores, y también con GDB.
  - E.g. los entornos de desarrollo *Visual Studio* o *Eclipse*.

- Lo primero es compilar el programa para que pueda ser depurado.
  - Para ello está la opción `-g`: `g++ -g -o myProgram myProgram.cpp`.
- Lo siguiente es iniciar el *debugger*: `gdb myProgram`
  - El `$` del Terminal desaparece; en su lugar se indica `(gdb)`
- El comando `list` nos permite ver el código del programa.
- Antes de lanzar el programa, establecer **puntos de interrupción**.
  - Son como marcadores que se pueden ponen a lo largo del código.
  - Cuando la ejecución del programa llega a estos, se detiene.
  - Para ponerlos se usa `break numlinea` (e.g. `(gdb)break 27`).
  - Para ver la lista de *breakpoints*: `info break`
  - Si queremos elimiar alguno: `delete numbreakpoint`

- Y ya se puede dar comienzo a la depuración, con el comando `run`
- El comando `continue` permite reanudar la ejecución tras *breakpoint*.
  - `next` avanza solo una línea del programa.
  - `step` avanza solo una línea y, si esta es una función, entra dentro.
- `print` nos permite ver el contenido de una variable.
- Si hay algún *bug*, el comando `where` nos hace un *backtrace*.
- Finalmente, `quit` termina la sesión de depuración.
- También se pueden usar las iniciales de los anteriores comandos.
  - E.g. r (run), s (step), n (next), p (print), b (break), q (quit).

```
01. #include <iostream>
02. using namespace std;
03.
04. double getDivision (int num, int den) {
05.   return static_cast<double> (num) / den;
06. }
07.
08. int main() {
09.   int n1 = 7, n2 = 3;
10.   cout << getDivision (n1, n2);
11.   n1 = 3; n2 = 0;
12.   cout << getDivision (n1, n2);
13.   return 0;
14. }
```

```
$ g++ -g -o div div.cpp
$ gdb div
```

```
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from div...
(gdb)
```

```
(gdb) break 9
Punto de interrupcion 1 at 0x11c5: file div.cpp, line 9.
(gdb) break 10
Punto de interrupcion 2 at 0x11d3: file div.cpp, line 10.
(gdb) break 12
Punto de interrupcion 3 at 0x1209: file div.cpp, line 12.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000000011c5 in main()
                                                   at div.cpp:9
2       breakpoint     keep y   0x00000000000011d3 in main()
                                                   at div.cpp:10
3       breakpoint     keep y   0x0000000000001209 in main()
(gdb)
```

```
(gdb) r
Starting program: div
[Depuracion de hilo usando libthread_db enabled]
Using host libthread_db library
    "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at div.cpp:9
9    int n1 = 7, n2 = 3;
(gdb) print n1
$1 = 0
(gdb) s

Breakpoint 2, main () at div.cpp:10
10   cout << getDivision (n1, n2);
(gdb) print n1
$2 = 7
(gdb) print n2
$3 = 3
(gdb)
```

```
(gdb) s
getDivision (num=7, den=3) at div.cpp:5
5    return static_cast<double> (num) / den;
(gdb) s
6  }
(gdb) s
main () at div.cpp:11
11   n1 = 3; n2 = 0;
(gdb) s

Breakpoint 3, main () at div.cpp:12
12   cout << getDivision (n1, n2);
(gdb) s
getDivision (num=3, den=0) at div.cpp:5
5    return static_cast<double> (num) / den;
(gdb) s
6  }
(gdb)
```

- El anterior programa no generaba fallo, por el casting a `double`.
  - La ejecución de la división por 0 mostrará INF o inf (infinito).
- Ahora cambiaremos la línea 5, para que se haga una división entera.
  - Y el tipo del valor devuelto, será `int` en lugar de `double`.
- Al ejecutar el programa como habitualmente (`./div`) se muestra:
  - `Excepción de coma flotante ('core' generado)`

```
01. #include <iostream>
02. using namespace std;
03.
04. int getDivision (int num, int den) {
05.   return num / den;
06. }
07.
08. int main() {
09.   int n1 = 7, n2 = 3;
10.   cout << getDivision (n1, n2);
11.   n1 = 3; n2 = 0;
12.   cout << getDivision (n1, n2);
13.   return 0;
14. }
```

```
(gdb) r
Starting program: div
[Depuracion de hilo usando libthread_db enabled]
Using host libthread_db library
    "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGFPE, Arithmetic exception.
0x000055555555519b in getDivision (num=3, den=0) at div.cpp:5
5    return num / den;
(gdb) where
#0  0x000055555555519b in getDivision (num=3, den=0) at div.cpp:5
#1  0x00005555555551f7 in main () at div.cpp:13
(gdb)
```

- Es una *suite* con diferentes herramientas de depuración.
  - Aunque la más empleada es *memcheck*.
  - Y su principal funcionalidad es *leak-check*.
- Uso ppal.: controlar la gestión que hacemos de memoria dinámica.
  - Fugas de memoria: se reserva memoria que ya no es accesible.
  - Tipos: seguras o probables (estás *jugando* con los punteros).
- Instalación: `sudo apt install valgrind`.
- Añadir opción `-g` al compilador para proporcionar info de *seg. faults*.
- Ejecución: `valgrind ./miEjecutable`
  - `valgrind --leak-check=full --show-leak-kinds=all ./miEjecutable`

- *leak-check* para control de fugas.
  - *no*: comprobación desactivada.
  - *summary* (opción por defecto): muestra n.º de fugas sin detalle.
  - *yes* o *full*: muestra n.º de fugas con detalle.
- *show-leak-kinds*: especificar qué tipos de fugas mostrar.
  - *definite*: olvidar borrar memoria reservada que ya no puedes borrar.
  - *indirect*: memoria que era accesible desde memoria ya inaccesible.
  - *reachable*: existe puntero a memoria no liberada cuando fin programa.
  - *possible*: no es seguro que exista puntero a memoria no liberada.
- *show-error-list=yes (-s)*: mostrar mensajes de error detectados.
- *-q*: modo silencioso (*quiet*) → solo mostrar mensajes de error.

- *getRandomInt* asigna valor a puntero y devuelve solo el valor.

```
int getRandomInt(int min, int max) {
    std::random_device rand_dev;
    std::default_random_engine rand_engine(rand_dev());
    std::uniform_int_distribution<int> uniform_dist(min, max);
    int *retval = new int(uniform_dist(rand_engine));
    return *retval;
}

int main() {
    std::cout << "The random number is " << getRandomInt(1, 5)
        << std::endl;
    return 0;
}
```

- Al perder acceso al puntero del atrib. hijo, perdemos acceso al padre.

```
struct node {
    int number;
    struct node *parent;
};

int main() {
    node *childItem = new node {.number = 1, .parent = new node
        {.number = 0, .parent = nullptr}};
    childItem = nullptr;
    return 0;
}
```

- Al incrementar puntero Valgrind no sabe si hay acceso al primero.

```
int *numbers;

void generateNumbers(int **numbers) {
    *numbers = malloc (sizeof(int) * 2);
}

int main() {
    generateNumbers (&numbers);
    numbers++;
    return 0;
}
```

- Sin incrementar puntero Valgrind mantiene puntero a memoria no liberada.

```c
int *numbers;

void generateNumbers(int **numbers) {
    *numbers = malloc (sizeof(int) * 2);
}

int main() {
    generateNumbers (&numbers);
    // numbers++;
    return 0;
}
```

- Herramienta estandar *de facto* para generar documentación de C++.
  - También soporta otros muchos lenguajes: C, C#, PHP, Java, Python...
  - Aunque Java o Python usan normal/ sus *tools*: Javadoc y Pydoc.
- Con código debida/ anotado, Doxygen genera documentación formal.
  - HTML, para manual/doc. on-line; LaTeX, como manual de referencia.
  - Otros: RTF, PostScript, *hyperlinked* PDF, páginas `man` de UNIX.
- Permite extraer la estructura del código desde los ficheros fuente.
  - Y visualizar relaciones con diferentes diagramas (colab., herencia, etc.).

- SFML: Simple and Fast Multimedia Library. Desarrollo de juegos.
  - Doxygen doc: *https://github.com/SFML/SFML/tree/master/doc*
  - Relacionada: OGRE3D, Object-Oriented Graphics Rendering Engine.
- OpenCV: Open Source Computer Vision. Algoritmos de visión.
  - Doxygen doc: *https://github.com/opencv/opencv/tree/master/doc*
- Magnum Graphics: middleware gráfico para juegos y visualizar datos.
  - Doxygen doc: *https://github.com/mosra/magnum/tree/master/doc*
- OpenFoam: dinámica fluidos, reacciones químicas, turbulencias, etc.
  - Doc: *https://github.com/OpenFOAM/OpenFOAM-6/tree/master/doc*
- CERN's Root Framework: desarrollo del entorno raíz del CERN. *https://github.com/root-project/root/tree/master/documentation/doxy*
  - Relacionada: GslWrapper, clase wrapper para la GNU Scientific Library.

- Instalación de todos los *features* de Doxygen:
  - `sudo apt-get install doxygen doxygen2man doxygen-doc doxygen-doxyparse doxygen-gui doxygen-latex`
- Para poder visualizar los árboles de relaciones entre clases:
  - `sudo apt-get install graphviz`
- Comprueba tu instalación/versión: `doxygen --version`

- El comportamiento de Doxygen se rige por un fichero: `Doxyfile`.
- Para generar un fichero `Doxyfile`: `doxygen -g <fichero>`
  - Normalmente se ejecuta sin el último parámetro.
- Este fichero se puede modificar.
- Cada proyecto tiene su fichero `Doxyfile`.
- Para usar `Doxygen`: ir al directorio del proyecto y lanzarlo (`doxygen`).
  - Este busca los códigos fuente, los *parsea* y genera la documentación.
  - La documentación estará en la carpeta especificada.
    - E.g. línea 61: `OUTPUT_DIRECTORY = "doxygen-doc"`
  - Doc. HTML (`doxygen-doc/html`): `$firefox index.html`
  - Doc. LaTeX(`doxygen-doc/latex`): `$make && evince refman.pdf`

```
#Linea 35:
PROJECT_NAME           = "My first Doxygen test"
#Linea 47:
PROJECT_BRIEF          = "This is a simple project to use Doxygen"
#Linea 54:
PROJECT_LOGO           = "./MyIcon.ico"
#Linea 61:
OUTPUT_DIRECTORY       = "doxygen-doc"
#Linea 363:
DISTRIBUTE_GROUP_DOC = YES
#Linea 438:
EXTRACT_ALL            = YES
#Linea 444:
EXTRACT_PRIVATE        = YES
```

```
#Linea 450:
EXTRACT_PACKAGE       = YES
#Linea 456:
EXTRACT_STATIC        = YES
#Linea 481:
EXTRACT_ANON_NSPACES = YES
#Linea 759:
WARN_NO_PARAMDOC      = YES
#Linea 867:
RECURSIVE             = YES
#Linea 876:
EXCLUDE               = README.md
#Linea 998:
SOURCE_BROWSER        = YES
```

```
#Linea 1004:
INLINE_SOURCES       = YES
#Linea 1017:
REFERENCED_BY_RELATION = YES
#Linea 1023:
REFERENCES_RELATION  = YES
#Linea 1031:
REFERENCES_LINK_SOURCE = NO
#Linea 2283:
UML_LOOK             = YES
#Linea 2334:
CALL_GRAPH           = YES
#Linea 2346:
CALLER_GRAPH         = YES
```

Los comentarios de Doxygen comienzan por `/**` y terminan por `*/`:

```
/**
 *  This is my first example using Doxygen
 *
 *  The asterisks to the left could be omitted, but they help to
 *  get a nice comment.
 */
```

Los comentarios de cabecera suelen contener *keywords*:

```
/**
 *  @file  MyClass.h
 *  @brief This is my first example using Doxygen
 *  @author Julio Vega
 *  @date  2022-11-07
 ***********************************************/

class MyClass {
  // [...]
};
```

Lo ideal es poner los comentarios mayor/ en los archivos cabecera (.h).

```
/**
 * \brief Adds two numbers.
 *
 * This function takes two numbers, adds them, and then returns
     the result.
 *
 * \param x The first number to add.
 * \param y The second number to add.
 * \return The sum of the two numbers.
 */
int add(int x, int y) {
    return x + y;
}
```

9. Depuración y documentación

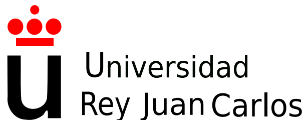
Julio Vega

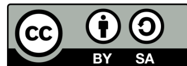
julio.vega@urjc.es

10. Manejo de ficheros

Julio Vega

julio.vega@urjc.es





©2024 **Julio Vega Pérez**
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. Introducción
2. Archivos y flujos
3. Escritura en un archivo secuencial
4. Lectura de un archivo secuencial
5. Escritura en un archivo de acceso aleatorio
6. Lectura de un archivo de acceso aleatorio

- Variables y colecciones almacenan datos temporalmente en RAM.
- Los archivos almacenan datos permanentemente en disco.
- Vamos a tratar con archivos secuenciales y de acceso aleatorio.
- Los datos en un fichero pueden ser crudos o con formato.

- La ud. más pequeña de dato que maneja un ordenador es el **bit**.
  - Y cada bit puede contener el valor 0 o el valor 1.
- Un **byte** a su vez está compuesto por ocho bits.
- Pero programar a bajo nivel, con bits, es muy difícil.
  - Más fácil, con formatos: dígitos, letras y símbolos especiales.
  - Este es el conjunto de caracteres que maneja un ordenador.
- Un programador usa caracteres; C++ proporciona el tipo `char`.
  - Un `char` ocupa un byte; para Unicode, en C++ se usa `wchar_t`.

- C++ considera cada archivo como una secuencia de bytes.
- Cada archivo termina con un caracter especial de terminación.
  - O con un patrón específico de bytes reconocido por el sistema.
- Al abrir un archivo se crea un objeto al cual se asocia un flujo.
  - Algunos objetos especiales son `cin`, `cout`, `cerr` y `clog`.
    - Estos se crean al usar la librería `<iostream>`.
    - `cin`: objeto flujo de entrada estándar (teclado).
    - `cout`: objeto flujo de salida estándar (pantalla).
    - `cerr` y `clog`: objetos flujo de error estándar.

- Para tratar ficheros en C++ necesitamos `<fstream>` e `<iostream>`.
- `<fstream>`: `basic_ifstream`, `basic_ofstream` y `basic_fstream`.
  - Para la entrada, salida y e/s de ficheros respectivamente.
- Además, `<fstream>` ofrece alias `typedef` para estas funciones.
  - E.g. `typedef ifstream` es una especialización de `basic_ifstream`.
    - `ifstream` permite la entrada de valores `char` desde fichero.
  - Y, de igual forma, `typedef ofstream` y `typedef fstream`.

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

int main() {
   ofstream outUsersFile ("users.dat", ios::out); // open file

   if (!outUsersFile) { // file couldn't be opened
      cerr << "File could not be opened" << endl;
      exit (1);
   }
```

```
  cout << "Enter data according to 'name surname phone' (press
      CTRL+C to finish):\n";

  string name;
  string surname;
  int phone;
  while (cin >> name >> surname >> phone) {
    outUsersFile << name << ' ' << surname << ' ' << phone <<
        endl;
  }
}
```

- Los objetos `ofstream` se abren por defecto en modo salida.
  - La línea `ofstream outUsersFile (''users.dat'', ios::out);`
    - Es equivalente a `ofstream outUsersFile (''users.dat'');`
  - En este modo, el fichero se trunca: se sobreescriben los datos.
  - También se puede crear el objeto `ofstream` y después abrirlo.
    - `ofstream outUsersFile;`
    - `outUsersFile.open(''users.dat'');`
- Un objeto `ofstream` también puede ser abierto en modo `app`.
  - Con `ios::app` se añaden los datos al final del fichero.

- Al modificar datos en un archivo secuencial, se puede corromper.
  - E.g. si tenemos fichero con contenido *[...] Julio Vega 677387389 [...]*.
  - Si queremos modificar *Vega* por *Vega Pérez*:
    - El nuevo registro quedaría así: *Julio Vega Pérez 677387389*.
    - Pero este nuevo registro ocupa seis caracteres más que el original.
    - Serían seis caracteres que se sobrescribirían del siguiente registro.
- Para actualizar un registro en un archivo secuencial correcta/.
  - Copiar a otro fich. todos los registros antes del que hay que modificar.
  - Añadir el registro nuevo a ese nuevo fichero.
  - Y final/ copiar todos los registros que hay después del reg. modificado.

- `ios::app` añade toda la salida al final del archivo.
- `ios::ate` puede añadir los datos en cualquier parte.
- `ios::in` abre un archivo en modo de entrada.
- `ios::out` abre un archivo en modo de salida.
- `ios::trunc` ignora el contenido del archivo (como `ios::out`).
- `ios::binary` abre un archivo para entrada/salida binaria.

- La clase `ios` define una variable enum `io_state`.
  - Cada flujo tiene este flag para comprobar los posibles errores.
  - Posibles valores: `goodbit`, `eofbit`, `badbit` o `failbit`.
  - Para comprobar: `int good()`, `int eof()`, `int bad()` o `int fail()`.
  - Con `int clear()` se borran los bits de error que se hayan activado.
- El operador `!` está sobrecargado en la librería `ios`.
  - E.g. nos informa sobre si la operación `open` tuvo éxito o no.
  - `(!outUsersFile)=true` $\iff$ (`failbit=1` o `badbit=1`) para `open`.
    - $\nexists$ fichero-r, no tenemos permisos (r o w), no queda espacio (w).
- Otra función sobrecargada de `ios` es `void*` de `cin`.
  - Permite convertir el flujo de entrada (`cin`) en un puntero.
  - $puntero = null \implies$ `cin=0` $\iff$ (`failbit=1` o `badbit=1`) para `cin`.
    - Es decir, `while cin` es `true` mientras no se activen bits de fallo.
  - Al introducir fin de archivo $\implies$ `failbit=1` $\implies$ fin while de `cin`.
    - Fin de archivo: sistemas UNIX `<Ctrl+d>`, Windows `<Ctrl+z>`.
  - `<Ctrl+d>` $\implies$ `main` termina $\implies$ destructor objeto `ofstream`.
    - También se puede cerrar explícita/ con `outUsersFile.close();`

```
int main() {
   ifstream inUsersFile ("users.dat", ios::in);

   if (!inUsersFile) {
      cerr << "File could not be opened" << endl;
      exit (1);
   }

   string name;
   string surname;
   int phone;
   cout << "Name\tSurname\tPhone\n";

   while (inUsersFile >> name >> surname >> phone)
      cout << name << "\t" << surname << "\t" << phone << endl;
}
```

- Para leer datos secuencial/ de un fichero, se leen de ppio. a fin.
- A veces se necesita leer todos los datos secuencial/ varias veces.
- `istream` y `ostream` ofrecen funciones para reposicionar puntero.
  - Un objeto `istream` tiene puntero *get* y, un `ostream`, *put*.
    - Para saber sus posiciones, están las funciones `tellg()` y `tellp()`.
  - En `istream` está la función `seekg` (*seek get*, buscar obtener).
    - E.g. instrucción `inUsersFile.seekg(0);` ubica puntero en pos. 0.
    - El arg. de `seekg` es un entero `long`. Y un 2.º arg. puede ser:
    - `ios::beg` opción por defecto para posiciona/ desde inicio del flujo.
    - `ios::cur` para un posiciona/ relativo a la pos. actual del flujo.
    - `ios::end` para un posiciona/ desde final (hacia atrás) del flujo.
  - En `ostream` está la función `seekp` (*seek put*, buscar poner).
    - Y su uso es similar al descrito para `seekg`, con los mismos posibles args.

```
int option = getUserOption(); // e.g. 1=showNames,
    2=showSurnames, 0=end...

while (option != 0) { // option != end
  while (!inUsersFile.eof()) { // shows the selected field
    inUsersFile >> name >> surname >> phone;
    if (option == 1)
      cout << name << endl;
    if (option == 2)
      cout << surname << endl;
    if (option == 3)
      cout << phone << endl;
  }

  inUsersFile.clear (); // reset eof for next loop
  inUsersFile.seekg (0); // pointer reposition to the beginning
  option = getUserOption(); // get a new request
}
```

- Estos archivos se usan para acceder a registro de forma instantánea.
  - El acceso a registro en un fich. secuencial tiene complejidad $O(n)$.
  - El acceso a registro en un fich. aleatorio tiene complejidad $O(1)$.
- C++ no ofrece una estructura para el manejo de ficheros.
  - Si se quieren usar ficheros con acceso aleatorio, hay que crearlos.
  - Las técnicas a emplear para crear un archivo aleat. son muy variadas.
  - E.g. usar registros de la misma long. fija. $\Longrightarrow$ fácil hallar un registro.

- Se puede escribir un `int` (cuatro bytes) con operador `<<`.
  - E.g. `outputFile << intNumber;`
  - Así se podría escribir desde un dígito hasta 11 (10 dígitos más signo).
    - Cada uno de estos requeriría un solo byte de almacenamiento.
- Otra opción podría ser usando función `write`.
  - E.g. `outputFile.write(reinterpret_cast<const char*> (&intNumber), sizeof (intNumber));`
  - Siempre escribiría la versión binaria de los cuatro bytes del tipo `int`.
  - `write` trata a su primer argumento como un grupo de bytes...
    - ...al ver el objeto en memoria como un puntero a byte (`const char*`).
  - Y desde esa posición, `write` envía el *n.º* de bytes según 2.º parám.
  - `write` necesita que el primer parámetro sea `const char*`
    - Y la expresión `&intNumber` devuelve puntero tipo `int*`.
    - `reinterpret_cast` es usado para convertir tipos de punteros.

```
#ifndef USER_H
#define USER_H
#include <string>
using namespace std;
class User {
public:
   User (string = "", string = "", int = 0);
   void setNumRecord (int);
   int getNumRecord () const;
   void setName (string);
   string getName () const;
   void setSurname (string);
   string getSurname () const;
   void setPhone (int);
   int getPhone () const;
private:
   int numRecord; char name [10]; char surname [15]; int phone;
};
#endif
```

```cpp
#include <string>
#include "User.h"
using namespace std;

User::User (string myName, string mySurname, int myPhone) {
  setName (myName);
  setSurname (mySurname);
  setPhone (myPhone);
}

int User::getNumRecord() const {
  return numRecord;
}
void User::setNumRecord (int record) {
  numRecord = record;
}
string User::getName() const {
  return name;
}
```

```
void User::setName (string myName) { // copy at most 10 char
  int length = myName.size();
  length = (length < 10 ? length : 9);
  myName.copy (name, length);
  name [length] = '\0'; // null character
}
string User::getSurname() const {
  return surname;
}
void User::setSurname (string mySurname) { // copy at most 15
    char
  int length = mySurname.size();
  length = (length < 15 ? length : 14);
  mySurname.copy (surname, length);
  surname [length] = '\0';
}
int User::getPhone() const { return phone; }
void User::setPhone (int myPhone) { phone = myPhone; }
```

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "User.h"
using namespace std;

int main() {
  ofstream outUsersFile ("users.dat", ios::out | ios::binary);

  if (!outUsersFile) { // ofstream could not open file
    cerr << "File could not be opened." << endl;
    exit (1);
  }

  User user; // fill with zeros each data member
  for (int i = 0; i < 10; i++) // write 10 empty records to file
    outUsersFile.write (reinterpret_cast <const char *> (&user),
       sizeof (User));
}
```

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "User.h"
using namespace std;

int main() {
  int numRecord;
  string name;
  string surname;
  int phone;
  fstream outUsersFile ("users.dat", ios::in | ios::out |
      ios::binary); // ios::in will require an existing file
  // Uses: "users.dat", generated in previous example
  if (!outUsersFile) { // fstream could not open file
    cerr << "File could not be opened." << endl;
    exit (1);
  }
  cout << "Enter record number (1 to 10, 0 to end)\n> ";
```

```
  User user;
  cin >> numRecord;
  while (numRecord > 0 && numRecord <= 10) {
    cout << "Enter name, surname and phone\n> ";
    cin >> name;
    cin >> surname;
    cin >> phone;
    user.setNumRecord (numRecord);
    user.setName (name);
    user.setSurname (surname);
    user.setPhone (phone);
    outUsersFile.seekp ((user.getNumRecord() - 1 ) *
       sizeof (User));
    outUsersFile.write (reinterpret_cast <const char *> (&user),
       sizeof (User));
    cout << "Enter record number (1 to 10, 0 to end)\n> ";
    cin >> numRecord;
  }
}
```

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
#include "User.h"
using namespace std;
void displayRecord (ostream &, const User &);

int main() {
  ifstream inUsersFile ("users.dat", ios::in | ios::binary);
  // Uses: "users.dat", generated in previous example

  if (!inUsersFile) { // fstream could not open file
    cerr << "File could not be opened." << endl;
    exit (1);
  }
  cout << left << setw (10) << "Record" << setw (11)
    << "Name" << setw (16) << "Surname" << left
    << setw (10) << right << "Phone" << endl;
```

```
  User user;
  inUsersFile.read (reinterpret_cast <char *>(&user),
    sizeof (User));

  while (inUsersFile && !inUsersFile.eof()) {
    if (user.getNumRecord() != 0)
       displayRecord (cout, user);

    inUsersFile.read (reinterpret_cast <char *>(&user),
      sizeof (User));
  }
}

void displayRecord (ostream &output, const User &record) {
   output << left << setw (10) << record.getNumRecord ()
      << setw (11) << record.getName()
      << setw (16) << record.getSurname()
      << setw (10) << right << record.getPhone() << endl;
}
```

10. Manejo de ficheros

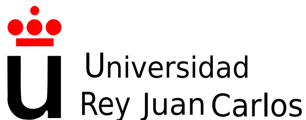
Julio Vega

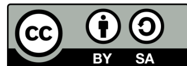
julio.vega@urjc.es

11. Patrones de diseño

Julio Vega

julio.vega@urjc.es





©2024 **Julio Vega Pérez**
Algunos derechos reservados.

Este trabajo se entrega bajo licencia CC-BY-SA 4.0.

Usted es libre de (a) compartir: copiar y redistribuir el material en cualquier medio o formato para cualquier propósito, incluso comercialmente; y (b) adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Contenidos

1. Introducción
2. Patrones elementales
3. Patrones de creación
4. Patrones estructurales

- El diseño software es un proceso iterativo de afinamiento de este.
- Cualquier software complejo requiere varias etapas.
  - Al principio se identifican los módulos más abstractos.
  - Progresivamente se va concretando el diseño de cada módulo.
- En el largo proceso del diseño software aparecen problemas.
  - Y estos problemas suelen repetirse con frecuencia en un DS.
  - Esto es, suelen darse diversos *patrones* de problemas/soluciones.
- **Patrones de diseño:** formas de resolver problemas comunes de DS.
- Igual que reutilizamos código, reutilizamos soluciones de DS.

- **Nombre:** corto y autodefinido, fácil de usar por diseñadores.
- **Problema:** qué resuelve el patrón y en qué contexto.
- **Solución:** normal/ describe las clases y relaciones entre objetos.
- **Ventajas/Desventajas:** complejidad, t.º ejec., portabilidad, etc.

- **De creación:** ofrecen solución sobre construcción clases, objetos, etc.
  - E.g. *Abstract Factory*, *Builder*, etc.
- **Estructurales:** jerarquía de clases, relaciones y composiciones.
  - E.g. *Adapter*, *Facade*, *Flyweight*, etc.
- **De comporta/:** cómo organizar ejecución, mensajes entre objetos.
  - E.g. *Visitor*, *Iterator*, *Observer*, etc.

- Son la base para la definición de patrones más complejos.
  - Como los citados previamente.
- Surgen de la relación *method-method*.
  - Esto es, cuando un método A llama desde su código a un método B.
  - Dicho de otro modo, cuando el método A depende de B.
    - Ya sea directa o indirectamente, como veremos a continuación.

```
class B {
  void methodB () {
    // ...
  }
}

class A {
  B b;
  void methodA () { // methodA depends on methodB
    b.methodB ();
    // ...
  }
}

main () {
  A a;
  a.methodA ();
}
```

```
class B {
  void methodB () { /* ... */ }
}
class C {
  B b;
  void methodC () { // methodC depends on methodB
    b.methodB ();
  }
}
class A {
  C c;
  void methodA () { // methodA depends on methodC
    c.methodC ();
  }
}
main () {
  A a;
  a.methodA ();
}
```

- Son originados por la relación *method-method*.
  - Y haciendo las combinaciones posibles entre los objetos y los métodos.
- **Recursion:** objetos y métodos implicados son los mismos.
- **Conglomeration:** si objetos son los mismos pero métodos diferentes.
- **Redirection:** objeto recibe invocación a método.
  - Y este método invoca al "mismo" método de otro objeto.
  - A un método cuya funcionalidad es igual aunque su implementación no.
- **Delegation:** cuando un método delega funcionalidad en otro.

```
class GUIButtonMaker {
  GUIButton makeGUIButton {
    // make the GUI button and return it
  }
}

class GUIMaker {
  GUIButtonMaker g;
  GUI makeGUI () {
    GUIButton button1 = g.makeGUIButton ();
    // GUIMaker object delegates making button to g instance
    // continues doing more stuff...
  }
}

main () {
  GUIMaker g;
  g.makeGUI ();
}
```

```
class Printer3D {
  void print (Model m) {
    //...
  }
}

class PrinterManager {
  Printer3D p;
  void print (Model m) {
    p.print (m); // PM object redirects its job to p instance
    // continues doing more stuff...
  }
}

main() {
  Model m;
  PrinterManager pm;
  pm.print (m);
}
```

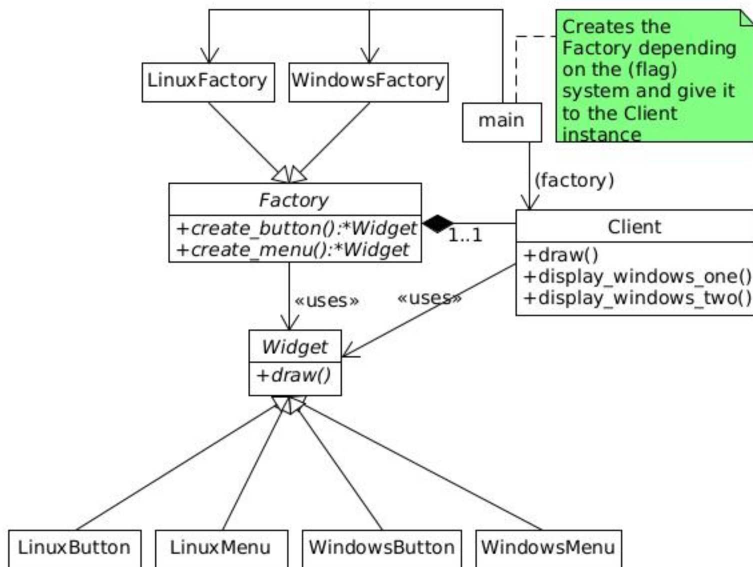
- Patrón usado cuando se requiere tener una única instancia de clase.
- Para crear instancias en C++ se puede usar el operador `new`.
  - Pero quizás necesario que no se permita crear más de una instancia.
  - E.g. el objeto `Balon` en el diseño de un juego de fútbol.
- Para asegurar una instancia → impedir clientes acceder a constructor.
  - ¿Cómo? Haciendo al constructor `private` o `protected`.
    - Si un cliente intenta crear instancia directa/ → error de compilación.
  - Y proporcionando un punto controlado por el que pedir la instancia.

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

class Ball {
public:
  static Ball* getTheBall();
  void move (int x, int y);
  void showPos ();
  // singletons shoul not be cloneable or assignable:
  Ball (Ball &otherBall) = delete;
  void operator= (const Ball&) = delete;

protected: // this is the KEY: constructor is protected!
  Ball():posx(0),posy(0){}; // it will be called by getTheBall()

private:
  int posx, posy;
  static Ball* singleBall; // pointer to the single instance
};
```

```
#include "Ball.h"

Ball* Ball::singleBall = nullptr;

Ball* Ball::getTheBall () {
  if (singleBall == nullptr)
    singleBall = new Ball ();
  else
    cout << "Error: trying to get another instance of a Ball
        singleton class!\n";

  return singleBall;
}

// [...] move and showPos functions are omitted but they are
    defined as always
```

```
#include "Ball.h"

int main () {
  Ball* ball = Ball::getTheBall ();
  int posx = 0, posy = 0;

  cout << "You got the ball! Choose its position (x, y):\n";
  cout << "> ";
  cin >> posx;
  cout << "> ";
  cin >> posy;

  ball->move (posx, posy);
  ball->showPos();

  // Ball* anotherBall1 = Ball::getTheBall(); // exec time error!
  // Ball* anotherBall2 = new Ball (); // comp. time error!
}
```

- Permite crear diferentes tipos de instancias aislando al cliente de ello.
- Al crecer pograma, el n.º de clases y sus jerarquías también lo hace.
- Este patrón permite crear diferentes objetos con diferentes jerarquías.
  - E.g. al construir los diferentes personajes de un juego de rol.
    - Cada personaje tendrá determinadas restricciones y relaciones.
  - E.g. al crear el sistema de ventanas según S.O. de un programa.
    - Dependiendo del S.O. las jerarquías subyacentes de creación difieren.
    - Crear el GUI es transparente para el cliente → lo hace la factoría.


LinuxFactory
WindowsFactory
main
Creates the Factory depending on the (flag) system and give it to the Client instance
(factory)
Factory
+create_button():*Widget
+create_menu():*Widget
1..1
Client
+draw()
+display_windows_one()
+display_windows_two()
«uses»
«uses»
Widget
+draw()
LinuxButton
LinuxMenu
WindowsButton
WindowsMenu

```
#include "Client.h"
#include "LinuxFactory.h"
#include "WindowsFactory.h"
#define LINUX // set the flag to Linux platform

int main() {
  Factory *factory;
#ifdef LINUX // switch statement to create a proper factory
  factory = new LinuxFactory;
#else // a Windows platform is being used
  factory = new WindowsFactory;
#endif

  Client *c = new Client(factory); // creates a proper GUI system
  c->draw(); // client doesn't know which draw function is called
}
```

- Se basa en definir una interfaz para crear instancias de objetos.
- Permite a las subclases decidir cómo se crean tales instancias.
    - De forma transparente para el cliente (como *Abstract Factory*).
- E.g. motorización de un coche, según selección del cliente.

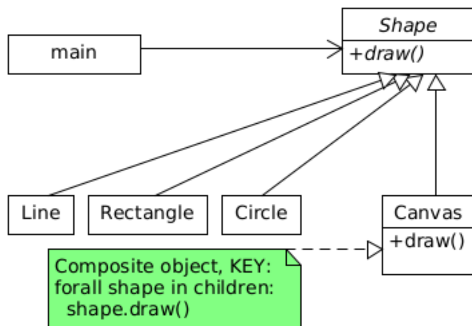
```
#include "Engine.h"

int main() {
  vector<Engine*> engines;
  int choice;
  while (true) {
    cout << "Choose engine: Gasoline=1 Diesel=2 Electric=3
        (Exit=0): ";
    cin >> choice;
    if (choice == 0) break;
    engines.push_back (Engine::makeEngine(choice));
    // factory method is called: depending on the engine, the
        object instance created will be different
  }
  for (int i = 0; i < engines.size(); i++)
    engines[i]->label();
  for (int i = 0; i < engines.size(); i++)
    delete engines[i];
}
```

- Proporciona abstracción cuando hay que crear diferentes objetos.
  - En un contexto en que se desconoce cuántos y cuáles van a ser.
- La idea ppal. es que los objetos deben poder clonarse en t.º ejec.
- Los dos patrones anteriores usan herencia y mét. abstractos.
  - Implementados por subclases que construyen y definen a cada objeto.
- Esto es un problema cuando el n.º de objetos es elevado o indeter/.
  - E.g. pensemos que un cliente puede configurar su motor como quiera.
    - Habría un gran n.º indeter/ de tipos de motores que se podrían hacer...
    - La solución ideal aquí será el uso del patrón *Prototype*.

- La clase interfaz `Engine` sería el `Prototype`.
- La ppal. diferencia: nuevo método `clone ()` en todos los objetos.
  - $\nexists$ *Factory Method*: `static Engine *makeEngine (int choice);`
  - Ahora tendremos: `virtual Engine* clone() const = 0;`
    - Y en cada subclase se implementa ese método devolviendo la instancia.
    - E.g. subclase Diesel: `Engine* clone() const{return new Diesel;}`
- No se necesita agente intermedio (factoría) para crear instancias.
  - La creación se realiza en la clase concreta que representa a la instancia.
  - Se puede tener factoría con los prototipos y esta invoque a `clone()`.
  - O se puede tener un gestor de prototipos. Hecho en nuestro ejemplo.
    - Que permita cargar/descargar los prototipos disponibles en t.º ejec.
    - Solución interesante para diseños ampliables en t.º ejec. (*plugins*).

```cpp
#include "EngineManager.h"

Engine* EngineManager::engineTypes[] = {
  0, new Diesel, new Gasoline, new Electric
};

Engine* EngineManager::makeEngine (int choice) {
  return engineTypes[choice]->clone();
}
```

- Permite diseñar objetos como estructuras recursivas tipo árbol.
- Se puede trabajar con esta estructura como si fuera un único objeto.
- Los nodos hoja de la estructura son objetos sin hijos.
  - En nuestro ejemplo, son: `Line`, `Rectangle` y `Circle`.
- Los nodos intermedios serán objetos *Composite*.
  - En nuestro ejemplo, el único objeto *Composite* es `Canvas`.
- Una misma operación puede ser usada por ambos tipos de nodos.
  - En ejemplo, vemos cómo todos los objetos usan la función `draw()`.
- Aunque nodo hoja herede todos los métodos, no todos le serán *útil*.
  - En ejemplo, los nodos hoja heredan `add()`, `remove()` y `getChild()`.
  - Son heredados pero solo tienen sentido para el nodo *Composite*.


Shape
+draw()
main
Line
Rectangle
Circle
Canvas
+draw()
Composite object, KEY:
forall shape in children:
shape.draw()

```
int main() {
  Line l;
  l.draw(); // every leaf object can draw itself
  Rectangle r;
  r.draw();
  Circle c;
  c.draw();

  Canvas canvas; // it composes a complete structure...
  canvas.add(&l);
  canvas.add(&r);
  canvas.add(&c);
  canvas.add(&r);
  canvas.draw(); // ...and draw the structure at once!

  return 0;
}
```

- Permite modificar responsabilidad o propiedades de un obj. en t.º ejec.
- Ofrece alternativas a las subclases para extender su funcionalidad.
- Ej.: supongamos que queremos añadir color a las distintas *shapes*.
  - No vamos a crear tres clases nuevas (¡¿y si fueran más?!).
    - `ColoredLine`, `ColoredRectangle` y `ColoredCircle`.
  - En su lugar, podemos crear una nueva clase `ColoredShape`.
    - Y que esta nos permita colorear las figuras que queramos.
- Tipos:
  - *Dynamic Decorator*: agrega al *decorated* obj. por ptr. o ref. en t.º ejec.
    - Op. +común. Cuando no sabemos *a priori* qué objetos se decorarán.
    - Permite jerarquía de clases más flexible que la herencia estática.
  - *Static Decorator*: hereda del *decorated* object. (t.º compilación).
    - Si conocemos de antemano qué objetos y cómo se decorarán.
    - Esta forma sería poco flexible y muy parecida al *Composite*.

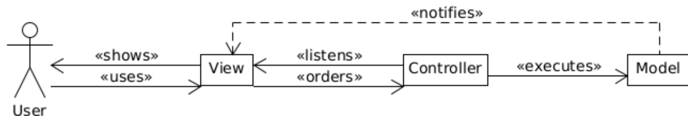
```
class ColoredShape : public Shape {
  public:
    ColoredShape (const Shape& s, const string &c) : shape{s},
        color{c} {
      cout << removeNumbers(typeid(s).name()) << " colored in "
          << c << endl;
    }
    void draw () const {
      cout << "Draw circle\n";
    }

  private:
    const Shape& shape;
    string color;
};
```

```
int main() {
  Line l; // basic objects
  l.draw();
  Rectangle r;
  r.draw();
  Circle c;
  c.draw();

  // dynamic because we decide at runtime which shape is colored
  ColoredShape blueL {l, "blue"}; // objects are colored
  ColoredShape greenR {r, "green"};
  ColoredShape redC {c, "red"};

  return 0;
}
```

- Este patrón eleva aún más el nivel de abstracción de un sistema.
- Oculta detalles de implementación para hacer más sencillo su uso.
- Proporciona un interfaz simplificado/unificado para el sistema.
    - Que oculta un complejo subsistema o subconjunto de interfaces.
- E.g. arrancar sistema complejo, como un coche, con solo girar llave.
    - Con ese gesto se activan, por debajo, otros subsistemas.
    - E.g. motor, sistemas frenos, luces, etc.

```
#include "LightSystem.h"
#include "BrakeSystem.h"

class Car { // Facade class
  public:
    void turnIgnitionOn () {
      engine.turnOn ();
      brakeSystem.turnOn ();
      lightSystem.turnOn ();
    }
  private:
    Engine engine;
    BrakeSystem brakeSystem;
    LightSystem lightSystem;
};
```

```
#include "Engine.h"
#include "Car.h"

int main() {
  Car myCar;
  myCar.turnIgnitionOn (); // very simple ignition

  return 0;
}
```



- Usado para aislar el dominio de aplicación del de presentación.
  - Dicho de otro modo: aislar la lógica del interfaz de un programa.
  - Así, una misma aplicación puede tener diferentes interfaces.
- En este patrón existen tres entidades bien diferenciadas:
  - **Modelo**: contiene únicamente los datos del programa.
  - **Vista**: interfaz de usuario que recibe órdenes de este.
    - Y también del controlador para mostrar info. o modificar interfaz.
  - **Controlador**: intermediario entre la vista y el modelo.
    - Recibe órdenes y traduce esa acción al dominio del modelo.
    - Estas órdenes son normal/ manejadas mediante *callbacks*.
    - Acciones: crear instancia obj., actualizar estados, pedir ops. al modelo...

```
// callback function: is called when data changes
typedef void (*DataChangedCallback)(string data);

class Model { // responsible for getting/setting data
public:
  Model ();
  Model (const string&);
  string getData ();
  void setData (const string&);
  void registerEvent (DataChangedCallback callback);

private:
  string data = "";
  DataChangedCallback event = nullptr;
};
```

```
class View { // responsible to show data to the user
public:
  View ();
  View (const Model&);
  void setModel (const Model&);
  void renderData ();
private:
  Model model;
};
```

```
#include "Model.h"
#include "View.h"

class Controller { // abstracts model from view and viceversa
public:
  Controller();
  Controller(const Model&, const View&);
  void setModel (const Model&);
  void setView (const View&);
  void boot();

private:
  Model model;
  View view;
};
```

```
void changeData (string data) {
  cout << "New data: " << data <<endl;
}

int main() {
  Model model("Model");
  View view(model);
  model.registerEvent (&changeData);

  Controller controller(model, view); // binds model and view

  controller.boot(); // app starts
  model.setData("Booting...");
  return 0;
}
```

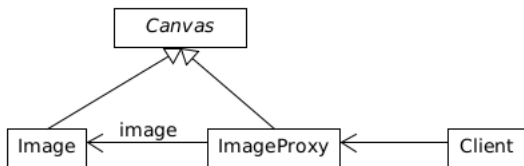
- Convierte el interfaz de una clase en otro demandado por cliente.
- Permite la comunicación entre clases con interfaces incompatibles.
- Muy útil si hay que usar necesaria/ una biblioteca externa.
  - Esta tendrá su interfaz cuya modificación supondría gran trabajo.
  - O que sea software privado sin posibilidad de modificar.
- Veamos ejemplo con adaptación de `LegacyRectangle` a `Rectangle`.

```
class Rectangle { // desidered interface (target)
  public:
    virtual void draw () = 0;
};
```

```
class LegacyRectangle { // legacy class (adaptee)
public:
  LegacyRectangle(int x1, int y1, int x2, int y2) {
    x1_ = x1;
    y1_ = y1;
    x2_ = x2;
    y2_ = y2;
    std::cout << "LegacyRectangle with [(x1,y1),(x2,y2)]
        coordinates is created\n";
  }
  void legacyDraw() {
    std::cout << "LegacyDraw rectangle is shown\n";
  }
private:
  int x1_;
  int y1_;
  int x2_;
  int y2_;
};
```

```
#include "Rectangle.h"
#include "LegacyRectangle.h"

// Adapter wrapper
class RectangleAdapter : public Rectangle, private
    LegacyRectangle {
public:
  RectangleAdapter (int x, int y, int w, int h):
    LegacyRectangle (x, y, x + w, y + h) {
      std::cout << "RectangleAdapter with [(x,y),(x+w,x+h)]
          coordinates is created\n";
    }

  void draw() {
    std::cout << "RectangleAdapter draw() method is called\n";
    legacyDraw ();
  }
};
```

```
int main() {
  int x = 10, y = 40, w = 320, h = 240;

  // I can use new interface (Rectangle) through the adapter
  Rectangle *r = new RectangleAdapter (x,y,w,h); // new coord.
      system

  r->draw(); // new drawing interface
  return 0;
}

/* Output:
LegacyRectangle with [(x1,y1),(x2,y2)] coordinates is created
RectangleAdapter with [(x,y),(x+w,x+h)] coordinates is created
RectangleAdapter draw() method is called
LegacyDraw rectangle is shown
*/
```



- Proporciona al cliente un objeto que actúa como sustituto del real.
- El obj. `Proxy` recibe petición y se la pasa al objeto real.
  - Y antes de pasarla hace algo: control de acceso, *caching*, etc.
  - El obj. real realiza casi todo el trabajo, el `Proxy`, controla acceso.
- El `Proxy` y el obj. real implementan un mismo interfaz.
  - Lo que permite al cliente tratar al `Proxy` como al obj. real.
- Uso: si se necesita añadir comporta/ a obj. sin modificar su clase.
- Ejemplo: mostrar una imagen cuya carga es costosa computacional/.
  - Sol.: `ImageProxy` que representa al obj. real `Image`.
  - Comporta/ modificado: obj. proxy puede cargar una sola vez imagen...
    - ...y la muestra tantas veces como solicite el cliente.

```
class Canvas {
public:
   virtual void request() = 0;
   virtual ~Canvas() {}
};
#endif
```

```
class Image : public Canvas {
public:
   void request() {
      cout << "Image (real subject): request()" << endl;
   }
};
```

```
#include "Canvas.h"
#include "ImageProxy.h"
#include "Image.h"

class ImageProxy : public Canvas {
private:
   Canvas* image; // ptr to the subject (canvas)
public:
   ImageProxy() : image (new Image()) {}
   ~ImageProxy() {
      delete image;
   }

   void request() { // forward calls to the (real) Image
      image->request();
   }
};
```

```
#include "ImageProxy.h"

int main() {
   ImageProxy imageProxy;
   imageProxy.request();
}
```

11. Patrones de diseño

Julio Vega

julio.vega@urjc.es