Tema IV

Funciones y Clases

***Objetivos*:**

- ▷ **Introducir el concepto de función y parámetro para no tener que repetir código.**
- ▷ **Introducir el principio de ocultación de información.**
- ▷ **Introducir el principio de encapsulación que permitirá empaquetar datos y funciones en un mismo módulo: la Clase.**
- ▷ **Construir objetos sencillos que cumplan principios básicos de diseño.**

Sugerencias: por favor, enviar un e-mail a JC.Cubero@decsai.ugr.es

IV.1. Funciones

IV.1.1. Fundamentos

IV.1.1.1. Las funciones realizan una tarea

A la hora de programar, es normal que haya que repetir las mismas acciones con distintos valores. Para no repetir el mismo código, tendremos que:

- ▷ **Identificar las acciones repetidas**
- ▷ **Identificar los valores que pueden variar (esto es, los parámetros)**
- ▷ **Definir una función que encapsule dichas acciones**

Las funciones como `sqrt`, `tolower`, etc., no son sino ejemplos de funciones incluidas en `cmath` y `cctype`, respectivamente, que resuelven tareas concretas y devuelven un valor.

Se suele utilizar una notación prefija, con parámetros (en su caso) separados por comas y encerrados entre paréntesis.

```
int main(){
   double lado1, lado2, hipotenusa, radicando;

   <Asignación de valores a los lados>

   radicando = lado1*lado1 + lado2*lado2;
   hipotenusa = sqrt(radicando);
```

Si queremos calcular la hipotenusa de dos triángulos rectángulos:

```
int main(){
   double lado1_A, lado2_A, lado1_B, lado2_B,
          hipotenusa_A, hipotenusa_B,
          radicando_A, radicando_B;

   <Asignación de valores a los lados>

   radicando_A = lado1_A*lado1_A + lado2_A*lado2_A;
   hipotenusa_A = sqrt(radicando_A);

   radicando_B = lado1_B*lado1_B + lado2_B*lado2_B;
   hipotenusa_B = sqrt(radicando_B);
   .......
}
```

¿No sería más claro, menos propenso a errores y más reutilizable el código si pudiésemos definir nuestra función `Hipotenusa`? El fragmento de código anterior quedaría como sigue:

```
int main(){
   double lado1_A, lado2_A, lado1_B, lado2_B,
          hipotenusa_A, hipotenusa_B;

   <Asignación de valores a los lados>

   hipotenusa_A = Hipotenusa(lado1_A, lado2_A);
   hipotenusa_B = Hipotenusa(lado1_B, lado2_B);
   ......
```

Los lenguajes de programación proporcionan distintos mecanismos para englobar un conjunto de sentencias en un paquete o *módulo (module)* . En este tema veremos dos tipos de módulos: las funciones y las clases.

IV.1.1.2. Definición

```
<tipo> <nombre-función> (<parámetros formales>) {
   <sentencias>
   return <expresión> ;
}
```

▷ **Por ahora, la definición se pondrá después de la inclusión de bibliotecas y antes del `main`. Antes de usar una función en cualquier sitio, hay que poner su definición.**

▷ **Diremos que**

`<tipo> <nombre-función> (<parám.formales>)`

es la *cabecera (header)* de la función.

> ***Los parámetros formales son los datos que la función necesita conocer del exterior, para poder hacer sus cómputos***

▷ **El cuerpo de la función debe contener:**

`return` <***expresión***>;

donde <*expresión*> ha de ser del mismo tipo que el especificado en la cabecera de la función (también puede ser un tipo *compatible*). El valor que contenga dicha expresión es el valor que devuelve la función cuando es llamada.

```
double Cuadrado(double entrada){
   return entrada*entrada;
}
```

El valor calculado por la función es el resultado de evaluar la expresión en la sentencia `return`

- **La llamada a una función constituye una expresión:**

  `Cuadrado(valor)` **es una expresión.**

- **En C++ no se pueden definir funciones dentro de otras. Todas están al mismo nivel.**

- **Como estilo de codificación, escribiremos la primera letra de las funciones en mayúscula. Si es un nombre compuesto, también irá en mayúcula la primera letra. Es el denominado estilo *UpperCamelCase* (también conocido como *estilo PascalCase*).**

IV.1.1.3. Parámetros formales y actuales

- **Los *parámetros formales (formal parameters)* son aquellos especificados en la cabecera de la función.**
  Al declarar un parámetro formal hay que especificar su tipo de dato. Si hay más de uno, se separan con comas.
  Los parámetros formales sólo se conocen dentro de la función.
  Si no hay ningún parámetro, basta poner `()` o, si se prefiere, `(void)`

- **Los *parámetros actuales (actual parameters)* son las expresiones pasadas como argumentos en la llamada a una función. El formato de la llamada es:**

  `<nombre-función> (<parámetros actuales>)`

  En la llamada no se especifica el tipo. Si hay más de uno, también se separan con comas.
  Si no hay ningún parámetro, la llamada será `<nombre-función>()`

  Nota:

  > **La traducción correcta de *actual* es *real*, pero también es válido el nombre *actual* en español para denotar el valor que se usa en el mismo momento de la llamada.**

```
double Cuadrado(double entrada){    // entrada: parámetro formal
   return entrada*entrada;
}
int main(){
   double resultado, valor;

   valor = 4;
   resultado = Cuadrado(valor);     // valor: parámetro actual
   cout << "El cuadrado de " << valor << " es " << resultado;
}
```

▶ *Flujo de control*

Cuando se ejecuta la llamada `resultado = Cuadrado(valor);` el flujo de control salta a la definición de la función.

- ▷ **Se realiza la correspondencia entre los parámetros.**
  El correspondiente parámetro formal recibe una copia del parámetro actual, es decir, se realiza la siguiente *asignación en tiempo de ejecución*:

  `<parámetro formal> = <parámetro actual>`

  En el ejemplo, `entrada = 4`

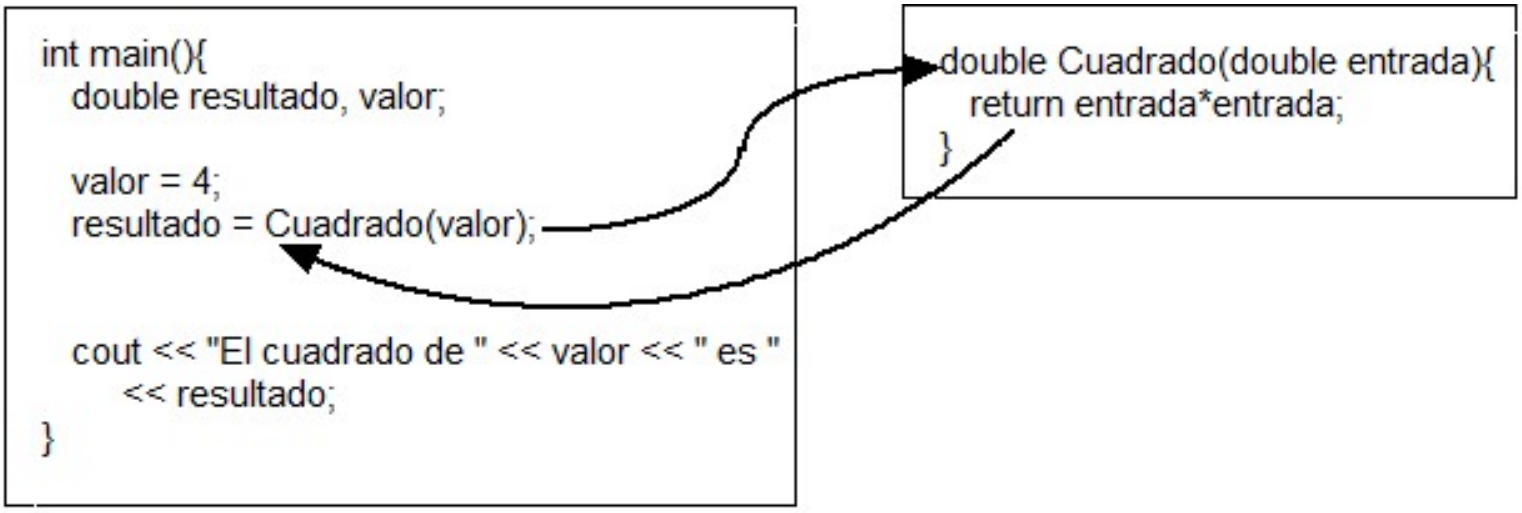
  Esta forma de pasar parámetros se conoce como *paso de parámetro por valor (pass-by-value)* .

- ▷ **Empiezan a ejecutarse las sentencias de la función y, cuando se llega a alguna sentencia `return <expresión>`, termina la ejecución de la función y devuelve el resultado de evaluar `<expresión>` al lugar donde se realizó la invocación.**

- ▷ **A continuación, el flujo de control prosigue por donde se detuvo al realizar la invocación de la función.**

```
double Cuadrado(double entrada){
   return entrada*entrada;
}
int main(){
   double resultado, valor;

   valor = 4;
   resultado = Cuadrado(valor);    // resultado = 16
   cout << "El cuadrado de " << valor << " es "
        << resultado;
}
```

entrada = valor



► *Correspondencia entre parámetros actuales y formales*

▷ **Debe haber exactamente el *mismo número* de parámetros actuales que de parámetros formales.**

```
double Cuadrado(double entrada){
   return entrada*entrada
}
int main(){
   double resultado;
   resultado = Cuadrado(5, 8);  // Error en compilación
   ......
```

▷ **Debemos garantizar que el parámetro actual tenga un valor correcto antes de llamar a la función.**

```
double Cuadrado(double entrada){
   return entrada*entrada
}
int main(){
   double resultado, valor;
   resultado = Cuadrado(valor);  // Error lógico
   ......
```

▷ **La correspondencia se establece por *orden de aparición*, uno a uno y de izquierda a derecha.**

```
double Resta(double valor_1, double valor_2){
   return valor_1 - valor2;
}
int main(){
   double un_valor = 5.0, otro_valor = 4.0;
   double resta;

   resta =  Resta(un_valor, otro_valor);  // 1.0
   resta =  Resta(otro_valor, un_valor);  // -1.0
```

- **El parámetro actual puede ser una expresión.**
  Primero se evalúa la expresión y luego se realiza la llamada a la función.

```
hipotenusa_A = Hipotenusa(3 * lado1_A , 3 * lado2_A);
```

  En la medida de lo posible, no abusaremos de este tipo de llamadas.

- **Cada parámetro formal y su correspondiente parámetro actual han de ser del *mismo tipo* (o *compatible*)**

```
double Cuadrado(double entrada){
   return entrada*entrada
}
int main(){
   double resultado;
   int valor = 7;
   resultado = Cuadrado(valor);   // casting automático
   ......
}
```

  ***Problema*: que el tipo del parámetro formal sea más *pequeño* que el actual. Si el parámetro actual tiene un valor que no cabe en el formal, se produce un desbordamiento aritmético, tal y como ocurre en cualquier asignación.**

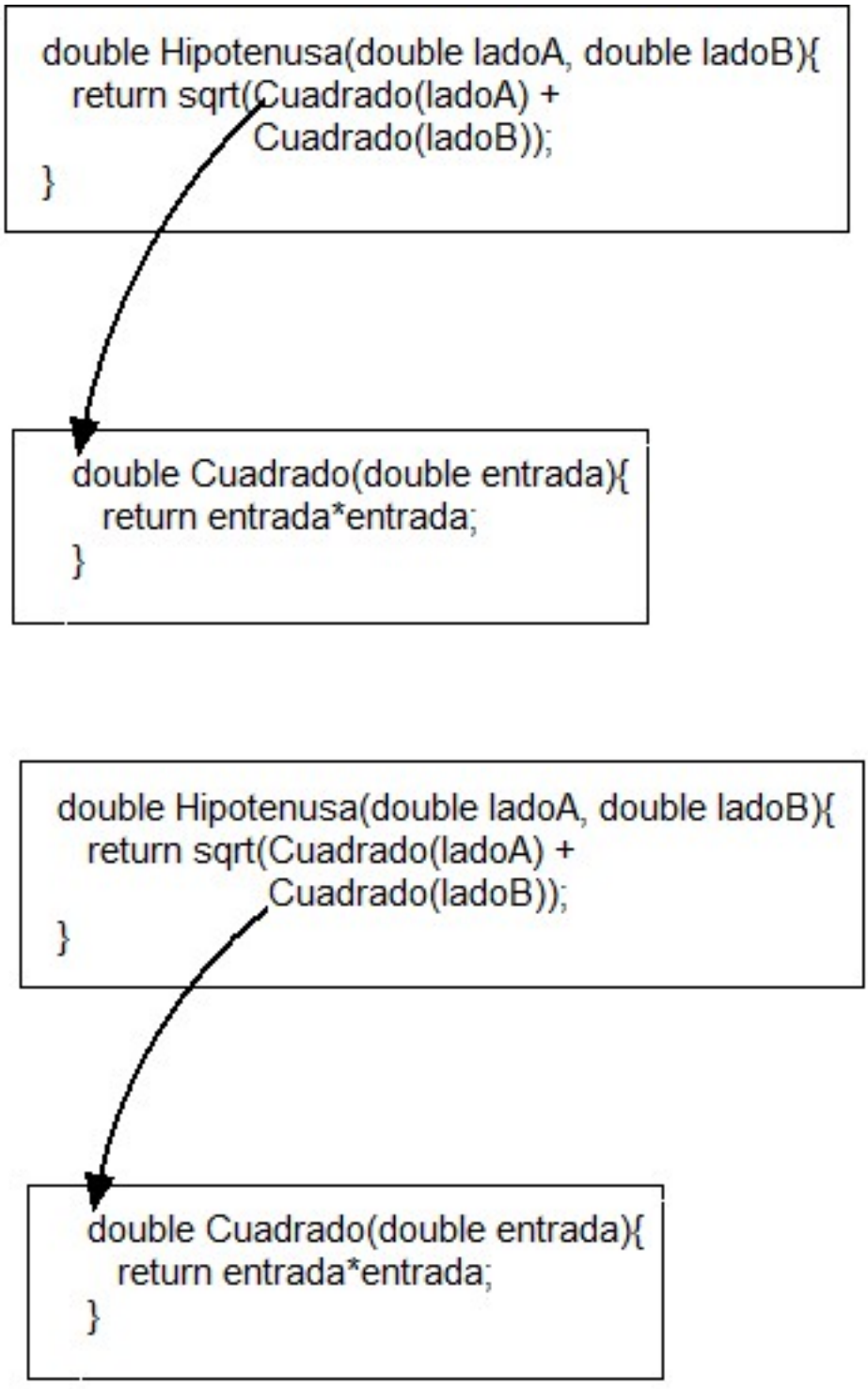
```
int DivisonEntera(int numerador, int denominador){
   return numerador / denominador;
}
int main(){

   ... DivisonEntera(400000000000, 10);   // Desbordamiento
```

▷ **Dentro de una función, se puede llamar a cualquier otra función que esté definida con anterioridad. El paso de parámetros entre funciones se rige por las mismas normas que hemos visto.**

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

double Cuadrado(double entrada){
   return entrada*entrada;
}

double Hipotenusa(double un_lado, double otro_lado){
   return sqrt(Cuadrado(un_lado) + Cuadrado(otro_lado));
}

int main(){
   double lado1_A, lado2_A, lado1_B, lado2_B,
          hipotenusa_A, hipotenusa_B;

   <Asignación de valores a los lados>

   hipotenusa_A = Hipotenusa(lado1_A, lado2_A);
   hipotenusa_B = Hipotenusa(lado1_B, lado2_B);
   ......
}
```

Si cambiamos el orden de definición de las funciones se produce un error de compilación.

Flujo de control:



▷ **Las modificaciones del parámetro formal no afectan al parámetro actual. Recordemos que el paso por valor conlleva trabajar con una copia del valor correspondiente al parámetro actual, por lo que éste no se modifica.**

```
double Cuadrado(double entrada){
   entrada = entrada * entrada;
   return entrada;
}

int main(){
   double resultado, valor = 3;

   resultado = Cuadrado(valor);

   // valor sigue siendo 3
   ......
}
```

Ejercicio. Defina la función `Hipotenusa`

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

double Hipotenusa(double un_lado, double otro_lado){
   return sqrt(un_lado*un_lado + otro_lado*otro_lado);
}

int main(){
   double lado1_A, lado2_A, lado1_B, lado2_B,
          hipotenusa_A, hipotenusa_B;

   <Asignación de valores a los lados>

   hipotenusa_A = Hipotenusa(lado1_A, lado2_A);
   hipotenusa_B = Hipotenusa(lado1_B, lado2_B);
   ............
}
```

http://decsai.ugr.es/jccubero/FP/IV_hipotenusa.cpp

Ejercicio. Comprobar si dos reales son iguales, aceptando un margen en la diferencia de 0.000001

```
bool SonIguales (double uno, double otro){
   return (abs(uno - otro) <= 1e-6);  // abs = Valor absoluto (cmath)
}
```

Ejercicio. Compruebe si un número es par.

```
bool EsPar(int n){
    if (n % 2 == 0)
       return true;
    else
       return false;
}
int main(){
   int un_numero;
   bool es_par_un_numero;

   cout << "Comprobar si un número es par.\n\n";
   cout << "Introduzca un entero: ";
   cin >> un_numero;

   es_par_un_numero = EsPar(un_numero);

   if (es_par_un_numero)
      cout << un_numero << " es par";
   else
      cout << un_numero << " es impar";
}
```

De forma más compacta:

```
bool EsPar (int n){
    return n % 2 == 0;
}
```

http://decsai.ugr.es/jccubero/FP/IV_par.cpp

La siguiente función compila correctamente pero se puede producir un error lógico durante su ejecución:

```
bool EsPar (int n)  {
    if (n%2 == 0)

        return true;
}
```

Una función debería devolver siempre un valor. En caso contrario, se produce un comportamiento indeterminado (undefined bahaviour) (recuerde lo visto en la página 301).

En resumen:

Definimos una única vez la función y la llamamos donde sea necesario. En la llamada a una función sólo nos preocupamos de saber su nombre y cómo se utiliza (los parámetros y el valor devuelto). Esto hace que el código sea:

- ***Menos propenso a errores***

  Después de un `copy-paste` del código a repetir, si queremos que funcione con otros datos, debemos cambiar una a una todas las apariciones de dichos datos, por lo que podríamos olvidar alguna.

- ***Más fácil de mantener***

  Ante posibles cambios futuros, sólo debemos cambiar el código que hay dentro de la función. El cambio se refleja automáticamente en todos los sitios en los que se realiza una llamada a la función

El programador debe identificar las funciones antes de escribir una sola línea de código. En cualquier caso, no siempre se detectan a priori las funciones, por lo que, una vez escrito el código, si detectamos bloques que se repiten, deberemos englobarlos en una función.

Durante el desarrollo de un proyecto software, primero se diseñan los módulos de la solución y a continuación se procede a implementarlos.

IV.1.1.4. Datos locales

Recuerde que el *ámbito (scope)* de un dato (variable o constante) `v` es el conjunto de todos aquellos sitios que pueden acceder a `v`.

Dentro de una función podemos declarar constantes y variables. Estas constantes y variables sólo se conocerán dentro de la función, por lo que se les denomina *datos locales (local data)* .

Usaremos los datos locales en dos situaciones:

1. **Para almacenar resultados temporales necesarios para hacer los cómputos requeridos en la función.**
2. **En muchas situaciones, declararemos un dato local asociado al valor que devolverá la función. Se le asignará un valor y se devolverá al final con una sentencia `return`.**

   Usaremos un nombre similar a la función pero en minúscula.

De hecho, los parámetros formales se pueden considerar datos locales.

```
<tipo> <nombre-función> (<parámetros formales>) {
   <constantes locales>
   <variables locales>
   <sentencias>
   return <expresión> ;
}
```

Al igual que ocurre con la declaración de variables del `main`, las variables locales a una función no inicializadas a un valor concreto tendrán un valor indeterminado al inicio de la ejecución de la función.

Ejemplo. **Calcule el factorial de un valor. Recordemos la forma de calcular el factorial de un entero `n`:**

```
fact = 1;
for (i = 2; i <= n ; i++)
   fact = fact * i;
```

Definimos la función `Factorial`:

- **La función únicamente necesita conocer el valor de `n`, que será de tipo `int`.**
- **Con un `int` de 32 bits, el máximo factorial computable es 12. Con un `long long` de 64 bits podemos llegar hasta 20.**

  Así pues, la cabecera será:

  ```
  long long Factorial(int n)
  ```

- **Para hacer los cómputos del factorial incluimos como datos locales a la función las variable `i` y `fact`. Éstas no se conocen fuera de la función.**

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

long long Factorial (int n){
   int i;
   long long fact = 1;

   for (i = 2; i <= n; i++)
      fact = fact * i;

   return fact;
}
```

```
int main(){
   int valor;
   long long resultado;

   cout << "Cómputo del factorial de un número\n";
   cout << "\nIntroduzca un entero: ";
   cin >> valor;

   resultado = Factorial(valor);
   cout  << "\nFactorial de " << valor << " = " << resultado;
}
```

http://decsai.ugr.es/jccubero/FP/IV_factorial.cpp

Dentro de una función no podemos acceder a los datos locales definidos en otras funciones ni a los de `main`.

```
long long Factorial (int n){
   int i;
   long long fact = 1;

   for (i = 2; i <= valor; i++)   //  Error de compilación
      fact = fact * i;

   return fact;
}
int main(){
   int valor:
   int resultado;

   cout << i;    //  Error de compilación
   n = 5;        //  Error de compilación

   cout << "\nIntroduzca valor";
   cin >> valor;

   resultado = Factorial(valor);
   cout  << "\nFactorial de " << valor << " = " << resultado;
}
```



Por tanto, los nombres dados a los parámetros formales pueden coincidir con los nombres de los parámetros actuales, o con los de cualquier dato de cualquier otra función: son datos distintos.

```
long long Factorial (int n){      // <- n de Factorial. OK
   int i;
   long long fact = 1;

   for (i = 2; i <= n; i++)
      fact = fact * i;

   return fact;
}

int main(){
   int n = 3;                     // <- n de main. OK
   int resultado;

   resultado = Factorial(n);
   cout  << "Factorial de " << n << " = "  << resultado;

         // Imprime en pantalla lo siguiente:
         // Factorial de 3 = 6
}
```

Ejemplo. **Vimos la función para calcular la hipotenusa de un triángulo rectángulo. Podemos darle el mismo nombre a los parámetros actuales y formales.**

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

double Hipotenusa(double un_lado, double otro_lado){
   return sqrt(un_lado*un_lado + otro_lado*otro_lado);
}

int main(){
   double un_lado, otro_lado, hipotenusa;

   cout << "\nIntroduzca primer lado";
   cin >> un_lado;
   cout << "\nIntroduzca segundo lado";
   cin >> otro_lado;

   hipotenusa = Hipotenusa(un_lado, otro_lado);
   cout << "\nLa hipotenusa vale " << hipotenusa;
}
```

Ejercicio. **Compruebe si un número es primo (recuerde el algoritmo visto en la página 240)**

```
bool EsPrimo(int valor){
   bool es_primo;
   int divisor;

   es_primo = true;

   for (divisor = 2 ; divisor < valor && es_primo ; divisor++)
      if (valor % divisor == 0)
         es_primo = false;

   return es_primo;
}

int main(){
   int un_numero;
   bool es_primo;

   cout << "Comprobar si un número es primo.\n\n";
   cout << "Introduzca un entero: ";
   cin >> un_numero;

   es_primo = EsPrimo(un_numero);

   if (es_primo)
      cout << un_numero << " es primo";
   else
      cout << un_numero << " no es primo";
}
```

http://decsai.ugr.es/jccubero/FP/IV_primo.cpp

Ejercicio. Calcule el MCD de dos enteros.

```
int MCD(int primero, int segundo){
/*
   Vamos dividiendo los dos enteros por todos los
   enteros menores que el menor de ellos hasta que:
   - ambos sean divisibles por el mismo valor
   - o hasta que lleguemos al 1
*/
   bool mcd_encontrado = false;
   int divisor, mcd;

   if (primero == 0 || segundo == 0)
      mcd = 0;
   else{
      if (primero > segundo)
         divisor = segundo;
      else
         divisor = primero;

      mcd_encontrado = false;

      while (!mcd_encontrado){
         if (primero % divisor == 0 && segundo % divisor == 0)
            mcd_encontrado = true;
         else
            divisor--;
      }
      mcd = divisor;
   }

   return mcd;
}
```

```
int main(){
   int un_entero, otro_entero, maximo_comun_divisor;

   cout << "Calcular el MCD de dos enteros.\n\n";
   cout << "Introduzca dos enteros: ";
   cin >> un_entero;
   cin >> otro_entero;

   maximo_comun_divisor = MCD(un_entero, otro_entero);

   cout << "\nEl máximo común divisor de " << un_entero
        << " y " << otro_entero << " es: " << maximo_comun_divisor;
}
```

http://decsai.ugr.es/jccubero/FP/IV_mcd_funcion.cpp

Ejemplo. **Construya una función para leer un entero estrictamente positivo.**

```
int LeePositivo(string mensaje){
   int a_leer;

   cout << mensaje;

   do{
      cin >> a_leer
   }while (a_leer <= 0);

   return a_leer;
}

int main(){
   int salario;

   salario = LeePositivo("\nIntroduzca el salario en miles de euros: ");
   ......
}
```

La lectura anterior no nos protege de desbordamientos. Para ello, habría que realizar una entrada sobre un tipo `string` y analizar la cadena.

IV.1.1.5. La Pila

Cada vez que se llama a una función, se crea dentro de una zona de memoria llamada *pila (stack)* , un compartimento de trabajo asociado a ella, llamado *marco de pila (stack frame)* .

- ▷ **Cada vez que se llama a una función se crea el marco asociado.**
- ▷ **En el marco asociado a cada función se almacenan, entre otras cosas:**
  - **Los parámetros formales.**
  - **Los datos locales (constantes y variables).**
  - **La *dirección de retorno* de la función.**
- ▷ **Cuando una función llama a otra, el marco de la función llamada se apila sobre el marco de la función desde donde se hace la llamada (de ahí el nombre de pila). Hasta que no termine de ejecutarse la última función llamada, el control no volverá a la anterior.**

Ejemplo. **En una competición deportiva, hay cinco equipos y tienen que jugar todos contra todos. ¿Cuántos partidos han de jugarse en total? Se supone que hay una sola vuelta. La respuesta es el número de combinaciones de cinco elementos tomados de dos en dos (sin repetición y no importa el orden)**

http://www.disfrutalasmatematicas.com/combinatoria/combinaciones-permutaciones.html

El combinatorio de dos enteros a y b se define como:

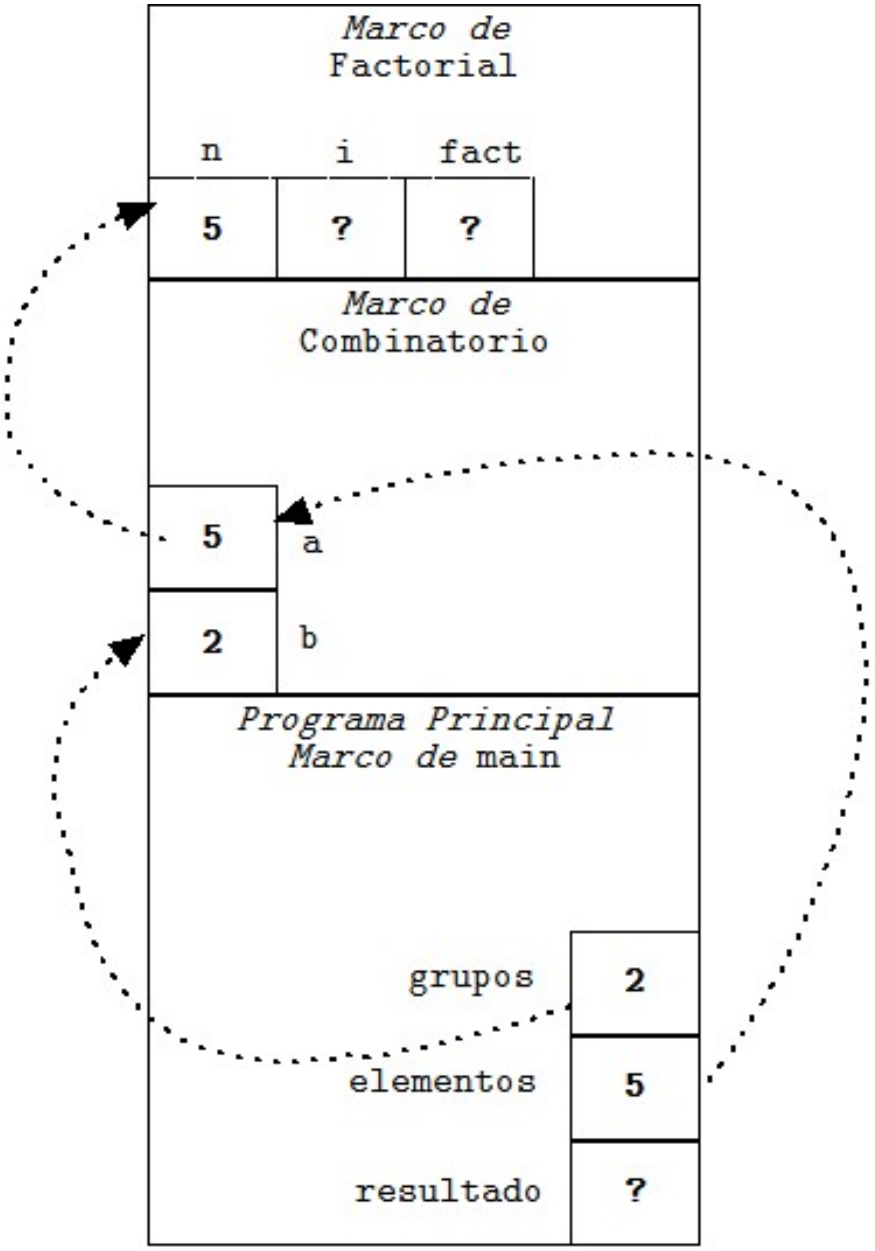
$$\binom{a}{b} = \frac{a!}{b!(a-b)!}$$

Construimos la función `Combinatorio` que llama a la función `Factorial`:

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

long long Factorial (int n){
   int i;
   long long fact = 1;

   for (i = 2; i <= n; i++)
      fact = fact * i;

   return fact;
}

long long Combinatorio(int a, int b){
   return Factorial(a) / (Factorial(b) * Factorial(a-b));
}

int main(){
```

```
    int elementos, grupos;
    long long resultado;

    cout << "Número Combinatorio.\n";
    cout << "Introduzca número total de elementos a combinar: ";
    cin >> elementos;
    cout << "Introduzca cuántos se escogen en cada grupo: ";
    cin >> grupos;

    resultado = Combinatorio(elementos, grupos);
    cout  << elementos << " sobre " << grupos << " = " << resultado;
}
```

http://decsai.ugr.es/jccubero/FP/IV_combinatorio.cpp



Pila

`main` es de hecho una función como otra cualquiera, por lo que también se almacena en la pila. Es la primera función llamada al ejecutarse el programa.

Nota:

La función **`main`** devuelve un entero al Sistema Operativo y puede tener más parámetros, pero en FP sólo veremos el caso sin parámetros. Por eso, la hemos declarado siempre como:

```
int main(){
   ......
}
```

▷ Si el programa termina con un error, debe devolver un entero distinto de 0.

▷ Si el programa termina sin errores, se debe devolver 0.

Puede indicarse incluyendo **`return 0;`** al final de **`main`** (antes de **`}`**)

En C++, si la función **`main`** no incluye una sentencia **`return`** y termina de ejecutarse correctamente el programa, se devuelve 0 por defecto, por lo que podríamos suprimir **`return 0;`** (sólo en la función **`main`**)

IV.1.1.6. Funciones void

Ejemplo. **Queremos construir un programa para calcular la hipotenusa de un triángulo rectángulo con una presentación al principio de la siguiente forma:**

```
int i, tope_lineas;


......
for (i = 1; i <= tope_lineas ; i++)
   cout << "\n************";
cout << "Programa básico de Trigonometría";
for (i = 1; i <= tope_lineas ; i++)
   cout << "\n************";
......
```

¿No sería más fácil de entender si el código del programa principal hubiese sido el siguiente?

```
......
Presentacion(tope_lineas);
......
```

En este ejemplo, `Presentacion` **resuelve la tarea de realizar la presentación del programa por pantalla, pero no calcula (devuelve) ningún valor, como sí ocurre con las funciones** `sqrt` **o** `Hipotenusa`**. Por eso, su llamada constituye una sentencia y no aparece dentro de una expresión. Este tipo particular de funciones que no devuelven ningún valor, se definen como sigue:**

```
void <nombre-función> (<parámetros formales>) {
   <constantes locales>
   <variables locales>
   <sentencias>
}
```

Los lenguajes suelen referirse a este tipo de funciones con el nombre *procedimiento (procedure)*

Obsérvese que no hay sentencia `return`. La función `void` termina cuando se ejecuta la última sentencia de la función

---

Nota:

Realmente, el lenguaje permite incluir una sentencia **`return ;`** en cualquier sitio para finalizar la ejecución del **`void`**, pero, como ya comentaremos en la página 425, debemos evitar esta forma de terminar una función.

El paso de parámetros y la definición de datos locales sigue las mismas normas que el resto de funciones.

Para llamar a una función `void`, simplemente, ponemos su nombre y la lista de parámetros actuales con los que realizamos la llamada:

```
void MiFuncionVoid(int parametro_formal, double otro_parametro_formal){
   ......
}
int main(){
   int parametro_actual;
   double otro_parametro_actual;
   ......
MiFuncionVoid(parametro_actual, otro_parametro_actual);
   ......
```

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

double Cuadrado(double entrada){
   return entrada*entrada;
}

double Hipotenusa(double un_lado, double otro_lado){
   return sqrt(Cuadrado(un_lado) + Cuadrado(otro_lado));
}

void Presentacion(int tope_lineas){
   int i;
   for (i = 1; i <= tope_lineas ; i++)
      cout << "\n************";
   cout << "Programa básico de Trigonometría";
   for (i = 1; i <= tope_lineas ; i++)
      cout << "\n************";
}
int main(){
   double lado1, lado2, hipotenusa;

   Presentacion(3);

   cout << "\n\nIntroduzca los lados del triángulo rectángulo: ";
   cin >> lado1;
   cin >> lado2;

   hipotenusa = Hipotenusa(lado1,lado2);

   cout << "\nLa hipotenusa vale " << hipotenusa;
}
```

Como cualquier otra función, las funciones `void` pueden llamarse desde cualquier otra función (`void` o cualquier otra), siempre que su definición vaya antes.

Ejercicio. **Aisle la impresión de las líneas de asteriscos en una función aparte, por si quisiéramos usarla en otros sitios.**

```
void ImprimeLineas (int num_lineas){
   int i;

   for (i = 1; i <= num_lineas ; i++)
      cout << "\n**********";
}

void Presentacion(int tope_lineas){
   ImprimeLineas (tope_lineas);
   cout << "Programa básico de Trigonometría";
   ImprimeLineas (tope_lineas);
}
```

IV.1.1.7. Sobrecarga de funciones (Ampliación)

C++ permite definir *sobrecarga de funciones (function overload)* , es decir, varias funciones con el mismo nombre pero que difieren:

- **O bien en el número de parámetros**
- **O bien en el tipo de los argumentos**

El compilador llamará a la versión que mejor encaje.

Dos versiones sobrecargadas no pueden diferenciarse, únicamente, en el tipo de dato devuelto, ya que el compilador no sabría a cuál llamar, ya que la expresión resultante podría asignarse a datos de diferentes tipos (por el casting automático)

Ejemplo.

```
double Max(double uno, double otro){
   ......
}
double Max(double uno, double otro, double tercero){
   ......
}
/*
// Error de compilación:
float Max(double uno, double otro){
   ......
}
*/
int main(){
   ......
   max = Max(3.5, 4.3);        // Primera versión
   max = Max(3.5, 4.3, 8.5);   // Segunda versión
   ......
```

Si el compilador encuentra una llamada *ambigua*, dará un error en tiempo de compilación:

Ejemplo.

```
double Max(double uno, double otro){
   ......
}
long Max(long uno, long otro){
   ......
}

int main(){
   int a, b, max;
   ......
   max = Max(a, b);        // Error de compilación
   ......
}
```

Los parámetros actuales `a` y `b` pueden ser transformados tanto a un `double` como a un `long`, por lo que la llamada es ambigua y el programa no se puede compilar.

Consejo*:** ***No abuse de la sobrecarga de funciones que únicamente difieran en los tipos de los parámetros formales

IV.1.2. Ámbito de un dato (revisión)

IV.1.2.1. Sobre las variables globales

Recuerde que el *ámbito (scope)* de un dato es el conjunto de todos aquellos sitios que pueden acceder a él.

► ***Lo que ya sabemos:***

Datos locales (declarados dentro de una función)

- ▷ **El ámbito es la propia función.**
- ▷ **No se puede acceder al dato desde otras funciones.**

***Nota*. Lo anterior se aplica también, como caso particular, a la función** `main`

Parámetros formales de una función

- ▷ **Se consideran datos locales de la función.**

► ***Lo nuevo (muy malo):***

Datos globales (global data)

- ▷ **Son datos declarados fuera de las funciones y del main.**
- ▷ **El ámbito de los datos globales está formado por todas las funciones que hay definidas con posterioridad** 

El uso de variables globales permite que todas las funciones las puedan modificar. Esto es pernicioso para la programación, fomentando la aparición de *efectos laterales (side effects)* .

Ejemplo. **Supongamos un programa para la gestión de un aeropuerto. Tendrá dos funciones:** `GestionMaletas` **y** `ControlVuelos`**. El primero controla las cintas transportadoras y como máximo puede gestionar 50 maletas. El segundo controla los vuelos en el área del aeropuerto y como máximo puede gestionar 30. El problema es que ambos van a usar el mismo dato global** `Max` **para representar dichos máximos.**

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

int Max;                // Variables globales
bool saturacion;

void GestionMaletas(){
   Max = 50;

   if (NumMaletasActual <= Max)
      [acciones maletas]
   else
      ActivarEmergenciaMaletas();
}
void ControlVuelos(){
   if (NumVuelosActual <= Max)
      [acciones vuelos]
   else{
      ActivarEmergenciaVuelos();
      saturacion = true;
   }
}
int main(){
   Max = 30;
   saturacion = false;

   while (!saturacion){
      GestionMaletas();   // Efecto lateral: Max = 50
      ControlVuelos();
   }
   ......
}
```

El uso de variables globales hace que los programas sean mucho más difíciles de depurar ya que la modificación de éstas puede hacerse en cualquier función del programa.

El que un lenguaje como C++ permita asignar un ámbito global a una variable, no significa que esto sea adecuado o recomendable.

El uso de variables globales puede provocar graves efectos laterales, por lo que su uso está completamente prohibido en esta asignatura.

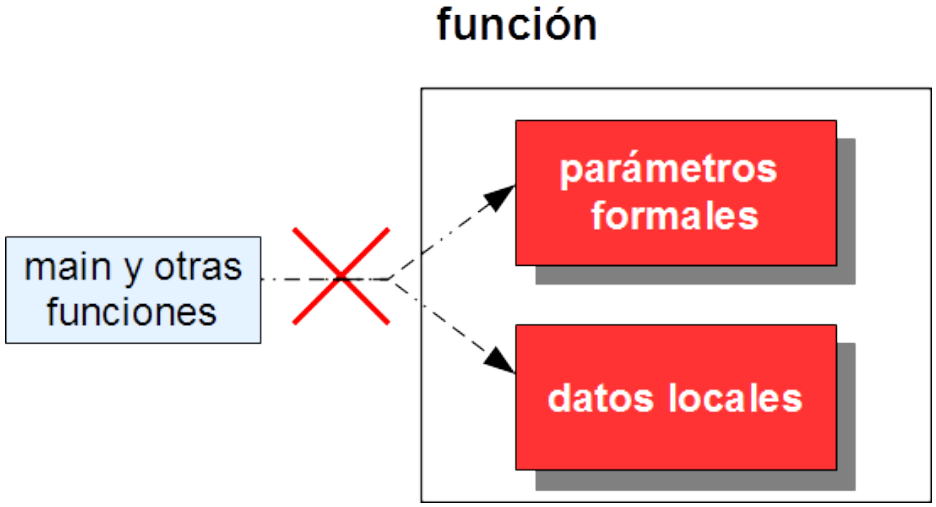
***Nota*. El uso de *constantes globales* es menos perjudicial ya que, si no pueden modificarse, no se producen efectos laterales**

IV.1.2.2. El principio de ocultación de información

¿Qué pasaría si los datos locales fuesen accesibles desde otras funciones?

- **Código propenso a errores, ya que podríamos modificar los datos locales por accidente y provocar errores.**
- **Código difícil de mantener, ya que no podríamos cambiar la definición de una función A, suprimiendo, por ejemplo, alguna variable local de A a la que se accediese desde otra función B.**

Cada función tiene sus propios datos (parámetros formales y datos locales) y no se conocen en ningún otro sitio. Esto impide que una función pueda interferir en el funcionamiento de otras.



En la llamada a una función no nos preocupamos de saber cómo realiza su tarea. Esto nos permite, por ejemplo, mejorar la implementación de una función sin tener que cambiar una línea de código del programa principal que la llama.

Ejemplo. **Sobre el ejercicio que comprueba si un número es primo (página 383) podemos mejorar la implementación, parando al llegar a la raíz cuadrada del valor sin haber encontrado un divisor (ver página 241)**

```
bool EsPrimo(int valor){
   bool es_primo;
   int divisor;
   double tope;

   es_primo = true;
   tope = sqrt(valor);

   for (divisor = 2 ; divisor <= tope && es_primo ; divisor++)
      if (valor % divisor == 0)
         es_primo = false;

   return es_primo;
}

int main(){
   ......
   No cambia nada
   ......
}
```

No importa si el lector no comprende el mecanismo matemático anterior.

Al haber *ocultado información* (los datos locales no se conocen fuera de la función), los cambios realizados dentro de la función no afectan al exterior. Ésto me permite seguir realizando las llamadas a la función al igual que antes (la cabecera no ha cambiado).

Ejemplo. **Retomamos el ejemplo del número combinatorio de la página 388. Para calcular los resultados posibles de la lotería primitiva, habría que calcular el número combinatorio 45 sobre 6, es decir,** `elementos = 45` **y** `grupos = 6`**. El resultado es** `13.983.816`**, que cabe en un** `long long`**. Sin embargo, el cómputo del factorial se desborda a partir de 20.**

Para resolver este problema, la implementación de la función `Combinatorio` **puede mejorarse simplificando la expresión matemática:**

$$\frac{a!}{b!(a-b)!} = \frac{a(a-1)\cdots(a-b+1)(a-b)!}{b!(a-b)!} = \frac{a(a-1)\cdots(a-b+1)}{b!}$$

Cambiemos la implementación de la función como sigue:

```
long long Combinatorio(int a, int b){
   int numerador, fact_b;

   numerador = 1;
   fact_b = 1;

   for (int i = 1 ; i <= b ; i++){
      fact_b = fact_b*i;
      numerador = numerador * (a - i + 1);
   }

   return numerador / fact_b;
}

int main(){
   No cambia nada
}
```

http://decsai.ugr.es/jccubero/FP/IV_combinatorio.cpp

Al estar aislado el código de una función dentro de ella, podemos cambiar la implementación interna de la función sin que afecte al resto de funciones (siempre que se mantenga inalterable la cabecera de la misma)

Lo visto anteriormente puede generalizarse en el siguiente principio básico en Programación:

Principio de Programación:

Ocultación de información (Information Hiding)

Al usar un componente software, no deberíamos tener que preocuparnos de sus detalles de implementación.

Como caso particular de componente software tenemos las funciones y los datos locales a ellas nos permiten ocultar los detalles de implementación de una función al resto de funciones.

IV.1.3. Parametrización de funciones

IV.1.3.1. ¿Cuántos parámetros pasamos?

¿Qué es mejor: muchos parámetros o pocos? Los justos y necesarios.

Ejemplo. **Faltan parámetros:**

```
.....
long long Factorial(){
   long long fact = 1;  int n;

   cin >> n;

   for (i = 2; i <= n; i++)
      fact = fact * i;

   return fact;
}

int main(){
   long long resultado;

   resultado = Factorial();
   cout  << "Factorial = "  << resultado;
}
```

Al leer el valor de `n` dentro de la función, ya no podemos usarla en otras plataformas y tampoco podemos usarla con enteros arbitrarios (que no provengan de la entrada estándar)

Las funciones que realicen un cómputo, no harán también operaciones de E/S

Ejemplo. Demasiados parámetros.

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

long long Factorial (int i, int n){
   long long fact = 1;

   while (i <= n){
      fact = fact * i;
      i++;
   }

   return fact;
}

int main(){
   int n = 5;
   long long resultado;
   int que_pinta_esta_variable_en_el_main = 1;

   // Resultado correcto:
   resultado = Factorial(que_pinta_esta_variable_en_el_main, n);

   // Resultado incorrecto:
   resultado = Factorial(4, n);
   ......
```

Ejemplo. **Demasiados parámetros. Construya una función para calcular la suma de los divisores de un entero (algoritmo visto en la página 278)**

```
int SumaDivisores (int valor){
   int suma = 0;
   int ultimo_divisor = valor/2;

   for (int divisor = 2; divisor <= ultimo_divisor; divisor++) {
      if (valor % divisor == 0)
         suma = suma + divisor;
   }

   return suma;
}
```

Llamada:

```
resultado = SumaDivisores(10);  // Resultado siempre correcto
```

Una versión con demasiados parámetros podría ser la siguiente:

```
int SumaDivisores (int valor, int ultimo_divisor){
   int suma = 0;

   for (int divisor = 2; divisor <= ultimo_divisor; divisor++) {
      if (valor % divisor == 0)
         suma = suma + divisor;
   }
   return suma;
}
```

Llamada:

```
resultado = SumaDivisores(10, 5);  // Resultado correcto
resultado = SumaDivisores(10, 4);  // Resultado incorrecto
```

Ejemplo. Demasiados parámetros:

Re-escribimos el ejemplo del MCD de la página 384. La función busca un divisor de dos números, empezando por el menor de ellos. ¿Y si la función exige que se calcule dicho mínimo fuera?

```
int MCD(int primero, int segundo, int el_menor_entre_ambos){
   bool mcd_encontrado = false;
   int divisor, mcd;

   if (primero == 0 || segundo == 0)
      mcd = 0;
   else{
      divisor = el_menor_entre_ambos;
      mcd_encontrado = false;

      while (!mcd_encontrado){
         if (primero % divisor == 0 && segundo % divisor == 0)
            mcd_encontrado = true;
         else
            divisor--;
      }
      mcd = divisor;
   }

   return mcd;
}

int main(){
   int un_entero, otro_entero, menor, maximo_comun_divisor;

   cout << "Calcular el MCD de dos enteros.\n\n";
   cout << "Introduzca dos enteros: ";
```

```
   cin >> un_entero;
   cin >> otro_entero;

   if (un_entero < otro_entero)
      menor = un_entero;
   else
      menor = otro_entero;

   // Resultado correcto:
   maximo_comun_divisor = MCD(un_entero, otro_entero, menor);

   // Resultado incorrecto:
   maximo_comun_divisor = MCD(un_entero, otro_entero, 9);
```

Para que la nueva versión de la función `MCD` funcione correctamente, *siempre* debemos calcular el menor, antes de llamar a la función. Si no lo hacemos bien, la función no realizará correctamente sus cómputos:

```
resultado = MCD(4, 8, 4);  // Resultado correcto
resultado = MCD(4, 8, 3);  // Resultado incorrecto
```

Observe que el parámetro `el_menor_entre_ambos` está completamente determinado por los valores de los otros dos parámetros. Debemos evitar este tipo de dependencias que normalmente indican la existencia de otra función que los relaciona.

Por tanto, nos quedamos con la solución de la página 384, en la que se calcula el menor dentro de la función.

Si se prevé su reutilización en otros sitios, el cómputo del menor puede definirse en una función aparte:

```
int Minimo(int un_entero, int otro_entero){
   if (un_entero < otro_entero)
      return un_entero;
   else
      return otro_entero;
}

int MCD(int primero, int segundo){
   bool mcd_encontrado = false;
   int divisor, mcd;

   if (primero == 0 || segundo == 0)
      mcd = 0;
   else{
      divisor = Minimo(primero, segundo);
      mcd_encontrado = false;

      while (!mcd_encontrado){
         if (primero % divisor == 0 && segundo % divisor == 0)
            mcd_encontrado = true;
         else
            divisor--;
      }
      mcd = divisor;
   }

   return mcd;
}

int main(){
```

```
    int un_entero, otro_entero, maximo_comun_divisor;

    cout << "Calcular el MCD de dos enteros.\n\n";
    cout << "Introduzca dos enteros: ";
    cin >> un_entero;
    cin >> otro_entero;

    maximo_comun_divisor = MCD(un_entero, otro_entero);

    cout << "\nEl máximo común divisor de " << un_entero
         << " y " << otro_entero << " es: " << maximo_comun_divisor;
}
```

http://decsai.ugr.es/jccubero/FP/IV_mcd_funcion.cpp

Los ejemplos anteriores ponen de manifiesto lo siguiente:

Todos los datos auxiliares que la función necesite para realizar sus cómputos serán datos locales.

Si el correcto funcionamiento de una función depende de la realización previa de una serie de instrucciones, éstas deben ir dentro de la función.

Los parámetros actuales deben poder variar de forma independiente unos de otros.

IV.1.3.2. ¿Qué parámetros pasamos?

Los parámetros nos permiten ejecutar el código de las funciones con distintos valores:

```
bool es_par;
int entero, factorial;
......
es_par    = EsPar(3);
es_par    = EsPar(entero);
factorial = Factorial(5);
factorial = Factorial(4);
factorial = Factorial(entero);
```

En estos ejemplos era evidente los parámetros que teníamos que pasar. En casos más complejos, no será así.

A la hora de establecer cuáles han de ser los parámetros de una función, debemos determinar:

- ▷ **Qué puede cambiar de una llamada a otra de la función.**
- ▷ **Qué factores influyen en la tarea que la función resuelve.**

Ejemplo. **Retomemos el ejemplo** `ImprimeLineas` **de la página 395.**

```
void ImprimeLineas (int num_lineas){
   for (int i = 1; i <= num_lineas ; i++)
      cout << "\n**********";
}
void Presentacion(int tope_lineas){
   ImprimeLineas (tope_lineas);
   cout << "Programa básico de Trigonometría";
   ImprimeLineas (tope_lineas);
}
```

`ImprimeLineas` **siempre imprime 12 asteriscos. ¿Y si también queremos que sea variable el número de asteriscos? Basta añadir un parámetro:**

```
void ImprimeLineas (int num_lineas, int numero_asteriscos){
   for (int i = 1; i <= num_lineas ; i++){
      cout << "\n";
      for (int j = 1; j <= numero_asteriscos; j++)
         cout << "*";
   }
}
```

Ahora podemos llamar a `ImprimeLineas` **con cualquier número de asteriscos. Por ejemplo, desde** `Presentacion`**:**

```
void Presentacion(int tope_lineas){
   ImprimeLineas (tope_lineas, 12);
   cout << "Programa básico de Trigonometría";
   ImprimeLineas (tope_lineas, 12);
}
```

En este caso, `Presentacion` **siempre imprimirá líneas con 12 asteriscos. Si también queremos permitir que cambie este valor, basta con incluirlo como parámetro:**

```
void Presentacion(int tope_lineas, int numero_asteriscos){
   ImprimeLineas (tope_lineas, numero_asteriscos);
   cout << "Programa básico de Trigonometría";
   ImprimeLineas (tope_lineas, numero_asteriscos);
}
```

Es posible que deseemos aislar la impresión de una línea de asteriscos en una función, para así poder reutilizarla en otros contextos. Nos quedaría:

```
void ImprimeAsteriscos (int num_asteriscos){
   for (int i = 1 ; i <= num_asteriscos ; i++)
      cout << "*";
}
void ImprimeLineas (int num_lineas, int num_asteriscos){
   for (int i = 1; i <= num_lineas ; i++){
      cout << "\n";
      ImprimeAsteriscos(num_asteriscos);
   }
}
void Presentacion(int tope_lineas, int num_asteriscos){
   ImprimeLineas (tope_lineas, num_asteriscos);
   cout << "Programa básico de Trigonometría";
   ImprimeLineas (tope_lineas, num_asteriscos);
}
```

Finalmente, si queremos poder cambiar el título del mensaje, basta pasarlo como parámetro:

```
void Presentacion(string mensaje, int tope_lineas, int num_asteriscos){
   ImprimeLineas (tope_lineas, num_asteriscos);
   cout << mensaje;
   ImprimeLineas (tope_lineas, num_asteriscos);
}
```

El programa principal quedaría así:

```
int main(){
   double lado1, lado2, hipotenusa;

   Presentacion("Programa básico de Trigonometría", 3, 32);

   cout << "\n\nIntroduzca los lados del triángulo rectángulo: ";
   cin >> lado1;
   cin >> lado2;

   hipotenusa = Hipotenusa(lado1, lado2);

   cout << "\nLa hipotenusa vale " << hipotenusa;
}
```

http://decsai.ugr.es/jccubero/FP/IV_hipotenusa_presentacion.cpp

Los parámetros nos permiten aumentar la flexibilidad en el uso de la función.

Ejemplo. **Retomemos el ejemplo de la altura del tema II. Si el criterio es el de la página 152, una persona es alta si mide más de 190 cm:**

```
bool EsAlta (int altura){
   return altura >= 190;
}
```

Pero si el criterio es el de la página 160, para determinar si una persona es alta también influye la edad, por lo que debemos incluirla como parámetro:

```
bool EsMayorEdad (int edad){
   return  edad >= 18;
}

bool EsAlta (int altura, int edad){
   if (EsMayorEdad(edad))
      return altura >= 190;
   else
      return altura >= 175;
}

int main(){
   int edad, altura;
   bool es_alta, es_mayor_edad;

   cout << "Mayoría de edad y altura.\n\n";
   cout << "Introduzca los valores de edad y altura: ";
   cin >> edad;
   cin >> altura;

   es_mayor_edad = EsMayorEdad(edad);
   es_alta = EsAlta(altura, edad);
```

```
    if (es_mayor_edad)
       cout << "\nEs mayor de edad";
    else
       cout << "\nEs menor de edad";

    if (es_alta)
       cout << "\nEs una persona alta";
    else
       cout << "\nNo es una persona alta";
}
```

Al diseñar la cabecera de una función, el programador debe analizar detalladamente cuáles son los factores que influyen en la tarea que ésta resuelve y así determinar los parámetros apropiados.

Si la función es muy simple, podemos usar los literales `190, 175`. Pero si es más compleja, seguimos el mismo consejo que vimos en el tema I y usamos constantes tal y como hicimos en la página 201:

```
bool EsMayorEdad (int edad){
   const int MAYORIA_EDAD = 18;

   return  edad >= MAYORIA_EDAD;
}

bool EsAlta (int altura, int edad){
   const int UMBRAL_ALTURA_JOVENES = 175,
             UMBRAL_ALTURA_ADULTOS = 190;
   int umbral_altura;

   if (EsMayorEdad(edad))
      umbral_altura = UMBRAL_ALTURA_ADULTOS;
   else
      umbral_altura = UMBRAL_ALTURA_JOVENES;

   return altura >= umbral_altura;
}

int main(){
   <No cambia nada>
}
```

http://decsai.ugr.es/jccubero/FP/IV_altura.cpp

Nota:

Si las constantes se utilizasen en otros sitios del programa, podrían ponerse como constantes globales

---

IV.1.4. Programando como profesionales

IV.1.4.1. Cuestión de estilo

¿Podemos incluir una sentencia `return` en cualquier sitio de la función?. Poder, se puede, pero no es una buena idea.

Ejemplo. **¿Qué ocurre en el siguiente código?**

```
long long Factorial (int n)  {
   int i;
   long long fact;

   fact = 1;
   for (i = 2; i <= n; i++)  {
      fact = fact * i;

      return fact;
   }
}
```

Si $n \geq 2$ se entra en el bucle, pero la función termina cuando se ejecuta la primera iteración, por lo que devuelve 2. Y si $n < 2$, no se ejecuta ningún `return` y por tanto se produce un *comportamiento indeterminado (undefined bahaviour)* .

Ejemplo. **Considere la siguiente implementación de la función `EsPrimo`:**

```
bool EsPrimo(int valor){
   int divisor;

   for (divisor = 2 ; divisor < valor; divisor++)
      if (valor % divisor == 0)
         return false;

   return true;
}
```

Sólo ejecuta `return true` cuando no entra en el condicional del bucle, es decir, cuando no encuentra ningún divisor (y por tanto el número es primo) Por lo tanto, la función se ejecuta correctamente pero el código es difícil de entender.

En vez de incluir un `return` dentro del bucle para salirnos de él, usamos, como ya habíamos hecho en este ejemplo, una variable `bool`. Ahora, viendo la cabecera del bucle, vemos todas las condiciones que controlan el bucle:

```
bool EsPrimo(int valor){
   int divisor;
   bool es_primo;

   es_primo = true;

   for (divisor = 2 ; divisor < valor && es_primo; divisor++)
      if (valor % divisor == 0)
         es_primo = false;

   return es_primo;
}
```

Así pues, debemos evitar construir funciones con varias sentencias

`return` *perdidas* dentro del código. En el caso de funciones con muy pocas líneas de código y siempre que el código quede claro, sí podríamos aceptarlo:

```
bool EsPar (int n){
    if (n % 2 == 0)
       return true;
    else
       return false;
}
```

***Consejo*:**

Salvo en el caso de funciones con muy pocas líneas de código, evite la inclusión de sentencias `return` ***perdidas dentro del código de la función. Use mejor un único*** `return` ***al final de la función.***

IV.1.4.2. Precondiciones

Una *precondición (precondition)* de una función es toda aquella restricción que deben satisfacer los parámetros para que la función pueda ejecutarse sin problemas.

```
long long Factorial (int n){
   int i;
   long long fact = 1;

   for (i = 2; i <= n; i++)
      fact = fact * i;

   return fact;
}
```

Si pasamos como parámetro a `n` un valor mayor de 20, se produce un desbordamiento aritmético. Indicamos esta precondición como un comentario antes de la función.

```
// Prec: 0 <= n <= 20
long long Factorial (int n){
   ......
}

// Prec: 0 <= a, b <= 20
long long Combinatorio(int a, int b){
   ......
}
```

¿Debemos comprobar dentro de la función si se satisfacen sus precondiciones? Y si lo hacemos, ¿qué acción debemos realizar en caso de que no se cumplan? Contestaremos a estas preguntas en la página 525

IV.1.4.3. Documentación de una función

Hay dos tipos de comentarios:

- ***Descripción del algoritmo que implementa la función***
  - ▷ **Describen cómo se resuelve la tarea encomendada a la función.**
  - ▷ **Se incluyen dentro del código de la función**
  - ▷ **Sólo describimos la esencia del algoritmo (tema II)**

Ejemplo. **El mayor de tres números**

```
int Max3 (int a, int b, int c){
   int max;
   /* Calcular el máximo entre a y b -> max
      Calcular el máximo entre max y c      */

   if (a > b)
      max = a;
   else
      max = b;

   if (c > max)
      max = c;

   return max;
}
```

En el examen es imperativo incluir la descripción del algoritmo.

▶ *Descripción de la cabecera de la función*

- ▷ **Describen qué tarea resuelve la función.**
- ▷ **También describen los parámetros (cuando no sea obvio).**
- ▷ **Se incluyen fuera, justo antes de la cabecera de la función**

```
// Calcula el máximo de tres enteros

int Max3 (int a, int b, int c){
   int max;
   /* Calcular el máximo entre a y b
      Calcular el máximo entre max y c
   */

   if (a > b)
      max = a;
   else
      max = b;

   if (c > max)
      max = c;

   return max;
}
```

Ejercicio. **Cambiar la implementación de a función `Max3` para que llame a una función `Max` que calcule el máximo de dos enteros.**

Incluiremos:

- **Una descripción breve del cometido de la función.**

> ***Consejo*:** ***Si no podemos resumir el cometido de una función en un par de líneas a lo sumo, entonces la función es demasiado compleja y posiblemente debería dividirse en varias funciones.***

- **Una descripción de lo que representan los parámetros (salvo que el significado sea obvio) También incluiremos las precondiciones de la función.**

```
// Combinatorio de dos números
// Prec: 0 <= a, b <= 20
long long Combinatorio(int a, int b){
   return Factorial(a)/(Factorial(b) * Factorial(a-b));
}
```

Ampliación:

Consulte el capítulo 19 del libro Code Complete de Steve McConnell sobre normas para escribir comentarios claros y fáciles de mantener.

IV.2. Clases

Hemos visto cómo las funciones introducen mecanismos para cumplir dos principios básicos:

- **Principio de una una única vez.**

  Identificando tareas y empaquetando las instrucciones que la resuelven.

- **Principio de ocultación de información.**

  A través de los datos locales.

En general, los lenguajes de programación proporcionan mecanismos de *modularización (modularization)* de código, para así poder cumplir dichos principios básicos.

En la siguiente sección introducimos las *clases*. Proporcionan un mecanismo complementario que permite cumplir los principios de programación:

- **Principio de una una única vez.**

  Identificando objetos y encapsulando en ellos *datos* y *funciones*.

- **Principio de ocultación de información.**

  Definiendo nuevas reglas de ámbito para los datos.

IV.2.1. Motivación. Clases y Objetos

Criterios que se han ido incorporando a lo largo del tiempo en los estándares de la programación:

Programación estructurada (Structured programming) **. Metodología de programación en la que la construcción de un algoritmo se basa en el uso de las estructuras de control vistas en el tema II, prohibiendo, entre otras cosas, el uso de estructuras de saltos arbitrarios del tipo** `goto`**.**

Programación modular (Modular programming) **: Cualquier metodología de programación que permita agrupar conjuntos de sentencias en *módulos* o *paquetes*.**

Programación procedural (Procedural programming) **. Metodología de programación modular en la que los módulos son las funciones.**

Las funciones pueden usarse desde cualquier sitio tras su declaración (puede decirse que son *funciones globales (global functions)*) y se comunican con el resto del programa a través de sus parámetros y del resultado que devuelven (siempre que no se usen variables globales)

La base del diseño de una solución a un problema usando programación procedural consiste en analizar los procesos o tareas que ocurren en el problema e implementarlos usando funciones.

***Nota*. Lamentablemente, no hay un estándar en la nomenclatura usada. Muchos libros llaman programación modular a la programación con funciones. Nosotros usamos aquí el término modular en un sentido genérico. Otros libros llaman programación estructurada a lo que nosotros denominamos programación procedural.**

Nota*. No se usa el término *programación funcional (functional programming)* para referirse a la programación con funciones, ya que se acuñó para otro tipo de programación, a saber, un tipo de *programación declarativa (declarative programming)

Programación orientada a objetos (Object oriented programming) **(PDO). Metodología de programación modular en la que los módulos o paquetes se denominan *objetos (objects)* .**

La base del diseño de una solución a un problema usando PDO consiste en analizar las entidades que intervienen en el problema e implementarlas usando objetos.

Un objeto aglutina en un único paquete datos y funciones. Las funciones incluidas en un objeto se denominan *métodos (methods)* . Los datos representan las características de una entidad y los métodos determinan su *comportamiento (behaviour)* .

Objeto: una ventana en Windows

- **Datos: posición esquina superior izquierda, altura, anchura, está_minimizada**
- **Métodos: Minimiza(), Maximiza(), Agranda(int tanto_por_ciento), etc.**

Objeto: una cuenta bancaria

- **Datos: identificador de la cuenta, saldo actual, descubierto que se permite**
- **Métodos: Ingresa(double cantidad), Retira(double cantidad), etc.**

Objeto: un triángulo rectángulo

- **Datos: los tres puntos que lo determinan A, B, C**
- **Métodos: ConstruyeHipotenusa(), ConstruyeSegmentoAB(), etc.**

Objeto: una fracción

- **Datos: Numerador y denominador**
- **Métodos: Simplifica(), Súmale(Fracción otra_fracción), etc.**

Objeto: una calculadora de nómina

- **Datos: salario base**
- **Métodos: AplicaSubidaSalarial(int edad, int num_hijos)**

Nota:

Observe que un objeto es un dato compuesto e incluye otros datos y métodos.

Para construir un objeto, primero tenemos que definir su estructura. Esto se hace con el concepto de clase.

Una *clase (class)* es un *tipo de dato* definido por el programador. Se usa para representar una entidad.

Con la clase especificamos las características comunes y el comportamiento de una entidad. Es como un patrón o molde a partir del cual construimos los objetos.

Un *objeto (object)* es un *dato* cuyo tipo de dato es una clase. También se dirá que un objeto es la *instancia (instance)* de una clase.

```
class MiClase{
   ......
};
int main(){
   MiClase un_objeto_instancia_de_MiClase;
   MiClase otro_objeto_instancia_de_MiClase;
   .......
```

```
class CuentaBancaria{              // <- clase
   ......
};
int main(){
   CuentaBancaria una_cuenta;      // <- un objeto
   CuentaBancaria otra_cuenta;     // <- otro objeto
   ......
```

Los objetos `una_cuenta` y `otra_cuenta` existirán mientras esté ejecutándose `main`.

IV.2.2. Encapsulación

- **En Programación Procedural, la modularización se materializa al incluir en el mismo componente software (la función) un conjunto de instrucciones.**
- **En PDO, la modularización se materializa al incluir en el mismo componente software (la clase) los datos y los métodos (funciones que actúan sobre dichos datos). Este tipo de modularización se conoce como *encapsulación (encapsulation)***

> ***La encapsulación es el mecanismo de modularización utilizado en PDO. La idea consiste en aunar datos y comportamiento en un mismo módulo.***

Una clase se compone de:

- ***Datos miembro (data member)* :**

  Son las características que definen una entidad.

  Todos los objetos que pertenecen a una misma clase tienen la misma estructura, pero cada objeto tiene un espacio en memoria distinto y por tanto unos valores propios para cada dato miembro. Diremos que el conjunto de valores específicos de los datos miembro en un momento determinado de un objeto conforman el *estado (state)* de dicho objeto.

- ***Funciones miembro (member functions)* o *métodos (methods)* :**

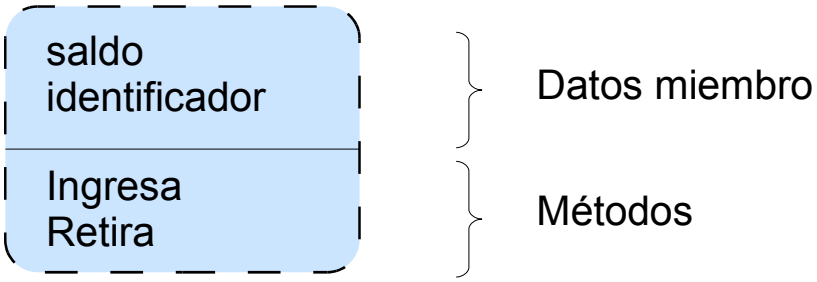
  Son funciones definidas dentro de la clase.

  Determinan el *comportamiento (behaviour)* de la entidad, es decir, el conjunto de operaciones que se pueden realizar sobre los objetos de la clase.

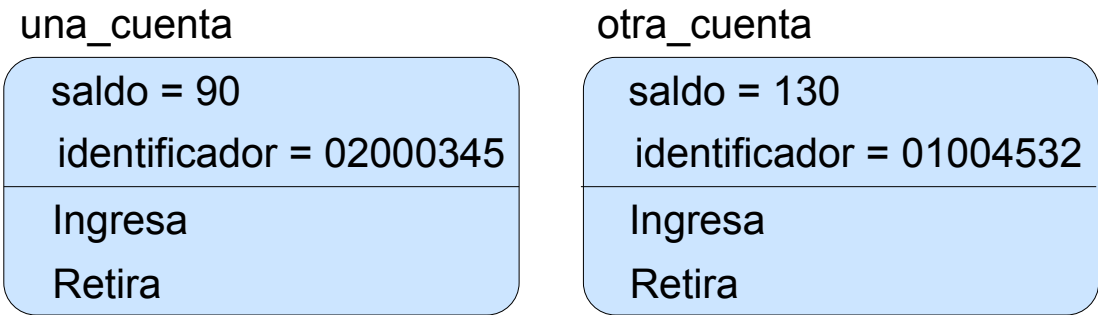
  La definición de los métodos es la misma para todos los objetos.

```
class CuentaBancaria{                  // <- clase
   ......
};
int main(){
   CuentaBancaria una_cuenta;          // <- un objeto
   CuentaBancaria otra_cuenta;         // <- otro objeto
   ......
```

Clase CuentaBancaria:



Objetos instancia de CuentaBancaria:



IV.2.2.1. Datos miembro

Para declarar un dato miembro dentro de una clase, hay que especificar su *ámbito (scope)* , que podrá ser público o privado. Esto se indica con el *especificador de acceso (access specifier)* `public` o `private`.

Empezamos viendo el ámbito público.

Todas las declaraciones incluidas después de `public:` son públicas, es decir, accesibles desde fuera del objeto. Desde el `main` o desde otros objetos, accederemos a los datos públicos de los objetos a través del nombre del objeto, un punto y el nombre del dato.

Cuando se crea un objeto, los datos miembro, como cualquier otro dato, contendrán un valor indeterminado.

```
class MiClase{
public:
   int dato;
};

int main(){
   MiClase un_objeto;     // un_objeto.dato   contiene   ?
   MiClase otro_objeto;   // otro_objeto.dato contiene   ?

   un_objeto.dato = 4;
   cout << un_objeto.dato;     // Imprime 4
   un_objeto.dato = 8;
   cout << un_objeto.dato;     // Imprime 8
   otro_objeto.dato = 7;
   cout << otro_objeto.dato;   // Imprime 7
```

Cada vez que modificamos un dato miembro, diremos que se ha modificado el estado del objeto.

Ejemplo. Cuenta bancaria.

Nota. Esta clase tiene problemas importantes de diseño que se irán arreglando a lo largo de este tema.

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

class CuentaBancaria{
public:
   double saldo;
   string identificador;
};

int main(){
   CuentaBancaria cuenta;           // ?, ?
   string identificador_cuenta;     // ""    <- Ver nota siguiente página
   double ingreso;                  // ?
   double retirada;                 // ?

   cout << "\nIntroduce identificador a asignar a la cuenta:";
   cin >> identificador_cuenta;
   cuenta.identificador = identificador_cuenta;

   cout << "\nIntroduce cantidad inicial a ingresar: ";
   cin >> ingreso;

   cuenta.saldo = ingreso;

   cout << "\nIntroduce cantidad a ingresar: ";
   cin >> ingreso;

   cuenta.saldo = cuenta.saldo + ingreso;
```


462

```
cout << "\nIntroduce cantidad a retirar: ";
cin >> retirada;

cuenta.saldo = cuenta.saldo - retirada;


CuentaBancaria otra_cuenta;  // Otro objeto de la clase CuentaBancaria

otra_cuenta.saldo = 30000;   // <- No afecta al otro objeto cuenta

saldo = 300;                 // Error de compilación.
                             // saldo no es un dato definido en main.
```

Algunas aclaraciones:

- **C++ inicializa siempre cualquier `string` a `""`.**
- **En vez de poner**

  ```
  cin >> identificador_cuenta;
  cuenta.identificador = identificador_cuenta;
  ```

  podríamos haber puesto directamente:

  ```
  cin >> cuenta.identificador;
  ```

- **Con las herramientas que conocemos, no puede leerse directamente un objeto por completo:**

  ```
  CuentaBancaria cuenta;


  cin >> cuenta;  // Error de compilación
  ```

Cuando se crea un objeto, los datos miembro no tienen ningún valor asignado por defecto, a no ser que se inicialicen dentro de la clase. En este caso, al crear el objeto, se le asignarán automáticamente los valores especificados en la inicialización.

Desde C++11, la inicialización puede realizarse en la definición del dato miembro:

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

class CuentaBancaria{
public:
   double saldo         = 0.0;
   string identificador = "";
   // Realmente, C++ ya inicializa los string a ""
};

int main(){
   CuentaBancaria cuenta;  // "", 0  Aquí se aplican las inicializaciones
                           //        al objeto cuenta

   cout << cuenta.saldo << "\n";   // 0
   cout << cuenta.identificador;   // ""
```


462

Las inicializaciones especificadas en la declaración de los datos miembro dentro de la clase, se aplican en el momento que se crea un objeto cualquiera de dicha clase.

IV.2.2.2. Métodos

Por ahora, hemos conseguido crear varios objetos (cuentas bancarias) con sus propios datos miembros (saldo, identificador). Ahora vamos a ejecutar métodos sobre dichos objetos.

Los métodos determinan el comportamiento de los objetos de la clase. Son como funciones definidas dentro de la clase.

- **El comportamiento de los métodos es el mismo para todos los objetos.**

  Al igual que ocurría con los datos miembro, podrán ser públicos o privados. Por ahora sólo consideramos métodos públicos.

  Desde el `main` o desde otros objetos, accederemos a los métodos públicos de los objetos a través del nombre del objeto, un punto, el nombre del método y entre paréntesis los parámetros (en su caso).

```
class MiClase{
public:
   int dato;

   // dato = sqrt(5);  Error de compilación. Las sentencias
   //                  deben estar dentro de los métodos
   void UnMetodo(){
      ......
   }
};
int main(){
   MiClase un_objeto;

   un_objeto.dato = 4;
   un_objeto.UnMetodo();
   ......
}
```

- **No existe un consenso entre los distintos lenguajes de programación, a la hora de determinar el tipo de letra usado para las clases, objetos, métodos, etc. Nosotros seguiremos básicamente el de Google (Buscar en Internet *Google coding style*):**
  - **Tanto los identificadores de las clases como de los métodos empezarán con una letra mayúscula.**

    Si el nombre es compuesto usaremos la primera letra de la palabra en mayúscula: `CuentaBancaria`
  - **Los identificadores de los objetos, como cualquier otro dato, empezarán con minúscula.**

    Si el nombre es compuesto usaremos el símbolo de subrayado para separar los nombres: `mi_cuenta_bancaria`

  Usaremos nombres para denotar las clases y verbos para los métodos de tipo* `void`*. Normalmente no usaremos infinitivos sino la tercera persona del singular.

  Para los métodos que devuelven un valor, usaremos nombres (el del valor que devuelve).

- **Los métodos pueden modificar el estado del objeto sobre el que actúan, es decir, pueden modificar los datos miembro. Acceden a ellos directamente.**

Ejemplo. **Añadimos métodos para ingresar y sacar dinero en la cuenta bancaria:**


462

```
class CuentaBancaria{                  // <- clase
public:
   double saldo          = 0.0;
   string identificador = "";
   // Realmente, C++ ya inicializa los string a ""

   void Ingresa(double cantidad){   // Dentro del método accedemos
      saldo = saldo + cantidad;     // directamente al dato miembro
   }
   void Retira(double cantidad){
      saldo = saldo - cantidad;
   }
};
int main(){
   CuentaBancaria una_cuenta;     // "", 0   un objeto
   CuentaBancaria otra_cuenta;    // "", 0   otro objeto

   una_cuenta.identificador = "20310381450100006529";   // "2...9", 0
   una_cuenta.Ingresa(25);                              // "2...9", 25
   una_cuenta.Retira(10);                               // "2...9", 15
   ......
   otra_cuenta.identificador = "20310381450100007518";  // "2...8", 0
   otra_cuenta.Ingresa(45);                             // "2...8", 45
   otra_cuenta.Retira(15);                              // "2...8", 30
   ......
```

Nota:

Observe que en la llamada `una_cuenta.Ingresa(25)` estamos pasando como parámetro un `int` a un `double`. Al igual que ocurría con las funciones, el paso de parámetros sigue las mismas directrices por lo que se producirá un casting automático y la correspondencia entre el parámetro actual y el formal se hará correctamente.

A destacar:

▷ **Los métodos `Ingresa` y `Retira` acceden al dato miembro `saldo` por su nombre.**

Los métodos acceden a los datos miembro directamente.

▷ **Los métodos `Ingresa` y `Retira` modifican alguno de los datos miembro.**

Los métodos pueden modificar el estado del objeto.

Ejercicio. **Definamos la clase** `SegmentoDirigido`

```
class SegmentoDirigido{
    public:
       double x_1, y_1, x_2, y_2;
};

int main(){
   SegmentoDirigido un_segmento;  // ?, ?, ?, ?

   un_segmento.x_1 = 3.4;
   un_segmento.y_1 = 5.6;
   un_segmento.x_2 = 4.5;
   un_segmento.y_2 = 2.3;

   cout << un_segmento.x_1;  //  3.4
   cout << un_segmento.x_2;  //  4.5
}
```


464

Recuerde que se puede especificar la inicialización de los datos miembro en la definición de la clase:

```
class SegmentoDirigido{
    public:
       double x_1 = 0.0,
              y_1 = 0.0,
              x_2 = 0.0,
              y_2 = 0.0;
};
int main(){
   SegmentoDirigido un_segmento, otro_segmento;

   cout << un_segmento.x_1 << " " << otro_segmento.x_1;  //  0.0 0.0
```


464

Ejercicio. **Defina sendos métodos `TrasladaHorizontal` y `TrasladaVertical` para trasladar un segmento un número de unidades.**


464

```
class SegmentoDirigido{
public:
   double x_1 = 0.0,
          y_1 = 0.0,
          x_2 = 0.0,
          y_2 = 0.0;

   void TrasladaHorizontal(double unidades){
      x_1 = x_1 + unidades;
      x_2 = x_2 + unidades;
   }
   void TrasladaVertical(double unidades){
      y_1 = y_1 + unidades;
      y_2 = y_2 + unidades;
   }
};
int main(){
   SegmentoDirigido un_segmento;

   un_segmento.x_1 = 3.4;
   un_segmento.y_1 = 5.6;
   un_segmento.x_2 = 4.5;
   un_segmento.y_2 = 2.3;

   un_segmento.TrasladaHorizontal(10);

   cout << un_segmento.x_1 << "," << un_segmento.y_1;      //  13.4,5.6
   cout << un_segmento.x_2 << "," << un_segmento.y_2;      //  14.5,2.3
}
```

Ejercicio. Calcule la longitud de un segmento dirigido.


464

```
class SegmentoDirigido{
public:
   double x_1 = 0.0,
          y_1 = 0.0,
          x_2 = 0.0,
          y_2 = 0.0;
   double Longitud(){
      double resta_abscisas = x_2 - x_1;
      double resta_ordenadas = y_2 - y_1;

      return sqrt(resta_abscisas * resta_abscisas +
                  resta_ordenadas * resta_ordenadas);
   }
   void TrasladaHorizontal(double unidades){
      x_1 = x_1 + unidades;
      x_2 = x_2 + unidades;
   }
   void TrasladaVertical(double unidades){
      y_1 = y_1 + unidades;
      y_2 = y_2 + unidades;
   }
};
int main(){
   SegmentoDirigido un_segmento;

   un_segmento.x_1 = 3.4;
   un_segmento.y_1 = 5.6;
   un_segmento.x_2 = 4.5;
   un_segmento.y_2 = 2.3;

   cout << "\nLongitud del segmento = " << un_segmento.Longitud();
```

IV.2.2.3. Controlando el acceso a los datos miembro

Ya vimos que si al llamar a una función siempre debemos realizar una serie de instrucciones, éstas deberían ir dentro de la función (página 415) Con los métodos pasa lo mismo.

Ejemplo. **En el ejemplo de la cuenta bancaria, ¿cómo implementamos una restricción real como que, por ejemplo, los ingresos sólo puedan ser positivos y las retiradas de fondos inferiores al saldo? ¿Lo comprobamos en el `main` antes de llamar a los métodos?**


462

```
class CuentaBancaria{
public:
   double saldo         = 0.0;
   string identificador = "";
   // Realmente, C++ ya inicializa los string a ""

   void Ingresa(double cantidad){
      saldo = saldo + cantidad;
   }
   void Retira(double cantidad){
      saldo = saldo - cantidad;
   }
};
int main(){
   CuentaBancaria cuenta;
   double ingreso, retirada;

   cuenta.identificador = "20310381450100006529";    // "2...9", 0

   cin >> ingreso;

   if (ingreso > 0)
      cuenta.Ingresa(ingreso);
```

```
cin >> retirada;

if (retirada > 0 && retirada <= cuenta.saldo)
   cuenta.Retira(retirada);
```

Cada vez **que realicemos un ingreso o retirada de fondos, habría que realizar las anteriores comprobaciones, por lo que es propenso a errores, ya que seguramente, alguna vez no lo haremos.**

Solución: Lo programamos dentro del método correspondiente. Siempre que se ejecute el método se realizará la comprobación automáticamente.

> ***Si la llamada a un método conlleva que siempre se realicen unas acciones previas, dichas acciones deben ir programadas dentro del método.***


```
class CuentaBancaria{
public:
   double saldo          = 0.0;
   string identificador = "";
   // Realmente, C++ ya inicializa los string a ""

   void Ingresa(double cantidad){
      if (cantidad > 0)
         saldo = saldo + cantidad;
   }

   void Retira(double cantidad){
      if (cantidad > 0 && cantidad <= saldo)
         saldo = saldo - cantidad;
   }
};
int main(){
   CuentaBancaria cuenta; // "", 0

   cuenta.identificador = "20310381450100006529";  // "2...9", 0
   cuenta.Ingresa(-3000); // "2...9", 0
   cuenta.Ingresa(25);    // "2...9", 25
   cuenta.Retira(-10);    // "2...9", 25
   cuenta.Retira(10);     // "2...9", 15
   cuenta.Retira(100);    // "2...9", 15
```

Los métodos permiten establecer la política de acceso a los datos miembro, es decir, determinar cuáles son las operaciones permitidas con ellos.

IV.2.2.4. Llamadas entre métodos dentro del propio objeto

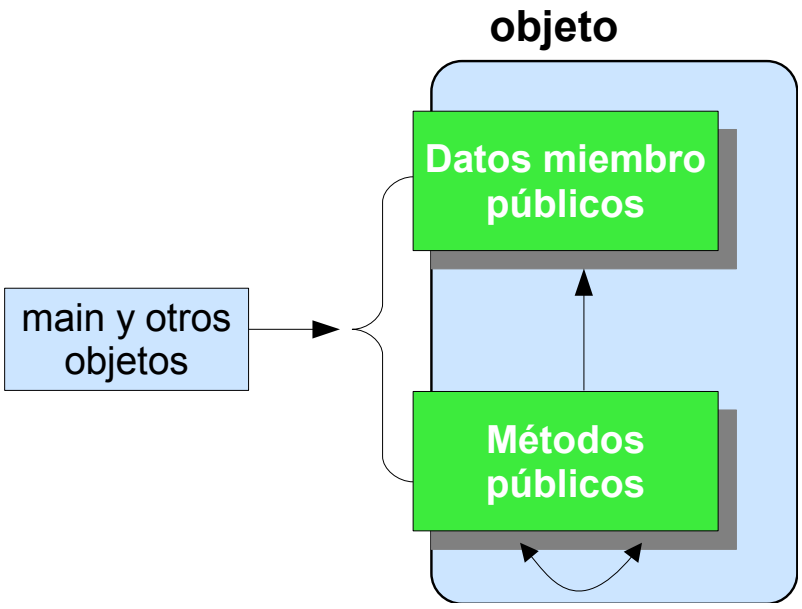
Ya sabemos cómo ejecutar los métodos sobre un objeto:

```
objeto.UnMetodo(...);
```

¿Pero pueden llamarse dentro del objeto unos métodos a otros? Si.

Todos los métodos de un objeto pueden llamar a los métodos del mismo objeto. La llamada se especifica indicando el nombre del método. No hay que anteponer el nombre de ningún objeto ya que en la definición de la clase no tenemos ningún objeto destacado.

```
class MiClase{
public:
   void UnMetodo(parámetros formales){
   }
   void OtroMetodo(...){
      UnMetodo(parámetros actuales);
   }
};
```



Ejemplo. **Implemente el método `Traslada` para que traslade el segmento tanto en horizontal como en vertical.**


464

```
class SegmentoDirigido{
public:
   double x_1 = 0.0,
          y_1 = 0.0,
          x_2 = 0.0,
          y_2 = 0.0;

   void TrasladaHorizontal(double unidades){
      x_1 = x_1 + unidades;
      x_2 = x_2 + unidades;
   }
   void TrasladaVertical(double unidades){
      y_1 = y_1 + unidades;
      y_2 = y_2 + unidades;
   }

   // Traslada en Horizontal y en Vertical

   void TrasladaRepitiendoCodigo (double und_horiz, double und_vert){
      x_1 = x_1 + und_horiz;
      x_2 = x_2 + und_horiz;
      y_1 = y_1 + und_vert;
      y_2 = y_2 + und_vert;
   }

   void Traslada(double und_horiz, double und_vert){
      TrasladaHorizontal(und_horiz);
      TrasladaVertical(und_vert);
   }
   ......
};
```

```
int main(){
   SegmentoDirigido un_segmento;     // 0.0, 0.0, 0.0, 0.0

   un_segmento.x_1 = 3.4;
   un_segmento.y_1 = 5.6;
   un_segmento.x_2 = 4.5;
   un_segmento.y_2 = 2.3;

   cout << "Segmento Dirigido.\n\n";
   cout << "Antes de la traslación:\n";
   cout << un_segmento.x_1 << " , " << un_segmento.y_1;     // 3.4 , 5.6
   cout << "\n";
   cout << un_segmento.x_2 << " , " << un_segmento.y_2;     // 4.5 , 2.3

   un_segmento.Traslada(5.0, 10.0);

   cout << "\n\nDespués de la traslación:\n";
   cout << un_segmento.x_1 << " , " << un_segmento.y_1;     // 8.4 , 15.6
   cout << "\n";
   cout << un_segmento.x_2 << " , " << un_segmento.y_2;     // 9.5 , 12.3
}
```

http://decsai.ugr.es/jccubero/FP/IV_segmento_dirigido_todo_public.cpp

Dentro de la clase, los métodos pueden llamarse unos a otros. Esto nos permite cumplir el principio de una única vez.

Ejercicio. **Aplique un interés porcentual al saldo de la cuenta bancaria.**


462

```
class CuentaBancaria{
public:
   double saldo         = 0;
   string identificador;  // C++ inicializa los string a ""

   void AplicaInteresPorcentualRepitiendoCodigo(int tanto_porciento){
      double cantidad;
      cantidad = saldo * tanto_porciento / 100.0;

      if (cantidad > 0)
         saldo = saldo + cantidad;
   }

   void AplicaInteresPorcentual (int tanto_porciento){
      Ingresa (saldo * tanto_porciento / 100.0);
   }
   void Ingresa(double cantidad){
      if (cantidad > 0)
         saldo = saldo + cantidad;
   }
   void Retira(double cantidad){
      if (cantidad > 0 && cantidad <= saldo)
         saldo = saldo - cantidad;
   }
};
int main(){
   CuentaBancaria cuenta; // "", 0

   cuenta.identificador = "20310381450100006529";  // "2...9", 0
   cuenta.Ingresa(25);                             // "2...9", 25
   cuenta.AplicaInteresPorcentual(3);              // "2...9", 25.75
```

http://decsai.ugr.es/jccubero/FP/IV_cuenta_bancaria_todo_public.cpp

IV.2.3. Ocultación de información

IV.2.3.1. Ámbito público y privado

- **En Programación Procedural, la ocultación de información se consigue con el ámbito local a la función (datos locales y parámetros formales).**
- **En PDO, la ocultación de información se consigue con el ámbito local a los métodos (datos locales y parámetros formales) y además con el ámbito `private` en las clases, tanto en los datos miembro como en los métodos.**

Recuperemos el ejemplo de la cuenta bancaria. Tenemos dos formas de ingresar 25 euros:

```
cuenta.Ingresa(25);
cuenta.saldo = cuenta.saldo + 25;
```

¿Y si hubiésemos puesto -3000?

```
int main()
   CuentaBancaria cuenta; // "", 0

   cuenta.identificador = "20310381450100006529";  // "2...9", 0
   cuenta.Ingresa(-3000); // "2...9", 0
   cuenta.saldo = -3000;  // "2...9", -3000
```


Para poder controlar las operaciones que están permitidas sobre el saldo, siempre deberíamos usar el método `Ingresa` y no acceder directamente al dato miembro `cuenta.saldo`. ¿Cómo lo imponemos? Haciendo que `saldo` sólo sea accesible desde dentro de la clase, es decir, que tenga ámbito privado.

Al declarar los miembros de una clase, se debe indicar su ámbito es decir, desde dónde se van a poder utilizar:

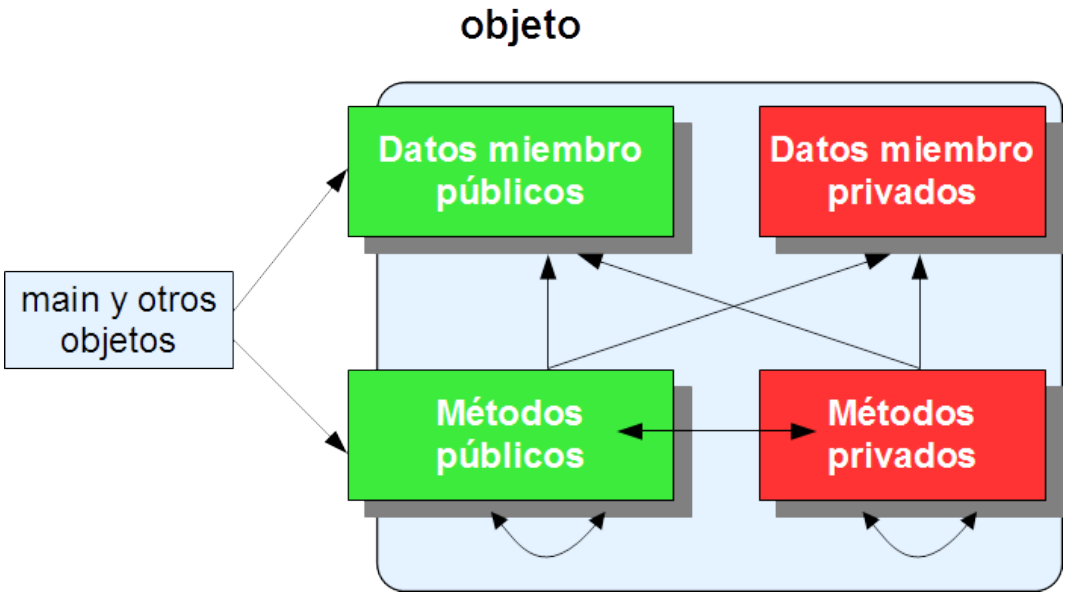
- ***Ámbito público (Public scope)*** **.**
  - **Se indica con el especificador de ámbito** `public`
  - **Los miembros públicos son visibles dentro y fuera del objeto.**
- ***Ámbito privado (Private scope)*** **.**
  - **Se indica con el especificador de ámbito** `private` **(éste es el ámbito por defecto si no se pone nada)**

```
class Clase{
private:
   int dato_privado;
   void MetodoPrivado(){
      ......
   }
public:
   int dato_publico;
   void MetodoPublico(){
      ......
   }
};
```

  - **Los miembros privados sólo se pueden usar desde dentro del objeto. No son accesibles desde fuera.**

```
int main(){
   Clase objeto;

   objeto.dato_publico = 4;
   objeto.dato_privado = 4;    // Error compilación
   objeto.MetodoPrivado();     // Error compilación
```

Los métodos (públicos o privados) del objeto acceden a los métodos y datos miembro (públicos o privados) por su nombre.



IV.2.3.2. UML: Unified modeling language

Para representar las clases se utiliza una notación gráfica con una caja que contiene el nombre de la clase. A continuación se incluye un bloque con los datos miembro y por último un tercer bloque con los métodos. Los datos miembro y métodos públicos se notarán con un símbolo + y los privados con -

La gráfica de la izquierda sería una representación con el estándar *UML* (Unified Modeling Language) *original*. Nosotros usaremos una adaptación como aparece a la derecha, semejante a la sintaxis de C++.

Clase
- dato_privado: int + dato_publico: double
- MetodoPrivado(param: double): int + MetodoPublico(param: int): void

UML *puro*

Clase
- int dato_privado + double dato_publico
- int MetodoPrivado(double param) + void MetodoPublico(int param)

UML *adaptado*

IV.2.3.3. Datos miembro privados

Supongamos que cambiamos el ámbito público de los datos miembro de la clase `Cuenta_Bancaria` a privado.


462

```
class CuentaBancaria{
private:
   double saldo         = 0.0;
   string identificador = "";
   // Realmente, C++ ya inicializa los string a ""
public:
   void Ingresa(double cantidad){
      if (cantidad > 0)
         saldo = saldo + cantidad;
   }
   void Retira(double cantidad){
      if (cantidad > 0 && cantidad <= saldo)
         saldo = saldo - cantidad;
   }
   void AplicaInteresPorcentual(int tanto_porciento){
      Ingresa (saldo * tanto_porciento / 100.0);
   }
};
int main(){
   CuentaBancaria cuenta;
```

Las siguientes sentencias provocan un error de compilación ya que los datos miembro son ahora privados:

```
   cuenta.identificador =  "20310381450100006529";
   cout << cuenta.saldo;
```

¿Cómo le asignamos entonces un valor?

Al trabajar con datos miembro privados debemos añadirle a la clase:

- **Métodos para modificar los datos miembro (aquellos que se deseen modificar desde fuera)**
- **Métodos para obtener el valor actual de los datos miembro (aquellos a los que se desee acceder desde fuera)**


462

```
class CuentaBancaria{
private:
   double saldo         = 0.0;
   string identificador = "";
   // Realmente, C++ ya inicializa los string a ""
public:
   void SetIdentificador(string identificador_cuenta){
      identificador = identificador_cuenta;
   }
   string Identificador(){
      return identificador;
   }
   void SetSaldo(double cantidad){
      if (cantidad > 0)
         saldo = cantidad;
   }
   double Saldo(){
      return saldo;
   }
   void Ingresa(double cantidad){
      if (cantidad > 0)
         saldo = saldo + cantidad;
   }
   void Retira(double cantidad){
      if (cantidad > 0 && cantidad <= saldo)
         saldo = saldo - cantidad;
```

```
   }
   void AplicaInteresPorcentual(int tanto_porciento){
      Ingresa (saldo * tanto_porciento / 100.0);
   }
};
int main(){
   CuentaBancaria cuenta;   // "", 0

   cuenta.SetIdentificador("20310381450100006529");  // "2...9", 0
   cuenta.Ingresa(25);                               // "2...9", 25
   cuenta.SetSaldo(-300);                            // "2...9", 25

   cout << cuenta.Saldo();      // <- Paréntesis, pues es un método
}
```

Parece razonable no incluir un método `SetSaldo` pues los cambios en el saldo deberían hacerse siempre con `Ingresa` y `Retira`. Para ello, basta eliminar `SetSaldo`.

http://decsai.ugr.es/jccubero/FP/IV_cuenta_bancaria_datos_miembro_private.cpp

De nuevo vemos que los métodos permiten establecer la política de acceso a los datos miembro. La clase nos quedaría así:

CuentaBancaria
- double saldo - string identificador
+ void SetIdentificador(string identificador_cuenta) + string Identificador() + double Saldo() + void Ingresa(double cantidad) + void Retira(double cantidad) + void AplicaInteresPorcentual(int tanto_por_ciento)

Salvo casos excepcionales, no definiremos datos miembro públicos. Siempre serán privados.

Esto nos permitirá controlar, desde dentro de la clase, las operaciones que puedan realizarse sobre ellos. Estas operaciones estarán encapsuladas en métodos de la clase.

Desde fuera de la clase, no se tendrá acceso directo a los datos miembro privados. El acceso será a través de los métodos anteriores.

Interfaz (Interface) **de una clase: Es el conjunto de datos y métodos públicos de dicha clase. Como usualmente los datos son privados, el término interfaz suele referirse al conjunto de métodos públicos.**

Ejercicio. **Cambiemos el ámbito (de `public` a `private`) de los datos miembro de la clase `SegmentoDirigido`. Obligamos cambiar los 4 datos simultáneamente.**

SegmentoDirigido
- double x_1 - double y_1 - double x_2 - double y_2
+ void SetCoordenadas (double origen_abscisa, double origen_ordenada, double final_abscisa, double final_ordenada) + double OrigenAbscisa() + double OrigenOrdenada() + double FinalAbscisa() + double FinalOrdenada() + double Longitud() + void TrasladaHorizontal(double unidades) + void TrasladaVertical(double unidades) + void Traslada(double en_horizontal, double en_vertical)

```
class SegmentoDirigido{
private:
   double x_1 = 0.0,
          y_1 = 0.0,
          x_2 = 0.0,
          y_2 = 0.0;
public:
   void SetCoordenadas(double origen_abscisa,
                       double origen_ordenada,
                       double final_abscisa,
                       double final_ordenada){
      if (! (origen_abscisa  == final_abscisa &&
             origen_ordenada == final_ordenada)){
         x_1 = origen_abscisa;
         y_1 = origen_ordenada;
         x_2 = final_abscisa;
         y_2 = final_ordenada;
      }
   }
   double OrigenAbscisa(){
      return x_1;
   }
   double OrigenOrdenada(){
      return y_1;
   }
   double FinalAbscisa(){
      return x_2;
   }
   double FinalOrdenada(){
      return y_2;
   }
   // Los métodos Longitud, Traslada, TrasladaHorizontal
   // y TrasladaVertical no varían
```

```
};
int main(){
   SegmentoDirigido un_segmento;   // 0.0, 0.0, 0.0, 0.0

   un_segmento.SetCoordenadas(3.4, 5.6, 4.5, 2.3);

   cout << "Segmento Dirigido.\n\n";
   cout << "Antes de la traslación:\n";
   cout << un_segmento.OrigenAbscisa() << " , "
        << un_segmento.OrigenOrdenada();          // 3.4 , 5.6
   cout << "\n";
   cout << un_segmento.FinalAbscisa() << " , "
        << un_segmento.FinalOrdenada();           // 4.5 , 2.3

   un_segmento.Traslada(5.0, 10.0);

   cout << "\n\nDespués de la traslación:\n";
   cout << un_segmento.OrigenAbscisa() << " , "
        << un_segmento.OrigenOrdenada();          // 8.4 , 15.6
   cout << "\n";
   cout << un_segmento.FinalAbscisa() << " , "
        << un_segmento.FinalOrdenada();           // 9.5 , 12.3

   SegmentoDirigido otro_segmento;                   // 0.0, 0.0, 0.0, 0.0
   otro_segmento.SetCoordenadas(3.4, 5.6, 3.4, 5.6); // 0.0, 0.0, 0.0, 0.0
   ......
```

Ejercicio. Representemos una circunferencia.

Circunferencia
- double centro_x - double centro_y - double radio
+ void SetCentro(double abscisa, double ordenada) + void SetRadio(double el_radio) + double AbscisaCentro() + double OrdenadaCentro() + double Radio() + double Longitud() + double Area() + void Traslada(double en_horizontal, double en_vertical)

```
const double PI = 6 * asin(0.5);

class Circunferencia{
private:
   double centro_x = 0.0;
   double centro_y = 0.0;
   double radio    = 1.0;   // Circunferencia goniométrica
public:
   void SetCentro(double abscisa, double ordenada){
      centro_x = abscisa;
      centro_y = ordenada;
   }
   void SetRadio(double el_radio){
      radio = el_radio;
   }
   double AbscisaCentro(){
      return centro_x;
   }
   double OrdenadaCentro(){
      return centro_y;
   }
```

```
      double Radio(){
         return radio;
      }
      double Longitud(){
         return 2*PI*radio;
      }
      double Area(){
         return PI*radio*radio;
      }
      void Traslada(double en_horizontal, double en_vertical){
         centro_x = centro_x + en_horizontal;
         centro_y = centro_y + en_vertical;
      }
   };
   int main(){
      Circunferencia mi_aro;                  // 0.0, 0.0, 1.0
      double longitud_mi_aro;

      mi_aro.SetCentro(4.5, 6.7);
      mi_aro.SetRadio(2.1);
      longitud_mi_aro = mi_aro.Longitud();    // 13.19468

      mi_aro.SetRadio(8.3);                   // Cambiamos el radio
      longitud_mi_aro = mi_aro.Longitud();    // 52,15044

      mi_aro.Traslada(10.1, 15.2);            // Traslación
      longitud_mi_aro = mi_aro.Longitud();    // 52,15044
```

http://decsai.ugr.es/jccubero/FP/IV_circunferencia.cpp

Ejemplo. **Defina la clase `Fecha` que representa un día, mes y año. Decidimos definir un único método para cambiar los tres valores a la misma vez, en vez de tres métodos independientes.**

Fecha	
- int	dia
- int	mes
- int	anio
+ void	SetDiaMesAnio (int el_dia, int el_mes, int el_anio)
+ int	Dia()
+ int	Mes()
+ int	Anio()
+ string	ToString()


499

```
class Fecha{
private:
   int dia,
       mes,
       anio;
public:
   void SetDiaMesAnio(int el_dia, int el_mes, int el_anio){
      bool es_bisiesto;
      bool es_fecha_correcta;
      const int anio_inferior = 1900;
      const int anio_superior = 2500;
      const int dias_por_mes[12] =
            {31,28,31,30,31,30,31,31,30,31,30,31};
            // Meses de Enero a Diciembre

      es_fecha_correcta =  1 <= el_dia &&
                           el_dia <= dias_por_mes[el_mes - 1] &&
                           1 <= el_mes   &&   el_mes <= 12 &&
                           anio_inferior <= el_anio   &&
                           el_anio <= anio_superior;
```

```
      es_bisiesto = (el_anio % 4 == 0 && el_anio % 100 != 0) ||
                    el_anio % 400 == 0;

      if (!es_fecha_correcta && el_mes == 2 && el_dia == 29 && es_bisiesto)
         es_fecha_correcta = true;

      if (es_fecha_correcta){
         dia = el_dia;
         mes = el_mes;
         anio = el_anio;
      }
   }
   int Dia(){
      return dia;
   }
   int Mes(){
      return mes;
   }
   int Anio(){
      return anio;
   }
   string ToString(){
      return to_string(dia) + "/" +   // to_string standard en C++11
             to_string(mes) + "/" +
             to_string(anio);

      // También sería correcto llamar a los métodos
      // Dia(), Mes(), Anio()
   }
};

int main(){
   Fecha nacimiento_JC;                        // ?, ?, ?
```

```
Fecha otra_fecha;                           // ?, ?, ?

nacimiento_JC.SetDiaMesAnio(27, 2, 1967);   // 27, 2, 1967
nacimiento_JC.SetDiaMesAnio(-8, 2, 1967);   // 27, 2, 1967

cout << nacimiento_JC.ToString();           // Imprime  27/2/1967

otra_fecha.SetDiaMesAnio(8, -2, 1967);      // ?, ?, ?
```

Nota:

Normalmente, no tendremos variables como **nacimiento_JC** ligadas a una persona en concreto. Lo usual es que tengamos un conjunto de objetos de tipo **Fecha**, asociados a distintas personas según un índice y guardados en un vector de objetos. Esto se verá en el próximo tema.

Nota:

Observe que se ha definido un vector como dato local de un método. Esto es correcto. El vector vivirá mientras se esté ejecutando el método. Consulte la página 558 para más detalle.

IV.2.3.4. Métodos privados

Si tenemos que realizar un mismo conjunto de operaciones en varios sitios de la clase, usaremos un método para así no repetir código. Si no queremos que se pueda usar desde fuera de la clase, lo declaramos `private`**.**

Ejemplo. **Sobre la clase** `CuentaBancaria`**, supongamos que no permitimos saldos superiores a 10000 euros. Para ello, definimos el método privado** `EsCorrecto` **que realiza las comprobaciones pertinentes. Llamamos a este método desde** `Ingresa` **y** `Retira`**.**

```
class CuentaBancaria{
private:
   double saldo         = 0.0;
   string identificador;  // C++ inicializa los string a ""

   bool EsCorrectoSaldo(double saldo_propuesto){
      return saldo_propuesto >= 0 && saldo_propuesto <= 10000;
   }
public:
   void SetIdentificador(string identificador_cuenta){
      identificador = identificador_cuenta;
   }
   string Identificador(){
      return identificador;
   }
   double Saldo(){
      return saldo;
   }
   void Ingresa(double cantidad){
      double saldo_resultante;

      if (cantidad > 0){
```

```
         saldo_resultante = saldo + cantidad;

         if (EsCorrectoSaldo (saldo_resultante))
            saldo = saldo_resultante;
      }
   }
   void Retira(double cantidad){
      double saldo_resultante;

      if (cantidad > 0){
         saldo_resultante = saldo - cantidad;

         if (EsCorrectoSaldo (saldo_resultante))
            saldo = saldo_resultante;
      }
   }
   void AplicaInteresPorcentual(int tanto_porciento){
      Ingresa (saldo * tanto_porciento / 100.0);
   }
};
int main(){
   CuentaBancaria cuenta;     // saldo = 0

   cuenta.Ingresa(50000);     // saldo = 0
   cuenta.Ingresa(8000);      // saldo = 8000

   bool es_correcto;
   es_correcto = cuenta.EsCorrectoSaldo(50000);   // Error de compilación
                                                  // Método private
```

Podemos comprobar que se repite el código siguiente:

```
if (EsCorrectoSaldo (saldo_resultante))
   saldo = saldo_resultante;
```

Si se desea, puede definirse el siguiente método *privado*:

```
void SetSaldo (double saldo_propuesto){
   if (EsCorrectoSaldo (saldo_propuesto))
      saldo = saldo_propuesto;
}
```

De forma que el método `Ingresa`, por ejemplo, quedaría así:

```
void Ingresa(double cantidad){
   double saldo_resultante;

   if (cantidad > 0){
      saldo_resultante = saldo + cantidad;
      SetSaldo(saldo_resultante);
   }
}
```

Observe que en el caso de que el saldo propuesto sea incorrecto, se deja el valor antiguo. Este criterio es correcto.

Recordemos que `SetSaldo` no queríamos que fuese `public` ya que sólo permitíamos ingresos y retiradas de fondos y no asignaciones directas.

La clase nos quedaría así:

CuentaBancaria
- double saldo - double identificador
- bool EsCorrectoSaldo(double saldo_propuesto) - bool EsCorrectoIdentificador(string identificador_propuesto) - void SetSaldo(double saldo_propuesto) + void SetIdentificador(string identificador_cuenta) + string Identificador() + double Saldo() + void Ingresa(double cantidad) + void Retira(double cantidad) + void AplicaInteresPorcentual(int tanto_porciento)

http://decsai.ugr.es/jccubero/FP/IV_cuenta_bancaria_metodos_privados.cpp

Los métodos privados se usan para realizar tareas propias de la clase que no queremos que se puedan invocar desde fuera de ésta.

Ejercicio. **Reescriba el ejemplo del segmento de la página 463 para que la comprobación de que las coordenadas son correctas se hagan en un método privado.**

```
class SegmentoDirigido{
private:
   double x_1 = 0.0,
          y_1 = 0.0,
          x_2 = 0.0,
          y_2 = 0.0;

   bool SonCorrectas(double abs_1, ord_1, abs_2, ord_2){
      return !(abs_1 == abs_2 && ord_1 == ord2);
   }
public:
   void SetCoordenadas(double origen_abscisa,
                       double origen_ordenada,
                       double final_abscisa,
                       double final_ordenada){
      if (SonCorrectas(origen_abscisa, origen_ordenada,
                       final_abscisa, final_ordenada)){
         x_1 = origen_abscisa;
         y_1 = origen_ordenada;
         x_2 = final_abscisa;
         y_2 = final_ordenada;
      }
   }
   ......
};
```

http://decsai.ugr.es/jccubero/FP/IV_segmento_dirigido_metodos_privados.cpp

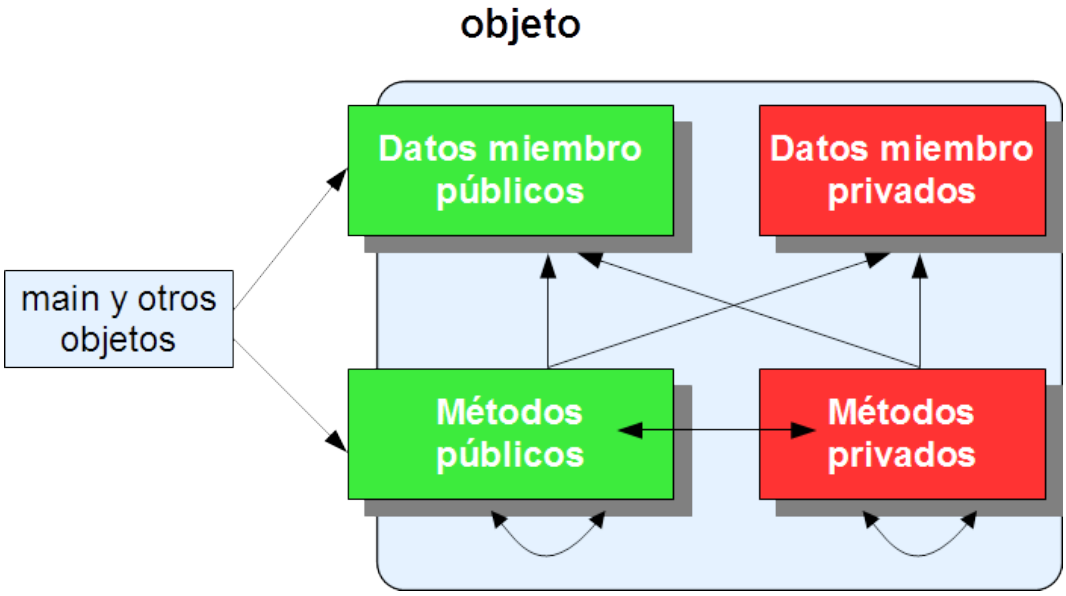
Ejercicio. **Reescriba el ejemplo de la Fecha de la página 468 para que la comprobación de que la fecha es correcta se haga en un método privado.**

Fecha
- int dia - int mes - int anio
- bool EsFechaCorrecta(int el_dia, int el_mes, int el_anio) + void SetDiaMesAnio(int el_dia, int el_mes, int el_anio) + int Dia() + int Mes() + int Anio() + string ToString()

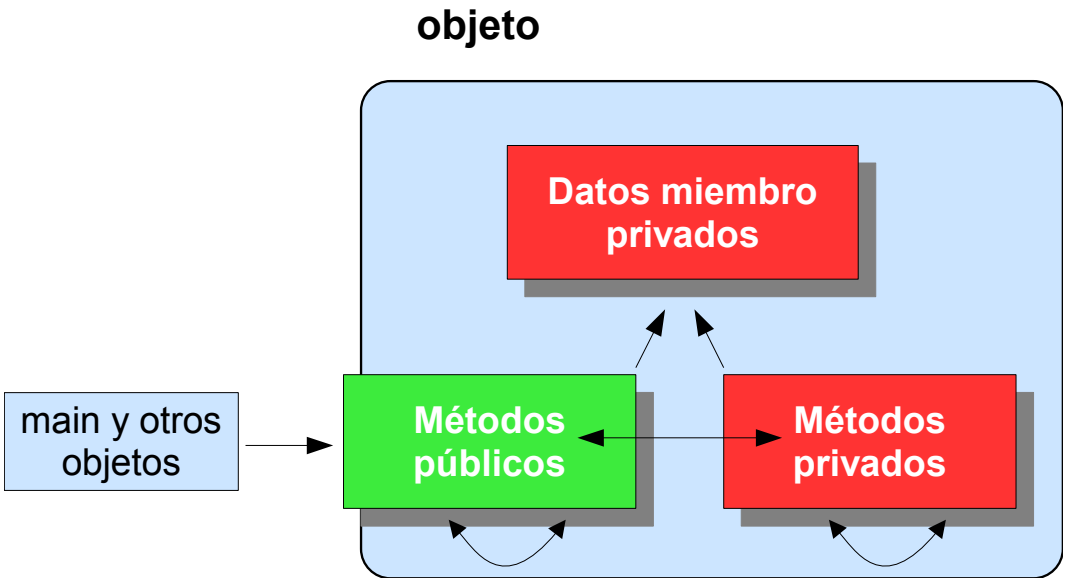

499

```
class Fecha{
private:
   int dia,
       mes,
       anio;

   bool EsFechaCorrecta(int el_dia, int el_mes, int el_anio){
      ......
   }
public:
   void SetDiaMesAnio(int el_dia, int el_mes, int el_anio){
      if (EsFechaCorrecta(el_dia, el_mes, el_anio)){
         dia  = el_dia;
         mes  = el_mes;
         anio = el_anio;
      }
   }
   ......
};
```

En resumen. Lo que C++ permite:



Lo que nosotros haremos:



IV.2.4. Constructores

IV.2.4.1. Estado inválido de un objeto

¿Qué ocurre si una variable no tiene asignado un valor? Cualquier operación que realicemos con ella devolverá un valor indeterminado.

```
int euros;       // euros = ?
cout << euros;   // Imprime un valor indeterminado
```

Lo mismo ocurre con un objeto que contenga datos miembro sin un valor determinado.

```
class Fecha {
private:
   int dia, mes, anio;
......
};

int main(){
   Fecha fecha_nacimiento_JC;  // dia = mes = anio = ?
                               // Estado inválido
```

Diremos que un objeto se encuentra en un momento dado en un *estado inválido (invalid state)* si algunos de sus datos miembros *esenciales* no tienen un valor correcto.

Queremos obligar a que un objeto esté en un estado válido desde el mismo momento de su definición. Para ello, debemos asegurar que sus datos miembro tengan valores *correctos*. ¿Qué entendemos por correcto? Depende de cada clase. Desde luego, si es un valor indeterminado (no asignado previamente), será incorrecto.

El uso de valores por defecto puede ser útil en las siguientes situaciones:

- **Una circunferencia por defecto podría ser la circunferencia goniométrica (centrada en el origen y de radio 1)**
- **Una cuenta bancaria se crea por defecto con saldo 0.**
- **Un segmento dirigido por defecto es el que va del origen (0,0) al mismo origen (0,0) (segmento degenerado)**

Sin embargo, los valores por defecto no resuelven los siguientes problemas:

- **¿Qué identificador por defecto asociamos a una nueva cuenta bancaria? No vale `""`. Debe ser un identificador válido.**
- **Si no admitimos segmentos degenerados, ¿cuál podría ser un segmento por defecto? ¿(0,0)-(0,1)? ¿(0,0)-(1,0)?, $\cdots$.**
- **Si tenemos la clase `Persona`, ¿qué nombre, DNI, edad, etc. se le asignan por defecto? No tiene sentido.**

Vamos a ver una herramienta más potente: el constructor. Éste nos permitirá, entre otras cosas, dar valores concretos a los datos miembro en el momento de la definición del objeto.

En resumen (ver también página 498):

Para evitar que un objeto esté en un estado inválido:

- ▷ *Cuando tenga sentido, podemos asignar valores por defecto a los datos miembro.*
- ▷ *En caso contrario, obligaremos a que en el momento de la creación de cada objeto, se le suministren unos valores concretos.*

  Para ello, se recurre a los constructores.

IV.2.4.2. Definición de constructores

Un *constructor (constructor)* es como un método sin tipo (`void`) de la clase en el que se incluyen todas las acciones que queramos realizar en el momento de construir un objeto (por ejemplo, asignar unos valores iniciales a los datos miembro).

- **El constructor se define dentro de la clase, en la sección `public`.**
- **Debe llamarse obligatoriamente, igual que la clase. No se pone `void`, sino únicamente el nombre de la clase.**
- **Cada vez que se cree un objeto, el compilador ejecutará las instrucciones especificadas en el constructor.**
- **Se pueden incluir parámetros, en cuyo caso, habrá que incluir los correspondientes parámetros actuales en el momento de la definición del objeto. En caso contrario, se produce un error de compilación.**

  Para asignar dichos valores a los datos miembro, se puede hacer con el operador de asignación, pero es más recomendable a través de la *lista de inicialización del constructor (constructor initialization list)* .

  Son construcciones del tipo `<dato_miembro> (<valor inicial>)` separadas por coma. La lista de inicialización del constructor va antes del paréntesis del constructor, y con dos puntos al inicio.

> ***Los constructores nos permiten ejecutar automáticamente un conjunto de instrucciones, cada vez que se crea un objeto.***
>
> ***En particular, si se les pasa como parámetro actual los valores a asignar a los datos miembro, permiten la creación de un objeto en un estado válido, en el mismo momento de su definición.***

Ejemplo. **Añada un constructor a la cuenta bancaria pasándole como parámetro el identificador de cuenta.**

```
class CuentaBancaria{
private:
   double saldo         = 0.0;
   string identificador;
   ......
public:
   /*
   CuentaBancaria(string identificador_cuenta){ // Correcto
      identificador = identificador_cuenta;
   }
   */
   CuentaBancaria(string identificador_cuenta)  // Preferible
      :identificador(identificador_cuenta)      // <- Lista inicialización
   {
   }
   ......
};
int main(){
   // CuentaBancaria cuenta;  <- Error de compilación

   CuentaBancaria cuenta("20310381450100007510");
      // cuenta: "20310381450100007510", 0.0
      // Estado válido
```

Si los parámetros actuales son variables, habrá que crear el objeto después de establecer dichas variables.

```
   cin >> identificador;

   CuentaBancaria cuenta(identificador);
```

Ahora que podemos dar el valor inicial al identificador dentro del constructor, sería lógico imponer que, posteriormente, éste no se pudiese cambiar. Para conseguirlo, bastaría con que el método `SetIdentificador` no fuese público. Nos quedaría:

CuentaBancaria	
- double	saldo
- double	identificador
- bool	EsCorrectoSaldo(double saldo_propuesto)
- bool	EsCorrectoIdentificador(string identificador_propuesto)
- void	SetSaldo(double saldo_propuesto)
- void	SetIdentificador(string identificador_cuenta)
+	CuentaBancaria(string identificador_cuenta)
+ string	Identificador()
+ double	Saldo()
+ void	Ingresa(double cantidad)
+ void	Retira(double cantidad)
+ void	AplicaInteresPorcentual(int tanto_porciento)

```
int main(){
   CuentaBancaria cuenta("20310381450100007510");
   CuentaBancaria otra_cuenta("20310381450100007511");

   // Las siguientes sentencias darían un error de compilación:

   // cuenta.SetIdentificador("20310381450100009876");
   // otra_cuenta.SetIdentificador("20310381450100003144");
```

Ejercicio. **Obligamos a pasar los cuatro puntos de un `SegmentoDirigido` en el constructor. No queremos que el segmento (0,0) - (0,0) sea el segmento por defecto.**

SegmentoDirigido
- double x_1 - double y_1 - double x_2 - double y_2
+ SegmentoDirigido((double origen_abscisa, double origen_ordenada, double final_abscisa, double final_ordenada) .....

```
class SegmentoDirigido{
private:
   double x_1,    // Ya no aceptamos 0.0, 0.0, 0.0, 0.0
          y_1,    // como segmento por defecto
          x_2,
          y_2;
public:
   SegmentoDirigido(double origen_abscisa,
                    double origen_ordenada,
                    double final_abscisa,
                    double final_ordenada)
      :x_1(origen_abscisa),
       y_1(origen_ordenada),
       x_2(final_abscisa),
       y_2(final_ordenada)
   {
   }
   // El resto de métodos no varían
};
int main(){
   // SegmentoDirigido un_segmento;   <- Error de compilación

   SegmentoDirigido un_segmento(3.4, 5.6, 4.5, 2.3);
   // estado válido
   ......
```

Ejercicio. **Añadimos un constructor a la clase `Fecha` que definimos en la página 476 para obligar a crear el objeto con los datos del día, mes y año.**

Fecha
- int dia - int mes - int anio
+ Fecha(int el_dia, int el_mes, int el_anio) + ...


499

```
class Fecha{
private:
   int dia,
       mes,
       anio;
   ......
public:
   Fecha(int el_dia, int el_mes, int el_anio)
      :dia(el_dia),
       mes(el_mes),
       anio(el_anio)
   {
   }
   ......
};
int main(){
   // Fecha nacimiento_JC;  // Error de compilación

   Fecha nacimiento_JC (27, 2, 1967);     // 27, 2, 1967
   Fecha nacimiento_Nadal (3, 6, 1986);   // 3, 6, 1986
```

Dentro de un constructor podemos ejecutar código y llamar a métodos de la clase.

```
class CuentaBancaria{
private:
   double saldo         = 0.0;
   string identificador_IBAN;
   string GetCodigoControl(string id_CCC){
      .......
   }
public:
   CuentaBancaria(string identificador_CCC){
      string codigo_control;
      codigo_control = GetCodigoControl(identificador_CCC);
      identificador_IBAN = "IBAN ES" +
                           codigo_control +
                           " " +
                           identificador_CCC;
   }
   ......
};
int main(){
   CuentaBancaria cuenta("20310381450100007510");
      // cuenta: "IBAN ES 17 20310381450100007510", 0.0
      // Estado válido
```

IV.2.4.3. Constructores sin parámetros

Si se desea, se puede proporcionar un constructor sin parámetros. Hay que indicar que, internamente, C++ proporciona un *constructor de oficio* sin parámetros oculto que permite crear el objeto y poco más.

```
class MiClaseSinConstructor{
   ......
};
int main(){
   MiClaseSinConstructor objeto;   // Constructor oculto de C++
}
```

Si el programador define cualquier constructor (con o sin parámetros) ya no está disponible el constructor de oficio.

```
class MiClase{
   ......
public:
   MiClase(){        // Único constructor disponible
      ......
   }
};
int main(){
   MiClase objeto;   // Se llama al constructor anterior
```

Nota:

En C++, cualquier constructor sin parámetros (ya sea definido por el programador o por el propio lenguaje) se le denomina constructor por defecto (default constructor)

Ejemplo. **Sobre la clase `Fecha`, definimos un constructor sin parámetros para que cree una fecha con los datos de la fecha actual.**

Fecha
- int dia - int mes - int anio
- void EsFechaCorrecta(int el_dia, int el_mes, int el_anio) + Fecha() + void SetDiaMesAnio (int el_dia, int el_mes, int el_anio) + int Dia() + int Mes() + int Anio() + string ToString()

Debemos usar la biblioteca `ctime`. No hay que comprender el código del constructor, sino entender cuándo se ejecuta.


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

class Fecha {
private:
   int dia, mes, anio;
   ......
public:
   Fecha(){
      time_t hoy_time_t;        // No hace falta entender este código
      struct tm * hoy_struct;

      hoy_time_t = time(NULL);
      hoy_struct = localtime ( &hoy_time_t );

      dia  = hoy_struct->tm_mday;
      mes  = hoy_struct->tm_mon + 1;
      anio = hoy_struct->tm_year + 1900 +1;
   }
   ......
};
int main(){
   Fecha fecha_hoy;                    // Constructor sin parámetros

   cout << fecha_hoy.ToString();
```

Ejemplo. **Supongamos que queremos generar números aleatorios entre 3 y 7. ¿Le parece que la siguiente secuencia es aleatoria?**

```
5 5 5 6 6 6 7 7 7 3 3 3
```

Parece obvio que no. Existen algoritmos para generar secuencias de *números pseudo-aleatorios (peudorandom numbers)* , es decir, secuencias de números con un comportamiento parecido al de una secuencia aleatoria.

Vamos a crear una clase para generar reales pseudo-aleatorios entre 0 y 1. En el constructor de la clase (que no tiene parámetros) programamos todo lo necesario para inicializar adecuadamente el generador de números pseudoaleatorios.

GeneradorAleatorioReales_0_1	
-	.....
+	GeneradorAleatorioReales_0_1()
+ double	Siguiente()

Cada vez que llamemos al método `Siguiente`, se generará el siguiente valor de la secuencia.

```
#include <random>  // para la generación de números pseudoaleatorios
#include <chrono>  // para la semilla
using namespace std;

class GeneradorAleatorioReales_0_1{
private:
   mt19937 generador_mersenne;   // Mersenne twister
   uniform_real_distribution<double> distribucion_uniforme;

   long long Nanosec(){
      return
      chrono::high_resolution_clock::now().time_since_epoch().count();
```

```
   }
public:
   GeneradorAleatorioReales_0_1(){
      distribucion_uniforme = uniform_real_distribution<double>(0.0, 1.0);
      const int A_DESCARTAR = 70000;
      // Panneton et al. ACM TOMS Volume 32 Issue 1, March 2006
      auto semilla = Nanosec();
      generador_mersenne.seed(semilla);
      generador_mersenne.discard(A_DESCARTAR);
   }
   double Siguiente(){
      return distribucion_uniforme(generador_mersenne);
   }
};

int main(){
   GeneradorAleatorioReales_0_1 aleatorio;

   for (int i=0; i<100; i++)
      cout << aleatorio.Siguiente() << " ";

   cout << "\n\n";
}
```

http://decsai.ugr.es/jccubero/FP/IV_generador_aleatorio_0_1.cpp

IV.2.4.4. Sobrecarga de constructores

Todos los métodos de una clase se pueden sobrecargar, tal y como se vio con las en el apartado de funciones (ver página 396)

Lo mismo ocurre con los constructores: se puede proporcionar más de un constructor, siempre que cambien en el tipo o en el número de parámetros. El compilador creará el objeto llamando al constructor correspondiente según corresponda con los parámetros actuales.

Esto nos permite crear objetos de maneras distintas, según nos convenga.

Ejemplo. **Queremos que estén disponibles los dos constructores que hemos visto de la clase** `Fecha`**:**

Fecha
- int dia - int mes - int anio
+ Fecha() + Fecha(int el_dia, int el_mes, int el_anio) .....


499

```
class Fecha{
private:
   int dia, mes, anio;
   ......
public:
   Fecha(){
      time_t hoy_time_t;       // No hace falta entender este código
      ......
   }
   Fecha(int el_dia, int el_mes, int el_anio)
      :dia(el_dia),
       mes(el_mes),
       anio(el_anio)
   {
   }
   ......
};
int main(){
   Fecha fecha_hoy;                    // Constructor sin parámetros
   Fecha nacimiento_JC (27, 2, 1967); // Constructor con parámetros

   cout << fecha_hoy.ToString();
   ......
```

Ejemplo. **Sobre la cuenta bancaria, proporcionamos dos constructores:**

- **Un constructor que obligue a pasar el identificador pero no el saldo (en cuyo caso se quedará con cero)**
- **Otro constructor que obligue a pasar el identificador y el saldo**

CuentaBancaria
- double saldo - double identificador
+ CuentaBancaria(string identificador_cuenta) + CuentaBancaria(string identificador_cuenta, double saldo_inicial) .....

```
class CuentaBancaria{
private:
   ......
public:
   CuentaBancaria(string identificador_cuenta)                          // 1
      :identificador(identificador_cuenta)
   { }
   CuentaBancaria(string identificador_cuenta, double saldo_inicial)   // 2
      :saldo(saldo_inicial),
       identificador(identificador_cuenta)
   { }
   ......
};
int main(){
   CuentaBancaria cuenta2("20310381450100007511");                 // 1
      // cuenta2: "20310381450100007511", 0.0
   CuentaBancaria cuenta1("20310381450100007510", 3000);           // 2
      // cuenta1: "20310381450100007510", 3000
   // CuentaBancaria cuenta;   Error de compilación
```

IV.2.4.5. Llamadas entre constructores

Si observamos el ejemplo anterior, podemos apreciar que hay cierto código repetido en los constructores:

```
identificador(identificador_cuenta)
```

Lo resolvemos llamando a un constructor desde el otro constructor. Debe hacerse en la lista de inicialización del constructor:

Ejemplo. **Llamamos a un constructor dentro del otro en el ejemplo de la cuenta bancaria.**

```
class CuentaBancaria{
private:
   double saldo   = 0.0;
   string identificador;  ......
public:
   CuentaBancaria(string identificador_cuenta)                          // 1
      :identificador(identificador_cuenta)
   { }
   CuentaBancaria(string identificador_cuenta, double saldo_inicial) // 2
      :CuentaBancaria(identificador_cuenta)
      // No puede haber nada más
   {
      saldo = saldo_inicial;
   }
   ......
};
int main(){
   CuentaBancaria cuenta2("20310381450100007511");                  // 1
      // cuenta2: "20310381450100007511", 0.0
   CuentaBancaria cuenta1("20310381450100007510", 3000);           // 2
      // cuenta1: "20310381450100007510", 3000
```

IV.2.4.6. Estado inválido de un objeto -revisión-

¿Qué ocurre si los datos suministrados en el constructor no son correctos? Tenemos un problema cuya resolución requiere de herramientas que no se verán en este curso.

Ejemplo. **Retomamos el ejemplo de la página 486 de la clase `Fecha`**

El constructor asignaba los parámetros actuales a los datos miembro:

```
public:
   Fecha(int el_dia, int el_mes, int el_anio)
      :dia(el_dia), mes(el_mes), anio(el_anio)
```

Pero, ¿qué ocurre si los parámetros actuales no son correctos?

```
int main(){
   Fecha una_fecha(-1, 2, -9);  // -1, 2 ,-9
   // una_fecha está en un estado inválido
```

La mejor solución a este problema pasa por lanzar una excepción dentro del constructor (ver Tema V) y no permitir la creación del objeto, pero está fuera de los objetivos del curso. Al menos, podríamos paliar el problema asignando un valor concreto a los datos miembro que nos indique que ha habido un problema.

- **A un objeto de la clase `Fecha`, le asignaríamos `-1, -1, -1`.**
- **A un objeto de la clase `Circunferencia`, le asignaríamos al radio `-1`. Al centro le podríamos asignar `(NAN,NAN)`.**
- **A un objeto de la clase `CuentaBancaria`, le asignaríamos `NAN` al saldo y `""` al identificador.**
- **A un objeto de la clase `SegmentoDirigido`, le asignaríamos `NAN` a las 4 coordenadas.**

Para asignar a los datos miembro dicho valor especial, lo podemos hacer de dos formas:

- **O bien usamos la inicialización de C++ 11 en la declaración de los datos miembro.**
- **O bien lo asignamos directamente en el constructor**

Lo vemos en el ejemplo de la siguiente página.

En resumen (ver también página 480)

> ***Para evitar que un objeto esté en un estado inválido en el mismo momento de su definición:***
>
> ***Si los parámetros actuales pasados al mismo constructor no son correctos, la mejor solución pasaría por lanzar una excepción (se verá en un tema posterior de ampliación) desde dentro del constructor y así impedir la creación del objeto.***
>
> ***Como no sabemos hacerlo, por ahora, les asignaremos (siempre que sea posible) un valor destacado que indique que ha habido un problema.***

Ejemplo. Clase `Fecha` completa

Fecha
- int dia - int mes - int anio
- bool EsFechaCorrecta(int el_dia, int el_mes, int el_anio) + Fecha(int el_dia, int el_mes, int el_anio) + Fecha() + void SetDiaMesAnio(int el_dia, int el_mes, int el_anio) + int Dia() + int Mes() + int Anio() + string ToString()

Opciones:

- **O bien usamos la inicialización de C++ 11 en la declaración de los datos miembro:**

```
class Fecha{
  int dia  = -1;
  int mes  = -1;
  int anio = -1;

  Fecha(int el_dia, int el_mes, int el_anio){
     if (EsFechaCorrecta(el_dia, el_mes, el_anio)){
        dia  = el_dia;
        mes  = el_mes;
        anio = el_anio;
     }
  }
  ......
```

▷ **O bien lo asignamos en el constructor:**

```
class Fecha {
   int dia;
   int mes;
   int anio;

   Fecha(int el_dia, int el_mes, int el_anio){
      if (EsFechaCorrecta(el_dia, el_mes, el_anio)){
         dia  = el_dia;
         mes  = el_mes;
         anio = el_anio;
      }
      else
      // lo mejor sería lanzar una excepción
         dia = mes = anio = -1;
   }
   ......
```

En cualquiera de los dos casos, el programa principal quedaría así:

```
int main(){
   Fecha una_fecha(-1, 2, -9);                // -1, -1 ,-1
   // una_fecha está en un estado inválido, pero "reconocible"

   Fecha nacimiento_JC (27, 2, 1967);         // 27, 2, 1967
   // estado válido

   nacimiento_JC.SetDiaMesAnio(-1, 2, 100);  // 27, 2, 1967
   // se queda como estaba. Estado válido.
```



http://decsai.ugr.es/jccubero/FP/IV_fecha.cpp

IV.2.5. Programando como profesionales

Conceptos fundamentales vistos hasta ahora en la construcción de una clase:

- **Ámbitos** `private` **y** `public`**.**
- **Encapsulación de datos y métodos dentro de una clase.**

Ahora bien,

- **¿Qué incluimos como dato miembro de una clase y qué pasamos como parámetros a sus métodos?**
- **¿Qué preferimos: pocas clases que hagan muchas cosas distintas o más clases que hagan cosas concretas?**

IV.2.5.1. Datos miembro vs parámetros de los métodos

Los datos miembro se comportan como datos *globales* dentro del objeto, por lo que son directamente accesibles desde cualquier método definido dentro de la clase y no hay que pasarlos como parámetros a dichos métodos.

Por tanto, ¿los métodos de las clases en PDO suelen tener menos parámetros que las funciones *globales* usadas en Programación Procedural?

¡Sí, por supuesto!

Ejemplo. **¿Cómo se definiría la longitud de un segmento con una función y no dentro de la clase `SegmentoDirigido`?**

- **Con una función debemos pasar 4 parámetros:**

```
Longitud(double x_1, double y_1, double x_2, double y_2){
   ......
}
```

- **Dentro de la clase, no:**

SegmentoDirigido
- double x_1 - double y_1 - double x_2 - double y_2
...... + double Longitud()

¿Incluimos la longitud como dato miembro del segmento?

SegmentoDirigido
- double x_1 - double y_1 - double x_2 - double y_2 ¿- double longitud?
...... + double Longitud()

No debemos incluir `longitud` como un dato miembro. Es una propiedad que puede calcularse en función de las coordenadas.

> ***Las propiedades que se puedan calcular a partir de los datos miembro, se obtendrán a través de un método.***

Ejemplo. **Defina la clase `Persona`. Representaremos su nombre, altura y edad.**

Persona
- string nombre - int edad - int altura
- bool EsCorrectaEdad(string edad_persona) - bool EsCorrectaAltura(string altura_persona) + Persona(string nombre_persona, int edad_persona, int altura_persona) + void SetNombre(string nombre_persona) + void SetEdad(int edad_persona) + void SetAltura(int altura_persona) + string Nombre() + int Edad() + int Altura()

```
class Persona{
private:
   string nombre;
   int edad;
   int altura;

   bool EsCorrectaEdad(int una_edad){
      return una_edad > 0 && una_edad < 120;
   }
   bool EsCorrectaAltura(int una_altura){
      return una_altura > 50 && una_altura < 250;
   }
public:
   Persona(string nombre_persona, int edad_persona, int altura_persona){
      if (EsCorrectaEdad(edad_persona)
         && EsCorrectaAltura(altura_persona)){
         nombre = nombre_persona;
         edad = edad_persona;
```

```
            altura = altura_persona;
        }
        else{
            // lo mejor sería lanzar una excepción
            nombre = "";
            edad   = -1;
            altura = -1;
        }
    }
    string Nombre(){
        return nombre;
    }
    int Edad(){
        return edad;
    }
    int Altura(){
        return altura;
    }
    void SetNombre(string nombre_persona){
            nombre = nombre_persona;
    }
    void SetEdad(int edad_persona){
        if (EsCorrectaEdad(edad_persona))
            edad = edad_persona;
    }
    void SetAltura(int altura_persona){
        if (EsCorrectaAltura(altura_persona))
            altura = altura_persona;
    }
};
```

Nota:

La edad es un dato temporal, por lo que sería mucho mejor representar la fecha de nacimiento. Lo podremos hacer en el tema V cuando veamos cómo declarar un objeto como dato miembro de otro.

Retomamos el ejemplo de la página 420. Habíamos definido estas funciones:

```
bool EsMayorEdad(int edad){
   return  edad >= 18;
}
bool EsAlta(int altura, int edad){
   if (EsMayorEdad(edad))
      return altura >= 190;
   else
      return altura >= 175;
}
int main(){
   ......
   es_mayor_edad = EsMayorEdad(48);
   es_alta       = EsAlta(186, 48);
```

¿Cómo quedaría si trabajásemos dentro de la clase?

Bastaría añadir los siguientes métodos

```
class Persona{
private:
   string nombre;
   int edad;
   int altura;
   ......
public:
   ......
   bool EsMayorEdad(){
      return  edad >= 18;
   }
   bool EsAlta(){
      if (EsMayorEdad())
         return altura >= 190;
      else
         return altura >= 175;
   }
};
int main(){
   Persona una_persona("JC", 48, 186);
   ......
   es_mayor_edad = una_persona.EsMayorEdad();
   es_alta       = una_persona.EsAlta();
```

http://decsai.ugr.es/jccubero/FP/IV_persona.cpp

Observe la diferencia:

- **Con funciones:**

```
......
es_mayor_edad = EsMayorEdad(48);
es_alta       = EsAlta(186, 48);
```

- **Con una clase:**

```
Persona una_persona("JC", 48, 186);
......
es_mayor_edad = una_persona.EsMayorEdad();
es_alta       = una_persona.EsAlta();
```

En el segundo caso, los datos importantes de la persona, ya están dentro del objeto `una_persona` y no hay que pasarlos como parámetros una y otra vez.

En cualquier caso, en numerosas ocasiones los métodos necesitarán información adicional que no esté descrita en el estado del objeto, como por ejemplo, la cantidad a ingresar o retirar de una cuenta bancaria, o el número de unidades a trasladar un segmento. Esta información adicional serán parámetros de los correspondientes métodos.

```
SegmentoDirigido un_segmento (1.0, 2.0, 1.0, 3.0);
un_segmento.TrasladaHorizontal(4);
```

Sólo incluiremos como datos miembro aquellas características esenciales que determinan una entidad.

La información adicional que se necesite para ejecutar un método, se proporcionará a través de los parámetros de dicho método.

No siempre es fácil determinar cuáles deben ser los datos miembros y cuáles los parámetros. Una heurística de ayuda es la siguiente:

> ***Cuando no tengamos claro si un dato ha de ser dato miembro o parámetro de un método: si suele cambiar continuamente durante la vida del objeto, es candidato a ser parámetro. Si no cambia o lo hace poco, es candidato a ser dato miembro.***

Ejemplo. **¿Incluimos en la cuenta bancaria las cantidades a ingresar y a retirar como datos miembro?**

```
class CuentaBancaria{
private:
   double saldo = 0.0;;
   string identificador;
   double cantidad_a_ingresar;
   double cantidad_a_retirar;
public:
   ......
   void SetCantidadIngresar(double cantidad){
      cantidad_a_ingresar = cantidad;
   }
   void Ingresa(){
      saldo = saldo + cantidad_a_ingresar;
   }
   void Retira(){
      saldo = saldo - cantidad_a_retirar;
   }
   ......
};

int main(){
   CuentaBancaria una_cuenta(40);

   una_cuenta.SetCantidadIngresar(25);
   una_cuenta.Ingresa();
```

La cantidad a ingresar/retirar no es un dato miembro. Es una información que se necesita sólo en el momento de ingresar/retirar.

Ejemplo. **Recuperamos el ejemplo del cómputo del salario de la página 194.**

Supongamos que gestionamos varias sucursales de la misma empresa y que los criterios de subida (experiencia, edad, número de hijos, etc) son los mismos, salvo que los límites pueden variar de una a otra. En cualquier caso, una vez establecidos ya no cambiarán. Supongamos también que el salario base no suele cambiar mucho dentro de la misma empresa.

Con una función tendríamos:

```
double SalarioFinal(int minimo_experiencia_alta,
                    int minimo_familia_numerosa,
                    int minimo_edad_senior,
                    double maximo_salario_bajo,
                    double salario_base,
                    int experiencia,
                    int edad,
                    int numero_hijos)
```

Queda mucho mejor una clase en la que:

- **Los datos que varían poco durante la vida del objeto serán datos miembro y los establecemos en el constructor. Por ejemplo, los límites para establecer el salario final.**
- **Los datos que van variando constantemente durante la vida del objeto serán parámetros a los métodos. Por ejemplo, la experiencia, edad y número de hijos de cada empleado.**
- **Habrá datos que puedan variar algo y que son necesarios para los métodos de la clase. Éstos serán datos miembro que se establecerán con métodos o en el constructor (preferible).**

CalculadoraSalario	
- int	minimo_experiencia_alta
- int	minimo_familia_numerosa
- int	minimo_edad_senior
- double	maximo_salario_bajo
- double	salario_base
- bool	EsCorrectoSalario(double un_salario)
+	CalculadoraSalario(int minimo_experiencia_alta_sucursal, int minimo_familia_numerosa_sucursal, int minimo_edad_senior_sucursal, int maximo_salario_bajo_sucursal)
+ void	SetSalarioBase(double salario_base_mensual_trabajador)
+ double	SalarioFinal(int experiencia, int edad, int numero_hijos)

```
......
int main(){
   // Orden de los parámetros en el constructor:
   // mínimo experiencia alta, mínimo familia numerosa,
   // mínimo edad senior, máximo salario bajo

   CalculadoraSalario calculadora_salario_Jaen(2, 2, 45, 1300);
   CalculadoraSalario calculadora_salario_Granada(3, 3, 47, 1400);
   ......
   calculadora_salario_Jaen.SetSalarioBase(salario_base_Jaen);
   ......
   // Bucle de lectura de datos:
   do{
      ......
      cin >> experiencia;
      cin >> edad;
      cin >> numero_hijos;
      ......
      salario_final =
         calculadora_salario_Jaen.SalarioFinal(experiencia,
                                               edad,
                                               numero_hijos);
      ......
   }while (hay_datos_por_procesar);
   ......
```

http://decsai.ugr.es/jccubero/FP/IV_actualizacion_salarial.cpp

IV.2.5.2. Principio de Responsabilidad única y cohesión de una clase

¿Qué preferimos: pocas clases que hagan muchas cosas distintas o más clases que hagan cosas concretas? Por supuesto, las segundas.

Principio en Programación Dirigida a Objetos.

Principio de Responsabilidad Única

(Single Responsability Principle)

Un objeto debería tener una única responsabilidad, la cual debe estar completamente encapsulada en la definición de su clase.

A cada clase le asignaremos una única responsabilidad. Esto facilitará:

- **La reutilización en otros contextos**
- **La modificación independiente de cada clase (o al menos paquetes de clases)**

Heurísticas:

- **Si para describir el propósito de la clase se usan más de 20 palabras, puede que tenga más de una responsabilidad.**
- **Si para describir el propósito de una clase se usan las conjunciones *y* u *o*, seguramente tiene más de una responsabilidad.**

Ejemplo. **¿Es correcto este diseño?:**

```
class CuentaBancaria{
private:
   double saldo;
   string identificador;
public:
   CuentaBancaria(double saldo_inicial){
      saldo = saldo_inicial;
   }
   string Identificador(){
      return identificador;
   }
   void SetIdentificador(string identificador_cuenta){
      identificador = identificador_cuenta;
   }
   double Saldo(){
      return saldo;
   }
   void ImprimeSaldo(){
      cout << "\nSaldo = " << saldo;
   }
   void ImprimeIdentificador(){
      cout << "\nIdentificador = " << identificador;
   }
};
```

La responsabilidad de esta clase es gestionar los movimientos de una cuenta bancaria Y comunicarse con el dispositivo de salida por defecto. Esto viola el principio de única responsabilidad. Esta clase no podrá reutilizarse en un entorno de ventanas, en el que `cout` no funciona.

Mezclar E/S y cómputos en una misma clase viola el principio de respon-

sabilidad única. Lamentablemente, ésto se hace con mucha frecuencia, incluso en multitud de libros de texto.

Jamás mezclaremos en una misma clase métodos de *cómputo* con métodos que accedan a los dispositivos de entrada/salida.

Ejemplo. **Retomamos el ejemplo de la fecha:**

¿Nos preguntamos si hubiera sido más cómodo sustituir el método `ToString` por `Imprimir`?:

```
class Fecha {
private:
   int dia, mes, anio;
   ......
public:
   ......
   void Imprime(){
      cout << string(dia) + "/" + to_string(mes)
              + "/" + to_string(anio);
   }
};
int main(){
   Fecha nacimiento_JC (27,2,1987);
   nacimiento_JC.Imprime();
```

De nuevo estaríamos violando el principio de responsabilidad única. Debemos eliminar el método `Imprimir` y usar el que teníamos antes `ToString`

```
class Fecha {
   ......
   string ToString(){
      return to_string(dia) + "/" + to_string(mes)
             + "/" + to_string(anio);
   }
};
int main(){
   Fecha nacimiento_JC (27,2,1987);
   cout << nacimiento_JC.ToString();
```

Un concepto ligado al principio de responsabilidad única es el de *cohesión (cohesion)* , que mide cómo de relacionadas entre sí están las funciones desempeñadas por un componente software (paquete, módulo, clase o subsistema en general)

> ***Por normal general, cuantos más métodos de una clase utilicen todos los datos miembros de la misma, mayor será la cohesión de la clase.***

***Ejemplo.* ¿Es correcto este diseño?**

ClienteCuentaBancaria
- double saldo - string identificador
- string nombre - string dni - int dia_ncmto - int mes_ncmto - int anio_ncmto
- void SetIdentificador(string identificador_cuenta) + ClienteCuentaBancaria(double saldo_inicial, string nombre_cliente, string dni_cliente, int dia_nacimiento, int mes_nacimiento, int anio_nacimiento) + string Identificador() + double Saldo() + void Ingresa(double cantidad) + void Retira(double cantidad) + void AplicaInteresPorcentual(int tanto_por_ciento)
+ string Nombre() + string DNI() + string FechaNacimiento() .....

Puede apreciarse que hay *grupos* de métodos que acceden a otros *grupos* de datos miembro. Esto nos indica que pueden haber varias responsabilidades mezcladas en la clase anterior.

Solución: Trabajar con dos clases: `CuentaBancaria` **y** `Cliente`**:**

Cliente
- string nombre - string dni - int dia_ncmto - int mes_ncmto - int anio_ncmto
+ Cliente(string nombre_cliente, string dni_cliente, int dia_nacimiento int mes_nacimiento, int anio_nacimiento) + string Nombre() + string DNI() + string FechaNacimiento() .....

Si necesitamos conectar ambas clases, podemos crear una tercera clase que contenga un `Cliente` **y una colección de cuentas asociadas. Esto se verá en el último tema.**

IV.2.5.3. Funciones vs Clases

¿Cuándo usaremos funciones y cuándo clases?

- **Como norma general, usaremos clases.**
- **Usaremos funciones para implementar tareas muy genéricas que operan sobre tipos de datos simples y que preveamos se pueden utilizar en programas distintos.**

```
bool   SonIguales(double uno, double otro)
bool   EsPrimo(int dato)
char   ToMayuscula(char letra)
double Redondea(double numero)
......
```

Ejemplo. **Retomamos el ejemplo del segmento de la página 475. El método privado `SonCorrectas` comprobaba que las coordenadas del punto inicial fuese distintas del punto final:**

```
bool SonCorrectas(double abs_1, ord_1, abs_2, ord_2){
   return !(abs_1 == abs_2 && ord_1 == ord2);
}
```

Al estar trabajando con reales, hubiese sido mejor realizar la comparación con un margen de error:

```
bool SonCorrectas(double abs_1, ord_1, abs_2, ord_2){
   return !(SonIguales(abs_1,abs_2) && SonIguales(ord_1,ord2));
}
```

¿Dónde definimos `SonIguales`? No es un método relativo a un objeto `Segmento`, por lo que no es un método público. Podría ser:

- ▷ **O bien un método privado.**
- ▷ **O bien una función global.**

Si el criterio de igualdad entre reales es específico para la clase (queremos un margen de error específico para las coordenadas de un segmento) usaremos un método privado.

Pero en este ejemplo, es de esperar que otras clases como `Triangulo`, `Cuadrado`, etc, usen el mismo criterio de igualdad entre reales. Por tanto, preferimos una función.

```
bool SonIguales(double uno, double otro) {
   return abs(uno-otro) <= 1e-6;
}

class SegmentoDirigido{
private:
   double x_1 = 0.0,
          y_1 = 0.0,
          x_2 = 0.0,
          y_2 = 0.0;
   bool SonCorrectas(double abs_1, double ord_1,
                     double abs_2, double ord_2){
      return !(SonIguales(abs_1,abs_2) && SonIguales(ord_1,ord2));
   }
public:
   void SetCoordenadas(double origen_abscisa,
                       double origen_ordenada,
                       double final_abscisa,
                       double final_ordenada){
      if (SonCorrectas(origen_abscisa, origen_ordenada,
                       final_abscisa, final_ordenada)){
         x_1 = origen_abscisa;
         y_1 = origen_ordenada;
         x_2 = final_abscisa;
         y_2 = final_ordenada;
      }
   }
   ......
};
class Triangulo{
   // Aquí también llamamos a SonIguales
   ......
};
```

Ejemplo. **Una clase `Fraccion` podría tener el siguiente esquema (los valores los pasamos en el constructor y no se permite su posterior modificación)**

Fraccion	
- int	numerador
- int	denominador
+	Fraccion(int el_numerador, int el_denominador)
+ int	Numerador()
+ int	Denominador()
+ void	Normaliza()
+ double	Division()
+ string	ToString()

```
int main(){
   Fraccion una_fraccion(4, 10);    // 4/10

   una_fraccion.Normaliza();        // 2/5
   cout << una_fraccion.ToString();
   ......
```

Para implementar el método `Normaliza`, debemos dividir numerador y denominador por el máximo común divisor. ¿Dónde lo implementamos? El cómputo del MCD no está únicamente ligado a la clase `Fraccion`, por lo que usamos una función global.

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

int MCD(int primero, int segundo){
   ......
}

class Fraccion{
   ......
   void Normaliza(){
      int mcd;

      mcd = MCD(numerador, denominador);
      numerador = numerador/mcd;
      denominador = denominador/mcd;
   }
};

int main(){
   Fraccion una_fraccion(4, 10);    // 4/10

   una_fraccion.Normaliza();        // 2/5
   cout << una_fraccion.ToString();
   ......
```

http://decsai.ugr.es/jccubero/FP/IV_fraccion.cpp

Como norma general, usaremos clases para modularizar los programas. En ocasiones, también usaremos funciones para implementar tareas muy genéricas que operan sobre tipos de datos simples.

IV.2.5.4. Comprobación y tratamiento de las precondiciones

En la página 426 se vieron las precondiciones de una función. La misma definición se aplica sobre los métodos. Ahora vamos a responder las siguientes preguntas:

- ▷ **¿Debemos comprobar dentro de la función/método si se satisfacen sus precondiciones?**
- ▷ **Si decidimos comprobarlo, ¿qué hacemos en el caso de que se violen?**

Ejemplo. **Función factorial.**

Versión en la que no se comprueba la precondición:

```
// Prec: 0 <= n <= 20
long long Factorial (int n){
   int i;
   long long fact = 1;

   for (i = 2; i <= n; i++)
      fact = fact * i;

   return fact;
}
```

Versión en la que se comprueba la precondición:

```
long long Factorial (int n){
   int i;
   long long fact = 1;

   if (n >= 0 && n <= 20){
      for (i = 2; i <= n; i++)
```

```
            fact = fact * i;
      }
      return fact;
   }
```

En este ejemplo, podemos optar por omitir la comprobación. Si se viola, el resultado es un desbordamiento aritmético y el factorial contendrá un valor indeterminado pero no supone un error grave del programa.

Ejemplo. **Clase** `CuentaBancaria`**.**

Versión en la que no se comprueba la precondición:

```
class CuentaBancaria{
   ......
   // Prec: cantidad > 0
   void Ingresa(double cantidad){
      saldo = saldo + cantidad;
   }
};
```

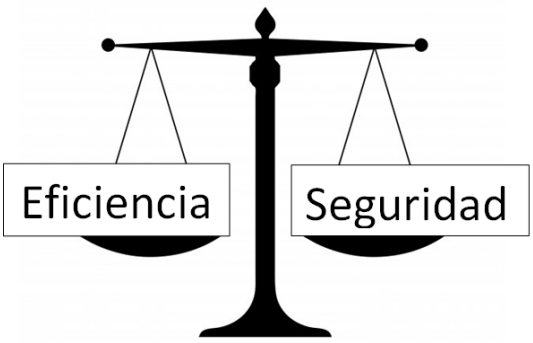
Versión en la que se comprueba la precondición:

```
class CuentaBancaria{
   ......
   void Ingresa(double cantidad){
      if (cantidad > 0)
         saldo = saldo + cantidad;
   }
};
```

En este caso, es mejor comprobar la precondición ya que no podemos permitir un saldo negativo.

¿Cuándo debemos comprobar las precondiciones?

Si la violación de una precondición de una función o un método público puede provocar errores de ejecución graves, dicha precondición debe comprobarse dentro del método.
En otro caso, puede omitirse (sobre todo si prima la eficiencia)



¿Comprobar Precondiciones dentro del método/función?

Supongamos que decidimos comprobar las precondiciones de una función/método y resulta que no se satisfacen ¿Qué hacemos? Posibilidades:

- **No hacer nada.**

  Si la cantidad a ingresar en una cuenta bancaria es negativa, no lo permitimos y dejamos la cantidad que hubiese.

```
class CuentaBancaria{
   ......
   void Ingresa(double cantidad){
      if (cantidad > 0)
         saldo = saldo + cantidad;
      // no hay else
   }
```

- **Realizar una acción por defecto.**

  Si la cantidad a ingresar es negativa, la convertimos a positivo y la ingresamos.

```
class CuentaBancaria{
   ......
   void Ingresa(double cantidad){
      if (cantidad > 0)
         saldo = saldo + cantidad;

      else
         saldo = saldo - cantidad;  // poco recomendable
   }
```

La elección de una alternativa u otra dependerá del problema concreto en cada caso, pero lo usual será no hacer nada. En el ejemplo anterior, el hecho de que la cantidad pasada como parámetro sea negativa, nos está indicando algún problema y muy posiblemente estaremos cometiendo un error al ingresar el valor en positivo.

Esta misma situación se presenta con los constructores. Ya vimos en la página 498 que si los parámetros actuales al constructor son incorrectos, lo único que podíamos hacer es una acción por defecto, a saber, asignarles un valor especial (aunque ya dijimos que lo mejor sería lanzar una excepción)

¿Y cómo sabemos que se ha violado la precondición desde el sitio en el que se invoca al método/función? ¿Lo siguiente sería correcto?

```
class CuentaBancaria{
   ......
   void Ingresa(double cantidad){
      if (cantidad > 0)
         saldo = saldo + cantidad;
      else
         cout << "La cantidad a ingresar no es positiva";
   }
};
```

Nunca haremos este tipo de notificaciones. Ningún método de la clase `CuentaBancaria` debe acceder al periférico de entrada o salida.

Soluciones para notificar el error:

- **Notificarlo *a la antigua usanza* devolviendo un código de error**
- **Mucho mejor: devolver una *excepción (exception)***

En el tema V veremos las excepciones. Veamos ahora la notificación devolviendo un código de error:

- **Si trabajamos con un método/función `void`, lo convertimos en un método/función que devuelva un código de error. El tipo devuelto será `bool` si sólo notificamos la existencia o no del error. Si hay varios posibles errores, lo mejor es usar un enumerado.**
- **Si el método/función no es un `void`, podemos usar el mismo valor devuelto para albergar el código de error, pero éste no debe ser uno de los valores *legales* de los devueltos por el método/función.**
- **Ninguna de las anteriores soluciones es aplicable a los constructores ya que éstos no pueden devolver ningún valor.**

Ejemplo. **En la cuenta bancaria transformamos el método `Ingresa` para que devuelva el código de error:**

```
void Ingresa(double cantidad) --> bool Ingresa(double cantidad)

class CuentaBancaria{
......
   bool Ingresa(double cantidad){
      bool error;

      error = (cantidad < 0);

      if (!error)
         saldo = saldo + cantidad;

      return error;
   }
};
int main(){
   ......
   bool error_cuenta;
   error_cuenta = cuenta.Ingresa(dinero);

   if (error_cuenta)
      cout << "La cantidad a ingresar no es positiva";
```

En cualquier caso, si devolvemos un código de error, en el sitio de la llamada, tendremos que poner *siempre* el `if` correspondiente, lo que resulta bastante tedioso.

La mejor solución pasa por devolver una excepción (ver tema V) dentro del método/función en el que se viola la precondición. Esto también es válido para el constructor.

En resumen:

Si decidimos que es necesario comprobar alguna precondición dentro de un método/función/constructor y ésta se viola, ¿qué debemos hacer como respuesta?

- ***O bien no hacemos nada (salvo lanzar una excepción, tal y como se indica en el siguiente párrafo) o bien realizamos una acción por defecto. La elección apropiada depende de cada problema, aunque lo usual será no hacer nada.***
- ***Además de lo anterior, si decidimos que es necesario notificar dicho error al cliente que ejecuta el método/función podemos devolver un código de error o, mucho mejor, lanzar una excepción (ver tema V). En el caso del constructor, sólo podemos lanzar una excepción.***

  La notificación jamás se hará imprimiendo un mensaje en pantalla desde dentro del método/función.

IV.2.6. Registros (structs)

IV.2.6.1. El tipo de dato struct

En algunas ocasiones, la clase que queremos construir es tan simple que no tiene métodos (comportamiento) asociados, ni restricciones sobre los valores asignables. Únicamente sirve para agrupar algunos datos. Para estos casos, usamos mejor un dato tipo *registro (record)* .

Un registro permite almacenar varios elementos de (posiblemente) distintos tipos y gestionarlos como uno sólo. Cada uno de los datos agrupados en un registro es un *campo* (son como los datos miembro públicos de las clases)

Los campos de un registro se suponen relacionados, que hacen referencia a una misma entidad. Cada campo tiene un *nombre*.

En C++, los registros se denominan *struct* .

Un ejemplo típico es la representación en memoria de un *punto* en un espacio bidimensional. Un punto está caracterizado por dos valores: abscisa y ordenada. En este caso, tanto abscisa como ordenada son del mismo tipo, supongamos que sea `int`.

```
struct Punto2D {
    double abscisa;
    double ordenada;
};
```

Se ha definido un tipo de dato registro llamado `Punto2D`. Los datos de este tipo están formados por dos campos llamados `abscisa` y `ordenada`, ambos de tipo `double`.

Cuando se declara un dato de este tipo:

```
Punto2D un_punto;
```

se crea una variable llamada `punto`, y a través de su nombre se puede acceder a los campos que la conforman mediante el operador punto (`.`). Por ejemplo, podemos establecer los valores de la abscisa y ordenada de `un_punto` a 4 y 6 respectivamente con las instrucciones:

```
un_punto.abscisa = 4.0;
un_punto.ordenada = 6.0;
```

Es posible declarar variables `struct` junto a la definición del tipo (aunque normalmente lo haremos de forma separada):

```
struct Punto2D {
    double abscisa;
    double ordenada;
} un_punto, otro_punto;
```

Un ejemplo de `struct` heterogéneo:

```
struct Persona {
    string nombre;
    string apellidos;
    string NIF;
    char   categoria;
    double salario_bruto;
};
```

No se supone ningún orden establecido entre los campos de un `struct` y no se impone ningún orden de acceso a éstos (por ejemplo, puede inicializarse el tercer campo antes del primero).

No puede leerse/escribirse un `struct`, sino a través de cada uno de sus campos, separadamente. Por ejemplo, para leer los valores de `un_punto` desde la entrada estándar:

```
cin >> un_punto.abscisa;
cin >> un_punto.ordenada;
```

y para escribirlos en la salida estándar:

```
cout << "(" << un_punto.abscisa << ", " << un_punto.ordenada << ")";
```

Es posible asignar un `struct` a otro y en consecuencia, un `struct` puede aparecer tanto como *lvalue* y *rvalue* en una asignación.

```
otro_punto = un_punto;
```

Los campos de un `struct` pueden emplearse en cualquier expresión:

```
dist_x = abs(un_punto.abscisa - otro_punto.abscisa);
```

siendo, en este ejemplo, `un_punto.abscisa` y `otro_punto.abscisa` de tipo `double`.

IV.2.6.2. Funciones/métodos y el tipo struct

Un dato `struct` puede pasarse como parámetro a una función y una función puede devolver un `struct`.

Por ejemplo, la siguiente función recibe dos struct y devuelve otro.

```
Punto2D PuntoMedio (Punto2D punto1, Punto2D punto2){
    Punto2D  punto_medio;

    punto_medio.abscisa = (punto1.abscisa + punto2.abscisa) / 2;
    punto_medio.ordenada = (punto1.ordenada + punto2.ordenada) / 2;

    return (punto_medio);
 }
```

IV.2.6.3. Ámbito de un struct

Una variable `struct` es una variable más y su ámbito es de cualquier variable:

- **Puede ser una variable global, aunque ya sabe que el uso de esta clase de variables no está permitido en esta asignatura.**
- **Como cualquier variable local a una función, puede usarse desde su declaración hasta el fín de la función en que ha sido declarada.**

  Por ejemplo, en la función `PuntoMedio` la variable `punto_medio` es una variable local y se comporta como cualquier otra variable local, independientemente de que sea un `struct`.
- **El ámbito más reducido es el nivel de bloque: si el `struct` se declarara dentro de un bloque sólo podría usarse dentro de él.**

  La variable `punto_medio` es una variable local de la función `PuntoMedio` y se comporta como cualquier otra variable local, independientemente de que sea un `struct`. Lo mismo podría decirse si el `struct` se declarara dentro de un bloque: sólo podría usarse dentro de ese bloque.

```
......
while (hay_datos_por_procesar) {
   Punto2D  punto;

   // punto sólo es accesible dentro del ciclo while.
   // Se crea un struct nuevo en cada iteración y "desaparece"
   // el de la anterior.
   // Evítelo por la recarga computacional ocasionada
 }
```

IV.2.6.4. Inicialización de los campos de un struct

Un `struct` puede inicializarse en el momento de su declaración. El objetivo es asignar un valor inicial a sus campos evitando que pueda usarse con valores basura.

El procedimiento es muy simple, y debe tenerse en cuenta el orden en que fueron declarados los campos del `struct`. Los valores iniciales se especifican entre llaves, separados por comas.

Por ejemplo, para inicializar un `struct Persona` podríamos escribir:

```
Persona una_persona = {"", "", "", 'A', 0.0};
```

que asigna los valores *cadena vacía* a los campos de tipo `string` (`nombre`, `apellidos` y `NIF`), el carácter `A` al campo `categoria` y el valor `0.0` al campo `salario_bruto`.

IV.2.7. Datos miembro constantes

Recuperamos el ejemplo de la cuenta bancaria. Recuerde que el método `SetIdentificador` era privado, por lo que, una vez asignado un identificador en el constructor, éste no podía cambiarse.

```
int main(){
   CuentaBancaria cuenta("20310381450100007510", 3000);

   // La siguiente sentencia da error de compilación
   // El método SetIdentificador es private:

   cuenta.SetIdentificador("20310381450100007511");
```

Desde fuera, hemos conseguido que el identificador sea constante. Pero ahora vamos a obligar a que también sea constante dentro de la clase.

Vamos a definir datos miembros constantes, es decir, que se producirá un error en tiempo de compilación si incluimos una sentencia en algún método de la clase que pretenda modificarlos.

Tipos de dato miembro constantes:

- ▷ ***Constantes a nivel de objeto (object constants)*** **: Cada objeto de la clase tiene su propio valor de constante.**
- ▷ ***Constantes estáticas (static constants)*** **o *constantes a nivel de clase (class constants)* : Todos los objetos de una clase comparten el mismo valor de constante.**

IV.2.7.1. Constantes a nivel de objeto

- **Cada objeto de una misma clase tiene su propio valor de constante.**
- **Se declaran dentro de la clase anteponiendo `const` al nombre de la constante.**
- **El valor a asignar lo recibe como un parámetro más en el constructor. La inicialización debe hacerse en un la lista de inicialización del constructor. Son construcciones del tipo `<dato_miembro> (<valor inicial>)` separadas por coma. La lista de inicialización del constructor va antes del paréntesis del constructor, y con dos puntos al inicio.**

```
class MiClase{
private:
   const double CTE_REAL;
   const string CTE_STRING;
public:
   MiClase(double un_real, int un_string)
      :CTE_REAL(un_real) ,   // Correcto. Aquí se le asigna el valor
       CTE_STRING(un_string)
   {
      CTE_REAL = un_real;    // Error de compilación. No es el sitio
   }
   void UnMetodo(){
      CTE_REAL   = 0.0;      // Error de compilación. Es constante
      CTE_STRING = "Si";     // Error de compilación. Es constante
   }
   ......
};
int main(){
   MiClase un_objeto(6.3, "Si");   // CTE_REAL = 6.3, CTE_STRING = "Si"
   MiClase un_objeto(5.8, "No");   // CTE_REAL = 5.8, CTE_STRING = "No"
```

Realmente, en la lista de inicialización se pueden definir (dar un valor inicial) todos los datos miembros, no sólo las constantes a nivel de objeto.

```
class MiClase{
private:
   double dato_miembro;
public:
   ......
   MiClase(double parametro)
      :dato_miembro(parametro)
   { }
   ......
}
```

De hecho, esta es la forma *recomendada* en C++ de inicializar los datos miembro. Por comodidad, nosotros usaremos inicialización del dato miembro en el mismo lugar de la declaración, salvo lógicamente los datos miembros constantes, que es obligatorio hacerlo en la lista de inicialización.

Ejercicio. **Recupere la clase `CuentaBancaria` y defina el identificador como una constante.**

```
class CuentaBancaria{
private:
   double saldo;
   const string IDENTIFICADOR;
public:
   CuentaBancaria(string identificador_cuenta, double saldo_inicial)
      :IDENTIFICADOR(identificador_cuenta)
   {
      SetSaldo(saldo_inicial);
   }
   CuentaBancaria(string identificador_cuenta)
      :IDENTIFICADOR(identificador_cuenta)
   {
      SetSaldo(0.0);
   }
   string Identificador(){
      return IDENTIFICADOR;
   }
   ......
};
int main(){
   CuentaBancaria una_cuenta("20310087370100001345", 100);
   ......
```

IV.2.7.2. Constantes a nivel de clase

Todos los objetos de una clase comparten el mismo valor de constante.

El valor a asignar se indica en la inicialización del dato miembro.

En UML, las constantes estáticas se resaltan subrayándolas.

Constantes estáticas NO enteras

▷ **Se declaran dentro de la definición de la clase:**

```
class <nombre_clase>{
   ......
   static const <tipo> <nombre_cte>;
   ......
};
```

▷ **Se definen fuera de la definición de la clase:**

```
const <tipo> <nombre_clase>::<nombre_cte> = <valor>;
```

```
class MiClase{
private:
   static const double CTE_REAL_PRIVADA;
   ......
};
const double MiClase::CTE_REAL_PRIVADA = 6.7;

int main(){
   MiClase un_objeto;
```

:: es el *operador de resolución de ámbito (scope resolution operator)* . Se utiliza, en general, para poder realizar la declaración de un miembro dentro de la clase y su definición fuera. Esto permite la *compilación se-*

parada (separate compilation) **. Se verá con más detalle en el segundo cuatrimestre.**

Nota:

En c++11 también se permite declarar una constante estática de la siguiente forma:

```
static constexpr double CTE_REAL = 4.0;
```

Constantes estáticas enteras

▷ **O bien como antes:**

```
class MiClase{
private:
   static const int CTE_ENTERA_PRIVADA;
   ......
};
const int MiClase::CTE_ENTERA_PRIVADA = 6;
```

▷ **O bien se definen en el mismo sitio de la declaración:**

```
class MiClase{
private:
   static const int CTE_ENTERA_PRIVADA = 4;
   ......
};
```

En el siguiente apartado vamos a necesitar las constantes estáticas enteras para dimensionar vectores dentro de una clase.

IV.2.8. Vectores y objetos

A lo largo de la asignatura seremos capaces de:

- ▷ **Incluir un vector como un dato miembro de un objeto.**
- ▷ **Definir un vector dentro de un método de un objeto.**
- ▷ **Definir un vector en el que cada componente sea un objeto.**

Empezamos viendo los dos primeros puntos y posteriormente trataremos el último.

IV.2.8.1. Los vectores como datos miembro

Los vectores de C++ son potentes pero pueden provocar errores de ejecución si accedemos a componentes no reservadas. Por ello, definiremos nuestras propias clases para trabajar con vectores de forma segura.

- ▷ **Para usar un vector clásico como dato miembro de una clase, debemos dimensionarlo o bien con un literal o bien con una constante global definida fuera de la clase o bien con una constante definida dentro de la clase (en este caso, debe ser estática)**

  En lo que sigue fomentaremos el uso de *constantes estáticas*.

- ▷ **Cuando se *cree* un objeto, se creará automáticamente el vector. Como cualquier dato miembro, el vector existirá mientras exista el objeto.**

- ▷ **Por ahora trabajamos con vectores de tipos simples. En el próximo tema veremos los vectores de objetos.**

```
class MiClase{
private:
   static const int TAMANIO = 30;
   char vector[TAMANIO];
   ......
};
int main(){
   MiClase objeto;  // Se crea una zona de memoria reservada
                    // para el objeto. Éste contiene un
                    // vector privado con 30 componentes
```

Dentro de la clase definiremos los métodos que acceden a las componentes del vector. Éstos establecerán la política de acceso a dichas componentes, como por ejemplo, si permitimos modificar cualquier componente en cualquier momento o si sólo permitimos añadir.

Ejercicio. **Retomamos el ejemplo del autobús de la página 308 y construimos la clase** `Autobus`

- **El nombre del conductor será un parámetro a pasar en el constructor.**
- **Definimos un método** `Situa` **indicando el nombre del pasajero y el del asiento que se le va a asignar:**

```
void Situa(int asiento, string nombre_pasajero)
```

Autobus	
- const int	MAX_PLAZAS
- string	pasajeros[MAX_PLAZAS]
- int	numero_actual_pasajeros
+	Autobus(string nombre_conductor)
+ void	Situa(int asiento, string nombre_pasajero)
+ string	Pasajero(int asiento)
+ string	Conductor()
+ bool	Ocupado(int asiento)
+ int	Capacidad()
+ int	NumeroActualPasajeros()

Esta clase se usará en la siguiente forma:

```
int main(){
   ......
   Autobus alsa(conductor);
   ......
   while (hay_datos_por_leer){
      alsa.Situa(asiento, nombre_pasajero);
      ......
   }
```

Veamos el programa completo:

```
int main(){
   // En la siguiente versión las cadenas introducidas
   // no deben tener espacios en blanco

   const string TERMINADOR = "-";
   string conductor,
          nombre_pasajero;
   int    asiento,
          capacidad_autobus;

   cout << "Autobús.\n";
   cout << "\nIntroduzca nombre del conductor: ";
   cin >> conductor;

   Autobus alsa(conductor);

   cout << "\nIntroduzca los nombres de los pasajeros y su asiento."
        << "Termine con " << TERMINADOR << "\n";
   cout << "\nNombre: ";
   cin >> nombre_pasajero;

   while (nombre_pasajero != TERMINADOR){
      cout << "Asiento: ";
      cin >> asiento;

      alsa.Situa(asiento, nombre_pasajero);

      cout << "\nNombre: ";
      cin >> nombre_pasajero;
   }

   cout << "\nTotal de pasajeros: " << alsa.NumeroActualPasajeros();
   cout << "\nConductor: " << alsa.Conductor() << "\n";
```

```
   capacidad_autobus = alsa.Capacidad();

   for (int i=1; i < capacidad_autobus; i++){
      cout << "\nAsiento número: " << i;

      if (alsa.Ocupado(i))
         cout << " Pasajero: " << alsa.Pasajero(i);
      else
         cout << " Vacío";
   }
}
```

Implementamos ahora la clase. Observe cómo ocultamos información al situar dentro de la clase las constantes `MAX_PLAZAS` y `VACIO`.

```
class Autobus{
private:
   const string VACIO = "";
   static const int MAX_PLAZAS = 50;
   string pasajeros[MAX_PLAZAS] = {VACIO};
   int numero_actual_pasajeros;
public:
   Autobus(string nombre_conductor){
      pasajeros[0] = nombre_conductor;
      numero_actual_pasajeros = 0; // El conductor no cuenta como pasajero

      for (int i=1; i < MAX_PLAZAS; i++)
         pasajeros[i] = VACIO;
   }
   void Situa(int asiento, string nombre_pasajero){
      if (asiento >= 1 && asiento < MAX_PLAZAS){
         if (! Ocupado(asiento)){
            pasajeros[asiento] = nombre_pasajero;
```

```
            numero_actual_pasajeros++;
         }
      }
   }
   // Prec: 0 < asiento < MAX_PLAZAS
   string Pasajero(int asiento){
      return pasajeros[asiento];
   }
   string Conductor(){
      return pasajeros[0];
   }
   // Prec: 0 < asiento < MAX_PLAZAS
   bool Ocupado(int asiento){
      return pasajeros[asiento] != VACIO;
   }
   int Capacidad(){
      return MAX_PLAZAS;
   }
   int NumeroActualPasajeros(){
      return numero_actual_pasajeros;
   }
};
```

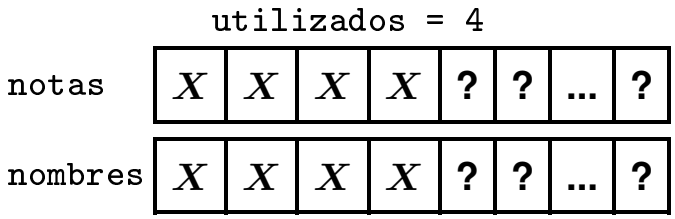
http://decsai.ugr.es/jccubero/FP/IV_autobus_clases.cpp

Ampliación:

Si se desea, el método **Situa** podría devolver un **bool** indicando si se ha podido situar correctamente al pasajero. En cualquier caso, si queremos notificar problemas durante la ejecución de un método es mejor hacerlo con el mecanismo de las excepciones tal y como se indica en la página 530

Ejemplo. **Retomamos el ejemplo de la página 306 creando la clase** `CalificacionesFinales`**.**

- ▷ **Añadiremos el nombre del alumno junto con su nota.**
- ▷ **Internamente usaremos un vector para almacenar los nombres y otro vector para las notas.**
- ▷ **Al contrario del ejemplo del autobús, no dejaremos huecos, sólo permitimos añadir datos y se irán llenando las componentes desde el inicio.**



Para forzar esta situación, la única forma de insertar nuevos datos será a través del siguiente método:

```
void Aniade(string nombre_alumno, double nota_alumno)
```

Observe que no pasamos como parámetro el índice de la componente a modificar. El método `Aniade` **siempre añade al final.**

- ▷ **Una vez añadido un alumno con su nota, no se puede borrar posteriormente.**

CalificacionesFinales	
- const int	MAX_ALUMNOS
- double	notas[MAX_ALUMNOS]
- string	nombres[MAX_ALUMNOS]
- int	utilizados
+	Notas()
+ int	Capacidad()
+ int	TotalAlumnos()
+ double	CalificacionAlumno(int posicion)
+ string	NombreAlumno(int posicion)
+ void	Aniade(string nombre_alumno, double nota_alumno)
+ double	MediaGlobal()
+ int	SuperanMedia()

Esta clase se usará en la siguiente forma:

```
int main(){
   Notas notas_FP;
   ......

   while (hay_datos_por_leer){
      ......
      notas_FP.Aniade(nombre, nota);
   }
```

Veamos el programa completo:

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

using namespace std;

class CalificacionesFinales{
private:
   static const int MAX_ALUMNOS = 100;
   double notas[MAX_ALUMNOS];
```

```
   string nombres[MAX_ALUMNOS];
   int utilizados;
public:
   CalificacionesFinales()
    :utilizados(0){
   }
   int Capacidad(){
      return MAX_ALUMNOS;
   }
   int TotalAlumnos(){
      return utilizados;
   }
   double CalificacionAlumno(int posicion){
      return notas[posicion];
   }
   string NombreAlumno(int posicion){
      return nombres[posicion];
   }
   void Aniade(string nombre_alumno, double nota_alumno){
      if (utilizados < MAX_ALUMNOS){
         notas[utilizados] = nota_alumno;
         nombres[utilizados] = nombre_alumno;
         utilizados++;
      }
   }
   double MediaGlobal(){
      double media = 0;

      for (int i = 0; i < utilizados; i++)
         media = media + notas[i];

      media = media / utilizados;
```

```
      return media;
   }
   int SuperanMedia(){
      double media = MediaGlobal();
      int superan_media = 0;

      for (int i = 0; i < utilizados; i++){
         if (notas[i] > media)
            superan_media++;
      }

      return superan_media;
   }
};
int main(){
   const string TERMINADOR = "-";
   int capacidad;
   string nombre;
   double nota, media, superan_media;
   CalificacionesFinales notas_FP;

   cout << "Introduzca nombre de alumno y su nota. "
        << TERMINADOR << " para terminar\n";

   capacidad = notas_FP.Capacidad();
   cin >> nombre;

   while (nombre != TERMINADOR && notas_FP.TotalAlumnos() < capacidad){
      cin >> nota;
      notas_FP.Aniade(nombre, nota);
      cin >> nombre;
   }
```

```
   media = notas_FP.MediaGlobal();
   superan_media = notas_FP.SuperanMedia();

   cout << "Media Aritmetica = " << media << "\n";
   cout << superan_media << " alumnos han superado la media\n";
}
```

http://decsai.ugr.es/jccubero/FP/IV_notas_clases.cpp

IV.2.8.2. Comprobación de las precondiciones trabajando con vectores

Recuerde lo indicado en la página 526:

Si la violación de una precondición de un método *público* puede provocar errores de ejecución graves, dicha precondición debe comprobarse dentro del método. En otro caso, puede omitirse.

Al trabajar con un vector dato miembro de una clase, si un método modifica componentes de dicho vector, entonces comprobaremos las precondiciones dentro del método.

Ejemplo. **En el ejemplo del autobús, hemos comprobado la precondición en el método `Situa` ya que si se viola y pasamos como parámetro un índice de asiento fuera del rango admitido, se podría modificar una zona de memoria no reservada, con consecuencias imprevisibles.**

▷ **Versión sin comprobar la precondición:**

```
// Prec: 0 < asiento < MAX_PLAZAS

void Situa(int asiento, string nombre_pasajero){
   if (! Ocupado(asiento)){
      pasajeros[asiento] = nombre_pasajero;
      numero_actual_pasajeros++;
   }
}
```

▷ **Versión comprobando la precondición:**

```
void Situa(int asiento, string nombre_pasajero){

   if (asiento >= 1 && asiento < MAX_PLAZAS){
      if (! Ocupado(asiento)){
         pasajeros[asiento] = nombre_pasajero;
         numero_actual_pasajeros++;
      }
   }
}
```

Por contra, si el índice pasado a `Pasajero` no es correcto, simplemente devolverá un valor basura, pero no corremos el peligro de modificar una componente no reservada. Por lo tanto, podemos evitar la comprobación de la precondición dentro del método `Pasajero`. Lo mismo ocurre con el método `Ocupado`.

En cualquier caso, recordemos que si accedemos a una componente no reservada, aunque no se modifique, el programa puede terminar ya que, en estos casos, C++ indica que el comportamiento es indeterminado.

Ejemplo. **En el ejemplo de las notas, hemos comprobado la precondición en el método `Aniade` ya que, si se viola e intentamos añadir por encima de la zona reservada, las consecuencias son imprevisibles.**

- **Versión sin comprobar la precondición:**

```
// Prec: 0 <= utilizados < MAX_ALUMNOS

void Aniade(string nombre_alumno, double calificacion_alumno){
   nota[utilizados] = calificacion_alumno;
   nombre[utilizados] = nombre_alumno;
   utilizados++;
}
```

- **Versión comprobando la precondición:**

```
void Aniade(string nombre_alumno, double calificacion_alumno){
   if (utilizados < MAX_ALUMNOS){
      nota[utilizados] = calificacion_alumno;
      nombre[utilizados] = nombre_alumno;
      utilizados++;
   }
}
```

Al no modificar componentes, hemos decidido no comprobar las precondiciones en los métodos `CalificacionAlumno` y `NombreAlumnno`

IV.2.8.3. Los vectores como datos locales de un método

- ▷ **Un vector puede ser un dato local de un método.**
- ▷ **Debe declararse con una constante definida dentro del método o en un ámbito superior.**
- ▷ **El vector se creará en la pila, en el momento de ejecutarse el método. Cuando éste termine, se libera la memoria asociada al vector. Lo mismo ocurre si fuese local a una función.**
- ▷ **Si se necesita inicializar el vector, se hará tal y como se indicó en la página 298**

```
class MiClase{
   ......
   void Metodo(){
      const int TAMANIO = 30;  // static cuando es dato miembro
      char vector[TAMANIO];
      ......
   }
};
int main(){
   MiClase objeto;
   objeto.Metodo()  // <- Se crea el vector local con 30 componentes
   ......           // Terminado el método, se destruye el vector local
```

Las funciones o métodos pueden definir vectores como datos locales para hacer sus cómputos. Pero no pueden devolverlos.

Lo que sí podremos hacer es devolver objetos que contienen vectores.

Ejemplo. Clase `Fecha`. Este ejemplo ya se vio en la página 499

```
class Fecha{
private:
   int dia, mes, anio;

   bool EsCorrecta(int el_dia, int el_mes, int el_anio){
      bool es_bisiesto;
      bool es_fecha_correcta;
      const int anio_inferior = 1900;
      const int anio_superior = 2500;
      const int dias_por_mes[12] =
            {31,28,31,30,31,30,31,31,30,31,30,31};

            // Meses de Enero a Diciembre

      es_fecha_correcta =  1 <= el_dia &&
                           el_dia <= dias_por_mes[el_mes - 1] &&
                           1 <= el_mes   &&   el_mes <= 12 &&
                           anio_inferior <= el_anio   &&
                           el_anio <= anio_superior;

      es_bisiesto = (el_anio % 4 == 0 && el_anio % 100 != 0) ||
                     el_anio % 400 == 0;

      if (!es_fecha_correcta && el_mes == 2 && el_dia == 29 && es_bisiesto)
         es_fecha_correcta = true;

      return es_fecha_correcta;
   }
   ......
```

IV.2.8.4. Los vectores como parámetros a un método

Los vectores clásicos que estamos viendo pueden pasarse como parámetros a una función o método. Sin embargo, no lo veremos en esta asignatura porque:

- **En C++, al pasar un vector como parámetro, únicamente se pasa una copia de la dirección de memoria inicial de las componentes, por lo que desde dentro de la función o método podemos modificarlas.**

  Esto rompe el concepto de *paso de parámetro por valor*.

- **Los vectores son una herramienta potente pero peligrosa si no se usan con cuidado. Por ello, se pretende acostumbrar al alumno a que construya sus propias clases si necesita almacenar varios valores. Serán los objetos de estas clases los que pasaremos como parámetro a otros métodos.**

En esta asignatura no pasaremos los vectores como parámetros de las funciones o métodos.

IV.2.9. La clase Secuencia de caracteres

IV.2.9.1. Métodos básicos

Ya sabemos que hay que trabajar con cuidado con los vectores con corchetes. Nos gustaría crear una clase genérica `MiVector` que protegiese adecuadamente las componentes de una incorrecta manipulación.

C++ proporciona clases específicas para trabajar de forma segura con vectores genéricos de datos como por ejemplo la plantilla `vector` de la `STL`. Se verán en el segundo cuatrimestre.

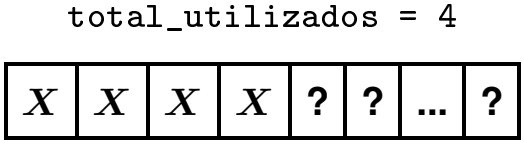
Con las herramientas que conocemos hasta ahora, vamos a tener que crear una clase para cada tipo de dato que queramos manejar:

```
MiVectorEnteros, MiVectorDoubles, etc
```

En lo que sigue nos ceñiremos al tipo de dato `char`. Queremos construir una clase `SecuenciaCaracteres` de forma que:

- ▷ **Represente una secuencia consecutiva de caracteres sin huecos.**
- ▷ **Internamente, usaremos un vector de tipo `char`:**
- ▷ **Los métodos de la interfaz pública de la clase protegerán convenientemente el acceso a las componentes del vector.**

  Por lo tanto, no permitiremos operaciones (como añadir o modificar componentes) que impliquen dejar huecos.



Una primera versión:

SecuenciaCaracteres	
- const int	TAMANIO
- char	vector_privado[TAMANIO]
- int	total_utilizados
+	SecuenciaCaracteres()
+ int	TotalUtilizados()
+ int	Capacidad()
+ void	Aniade(char nuevo)
+ void	Modifica(int indice, char nuevo)
+ char	Elemento(int indice)

- ▷ `TotalUtilizados` **devuelve el número de componentes ocupadas.**
- ▷ `Capacidad` **devuelve el tamaño del vector (número de componentes reservadas en memoria)**
- ▷ `Aniade` **añade al final un valor más.**
- ▷ `Modifica` **modifica el valor de una componente ya existente (previamente añadida a través del método** `Aniade`**).**

  Los métodos `Aniade` **y** `Modifica` **pueden modificar componentes del vector. Para impedir que se acceda a componentes no reservadas, comprobaremos la precondición de que la posición esté en el rango correcto.**
- ▷ `Elemento` **devuelve el valor de la componente en el índice señalado.**

Un ejemplo de uso de esta clase:

```
int main(){
   SecuenciaCaracteres secuencia;
   int num_elementos;

   secuencia.Aniade('h');   // -> h
   secuencia.Aniade('o');   // -> ho
   secuencia.Aniade('l');   // -> hol
   secuencia.Aniade('a');   // -> hola

   num_elementos = secuencia.TotalUtilizados()

   for (int i = 0; i < num_elementos; i++)
      cout << secuencia.Elemento(i) << " ";

   secuencia.vector_privado[1] = 'g';  // Error compilac. Dato privado.
   secuencia.vector_privado[70] = 'g'; // Error compilac. Dato privado.

   secuencia.Modifica(9,'l');  // -> hola   Acceso incorrecto
   secuencia.Modifica(0,'l');  // -> lola
```

Definamos los métodos:

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int total_utilizados;
public:
   SecuenciaCaracteres()
      :total_utilizados(0) {
   }
   int TotalUtilizados(){
      return total_utilizados;
   }
   int Capacidad(){
      return TAMANIO;
   }
   void Aniade(char nuevo){
      if (total_utilizados < TAMANIO){
         vector_privado[total_utilizados] = nuevo;
         total_utilizados++;
      }
   }
   void Modifica(int posicion, char nuevo){
      if (posicion >= 0 && posicion < total_utilizados)
         vector_privado[posicion] = nuevo;
   }
   char Elemento(int indice){
      return vector_privado[indice];
   }
};
```

http://decsai.ugr.es/jccubero/FP/IV_secuencia_base.cpp

¿Debería añadirse el siguiente método?

```
class SecuenciaCaracteres{
   ......
   void SetTotalUtilizados(int nuevo_total){
      total_utilizados = nuevo_total;
   }
};
```

Obviamente no. La gestión de `total_utilizados` es responsabilidad de la clase, y ha de actualizarse automáticamente dentro de la clase y no de forma manual. Un par de excepciones se ven en los siguientes ejemplos.

Ejercicio. **Definir un método para *borrar* todos los caracteres.**

```
class SecuenciaCaracteres{
   ......
   void EliminaTodos(){
      total_utilizados = 0;
   }
};
```

Ejercicio. **Definir un método para *borrar* el último carácter.**

```
class SecuenciaCaracteres{
   ......
   void EliminaUltimo(){
      total_utilizados--;
   }
};
```

Ejercicio. **Incluso sería aceptable definir un método para *truncar* a partir de una posición. Pero siempre sin dejar huecos.**

```
class SecuenciaCaracteres{
   ......
   void Trunca(int nuevo_total){
      if (0 <= nuevo_total && nuevo_total < total_utilizados)
         total_utilizados = nuevo_total;
   }
};
```

Si queremos una secuencia de reales:

```
class SecuenciaDoubles{
private:
   static const int TAMANIO = 50;
   double vector_privado[TAMANIO];
   int total_utilizados;
public:
   SecuenciaDoubles()
      :total_utilizados(0)   {  }

   < Los métodos TotalUtilizados() y Capacidad() no varían >

   void Aniade(double nuevo){
      if (total_utilizados < TAMANIO){
         vector_privado[total_utilizados] = nuevo;
         total_utilizados++;
      }
   }
   double Elemento(int indice){
      return vector_privado[indice];
   }
};
```

Los lenguajes de programación ofrecen recursos para no tener que escribir el mismo código para tipos distintos:

- ▷ ***Plantillas (Templates)*** **en C++**
- ▷ ***Genéricos (Generics)*** **en Java y .NET**

Por ahora, tendremos que duplicar el código y crear una clase para cada tipo de dato. 

Veamos cómo quedarían implementados los métodos de búsqueda vistos en la sección III.2.1 dentro de la clase `SecuenciaCaracteres`:

IV.2.9.2. Métodos de búsqueda

Búsqueda secuencial

¿Devolvemos un `bool`? Mejor si devolvemos la posición en la que se ha encontrado. En caso contrario, devolvemos un valor imposible de posición (por ejemplo, -1)

***Cabecera*:**

```
class SecuenciaCaracteres{
   ......
   int PrimeraOcurrencia (char buscado){ ...... }
};
```

***Llamada*:**

```
int main(){
   SecuenciaCaracteres secuencia;
   ......
   pos_encontrado = secuencia.PrimeraOcurrencia('l');

   if (pos_encontrado == -1)
      cout << "\nNo encontrado";
   else
      cout << "\nEncontrado en la posición " << pos_encontrado;
}
```

Implementación:

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int total_utilizados;
public:
   ......
   int PrimeraOcurrencia (char buscado){
      bool encontrado = false;
      int i = 0;

      while (i < total_utilizados  &&  !encontrado){
         if (vector_privado[i] == buscado)
            encontrado = true;
         else
            i++;
      }

      if (encontrado)
         return i;
      else
         return -1;
   }
};
```

Observe que también podríamos haber puesto lo siguiente:

```
   int PrimeraOcurrencia (char buscado){
      .....
      while (i < total_utilizados  &&  !encontrado)
         if (Elemento(i) == buscado)
      ....
   }
};
```

- **La llamada al método `Elemento` es más *segura* ya que dentro de él podemos comprobar la condición de que el índice sea correcto.**
- **Por otra parte, la llamada conlleva una recarga computacional. Es más rápido acceder directamente a la componente del vector privado. No sería significativa si sólo se realizase una llamada, pero está dentro de un bucle, por lo que la recarga sí puede ser importante.**

Por tanto, normalmente optaremos por trabajar dentro de los métodos de la clase accediendo directamente al vector privado con la notación corchete.

> ***Consejo*:** *Procure implementar los métodos de una clase lo más eficientemente posible, pero sin que ello afecte a ningún cambio en la interfaz de ésta (las cabeceras de los métodos públicos)*

A veces, podemos estar interesados en buscar entre una componente izquierda y otra derecha, ambas inclusive.

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int total_utilizados;
public:
   ....................................
   int PrimeraOcurrenciaEntre (int pos_izda, int pos_dcha, char buscado){
      int i = pos_izda;
      bool encontrado = false;

      while (i <= pos_dcha  &&  !encontrado)
         if (vector_privado[i] == buscado)
            encontrado = true;
         else
            i++;

      if (encontrado)
         return i;
      else
         return -1;
   }
};
```

La clase puede proporcionar los dos métodos `PrimeraOcurrencia` y `PrimeraOcurrenciaEntre`. Pero para no repetir código, cambiaríamos la implementación del primero como sigue:

```
int PrimeraOcurrencia (char buscado){
   return PrimeraOcurrenciaEntre (0, total_utilizados - 1, buscado);
}
```

Hasta ahora, nos quedaría:

SecuenciaCaracteres	
- const int	TAMANIO
- char	vector_privado[TAMANIO]
- int	total_utilizados
+	SecuenciaCaracteres()
+ int	TotalUtilizados()
+ int	Capacidad()
+ void	Aniade(char nuevo)
+ void	Modifica(int pos_a_modificar, char valor_nuevo)
+ char	Elemento(int indice)
+ void	EliminaTodos()
+ int	PrimeraOcurrenciaEntre(int pos_izda, int pos_dcha, char buscado)
+ int	PrimeraOcurrencia(char buscado)

Búsqueda binaria

***Precondiciones de uso*: Se aplica sobre un vector ordenado.**

No comprobamos que se cumple la precondición dentro del método ya que resultaría muy costoso comprobar que el vector está ordenado cada vez que ejecutamos la búsqueda.

La cabecera del método no cambia, aunque para destacar que no es una búsqueda cualquiera, sino que necesita una precondición, cambiamos el nombre del identificador.

***Cabecera*:**

```
class SecuenciaCaracteres{
   ......
   int BusquedaBinaria (char buscado){ ...... }
};
```

***Llamada*:**

```
int main(){
   SecuenciaCaracteres secuencia;
   ......
   pos_encontrado = secuencia.BusquedaBinaria('B');

   if (pos_encontrado == -1)
      cout << "\nNo encontrado";
   else
      cout << "\nEncontrado en la posición " << pos_encontrado;

}
```

Implementación:

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int  total_utilizados;
public:
   ......
   int BusquedaBinaria (char buscado){
      int izda, dcha, centro;
      bool encontrado = false;

      izda = 0;
      dcha = total_utilizados - 1;

      while (izda <= dcha  &&  !encontrado){
         centro = (izda + dcha) / 2;

         if (vector_privado[centro] == buscado)
            encontrado = true;
         else if (buscado < vector_privado[centro])
            dcha = centro - 1;
         else
            izda = centro + 1;
      }

      if (encontrado)
         return centro;
      else
         return -1;
   } };
```

Buscar el mínimo

¿Devolvemos el mínimo de la secuencia? Mejor si devolvemos su posición por si luego quisiéramos acceder a la componente.

> ***Como norma general, cuando un método busque algo en un vector, debe devolver la posición en la que se encuentra y no el elemento en sí.***

***Cabecera*:**

```
class SecuenciaCaracteres{
   ......
   int PosicionMinimo() { ...... }
};
```

***Llamada*:**

```
int main(){
   SecuenciaCaracteres secuencia;
   ......
   pos_minimo = secuencia.PosicionMinimo();
```

Implementación:

```
class SecuenciaCaracteres{
   ......
   int PosicionMinimo(){
      int posicion_minimo;
      char minimo;

      if (total_utilizados > 0){
         minimo = vector_privado[0];
         posicion_minimo = 0;

         for (int i = 1; i < total_utilizados ; i++){
            if (vector_privado[i] < minimo){
               minimo = vector_privado[i];
               posicion_minimo = i;
            }
         }
      }
      else
         posicion_minimo = -1;

      return posicion_minimo;
   }
};
```

Buscar el mínimo en una zona del vector

Debemos pasar como parámetro al método los índices `izda` y `dcha` que delimitan la zona en la que se quiere buscar.

`v = (h,b,t,c,f,i,d,f,?,?,?)  izda = 2 , dcha = 5`

`PosicionMinimoEntre(2, 5)` **devolvería la posición** `3` **(índice del vector)**

***Cabecera*:**

```
class SecuenciaCaracteres{
   .......
   // Precond: 0 <= izda <= dcha < total_utilizados
   int PosicionMinimoEntre(int izda, int dcha){ ...... }
};
```

***Llamada*:**

```
int main(){
   SecuenciaCaracteres secuencia;
   ......
   pos_minimo = secuencia.PosicionMinimoEntre(1,3);
```

Implementación:

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int total_utilizados;
public:
   ........
   // Precond: 0 <= izda <= dcha < total_utilizados
   int PosicionMinimoEntre(int izda, int dcha){
      int posicion_minimo = -1;
      char minimo;

      minimo = vector_privado[izda];
      posicion_minimo = izda;

      for (int i = izda+1 ; i <= dcha ; i++)
         if (vector_privado[i] < minimo){
            minimo = vector_privado[i];
            posicion_minimo = i;
         }

      return posicion_minimo;
   }
};
```

Al igual que hicimos con el método `PrimeraOcurrencia`, la implementación de `PosicionMinimo` habría que cambiarla por:

```
   int PosicionMinimo(){
      return PosicionMinimoEntre(0, total_utilizados - 1);
   }
```

Nos quedaría por ahora:

SecuenciaCaracteres	
- const int	TAMANIO
- char	vector_privado[TAMANIO]
- int	total_utilizados
+	SecuenciaCaracteres()
+ int	TotalUtilizados()
+ int	Capacidad()
+ void	Aniade(char nuevo)
+ void	Modifica(int pos_a_modificar, char valor_nuevo)
+ char	Elemento(int indice)
+ void	EliminaTodos()
+ int	PrimeraOcurrenciaEntre(int pos_izda, int pos_dcha, char buscado)
+ int	PrimeraOcurrencia(char buscado)
+ int	BusquedaBinaria(char buscado)
+ int	PosicionMinimo()
+ int	PosicionMinimoEntre(int izda, int dcha)

http://decsai.ugr.es/jccubero/FP/IV_secuencia_busqueda.cpp

Inserción de un valor

La inserción de un valor dentro del vector implica desplazar las componentes que hay a su derecha. Debemos comprobar que los accesos son correctos para evitar modificar componentes no reservadas.

***Cabecera*:**

```
class SecuenciaCaracteres{
   ......
   void Inserta(int pos_insercion, char valor_nuevo){ ...... }
};
```

***Llamada*:**

```
int main(){
   SecuenciaCaracteres secuencia;
   ......
   secuencia.Inserta(2, 'r');
```

Implementación:

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int total_utilizados;
public:
   ......
   void Inserta(int pos_insercion, char valor_nuevo){
      if (total_utilizados < TAMANIO  &&  pos_insercion >= 0
         &&  pos_insercion <= total_utilizados){

         for (int i = total_utilizados ; i > pos_insercion ; i--)
            vector_privado[i] = vector_privado[i-1];

         vector_privado[pos_insercion] = valor_nuevo;
         total_utilizados++;
      }
   }
```

Eliminación de un valor

Debemos pasar como parámetro la posición a eliminar.

***Cabecera*:**

```
class SecuenciaCaracteres{
   ......
   void Elimina(int pos_a_eliminar){
      ......
   }
};
```

***Llamada*:**

```
int main(){
   SecuenciaCaracteres secuencia;
   ......
   secuencia.Elimina(2);
```

Implementación:

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int total_utilizados;
public:
   ..........
   // Elimina una componente, dada por su posición
   void Elimina (int posicion){
      if (posicion >= 0 && posicion < total_utilizados){
         int tope = total_utilizados-1;

         for (int i = posicion ; i < tope ; i++)
            vector_privado[i] = vector_privado[i+1];

         total_utilizados--;
      }
   }
};
```

Nos quedaría por ahora:

SecuenciaCaracteres	
- const int	TAMANIO
- char	vector_privado[TAMANIO]
- int	total_utilizados
+	SecuenciaCaracteres()
+ int	TotalUtilizados()
+ int	Capacidad()
+ void	Aniade(char nuevo)
+ void	Modifica(int pos_a_modificar, char valor_nuevo)
+ char	Elemento(int indice)
+ void	EliminaTodos()
+ int	PrimeraOcurrenciaEntre(int pos_izda, int pos_dcha, char buscado)
+ int	PrimeraOcurrencia(char buscado)
+ int	BusquedaBinaria(char buscado)
+ int	PosicionMinimoEntre(int izda, int dcha)
+ int	PosicionMinimo()
+ void	Inserta(int pos_insercion, char valor_nuevo)
+ void	Elimina(int pos_a_eliminar)

http://decsai.ugr.es/jccubero/FP/IV_secuencia_inserta_elimina.cpp

IV.2.9.3. Llamadas entre métodos

Ejercicio. **Borre el elemento mínimo de un vector.**

¿Construimos el siguiente método?

```
void EliminaMinimo()
```

Lo normal será que no lo hagamos ya que es mejor definir el método

```
void Elimina (int posicion_a_eliminar)
```

y realizar la siguiente llamada:

```
int main(){
   SecuenciaCaracteres cadena;
   ......
   cadena.Elimina( cadena.PosicionMinimo() );
```

A la hora de diseñar la interfaz de una clase (el conjunto de métodos públicos), intente construir un conjunto minimal de operaciones primitivas que puedan reutilizarse lo máximo posible.

En cualquier caso, si se prevé que la eliminación del mínimo va a ser una operación muy usual, puede añadirse como un método de la clase, pero eso sí, en la implementación se llamará a los otros métodos.

```
   void EliminaMinimo(){
      Elimina( PosicionMinimo() );
   }
```

IV.2.9.4. Algoritmos de ordenación

Se usarán para ordenar los caracteres según el orden de la tabla ASCII. Obviamente, sólo tendrá sentido en ciertas situaciones. Por ejemplo, si representamos las notas según una escala ECTS (con letras) estamos interesados en ordenar dichas notas. Por contra, habrá pocos problemas que necesiten ordenar los caracteres del nombre de una persona, por ejemplo.

***Cabecera*:**

```
class SecuenciaCaracteres{
   ......
   void Ordena(){ ...... }
};
```

***Llamada*:**

```
int main(){
   SecuenciaCaracteres secuencia;
   ......
   secuencia.Ordena();
```

Para diferenciar los distintos algoritmos de ordenación usaremos nombres de métodos distintos.

Es importante que observe el uso de otros métodos de la clase, para implementar los distintos algoritmos de ordenación.

Ordenación por selección

Implementación:

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int total_utilizados;

   void IntercambiaComponentes_en_Posiciones(int pos_izda, int pos_dcha){
      char intercambia;

      intercambia = vector_privado[pos_izda];
      vector_privado[pos_izda] = vector_privado[pos_dcha];
      vector_privado[pos_dcha] = intercambia;
   }
   int PosicionMinimoEntre(int izda, int dcha){
      ......
   }
   .......
public:
   void Ordena_por_Seleccion(){
      int pos_min;

      for (int izda = 0 ; izda < total_utilizados ; izda++){
         pos_min = PosicionMinimoEntre(izda, total_utilizados-1);
         IntercambiaComponentes_en_Posiciones(izda, pos_min);
      }
   }
   ......
```

Observe que no hemos comprobado las precondiciones en el método privado `IntercambiaComponentes_en_Posiciones`. Esto lo hemos hecho para aumentar la eficiencia de los métodos públicos que lo usen (por ejemplo, los métodos de ordenación) Si quisiéramos ofrecer un método público que hiciese lo mismo, sí habría que comprobar las precondiciones.

> ***Para aumentar la eficiencia, a veces será conveniente omitir la comprobación de las precondiciones en los métodos privados.***

Observe la diferencia de este consejo sobre los métodos privados con el que vimos en la página 555 sobre los métodos públicos.

Ordenación por inserción

Implementación:

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int total_utilizados;
public:
   ..............
   void Ordena_por_Insercion(){
      int izda, i;
      char a_desplazar;

      for (izda = 1; izda < total_utilizados; izda++){
         a_desplazar = vector_privado[izda];

         for (i = izda; i > 0 && a_desplazar < vector_privado[i-1]; i--)
            vector_privado[i] = vector_privado[i-1];

         vector_privado[i] = a_desplazar;
      }
   }
};
```

Ordenación por burbuja

▶ *Primera Aproximación*

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int total_utilizados;

   void IntercambiaComponentes_en_Posiciones(int pos_izda, int pos_dcha){
      char intercambia;

      intercambia = vector_privado[pos_izda];
      vector_privado[pos_izda] = vector_privado[pos_dcha];
      vector_privado[pos_dcha] = intercambia;
   }
public:
  ..............
  void Ordena_por_Burbuja(){
      int izda, i;

      for (izda = 0; izda < total_utilizados; izda++)
        for (i = total_utilizados-1 ; i > izda ; i--)
          if (vector_privado[i] < vector_privado[i-1])
             IntercambiaComponentes_en_Posiciones(i, i-1);
   }
};
```

▶ *Segunda Aproximación*

```
class SecuenciaCaracteres{
private:
   static const int TAMANIO = 50;
   char vector_privado[TAMANIO];
   int total_utilizados;

   void IntercambiaComponentes_en_Posiciones(int pos_izda, int pos_dcha){
       ..............
   }
public:
   ..............
   void Ordena_por_BurbujaMejorado(){
      int izda, i;
      bool cambio;

      cambio = true;

      for (izda = 0; izda < total_utilizados && cambio; izda++){
        cambio = false;

        for (i = total_utilizados-1;  i > izda; i--)
           if (vector_privado[i] < vector_privado[i-1]){
             IntercambiaComponentes_en_Posiciones(i, i-1);
             cambio = true;
           }
      }
   }
};
```

Nos quedaría por ahora:

SecuenciaCaracteres	
- const int	TAMANIO
- char	vector_privado[TAMANIO]
- int	total_utilizados
- void	IntercambiaComponentes_en_Posiciones(int pos_izda, int pos_dcha)
+	SecuenciaCaracteres()
+ int	TotalUtilizados()
+ int	Capacidad()
+ void	Aniade(char nuevo)
+ void	Modifica(int pos_a_modificar, char valor_nuevo)
+ char	Elemento(int indice)
+ void	EliminaTodos()
+ int	PrimeraOcurrenciaEntre(int pos_izda, int pos_dcha, char buscado)
+ int	PrimeraOcurrencia(char buscado)
+ int	BusquedaBinaria(char buscado)
+ int	PosicionMinimoEntre(int izda, int dcha)
+ int	PosicionMinimo()
+ void	Inserta(int pos_insercion, char valor_nuevo)
+ void	Elimina(int pos_a_eliminar)
+ void	Ordena_por_Seleccion()
+ void	Ordena_por_Insercion()
+ void	Ordena_por_Burbuja()
+ void	Ordena_por_BurbujaMejorado()

http://decsai.ugr.es/jccubero/FP/IV_secuencia_ordenacion.cpp

IV.2.10. La clase string

IV.2.10.1. Métodos básicos

Ya vimos en la página 311 que el tipo `string` usado hasta ahora es, realmente, una clase por lo que los datos de tipo `string` son objetos.

Podemos pensar que, internamente, trabaja con un dato miembro privado que es un vector de `char` de tamaño variable.

string	
-	.....
+ void	push_back(char nuevo)
+ size_type	capacity()
+ size_type	size()
+ void	clear()
+ reference	at(size_type indice)

Los métodos `push_back` y `size` ya los conocemos. Sin entrar en detalles:

- ▷ **El método `capacity` devuelve la capacidad máxima actual (puede variar) y el método `clear` la deja con `""`.**
- ▷ **`size_type` es un tipo entero que garantiza albergar el tamaño necesario de un `string`.**
- ▷ **`reference` es una referencia (se verá en el segundo cuatrimestre). Por ahora, puede usarlo pensando que devuelve un `char` del string.**

La clase `SecuenciaCaracteres` se ha definido con métodos similares a los existentes en un `string`:

`string`	`SecuenciaCaracteres`
`push_back`	`Aniade`
`capacity`	`Capacidad`
`size`	`TotalUtilizados`
`clear`	`EliminaTodos`
`at`	`Elemento`

Ampliación:

Otros métodos interesantes (no hay que aprenderlos y no entran en el examen):

`at`. Accede (y modifica) a las componentes de forma segura.
`append`. Añade una cadena al final de otra.
`insert`. Inserta una subcadena a partir de una posición.
`replace`. Reemplaza una serie de caracteres consecutivos por una subcadena.
`erase`. Borra un número de caracteres, a partir de una posición.
`find`. Encuentra un carácter o una subcadena.
`substr`. Obtiene una subcadena.
`reserve`. Reserva un número de componentes (número que luego puede aumentar o disminuir). Útil cuando deseemos trabajar con un string grande, para que así, el programa no tenga que estar aumentando el tamaño del vector interno conforme se va llenando (lo que puede ser muy costoso).

La clase `string` permite además operaciones más *sofisticadas* como:

- **Asignar un valor concreto al objeto `string` a través de un literal de cadena de caracteres.**
- **Imprimir su contenido con `cout`.**
- **Acceder a las componentes individuales como si fuese un vector clásico, con la notación corchete.**
- **Concatenar dos cadenas o una cadena con un carácter usando el operador `+`**
- **Al crear un `string` por primera vez, éste contiene la *cadena vacía (empty string)* , a saber `""`**

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

int main(){
   string cadena;

   if (cadena == "")
      cout << "Vacía";                  // Vacía

   cadena.push_back('F');
   cout << "\n" << cadena;              // F

   cadena = "Funda";                    // Funda
   cadena = cadena + "ment";            // Fundament
   cadena = cadena + 'o';               // Fundamento
   cout << "\n" << cadena;              // Fundamento
   cadena.push_back('s');
   cout << "\n" << cadena;              // Fundamentos
   cout << "\n" << cadena.capacity();   // 20
   cout << "\n" << cadena.size();       // 11
   cadena[9] = 'a';
   cout << "\n" << cadena;              // Fundamentas
   cadena[12] = 'z';                    // Comportamiento indeterminado
```

IV.2.10.2. El método ToString

En numerosas ocasiones, será bastante útil proporcionar un método `ToString` a las clases que construyamos. Ya lo hicimos con la clase `Fecha`, por ejemplo (ver página 468) Para una clase `CalificacionesFinales` podría devolver una cadena con los nombres y notas de todos los alumnos.

En la clase `SecuenciaCaracteres`, está clara su utilidad:

```
class SecuenciaCaracteres{
......
   string ToString(){
      // Si el número de caracteres en memoria es muy grande,
      // es mucho más eficiente reservar memoria previamente
      // y usar push_back

      string cadena;

      cadena.reserve(total_utilizados);

      for (int i=0; i < total_utilizados; i++)
         cadena.push_back(vector_privado[i]);
         // cadena = cadena + vector_privado[i]  <- Evitarlo.
         // Muy muy ineficiente con muchas iteraciones;

      return cadena;
   }
};
int main(){
   SecuenciaCaracteres secuencia;
   ......
   cout << "\n" << secuencia.ToString();
}
```

IV.2.10.3. Lectura de un string con getline

¿Cómo leemos una variable de tipo `string`? `cin` se salta los separadores:

```
string cadena1, cadena2;

cout << "\nIntroduzca nombre: ";  // "Juan    Carlos"
cin >> cadena1;
cin >> cadena2;
cout << "\n\n" << "-" << cadena1 << "-" << cadena2 << "-";
                                    // -Juan-Carlos-
```

Si queremos leer los separadores dentro de un `string` usaremos la siguiente función que usa *getline* (no hace falta entender cómo lo hace)

```
string LeeCadenaHastaEnter(){
   string cadena;
   // Nos saltamos todos los \n que pudiera haber al principio:

   do{
      getline(std::cin, cadena);
   }while (cadena == "");

   return cadena;
}
int main(){
   string cadena1, cadena2;

   cout << "\nIntroduzca nombre: ";
   cadena1 = LeeCadenaHastaEnter();  // Juan Carlos Cubero
   cout << "\nIntroduzca nombre: ";  // Pedro Jiménez
   cadena2 = LeeCadenaHastaEnter();
   cout << "\n\n" << "-" << cadena1 << "-" << cadena2 << "-";
            // -Juan Carlos Cubero-Pedro Jiménez-
```

Bibliografía recomendada para este tema:

- **A un nivel menor del presentado en las transparencias:**
  - **Capítulo 3 (para las funciones) y primera parte del capítulo 6 (para las clases) de Deitel & Deitel**
- **A un nivel similar al presentado en las transparencias:**
  - **Capítulos 6 y 13 de Gaddis.**
  - **Capítulos 4, 5 y 6 de Mercer**
  - **Capítulo 4 de Garrido (sólo funciones, ya que este libro no incluye nada sobre clases)**
- **A un nivel con más detalles:**
  - **Capítulos 10 y 11 de Stephen Prata.**