Tema 3. Abstracción Procedimental

Vicente Benjumea García

Introducción a la Programación
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Tema 3. Abstracción procedimental

- Subprogramas: procedimientos, funciones y parámetros.
  - Definición de procedimientos y funciones. Parámetros formales.
  - Paso de parámetros por valor y por referencia.
  - Invocación a procedimientos y funciones. Parámetros actuales.
- Diseño descendente y abstracción procedimental.
- Criterios de modularización: Acoplamiento y Cohesión.
- Variables locales y globales.
- Precondiciones y postcondiciones. Asertos y excepciones.
- Algunas Funciones de la Biblioteca `<cmath>`.
- Recursividad.
  - Concepto de recursividad.
  - Recursividad frente a iteración.

Abstracción Procedimental: Subprogramas

Subprogramas

- Un **subprograma** define un **bloque de código independiente** que resuelve un determinado subproblema de forma parametrizada, y puede ser ejecutado (invocado) múltiples veces, aplicado a diferentes argumentos (valores).
- Los **subprogramas** permiten la aplicación de la **Abstracción Procedimental**, y de la metodología de programación de **Diseño Descendente** y **Refinamientos Sucesivos** a la resolución de problemas y desarrollo de programas.
  - Divide un problema en subproblemas, y asocia cada subproblema con un subprograma que lo resuelva.
  - Permite la resolución de un problema y el desarrollo de un programa **por partes**.
- Los **subprogramas** hacen posible la **reutilización** del código.

Efectos de los subprogramas

- Un subprograma **sólo puede tener los efectos** que sean especificados.

Abstracción Procedimental: Subprogramas. Ejemplos

Ejemplos

1. Desarrolle un programa que muestre el valor absoluto de un número leído de teclado.
2. Desarrolle un programa que muestre los valores absolutos de dos números leídos de teclado.
3. Desarrolle un programa que muestre el valor absoluto del menor de dos números leídos de teclado.
4. Desarrolle un programa que muestre el valor menor de los valores absolutos de dos números leídos de teclado.
5. Desarrolle un programa que muestre de forma ordenada (menor y mayor) los valores de dos números leídos de teclado.

Subprogramas: Procedimientos y Funciones

- Un **subprograma** define un **bloque de código independiente** que resuelve un determinado subproblema de forma parametrizada, y puede ser ejecutado (invocado) múltiples veces, aplicado a diferentes argumentos (valores).
  - **Funciones**: calculan **un único** valor a partir de la información de entrada.
  - **Procedimientos**: procesamiento **general** de información.
- La **definición** de un subprograma debe aparecer **antes** de su **invocación**.
  - La **cabecera** de un subprograma especifica el **tipo** del valor devuelto (o `void`), el **nombre** del subprograma y los **parámetros formales**.
  - El **cuerpo** del subprograma especifica la secuencia de **acciones** que resuelven un subproblema. Define sus propias **variables locales** de trabajo.
- Donde sea necesaria la resolución del subproblema, se realizará una **invocación** al subprograma, especificando el **nombre** del subprograma y los **parámetros actuales**.

```
⇑ // Función ⇓          ⇓
int menor(int x, int y)
{
    int z = x; ◀◀◀
    if (y < z) {
        z = y;
    }
    return z;  ◀◀◀
}
```

```
// Procedimiento  ⇕          ⇕
void ordenar(int& x, int& y)
{
    if (x > y) {
        int z = x; ◀◀◀
        x = y;
        y = z;
    }
}
```

```
// Principal
int main()
{
    int a, b; ◀◀◀
    cin >> a >> b;
▶▶  int c = menor(a, b);
▶▶  ordenar(a, b);
    bool ok = (c == a);
}
```

Definición de Subprogramas. Funciones

- La **definición** de una función debe aparecer **antes** de su **invocación**.
- **Definición de Funciones**: calculan **un único** valor a partir de la información de entrada.
  - La **cabecera** de una función especifica el **tipo** del valor devuelto, el **nombre** de la función y los **parámetros formales** (información de entrada).
  - El **cuerpo** de la función especifica la secuencia de **acciones** que resuelven un subproblema. Define sus propias **variables locales** de trabajo.
  - El **valor devuelto** por la función es el resultado de evaluar la expresión de la sentencia `return`.
  - El **cuerpo** de una función sólo debe tener una **única** sentencia `return`, y será la **última sentencia** del cuerpo de la función. Cualquier otra utilización de la sentencia `return` **no está permitida**.

```
 ⇑           ⇓
int vabs(int x)
{
   if (x < 0) {
      x = -x;
   }
   return x;   ◀◀◀
}
```

```
 ⇑            ⇓      ⇓
int menor(int x, int y)
{
    int z = x; ◀◀◀
    if (y < z) {
        z = y;
    }
    return z;   ◀◀◀
}
```

```
int main()
{
  int a, b; ◀◀◀
  cin >> a >> b;
  int c = vabs(a);
  int d = vabs(b);
  int e = menor(c, d);
  cout << menor(vabs(a), vabs(b));
} // return es opcional en main
```

Definición de Subprogramas. Procedimientos

- La **definición** de un procedimiento debe aparecer **antes** de su **invocación**.
- **Definición de Procedimientos**: procesamiento **general** de información.
  - La **cabecera** de un procedimiento especifica que no devuelve ningún valor (`void`), el **nombre** del procedimiento y los **parámetros formales**.
  - El **cuerpo** del procedimiento especifica la secuencia de **acciones** que resuelven un subproblema. Define sus propias **variables locales** de trabajo.
  - **Procesa información** que se transfiere a través de los parámetros.
  - El **cuerpo** de un procedimiento **no** debe tener ninguna sentencia `return`.

```
                ⇑
void leer(int& x)
{
  cout << "Número: ";
  cin >> x;
}
                ⇓
void mostrar(int x)
{
  cout << "Resultado: ";
  cout << x << endl;
}
```

```
                    ⇕       ⇕
void ordenar(int& x, int& y)
{
    if (x > y) {
        int z = x; ◀◀◀
        x = y;
        y = z;
    }
}
```

```
int main()
{
  int a, b;
  leer(a);
  leer(b);
  ordenar(a, b);
  mostrar(a);
  mostrar(b);
}
// return es opcional
// en main
```

Parámetros Formales

- Todo el intercambio/transferencia de información con un subprograma se realiza a través de los **parámetros** (argumentos), *y del valor devuelto por las funciones*.
- **Parámetros Formales:** aparecen en la **definición** del subprograma.
- **Parámetros Actuales:** aparecen en la **invocación** al subprograma.
- Los parámetros poseen:
  - **Tipo** que define el intercambio de información.
  - **Nombre** con el que referenciarlos.
  - **Direccionalidad** que determina la dirección de la transferencia de información.
    - (⇓) **Entrada**: el subprograma **recibe** información a través del parámetro.
    - (⇑) **Salida**: el subprograma **envía** información a través del parámetro.
    - (⇕) **Entrada/Salida**: el subprograma **recibe** información a través del parámetro, la modifica, y la **envía** ya modificada a través del mismo parámetro.

- Los *parámetros* también pueden ser denominados **argumentos**, tanto *formales* como *actuales*.
- A veces se usa *parámetro* como sinónimo de *parámetro formal* (en la definición del subprograma), y se usa *argumento* como sinónimo de *parámetro actual* (en la invocación del subprograma), depende del contexto en el que se utilice.

- La transferencia de información entre subprogramas se implementa en la práctica mediante el **paso por valor** y el **paso por referencia**.

Parámetros Formales. Direccionalidad

- El flujo de información posee una determinada **direccionalidad**:
  - (⇓) **Entrada**: el subprograma **recibe** información a través del parámetro.
  - (⇑) **Salida**: el subprograma **envía** información a través del parámetro.
  - (⇕) **Entrada/Salida**: el subprograma **recibe** información a través del parámetro, la modifica, y la **envía** ya modificada a través del mismo parámetro.

```
                ⇑
void leer(int& x)
{
   cout << "Datos";
   cin >> x;
}



int main()
{
   int a;
   leer(a);
}
```

```
 ⇑              ⇓      ⇓
int menor(int x, int y)
{
    int z = x;
    if (y < z) {
        z = y;
    }
    return z;
}

int main()
{
   int a = 7;
   int c = menor(a, 3+2);
}
```

```
                    ⇕       ⇕
void ordenar(int& x, int& y)
{
    if (x > y) {
        int z = x;
        x = y;
        y = z;
    }
}

int main()
{
   int a = 5, b = 3;
   ordenar(a, b);
}
```

Paso de Parámetros por Valor y por Referencia

Paso de Parámetros por Valor

- Parámetros de **Entrada de Tipos Simples**: se utiliza el **paso por valor**.
- En el **paso por valor** (`int x`) el parámetro formal actúa como una **nueva variable independiente** inicializada con una **copia del valor** del parámetro actual.
  - El parámetro formal **es un objeto distinto** del parámetro actual (aislados).
  - La modificación en el parámetro formal **no afecta** al parámetro actual.

	Tipos Simples		Tipos Compuestos	
	(⇓)Ent	(⇑)Sal(⇕)E/S	(⇓)Ent	(⇑)Sal(⇕)E/S
Par. Formal	P.Valor (int x)	P.Referencia (int& x)	P.Ref.Cte (const Per& x)	P.Referencia (Per& x)

```
 ⇑          ⇓
int vabs(int x)
{
   if (x < 0) {
      x = -x;
   }
   return x;
}
```

```
                ⇓
void mostrar(int x)
{
  cout << "Resultado: ";
  cout << x << endl;
}
```

```
int main()
{
  int a, b;
  cin >> a >> b;
  int c = vabs(a);
  mostrar(c);
  mostrar(vabs(b));
}
```

Paso de Parámetros por Valor y por Referencia

Paso de Parámetros por Referencia

- Parámetros de **Salida** y **Entrada/Salida**: se utiliza el **paso por referencia**.
- En el **paso por referencia** (`int& x`) el parámetro formal se **vincula a la variable** situada como parámetro actual, referenciando ambos al mismo objeto.
  - El parámetro formal **está vinculado a la variable** del parámetro actual.
  - La modificación en el parámetro formal **también modifica** al parámetro actual.

	Tipos Simples		Tipos Compuestos	
	(⇓)Ent	(⇑)Sal(⇕)E/S	(⇓)Ent	(⇑)Sal(⇕)E/S
Par. Formal	P.Valor (int x)	P.Referencia (int& x)	P.Ref.Cte (const Per& x)	P.Referencia (Per& x)

```
          ⇑
void leer(int& x)
{
  cout << "Número: ";
  cin >> x;
}
```

```
                ⇕       ⇕
void ordenar(int& x, int& y)
{
    if (x > y) {
        int z = x;
        x = y;
        y = z;
    }
}
```

```
int main()
{
  int a, b;
  leer(a);
  leer(b);
  ordenar(a, b);
  cout << a << " " << b;
}
```

Paso de Parámetros por Valor y por Referencia

Paso de Parámetros por Referencia Constante

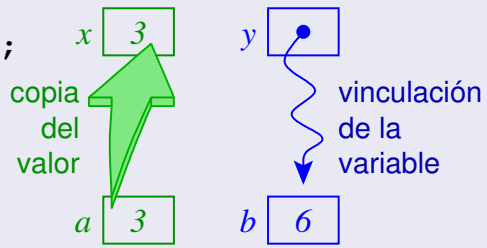
- Parámetros de **Entrada de Tipos Compuestos** (tema 4): se utiliza el **paso por referencia constante**.
- En el **paso por referencia constante** (`const Per& x`) el parámetro formal constante se **vincula al valor** del parámetro actual, referenciándolo de forma **constante** y permitiendo acceder a dicho valor.
  - El parámetro formal **constante está vinculado al valor** del parámetro actual.
  - No es posible modificar el parámetro formal, ya que es **constante**.

	Tipos Simples		Tipos Compuestos	
	(⇓)Ent	(⇑)Sal(⇕)E/S	(⇓)Ent	(⇑)Sal(⇕)E/S
Par. Formal	P.Valor (int x)	P.Referencia (int& x)	P.Ref.Cte (const Per& x)	P.Referencia (Per& x)

Paso de Parámetros por Valor y por Referencia. Ejemplo

- El **paso por valor** de tipos **simples** permite realizar la transferencia de información de **entrada** de forma *eficiente*.
  - En el **paso por valor** el parámetro formal actúa como una **nueva variable independiente** inicializada con una **copia del valor** del parámetro actual.
- El **paso por referencia** permite realizar la transferencia de información de **salida** y **entrada/salida** de forma *eficiente*.
  - En el **paso por referencia** el parámetro formal se **vincula a la variable** situada como parámetro actual, utilizando ambos la misma zona de memoria.

```
void subprograma(int x, int& y)
{
    y = 2 * x;
}

int main()
{
    int a, b;
    a = 3;
    subprograma(a, b);
}
```



Utilización de Subprogramas: Invocación

- La **definición** de un subprograma debe aparecer **antes** de su **invocación**.
- La invocación (llamada) se realiza mediante el **nombre** seguido por los **parámetros actuales**.
- La invocación a una **función** no puede constituir por sí sola una sentencia, sino que debe aparecer dentro de alguna estructura que **utilice** el valor resultado de la función.
- La invocación a un **procedimiento** constituye por sí sola una sentencia que puede ser utilizada como tal en el cuerpo de subprogramas y de `main`.
- Un subprograma puede ser invocado múltiples veces, aplicado a los mismos o a diferentes argumentos (parámetros actuales).
- Un subprograma puede invocar a otros subprogramas (definidos previamente).

```
 ⇑           ⇓     ⇓
int menor(int x, int y)
{
    int z = x;
    if (y < z) {
        z = y;
    }
    return z;
}
```

```
                  ⇕     ⇕
void ordenar(int& x, int& y)
{
    if (x > y) {
        int z = x;
        x = y;
        y = z;
    }
}
```

```
int main()
{
    int a, b;
    cin >> a >> b;
▶▶  int c = menor(a, b);
▶▶  ordenar(a, b);
    bool ok = (c == a);
}
```

Utilización de Subprogramas. Parámetros Actuales

- El **número de parámetros** actuales debe **coincidir** con el número de parámetros formales.
- **Correspondencia posicional** entre parámetros actuales y formales.
- El **tipo** del parámetro actual debe **coincidir** con el tipo del correspondiente parámetro formal.
- Un parámetro formal de **salida** o **entrada/salida** *(paso por referencia)* requiere que el parámetro actual sea una **variable**.
- Un parámetro formal de **entrada** *(paso por valor o referencia constante)* requiere que el parámetro actual sea una **variable**, **constante** o **expresión**.

	Tipos Simples		Tipos Compuestos	
	(⇓)Ent	(⇑)Sal(⇕)E/S	(⇓)Ent	(⇑)Sal(⇕)E/S
Par. Formal	P.Valor (int x)	P.Referencia (int& x)	P.Ref.Cte (const Per& x)	P.Referencia (Per& x)
Par. Actual	Constante Variable Expresión	Variable	Constante Variable Expresión	Variable

Utilización de Subprogramas. Flujo de Ejecución

La ejecución del programa **comienza** en la función **main**.

- La función **main** es *especial* y no necesita **return** (por defecto es `return 0;`)

Cuando se produce una **invocación** a un subprograma:

1. Se establecen las *vias de comunicación* entre los algoritmos llamante y llamado.
   - **Copia de valores** en el *paso por valor*.
   - **Vinculación de variables** en el *paso por referencia*.
2. El **flujo de ejecución** salta a ejecutar la primera instrucción del cuerpo del subprograma llamado, ejecutándose éste.
3. Las **variable locales** se crearán a medida que se *ejecuten* sus definiciones.
4. Cuando finaliza la ejecución del subprograma, los parámetros y variables locales previamente creadas se **destruyen** y el **flujo de ejecución** continúa por la siguiente instrucción a la invocación realizada.

```
 ⇑           ⇓     ⇓
int menor(int x, int y)
{
    int z = x; ◀◀◀
    if (y < z) {
        z = y;
    }
    return z;  ◀◀◀
}
```

```
                  ⇕     ⇕
void ordenar(int& x, int& y)
{
    if (x > y) {
        int z = x; ◀◀◀
        x = y;
        y = z;
    }
}
```

```
int main()
{
    int a, b; ◀◀◀
    cin >> a >> b;
▶▶  int c = menor(a, b);
▶▶  ordenar(a, b);
    bool ok = (c == a);
}
```

Utilización de Subprogramas. Parámetros. Ejemplo

- Supongamos que se define un subprograma de la siguiente manera:

```
void prueba(int a, int b, double& c, double& d, char e)
{ /* ... */ }
```

- Supongamos que se define el programa principal de la siguiente manera:

```
int main()
{
    double x = 2, y = 3;
    int    m = 4;
    char   c = 'z';

    prueba(m+3,  10,    x,   y,  c);
    prueba(m,    19,    x,   y);
    prueba(35, m*10,    x,   c,  y);
    prueba(m,   3.5,    x,   y,  c);
    prueba(30,   10,    x, x+y,  c);
    prueba(30,   10,    m,   x,  c);
    prueba(m,   m*m,    y,   x,  c);
    prueba(m,    10, 35.0,   y, 'E');
    prueba(30,   10,    c,   d,  e);
}
```

- ¿ Que llamadas a **prueba** desde **main** son incorrectas, y cuál es la razón ?

Declaración de Subprogramas: Prototipos

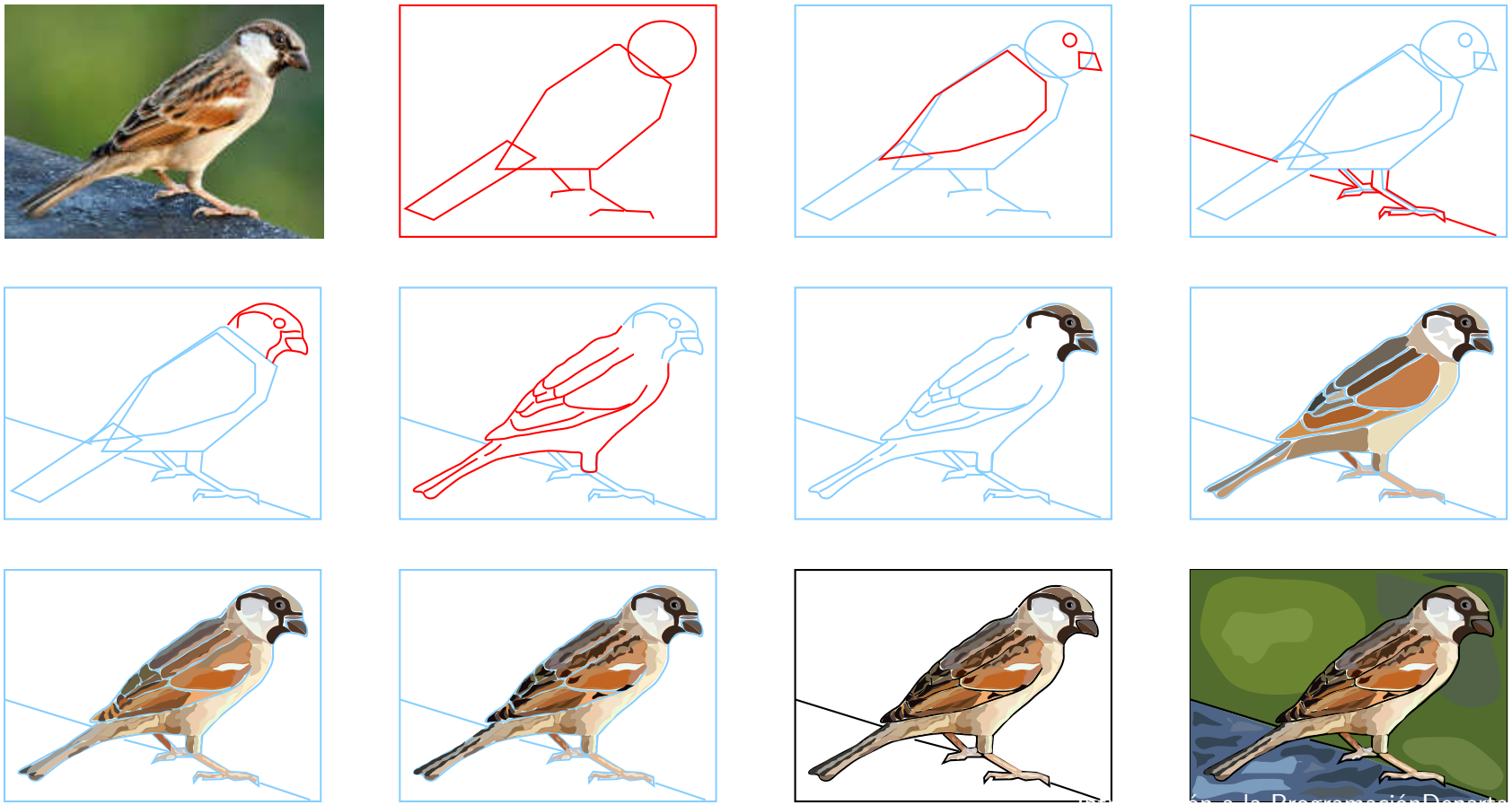
- La **definición** de un subprograma debe aparecer **antes** de su **invocación**.
- **Prototipo**: es posible declarar un determinado subprograma sin necesidad de definirlo explícitamente. **Aunque NO se recomienda**.
  - Define la **cabecera** del subprograma, terminada en **punto-y-coma**, y el cuerpo del subprograma no se define.
  - Permite **invocar** a un subprograma desde cualquier sitio posterior la declaración de su prototipo.
  - El subprograma deberá ser definido en algún otro lugar del programa.
  - Son útiles para diseñar bibliotecas y para casos de recursividad indirecta.

```
// -- Prototipos ----
int menor(int x, int y);        <<<
void ordenar(int& x, int& y);   <<<
// -- Principal ----
int main()
{
     int a, b;
     cin >> a >> b;
     int c = menor(a, b);
     ordenar(a, b);
     bool ok = (c == a);
}
```

```
int menor(int x, int y) {
    int z = x;
    if (y < z) {
        z = y;
    }
    return z;
}
void ordenar(int& x, int& y) {
    if (x > y) {
        int z = x;
        x = y;
        y = z;
    }
}
```

Diseño Descendente (I)

- Los problemas reales suelen ser **demasiado** grandes y **complejos** para abordar su solución completa en su totalidad. Por ello, abordamos su solución **por partes**.
  - Utilizamos la **abstracción** para **identificar** los **conceptos importantes** de cada nivel, dejando los detalles para refinamientos posteriores, y aplicamos **refinamientos sucesivos**, utilizando la abstracción en cada nivel.

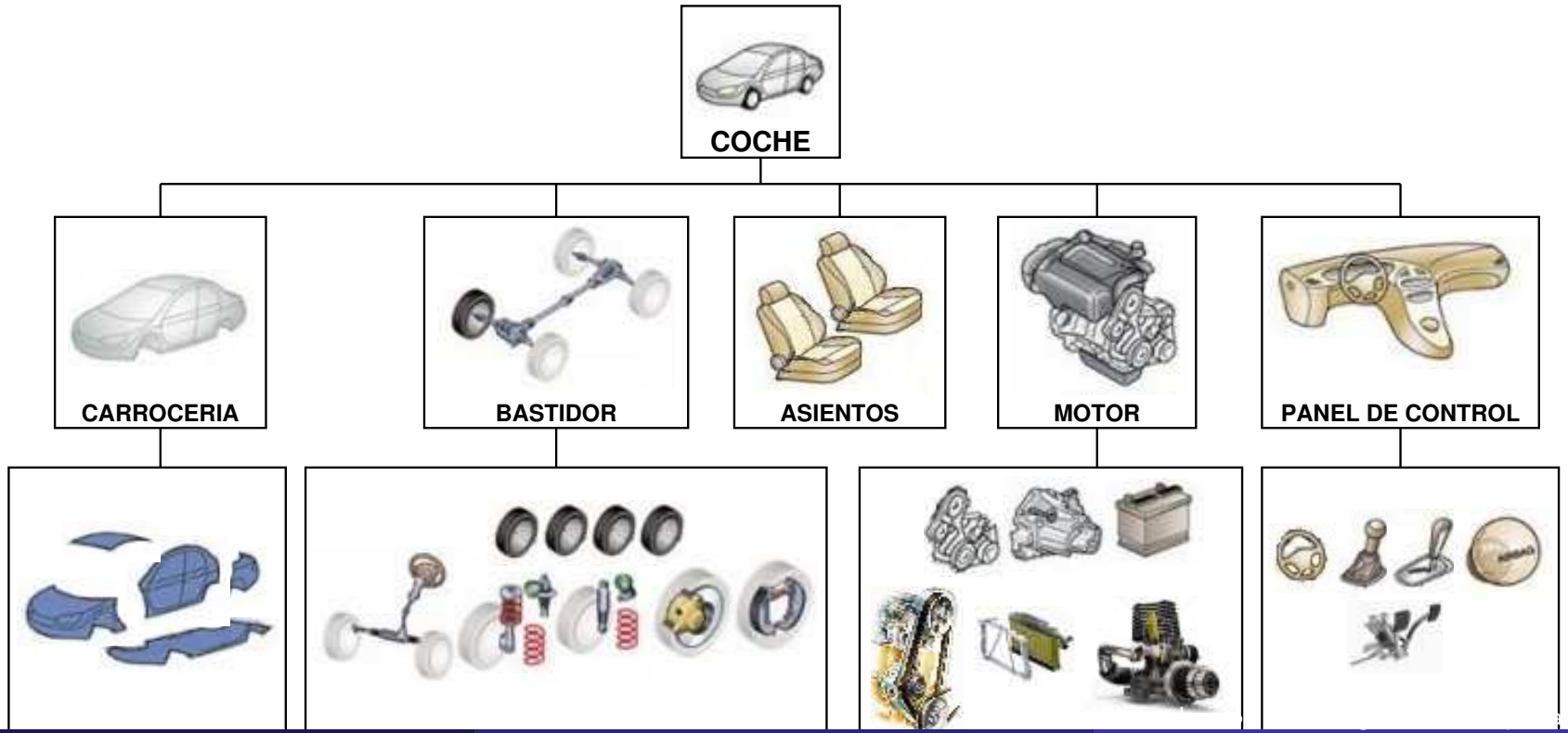


Diseño Descendente (II)

- Los problemas reales suelen ser **demasiado** grandes y **complejos** para abordar su solución completa en su totalidad. Por ello, abordamos su solución **por partes**.
  - Utilizamos la **abstracción** para **identificar** los **conceptos importantes** de cada nivel, dejando los detalles para refinamientos posteriores, y aplicamos **refinamientos sucesivos**, utilizando la abstracción en cada nivel.

Diseño Descendente: Abstracción y Refinamientos

- La **Abstracción** es la herramienta mental que nos permite **analizar**, **comprender**, y **construir** sistemas complejos.
  - Realizamos el análisis, la comprensión, y la construcción de sistemas complejos por **niveles de abstracción**.
  - Utilizamos la **abstracción** para **identificar** los **conceptos importantes** de cada nivel, dejando los detalles para refinamientos posteriores.
  - Aplicamos **Refinamientos Sucesivos**, utilizando la abstracción en cada nivel.



Diseño Descendente: Abstracción Procedimental

- En la mayoría de los problemas reales, el **algoritmo** que los soluciona es **demasiado** grande y **complejo** para codificarlo mediante un único programa monolítico.
- Desventajas de este tipo de programas monolíticos:
  - Dificultad para afrontar directamente la **solución** de un problema complejo.
  - Dificultad para **adaptar** los programas a nuevas circunstancias.
  - Dificultad para detectar y corregir **errores**.
  - Imposibilidad de **reutilizar** partes del programa en otros programas.
- Es preferible codificar un programa **por partes** (utilizando subprogramas), empleando la **abstracción procedimental** y los **refinamientos sucesivos** para simplificar la codificación de los **subprogramas** que componen el programa completo.

Diseño Descendente: Refinamientos Sucesivos

- La metodología de **Diseño Descendente** (*refinamientos sucesivos*, *top-down*, *divide y vencerás*) nos permite afrontar la solución de un problema, y la codificacion del programa, **por partes**, de tal forma que no es necesario resolver el problema completamente como un todo, sino que se puede resolver poco a poco, mediante **refinamientos sucesivos**.
- **Refinamientos sucesivos**: un **problema** se descompone en varios **subproblemas** más simples, y estos a su vez se vuelven a descomponer en otros **subproblemas** más simples todavía, y así sucesivamente hasta llegar a un nivel de descomposición que permita la solución sencilla de los diferentes subproblemas.
- En cada paso de refinamiento, dado un determinado problema, la **abstracción** nos permite **identificar** adecuadamente cuales son los **subproblemas** en los que se descompone, identificando los conceptos importantes de un determinado nivel, y dejando los detalles para refinamientos posteriores.
- La **solución** al problema completo viene dada por la **composición** de cada una de las **soluciones** a los subproblemas más simples.

Abstracción Procedimental e Interfaces

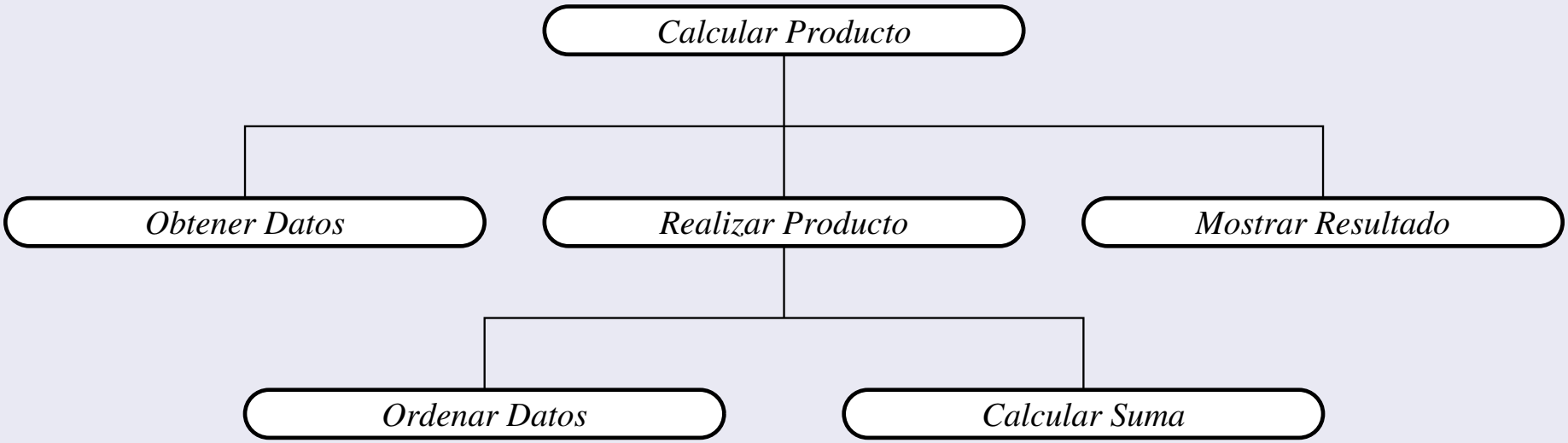
- Utilizamos **abstracción** y aplicamos **refinamientos sucesivos**:
  - Divide un **problema** en **subproblemas**.
    - Asocia un **subprograma** a cada subproblema.
  - Determinamos la **funcionalidad** del subprograma.
    - ¿ Que subproblema resuelve ?
  - Determinamos la **transferencia de información** (interfaz) del subprograma.
    - ¿ Que información necesita y que información produce? ¿ Bajo que condiciones ?
  - No importa **como se resuelve** el subproblema.
    - Será abordado en un nivel de refinamiento posterior.
- En programación, la **interfaz** define la forma en que se comunican y cooperan dos subprogramas. Define la transferencia de información entre subprogramas.
  - La información que necesita, la información que produce, y que condiciones debe cumplir la información transferida.
- Objetivo: división en **subprogramas independientes** (minimizar las dependencias con otros subprogramas).
  - Funcionalidad clara y bien definida.
  - Transferencia de información simple y bien definida. Cuanto más **simple**:
    - El subprograma tendrá menos dependencias y estará más **aislado** del entorno.
    - El subprograma será más **fácil** de utilizar y se producirán **menos errores** en su utilización.

Diseño Descendente: Ventajas

- El *diseño descendente*, la *abstracción procedimental* y los *refinamientos sucesivos* tienen numerosas ventajas en el desarrollo de software:
  - Simplifica el diseño y la solución del programa, ya que se realiza **por partes**, poco a poco.
  - Permite que el programador esté **concentrado** en la solución individual de cada subproblema/subprograma concreto.
  - Los subprogramas **encapsulan** y **aislan** las diferentes tareas que componen un programa.
  - Simplifica la comprensión y mejora la **legibilidad** del programa.
  - Si el método para solucionar una tarea debe cambiar, el **aislamiento** evita que dicho cambio influya en las otras tareas.
  - Facilita la **modificación**, adaptación y **evolución** del software.
  - Facilita la **detección** y **corrección** de errores (depuración).
  - Posibilidad de **reutilización** del subprograma en otro contexto.

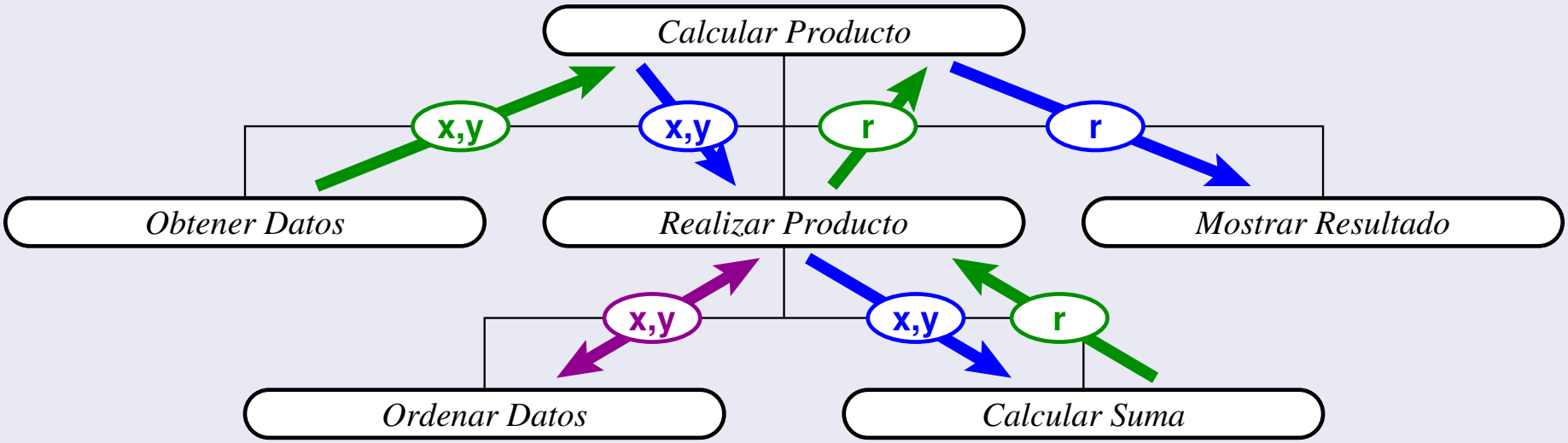
Diseño Descendente: Abstracción Procedimental. Ejemplo

- Calcular el producto de dos números mediante sumas sucesivas, minimizando el número de iteraciones realizadas.



Diseño Descendente: Abstracción Procedimental. Ejemplo

- Calcular el producto de dos números mediante sumas sucesivas, minimizando el número de iteraciones realizadas.



Diseño Descendente: Abstracción Procedimental. Ejemplo

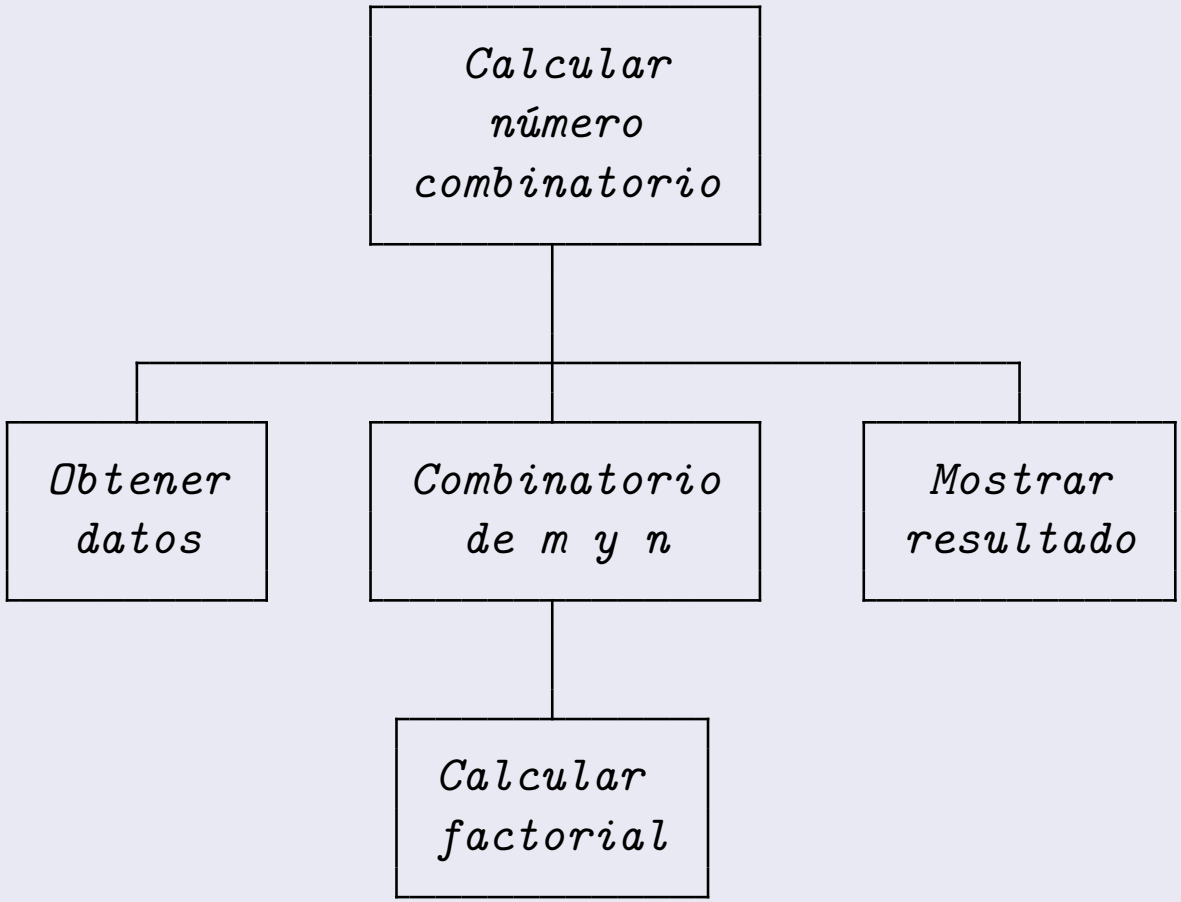
```
#include <iostream>
using namespace std;
                      ⇑      ⇑
void ordenar(int& x, int& y) // P.Ref
{
  if (x > y) {
    int z = x;  // variable local
    x = y;
    y = z;
  }
}
 ⇑                    ⇓      ⇓
int calcularSuma(int x, int y)
{
  int suma = 0;    // variable local
  for (int i = 0; i < x; ++i) {
    suma = suma + y;
  }
  return suma; // Devuelve el valor
}

// Se debe definir un subprograma
// antes de su utilización
```

```
 ⇑                          ⇓      ⇓
int realizarProducto(int x, int y)
{
  ordenar(x, y);
  return calcularSuma(x, y);
}
                    ⇑      ⇑
void leerDatos(int& x, int& y) // P.Ref
{
  cout << "Introduce dos números";
  cin >> x >> y;
}
                           ⇓
void mostrarResultado(int r) // P.Valor
{
  cout << "Resultado: " << r << endl;
}
int main()
{
  int a, b, c;     // variables locales
  leerDatos(a, b);
  c = realizarProducto(a, b);
  mostrarResultado(c);
}
```

Diseño Descendente: Abstracción Procedimental. Ejemplo

- Cálculo de un número combinatorio, considerando que $m \geq n$:

$$\binom{m}{n} = \frac{m!}{n! \cdot (m-n)!}$$



Diseño Descendente: Abstracción Procedimental. Ejemplo

```
#include <iostream>
using namespace std;
void leerDatos(int& m, int& n) {
    cout << "Introduce m y n (m >= n): ";
    cin >> m >> n ;
    while (m < n) {
        cout << "Error. Introduce m y n (m >= n): ";
        cin >> m >> n ;
    }
}
void mostrarResultado(int m, int n, long res) {
    cout << "El número combinatorio de " << m << " sobre " << n << " es " << res << endl;
}
long factorial(int x) {
    long fact = 1;
    for (int i = 2; i <= x; ++i) {
        fact = fact * i;
    }
    return fact;
}
long combinatorio(int m, int n) {
    return factorial(m) / ( factorial(n) * factorial(m-n) );
}
int main() {
    int m, n;
    leerDatos(m, n);
    long res = combinatorio(m, n);
    mostrarResultado(m, n, res);
}
```

Diseño Descendente: Abstracción Procedimental. Ejemplo

- Desarrolle un programa que lea un número (≥0), en caso de datos erróneos mostrará un mensaje adecuado. En otro caso, muestra el valor de cada dígito del número, en orden inverso, separados por espacios. Por ejemplo, para el número 12345678, muestra 8 7 6 5 4 3 2 1

Diseño Descendente: Abstracción Procedimental. Ejemplo

```cpp
#include <iostream>
using namespace std;
void leer(int& num) {
    cout << "Introduzca número: ";
    cin >> num;
}
int n_digitos(int num)
{
    int i = 0;
    do {
        num = num / 10;
        ++i;
    } while (num != 0);
    return i;
}
int digito(int num, int pos)
{
    for (int j = 0; j < pos; ++j) {
        num = num / 10;
    }
    return num % 10;
}
```

```cpp
void mostrar_digitos(int num) // O(n^2)
{
    int nd = n_digitos(num);
    for (int i = 0; i < nd; ++i) {
        cout <<  digito(num, i) << " ";
    }
    cout << endl;
}
void mostrar_digitos_alternativo(int num) // O(n)
{
    do {
        cout <<  (num % 10) << " ";
        num = num / 10;
    } while (num > 0);
    cout << endl;
}
int main()
{
    int num;
    leer(num);
    if (num < 0) {
        cout << "Error" << endl;
    } else {
        mostrar_digitos(num);
    }
}
```

Diseño Descendente: Abstracción Procedimental. Ejemplo

Procesamiento Secuencial

- Desarrolle un programa que lea una secuencia de números terminada en cero, y muestre aquellos números de la secuencia leída que sean primos. Por ejemplo:

```
Introduce una secuencia de números terminada en cero:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 0
Primos: 2 3 5 7 11 13 17 19
```

Diseño Descendente: Abstracción Procedimental. Ejemplo

```
bool es_primo(int num)
{
    bool esprimo = (num > 1);
    for (int div = 2; (div <= num/2) && esprimo ; ++div) {
        if ((num % div) == 0) {
            esprimo = false;
        }
    }
    return esprimo;
}
void procesar(int num)
{
    if (es_primo(num)) {
        cout << num << " ";
    }
}
int main()
{
    int num;
    cout << "Introduce una secuencia de números terminada en cero: " << endl;
    cin >> num ;
    cout << "Primos: " ;          // En el procesamiento secuencial
    while (num != 0) {            // la entrada de datos se entremezcla
        procesar(num);            // con el procesamiento de los datos
        cin >> num ;              // y con el control del bucle
    }
    cout << endl;
}
```

Diseño Descendente: Abstracción Procedimental. Ejemplo

Procesamiento Secuencial

- Desarrolle un programa que lea una secuencia de números terminada en cero, y compruebe si en la secuencia han aparecido tres números primos ascendentes consecutivos. Por ejemplo:

```
Introduce secuencia:
1 2 5 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 0
Resultado: false
```

```
Introduce secuencia:
1 2 3 4 5 6 7 11 13 14 15 16 17 18 19 0
Resultado: true
```

Diseño Descendente: Abstracción Procedimental. Ejemplo

```
bool es_primo(int num)
{
    bool esprimo = (num > 1);
    for (int div = 2; (div <= num/2) && esprimo ; ++div) {
        if ((num % div) == 0) {
            esprimo = false;
        }
    }
    return esprimo;
}
bool primos_ascendentes(int ant2, int ant1, int act)
{
    return ((ant2 < ant1 && ant1 < act)
            && es_primo(ant2)
            && es_primo(ant1)
            && es_primo(act));
}
void procesar(int ant2, int ant1, int act, bool& ok)
{
    if (primos_ascendentes(ant2, ant1, act)) {
        ok = true;
    }
}
```

```
void leer(int& ant2, int& ant1, int& act)
{
    ant2 = ant1;
    ant1 = act;
    cin >> act;
}
int main()
{
    bool ok = false;
    int ant2 = 0;
    int ant1 = 0;
    int act = 0;
    cout << "Introduce secuencia: " << endl;
    leer(ant2, ant1, act);
    while (act != 0) {
        procesar(ant2, ant1, act, ok);
        leer(ant2, ant1, act);
    }
    cout << "Resultado: " << boolalpha << ok
         << endl;
}
```

Diseño Descendente: Abstracción Procedimental. Ejemplo

Secuencia de Fibonacci

- Desarrolle un programa que lea un número **N** por teclado mayor o igual a cero y calcule e imprima el n-ésimo número de la secuencia de Fibonacci.
  - Los dos primeros números de esta secuencia son el cero y el uno, y a partir de éstos, cada número de la secuencia se calcula realizando la suma de los dos anteriores, es decir: $f_0 = 0$; $f_1 = 1$; $f_n = f_{n-1} + f_{n-2}$.
  - Los primeros números de la secuencia de Fibonnaci son: $0, 1, 1, 2, 3, 5, 8, 13, 21, \cdots$
  - Por ejemplo, para *N=8* mostrará 21, y para *N=20* mostrará 6765.

Diseño Descendente: Abstracción Procedimental. Ejemplo

```
void leer(int& num) {
    cout << "Introduce número: ";
    cin >> num;
}
int fibonacci(int n)
{
    int fk, fk1, fk2;           // variables para almacenar F_k, F_k-1 y F_k-2
    if (n < 2) {                // si N es menor que 2
        fk = n;                         // el resultado F_0 es 0 y F_1 es 1
    } else {                    // en otro caso
        fk1 = 0;                        // primer valor de la sucesión F_0 es 0
        fk  = 1;                        // segundo valor de la sucesión F_1 es 1
        for (int k = 2; k <= n; ++k) { // iterar para los valores de K ∈ { 2...N }
            fk2 = fk1;                    //  asignar a F_k-2 el valor de F_k-1 anterior
            fk1 = fk;                     //  asignar a F_k-1 el valor de F_k anterior
            fk = fk1 + fk2;               //  asignar a F_k el nuevo valor F_k-1 +F_k-2
        }
    }
    return fk;                  // devolver el resultado F_n calculado
}
int main() {
    int n;
    leer(n);
    if (n < 0) {
        cout << "Error" << endl;
    } else {
        cout << "Valor: " << fibonacci(n) << endl;
    }
}
```

Criterios de Modularización: Acoplamiento y Cohesión

- No existen métodos objetivos para determinar cómo **descomponer un problema** en subprogramas, es una labor **subjetiva**.
- No obstante, se siguen algunos criterios que pueden guiarnos para descomponer un problema y modularizar adecuadamente.
  - **Acoplamiento**
  - **Cohesión**

Objetivos para una adecuada modularización

- El diseñador de software debe buscar un **bajo acoplamiento** entre subprogramas y una **alta cohesión** dentro de cada uno.

- Si **no** es posible **analizar** y comprender un subprograma de forma aislada e **independiente** del resto, entonces podemos deducir que la división modular **no es adecuada**, y tendrá un alto acoplamiento.
- La **funcionalidad** de un subprograma debe ser **simple**, estar claramente definida, y ser fácil de expresar. En otro caso, tendrá una baja cohesión.

- Una modularización inadecuada afecta negativamente a la calificación.

Criterios de Modularización: Acoplamiento y Cohesión

Acoplamiento

- Un objetivo en el diseño descendente es crear **subprogramas independientes**.
- Sin embargo, debe haber alguna **interacción** entre los subprogramas para formar un sistema coherente.
- El **acoplamiento** representa el **grado de dependencia** de un subprograma respecto de otro.
- Se desea **minimizar el acoplamiento**, es decir, maximizar la independencia.
- Si **no** es posible **analizar** y comprender un subprograma de forma aislada e **independiente** del resto, entonces tiene una alta dependencia con el resto.

Cohesión

- La **cohesión** hace referencia al **grado de relación** entre las diferentes **partes internas** de un subprograma.
- Se desea **maximizar la cohesión** dentro de cada subprograma.
- Si la cohesión es alta, entonces un subprograma realiza una **única tarea**, con una funcionalidad clara y bien definida.
- Si la cohesión es baja, hay gran diversidad entre las distintas tareas realizadas dentro de un subprograma, haciendo difícil su comprensión y modificación.

Criterios de Modularización. Ejemplo 1 (v1)

Versión con Alto Acoplamiento y Baja Cohesión => Código Inadecuado

```
void cuenta_divisores(int numero, int& cnt)
{
    for (int i = 1; i <= numero; ++i) {
        if (numero % i == 0) {
            ++cnt;
        }
    }
    cout << "Resultado: " << cnt << endl;
}
```

```
int main()
{
    int numero, cnt = 0;
    cout << "Introduce número: ";
    cin >> numero;
    cuenta_divisores(numero, cnt);
}
```

- Nótese que la corrección de **cuenta_divisores()** depende de que la variable **cnt** se inicialice con el valor **cero** en **main()**, por lo que hay un **alto acoplamiento** entre ambos subprogramas.
- El procedimiento **cuenta_divisores()** calcula la cuenta de divisores, y **además** muestra el resultado en pantalla, que son tareas poco relacionadas, por lo que hay una **baja cohesión** en ese subprograma.

Criterios de Modularización. Ejemplo 1 (v2)

Versión con Bajo Acoplamiento y Alta Cohesión => Código Adecuado

```
int cuenta_divisores(int numero)
{
    int cnt = 0;
    for (int i = 1; i <= numero; ++i) {
        if (numero % i == 0) {
            ++cnt;
        }
    }
    return cnt;
}
void mostrar(int cnt)
{
    cout << "Resultado: " << cnt << endl;
}
```

```
void leer(int& numero)
{
    cout << "Introduce número: ";
    cin >> numero ;
}
int main()
{
    int numero;
    leer(numero);
    mostrar( cuenta_divisores(numero) );
}
```

- Cada subprograma está aislado y es independiente del resto, por lo que hay **bajo acoplamiento** entre los subprogramas.
- Cada subprograma sólo realiza una única tarea, con una funcionalidad clara y bien definida, por lo que hay **alta cohesión** dentro de cada subprograma.

Criterios de Modularización. Ejemplo 2

Buscar número perfecto

- Desarrolle un programa que busque y muestre el primer **número perfecto** mayor o igual que un valor leído de teclado.
- Un número es perfecto si es igual a la suma de sus divisores (salvo él mismo).
  - Por ejemplo, 28 es perfecto ya que $28 = 1 + 2 + 4 + 7 + 14$

Criterios de Modularización. Ejemplo 2 (v1)

```
void comprobar(int& a, int& b, int& i) {
    if (a % i == 0) {
        b = b + i;
    }
}
int calculo(int& a, int& b)
{
    for (int i = 1; i <= a/2; i++) {
        comprobar(a, b, i);
    }
    return b;
}
void resultado(int& a, int& b) {
    if (a == b) {
        cout << "Número perfecto: " << a << endl;
    } else {
        ++a;
    }
}
int main()
{
    int a, b = 0;
    cout << "Introduce valor inicial: ";
    cin >> a ;
    do {
        calculo(a, b);
        resultado(a, b);
    } while (a != b);
}
```

Código adaptado realizado por un alumno

ALTO ACOPLAMIENTO Y BAJA COHESIÓN

Nombres Inadecuados
Parámetros Inadecuados
Difícil de Analizar
Difícil de Comprender

El PROGRAMA NO ES CORRECTO

Criterios de Modularización. Ejemplo 2 (v1)

```
void comprobar(int& a, int& b, int& i) {
    if (a % i == 0) {
        b = b + i;
    }
}
int calculo(int& a, int& b)
{
    for (int i = 1; i <= a/2; i++) {
        comprobar(a, b, i);
    }
    return b;
}
void resultado(int& a, int& b) {
    if (a == b) {
        cout << "Número perfecto: " << a << endl;
    } else {
        ++a;
    }
}
int main()
{
    int a, b = 0;
    cout << "Introduce valor inicial: ";
    cin >> a ;
    do {
        calculo(a, b);
        resultado(a, b);
    } while (a != b);
}
```

- **Comprobar** tiene un ALTO ACOPLAMIENTO funcional con **Cálculo**. Gran dependencia funcional, **Cálculo** no tiene sentido de forma aislada

- **Cálculo** tiene un ALTO ACOPLAMIENTO funcional con **Comprobar**
- Devuelve la suma de divisores con RETURN y a través del parámetro B
- El parámetro B es de entrada/salida y contiene el valor inicial de la suma
- Tiene un ALTO ACOPLAMIENTO con **MAIN**, ya que el valor inicial de la suma (B) depende de la llamada desde el exterior

- **Resultado** tiene una BAJA COHESIóN, ya que realiza diversas tareas poco relacionadas (mostrar resultado e incremento de A para bucle)
- **Resultado** tiene un ALTO ACOPLAMIENTO funcional con **MAIN** en el control del bucle

- Siempre se utiliza el PASO POR REFERENCIA, por lo que aumentan las dependencias y el ACOPLAMIENTO entre subprogramas

- **MAIN** tiene una BAJA COHESIóN, ya que realiza diversas tareas poco relacionadas (entrada de datos, cómputo y salida)
- **MAIN** tiene un ALTO ACOPLAMIENTO funcional con **Resultado** en el control del bucle
- El control del bucle es confuso, complejo y erróneo
- La salida de resultados debería estar fuera del bucle

- **Error:** El valor de la variable B no se inicializa para cada iteración
- **Error:** Si (A==B+1) termina el bucle sin encontrar el número perfecto

Criterios de Modularización. Ejemplo 2 (v2)

```
inline bool es_divisible(int num, int den)
{
    return num % den == 0;
}
int suma_divisores(int num)
{
    int suma = 0;
    for (int i = 1; i <= num / 2; ++i) {
        if (es_divisible(num,  i)) {
            suma += i;
        }
    }
    return suma;
}
inline bool es_perfecto(int num)
{
    return num == suma_divisores(num);
}
int buscar_perfecto(int num)
{
    while (! es_perfecto(num)) {
        ++num;
    }
    return num;
}
```

```
void leer(int& num)
{
    cout << "Introduce valor inicial: ";
    cin >> num ;
}

void mostrar(int num)
{
    cout << "Número perfecto: " << num << endl
}

int main()
{
    int num;
    leer(num);
    mostrar( buscar_perfecto(num) );
}
```

BAJO ACOPLAMIENTO
Y
ALTA COHESIÓN

Variables Locales

- Las **variables locales** son:
  - Las variables declaradas dentro de un subprograma (incluida **`main`**).
  - Los parámetros formales de un subprograma.
- Una variable local se **crea** cuando el flujo de ejecución ejecuta la sentencia donde está declarada.
- Una variable local sólo puede ser **accedida dentro** del subprograma (o bloque) donde está declarada, a partir del punto donde ha sido declarada (regla de ámbito).
- Una variable local se **destruye** cuando el flujo de ejecución termina la ejecución del subprograma (o bloque) donde ha sido declarada.
- Cuando una variable local se declara **dentro de un bucle**, entonces se **crea** y se **destruye** en **cada iteración** (el valor **no perdura** entre iteraciones).

```
int menor(int x, int y)
{                ▲       ▲
    int z = x; ◀◀◀
    if (y < z) {
        z = y;
    }
    return z;
}
```

```
void ordenar(int& x, int& y)
{                    ▲       ▲
    if (x > y) {
        int z = x; ◀◀◀
        x = y;
        y = z;
    }
}
```

```
int main()
{
    int a, b; ◀◀◀
    cin >> a >> b;
    int c = menor(a, b);
    ordenar(a, b);
    bool ok = (c == a);
}
```

Variables Globales y Efectos Laterales

- Las **variables globales** son aquellas que se **declaran fuera** de los subprogramas (incluida `main`).
- Las variables globales pueden ser accedidas desde cualquier subprograma que aparezca posterior a su declaración.
- En nuestro contexto, se denonima **efecto lateral** al intercambio de información entre dos subprogramas realizado a través de variables globales.
- Debido a los problemas asociados a las variables globales y efectos laterales:

LA UTILIZACIÓN DE VARIABLES GLOBALES Y EFECTOS LATERALES ESTÁ PROHIBIDA

- **Problemas** asociados a las variables globales y efectos laterales:
  - Aumentan el **acoplamiento** entre los subprogramas que las utilizan.
  - Se crean **dependencias invisibles** entre subprogramas.
  - **Reducen** la posibilidad de **reutilización** de código.
  - Aumenta la posibilidad de **cometer errores** que son difíciles de encontrar y corregir.

Variables Globales y Efectos Laterales. Ejemplo

LA UTILIZACIÓN DE VARIABLES GLOBALES Y EFECTOS LATERALES ESTÁ PROHIBIDA

```
#include <iostream>
using namespace std;
const int MAX = 30;

int vble_global; // VARIABLE GLOBAL (PROHIBIDA)

void Sub1()
{
    int vble_local;
    vble_local = vble_global * MAX; // EFECTO LATERAL (PROHIBIDO)
    vble_global = vble_local + MAX; // EFECTO LATERAL (PROHIBIDO)
}
int main()
{
    int i,j;
    vble_global = 5;                // EFECTO LATERAL (PROHIBIDO)
    Sub1();
    cout << vble_global << endl;    // EFECTO LATERAL (PROHIBIDO)
}
```

Sobrecarga de Subprogramas

- Se denomina sobrecarga cuando distintos subprogramas se denominan con el mismo identificador.
- En C++ es posible sobrecargar subprogramas siempre y cuando tengan parámetros diferentes.

```
inline double media(int x, int y, int z)
{
    return double(x + y + z) / 3.0;
}
inline double media(int x, int y)
{
    return double(x + y) / 2.0;
}
```

- Con propósitos de eficiencia, cuando el cuerpo de un subprograma se reduce a una **simple expresión**, es posible definirlo `inline`.
  - En caso de `inline`, el **compilador optimiza** el código, **reemplazando** la **llamada** a un subprograma por el propio código del **cuerpo** del subprograma.
  - Se mantienen las ventajas de la **modularización** y **abstracción**.

```
double k = 5 * media(a, b, c) ;
double h = 5 * media(a, b) ;
```

```
double k = 5 * (double(a + b + c) / 3.0) ;
double h = 5 * (double(a + b) / 2.0) ;
```

Asertos y Excepciones

Asertos

- Se debe incluir la biblioteca `<cassert>`.
- Los asertos sirven para comprobar y **detectar errores de programación**.
  - Los asertos sirven para comprobar, en determinados puntos del programa, que determinadas **condiciones deben ser ciertas**, y si no son ciertas, entonces indican que se ha producido un **error de programación** en nuestro programa.
  - Forman parte del *código de depuración*.
- La sentencia `assert(` *expr-lógica* `);` evalúa la expresión lógica, y si ésta es `true`, entonces no hace nada, pero si la expresión lógica se evalúa a `false`, entonces se **aborta** la ejecución del programa con un mensaje indicando la situación de error.
- Los asertos se pueden desactivar con la opción de compilación -DNDEBUG, en cuyo caso la sentencia `assert(...)` no hace nada *(desaparece)*.

```
#include <cassert>
int main()
{
    ...
    assert(divisor != 0);
    int cociente = dividendo / divisor;
    int resto = dividendo % divisor;
    assert(dividendo == (divisor * cociente + resto));
    ...
}
```

Excepciones

- El lanzamiento de excepciones sirve para **informar** que se ha producido una **situación de error** excepcional durante la ejecución del programa.
  - Cuando sucede alguna situación de error en un programa, entonces se puede **lanzar** (`throw`) una excepción que informe de la situación de error excepcional.
  - La excepción que se lanza puede ser de cualquier tipo, aunque nosotros simplemente utilizaremos un **mensaje** (`"..."`) que informe del error encontrado.
- Si se lanza una excepción, entonces la ejecución del programa **aborta** en ese punto.
  - Las excepciones se suelen lanzar dentro se sentencias `if` que comprueben la situación de error.
  - Cuando se lanza una excepción, hay situaciones avanzadas en las que se puede **capturar** la excepción y **recuperar** la situación de error.

```
int main()
{
    ...
    if (divisor == 0) {
        throw  "Error: división por cero";
    }
    int cociente = dividendo / divisor;
    int resto = dividendo % divisor;
}
```

Precondiciones y Postcondiciones

Precondiciones

- **Precondición** es una condición que debe ser cierta **antes** de la invocación a un subprograma. Especifica las **condiciones** necesarias para poder ejecutar dicho subprograma.
  - Las **precondiciones** se pueden codificar mediante una sentencia `assert(...)`.
  - Las **precondiciones** también se pueden codificar mediante una sentencia `if` y el lanzamiento `throw` de una **excepción** si no se cumple la precondición.
  - Ayudan a **detectar errores** de programación si el subprograma que invoca **no cumple las precondiciones** especificadas por el subprograma invocado.

Postcondiciones

- **Postcondición** es una condición que debe ser cierta **tras** la ejecución de un subprograma. Especifica el **comportamiento** de dicho subprograma.
  - Las **postcondiciones** se pueden codificar mediante una sentencia `assert(...)`.
  - Ayudan a **detectar errores** de programación al comprobar si el **comportamiento** del subprograma no es adecuado.

- *A veces no es posible codificar las precondiciones y postcondiciones adecuadamente.*

Precondiciones y Postcondiciones. Ejemplo (v1)

```
#include <iostream>
#include <cassert>
using namespace std;
void dividir(int dividendo, int divisor, int& cociente, int& resto)
{
    if (divisor == 0) {                                    // PRECOND
        throw  "Precond-Error: argumentos erróneos";
    }
    cociente = dividendo / divisor;
    resto = dividendo % divisor;
    assert(dividendo == (divisor * cociente + resto)); // POSTCOND
}
int main()
{
    int dividendo, divisor, cociente, resto;
    cout << "Introduce el dividendo y divisor: ";
    cin >> dividendo >> divisor;
    // Error, antes de invocar a dividir hay que comprobar que divisor != 0
    dividir(dividendo, divisor, cociente, resto);
    cout << "Cociente: " << cociente << "  Resto: " << resto << endl;
}
```

Precondiciones y Postcondiciones. Ejemplo (v2)

```
#include <iostream>
#include <cassert>
using namespace std;
void dividir(int dividendo, int divisor, int& cociente, int& resto)
{
    if (divisor == 0) {                                    // PRECOND
        throw  "Precond-Error: argumentos erróneos";
    }
    cociente = dividendo / divisor;
    resto = dividendo % divisor;
    assert(dividendo == (divisor * cociente + resto)); // POSTCOND
}
int main()
{
    int dividendo, divisor, cociente, resto;
    cout << "Introduce el dividendo y divisor: ";
    cin >> dividendo >> divisor;
    if (divisor == 0) {
        cout << "Error: división por cero" << endl;
    } else {
        dividir(dividendo, divisor, cociente, resto);
        cout << "Cociente: " << cociente << "  Resto: " << resto << endl;
    }
}
```

Algunas Funciones de la Biblioteca <cmath>

Funciones de <cmath>	Significado
`double hypot(double x, double y)`	hipotenusa de x e y ($\equiv \sqrt{x^2 + y^2}$)
`double sqrt(double x)`	raiz cuadrada de x, $\sqrt{x}$, $x \geq 0$
`double cbrt(double x)`	raiz cúbica de x, $\sqrt[3]{x}$
`double pow(double x, double y)`	x^y
`double exp(double x)`	e^x
`double exp2(double x)`	2^x
`double log(double x)`	logaritmo natural, $ln(x)$, $x > 0$
`double log2(double x)`	logaritmo binario, $log_2(x)$, $x > 0$
`double log10(double x)`	logaritmo decimal, $log_{10}(x)$, $x > 0$
`double ceil(double x)`	menor entero $\geq x$, $\lceil x \rceil$
`double floor(double x)`	mayor entero $\leq x$, $\lfloor x \rfloor$
`double trunc(double x)`	valor entero de x, sin decimales
`double round(double x)`	valor entero más cercano a x
`double fabs(double x)`	valor absoluto de x, $\lvert x \rvert$
`double fmod(double x, double y)`	resto de x / y
`double sin(double r)`	seno, $sin(r)$ (en radianes)
`double cos(double r)`	coseno, $cos(r)$ (en radianes)
`double tan(double r)`	tangente, $tan(r)$ (en radianes)
`double asin(double x)`	arco seno, $arcsin(x)$, $x \in [-1,1]$
`double acos(double x)`	arco coseno, $arccos(x)$, $x \in [-1,1]$
`double atan(double x)`	arco tangente, $arctan(x)$
`double atan2(double y, double x)`	arco tangente, $arctan(y/x)$

Algunas Funciones de la Biblioteca <cmath>. Ejemplo

```
#include <iostream>
#include <cmath>
using namespace std;
void leer(double& x, double& y)
{
    cout << "Introduce los valores de  los dos catetos: ";
    cin >> x >> y;
}
void mostrar(double x)
{
    cout << "Hipotenusa: " << x << endl;
}
int main()
{
    double x, y;
    leer(x, y);
    double h1 = hypot(x, y);
    mostrar(h1);
    double h2 = sqrt(x*x + y*y);
    mostrar(h2);
    double h3 = sqrt(pow(x, 2) + pow(y, 2));
    mostrar(h3);
}
```

Recursividad

- Técnica de programación **alternativa** al uso de estructuras iterativas para la resolución de procesos repetitivos.
  - Soluciones elegantes, simples, estructuradas y modulares.
- **Subprograma recursivo**:
  - El que se **invoca a sí mismo** para resolver una ***"versión más pequeña"*** del problema para el que ha sido diseñado.
- También es posible que un subprograma **A** pueda invocar a otro subprograma **B**, el cual puede volver a invocar al subprograma **A** mediante **recursividad indirecta**.
  - En este caso, es necesario declarar el prototipo del subprograma.

Recursividad. Ejemplo 1

- Ejemplo: calcular el **factorial** de un número de forma iterativa.

$$n! = \begin{cases} 1, & si\ n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 1, & si\ n > 0 \end{cases}$$

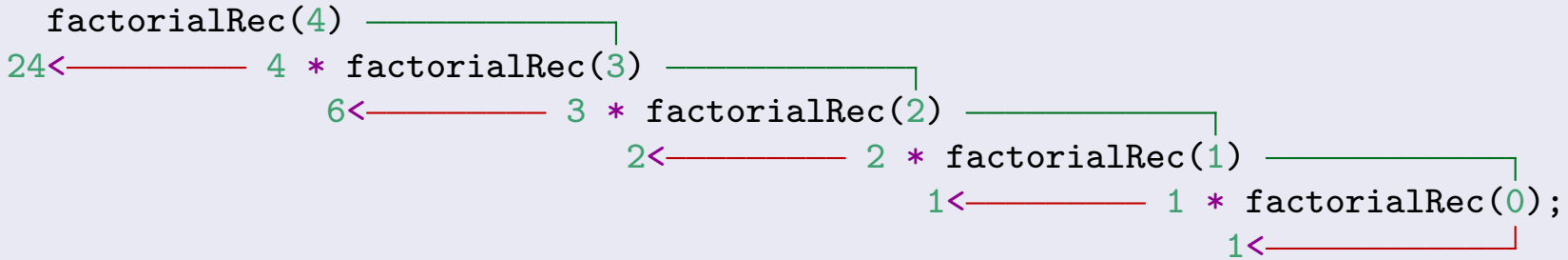
```
long factorialIterativo(int n)
{
    long fn = 1;
    for (int i = 2; i <= n; ++i) {
        fn = fn * i;
    }
    return fn;
}
```

Recursividad. Ejemplo 2

- Ejemplo: calcular el **factorial** de un número de forma recursiva.

$$n! = \begin{cases} 1, & si\ n = 0 \\ n \cdot (n-1)!, & si\ n > 0 \end{cases}$$

```
long factorialRec(int n)
{
    long fn;
    if (n == 0) {                          caso base
        fn = 1;
    } else {
        fn = n * factorialRec(n-1);        llamada recursiva
    }                                      problema más pequeño
    return fn;
}
```



Recursividad

Concepto de Recursividad

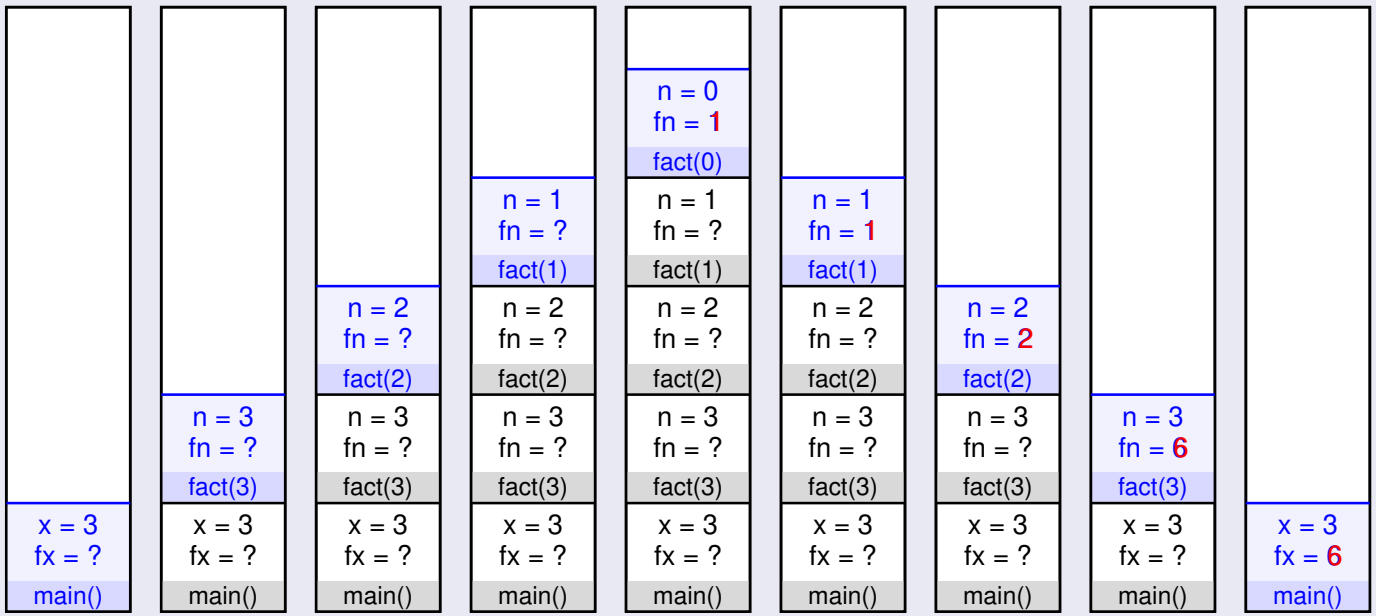
- Los subprogramas recursivos se **invocan a sí mismos**.
- Cada **llamada recursiva** se hace con parámetros de *menor tamaño* que el de la anterior llamada. Así, cada vez se está invocando **a otro problema idéntico pero de menor tamaño**.
- Existe un caso especial, o **caso base**, en el que **no se utiliza la recursividad**.
- La forma en la que el tamaño del **problema disminuye** en cada llamada recursiva asegura que se **llegará a este caso base**, y finalice la recursividad.

En general, para determinar si un algoritmo recursivo está **bien diseñado** debemos plantearnos tres preguntas:

1. ¿ Existe uno o varios **casos base** del subprograma, y éste funciona correctamente para ellos ? ¿ Se alcanzan ?
2. ¿ Cada llamada recursiva al subprograma se refiere a un **caso más pequeño** del problema original ?
3. Suponiendo que las llamadas recursivas funcionan correctamente, así como el caso base, ¿ funciona correctamente todo el subprograma ?

Gestión de Memoria en la Invocación a Subprogramas

- En cada llamada a un subprograma, tanto los parámetros como las variables locales se crean nuevas en un nuevo **bloque de memoria de trabajo**.
- Cuando termina la ejecución de un subprograma, su **bloque de memoria de trabajo** se elimina.




```
int fact(int n)
{
  int fn;
  if (n == 0) {
    fn = 1;
  } else {
    fn = n * fact(n-1);
  }
  return fn;
}

int main()
{
  int x, fx;
  cout << "Número: ";
  cin >> x;
  fx = fact(x);
  cout << fx << endl;
}
```

Recursividad. Ejemplo 3

- La serie de **Fibonacci** es una secuencia de números que aparece con frecuencia en matemáticas y en la naturaleza.
  - La serie comienza con los dos primeros números **0** y **1**, y cada número del resto de la secuencia se calcula como la suma de los dos números anteriores: **0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...**

$$fib(n) = \begin{cases} n, & si\ n < 2 \\ fib(n-1) + fib(n-2), & si\ n \geq 2 \end{cases}$$

```
int fib(int n)
{
    int fn;
    if (n < 2) {                                  caso base
        fn = n;
    } else {
        fn = fib(n-1) + fib(n-2);      llamadas recursivas
    }                                  problemas más pequeños
    return fn;
}
```

Recursividad vs. Iteración

- La recursividad es una técnica de programación potente para resolver problemas, que a menudo produce **soluciones simples y claras**.
- Sin embargo, la recursividad también tiene algunas desventajas, las cuales se enmarcan en el campo de la eficiencia. **Muchas veces un algoritmo iterativo es más eficiente que su correspondiente recursivo**. Existen dos factores que contribuyen a ello:
  1. La **sobrecarga** asociada con las llamadas a subprogramas.
  2. La **ineficiencia** inherente de algunos algoritmos recursivos.

Recursividad vs. Iteración

Sobrecarga asociada con las llamadas a subprogramas

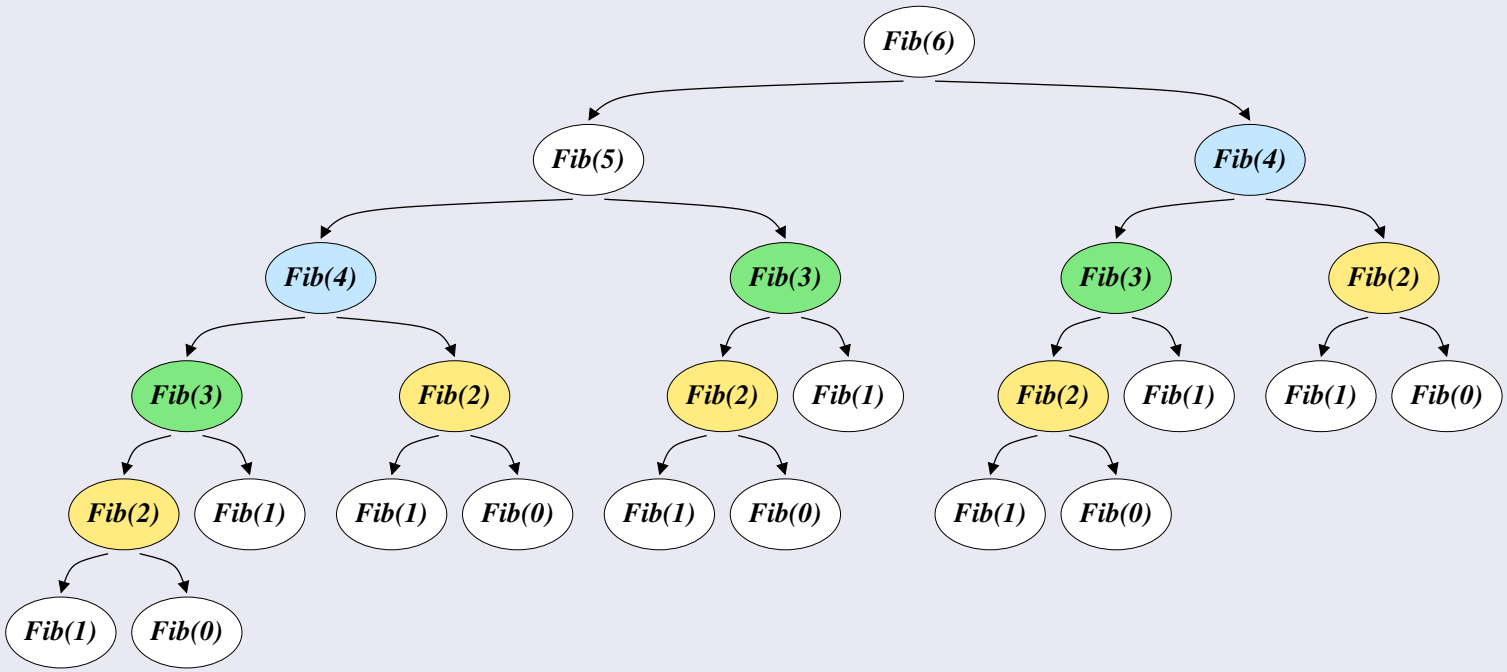
- Cuando se produce una **llamada** a un subprograma se debe realizar el **paso de parámetros**, y se deben crear las **variables locales** del subprograma llamado.
- Cuando **finaliza** la ejecución del subprograma llamado, se deben **destruir** todas esas variables locales y los parámetros.
- Conlleva una **sobrecarga** en tiempo de ejecución y en utilización de memoria.
- **Esta sobrecarga es mayor cuando se utiliza la recursividad**, ya que una simple llamada inicial a un subprograma puede generar un gran número de llamadas recursivas.
- Muchos **compiladores** pueden realizar **optimizaciones** en *algunos casos* (por ej. la **recursividad de cola** en el subprograma `factorialRec`), de forma que la ejecución sea tan eficiente como la del **equivalente iterativo**.

- No debemos usar la recursividad cuando sea posible diseñar un **algoritmo iterativo igual de sencillo**. Por ejemplo, deberíamos usar la función iterativa `factorialIterativo` en lugar de su equivalente recursiva `factorialRec`.

Recursividad vs. Iteración

Ineficiencia inherente de algunos algoritmos recursivos

- Esta ineficiencia es debida al **método empleado** para resolver el problema concreto que se esté abordando, debido a su simplicidad.
- Por ejemplo, en la función `fib(int)` anterior, apreciamos un gran inconveniente: los mismos valores son calculados varias veces.
  - Por ejemplo, para calcular `fib(6)`, tenemos que calcular `fib(4)` dos veces, `fib(3)` tres veces, y `fib(2)` cinco veces. Mientras más grande sea `n`, mayor ineficiencia.



Recursividad vs. Iteración

- Se puede diseñar un algortimo iterativo que calcule los números de la serie de Fibonacci de manera más eficicente.

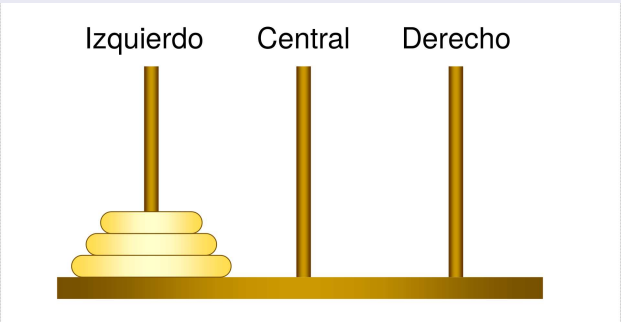
```
int fib(int n)
{
    int fk, fk1, fk2;            // variables para almacenar F_k, F_k-1 y F_k-2
    if (n < 2) {                 // si N es menor que 2
        fk = n;                          // el resultado F_0 es 0 y F_1 es 1
    } else {                     // en otro caso
        fk1 = 0;                         // primer valor de la sucesión F_0 es 0
        fk  = 1;                         // segundo valor de la sucesión F_1 es 1
        for (int k = 2; k <= n; ++k) { // iterar para los valores de K ∈ { 2...N }
            fk2 = fk1;                     //  asignar a F_k-2 el valor de F_k-1 anterior
            fk1 = fk;                      //  asignar a F_k-1 el valor de F_k anterior
            fk = fk1 + fk2;                //  asignar a F_k el nuevo valor F_k-1+F_k-2
        }
    }
    return fk;                   // devolver el resultado F_n calculado
}
```

Recursividad vs. Iteración

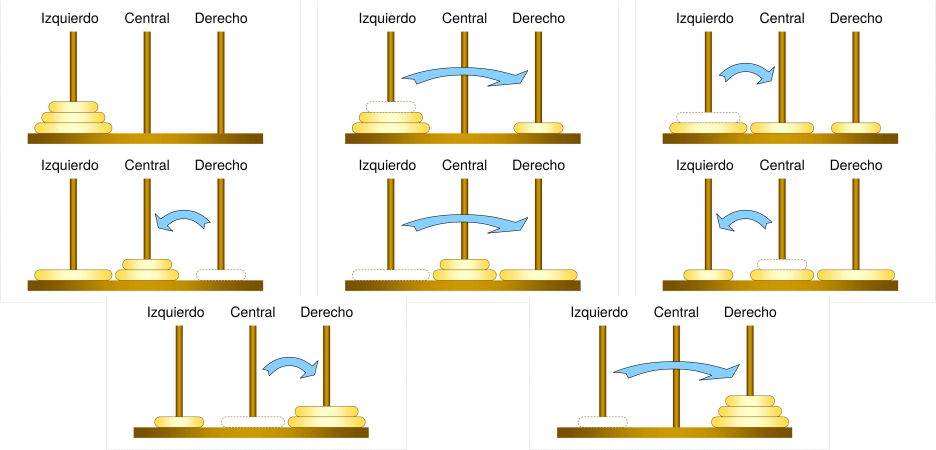
- A pesar de todo esto, en muchas circunstancias el uso de la **recursividad** permite a los programadores especificar **soluciones naturales y sencillas** que serían difíciles de resolver mediante técnicas iterativas.
- En estos casos la sencillez y naturalidad de la solución compensa la posible ineficiencia en la ejecución.

Torres de Hanoi

- Se tienen **3 palos** de madera (izquierdo, central, derecho). El palo izquierdo tiene ensartados un montón de discos concéntricos de tamaño decreciente, de manera que el **disco más pequeño está arriba, y el mayor está abajo**.
- Problema: **mover todos los discos del palo izquierdo al derecho** respetando las siguientes reglas:
  - Sólo se puede mover un disco cada vez.
  - No se puede poner un disco encima de otro más pequeño.
  - Después de un movimiento, todos los discos han de estar en alguno de los tres palos.

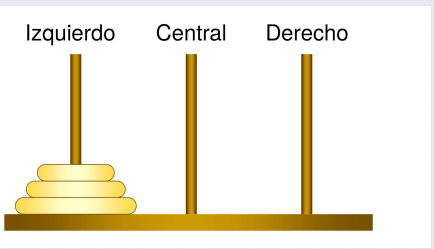


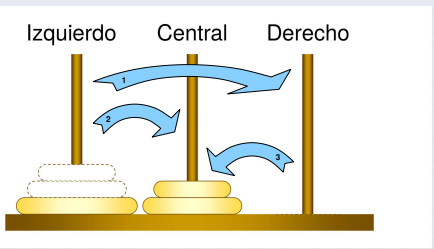
Recursividad. Torres de Hanoi

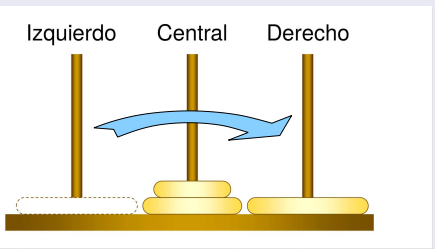


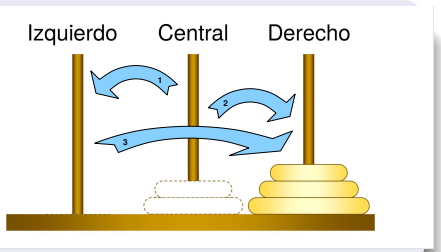
Torres de Hanoi. Solución Recursiva

- Si el **número de discos N a mover es igual a 1**, el problema se resuelve inmediatamente moviendo directamente el disco del palo **origen** al **destino**.
- Si el **número de discos N a mover es mayor que 1**, desde un palo **origen** (en nuestro caso el izquierdo) a un palo **destino** (en nuestro caso el derecho), utilizando el otro palo como **auxiliar** (en nuestro caso el central):
  1. Mover los **N-1** discos superiores del palo **origen** al palo **auxiliar**, utilizando en este paso el palo destino como palo auxiliar.
  2. Mover el disco que queda del palo origen al destino.
  3. Mover los **N-1** discos del palo auxiliar al palo destino, utilizando el palo origen como palo auxiliar.









Recursividad. Torres de Hanoi

```
const int IZQUIERDO = 1;
const int CENTRAL   = 2;
const int DERECHO   = 3;

void escribirPalo(int p)
{
    switch (p){
    case IZQUIERDO: cout << "izquierdo"; break;
    case CENTRAL:   cout << "central";   break;
    case DERECHO:   cout << "derecho";   break;
    }
}
void mueveUno(int origen, int destino)
{
    escribirPalo(origen);
    cout << " -> ";
    escribirPalo(destino);
    cout << endl;
}
void mueve(int n, int origen, int auxiliar, int destino)
{
    if (n == 1) {
        mueveUno(origen, destino);
    } else {
        mueve(n-1, origen, destino, auxiliar);
        mueveUno(origen, destino);
        mueve(n-1, auxiliar, origen, destino);
    }
}
```

```
Introduce número de discos: 3
Movimientos de discos:
izquierdo -> derecho
izquierdo -> central
derecho -> central
izquierdo -> derecho
central -> izquierdo
central -> derecho
izquierdo -> derecho
```

```
int main()
{
  int n;
  cout << "Introduce número de discos: ";
  cin >> n;
  cout << "Movimientos de discos: " << endl;
  mueve(n, IZQUIERDO, CENTRAL, DERECHO);
}
```

Recursividad. Ejemplo MCD

El máximo común divisor (*mcd*) de dos números enteros positivos **p** y **q** es el mayor entero **d** que divide a ambos.
Un algoritmo muy conocido para calcularlo es el de *Euclides*. Éste utiliza dos variables, que contienen inicialmente a cada uno de los números, y trata de hacer el valor de ambas variables sea el mismo.
Para ello, irá restando el valor manor a la variable con mayor valor, hasta que ambas variables contengan el mismo valor. En dicho momento, el valor obtenido en cualquiera de ellas es el máximo común divisor de los dos números iniciales.

- Por ejemplo, si P = 18 y Q = 12, el algoritmo hará que P y Q vayan tomando los siguientes valores:

Inicialmente	P == 18	y	Q == 12	(P > Q => P = P - Q)
Después	P == 6	y	Q == 12	(Q > P => Q = Q - P)
Después	P == 6	y	Q == 6	(P == Q => El mcd es 6)

Desarrolla un programa según el algoritmo anterior siguiendo un enfoque **recursivo**.

Recursividad. Ejemplo MCD

```
#include <iostream>
using namespace std;
void leer(int& P, int& Q)
{
    cout << "Introduzca dos numeros positivos: ";
    cin >> P >> Q;
}
int mcd(int P, int Q)
{
    int res;
    if (P == Q) {
        res = P;
    } else if (P > Q) {
        res = mcd(P-Q, Q);
    } else {
        res = mcd(P, Q-P);
    }
    return res;
}
int main()
{
    int P, Q;
    leer(P, Q);
    if ((P <= 0) || (Q <= 0)) {
        cout << "Error" << endl;
    } else {
        cout << "El MCD es: " << mcd(P, Q) << endl;
    }
}
```