Tema 3. Abstracción Procedimental

Vicente Benjumea García

Programación-I
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.
  - Paso de parámetros por valor y punteros.
  - Invocación a procedimientos y funciones. Argumentos.
- Diseño descendente y abstracción procedimental.
- Criterios de modularización: Acoplamiento y Cohesión.
- Asertos, precondiciones y postcondiciones.
- Variables locales, globales y estáticas.
- Algunas Funciones de la Biblioteca `<math.h>`.
- Introducción a la Recursividad.

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** (parámetros formales).
  - El **cuerpo** del subprograma especifica la secuencia de **acciones** que resuelven un subproblema, utilizando los parámetros. Define sus propias **variables locales**.
- Donde sea necesaria la resolución del subproblema, se **invocará** al subprograma, especificando el **nombre** del subprograma y los **argumentos** (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) { // Atención
        int z = *x; ◀◀◀
        *x = *y;
        *y = z;
    }
}
```

```
// Principal
int main()
{
    int a, b; ◀◀◀
    scanf(" %d %d", &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** (información de entrada).
  - El **cuerpo** de la función especifica la secuencia de **acciones** que resuelven un subproblema, utilizando los parámetros. Define sus propias **variables locales**.
  - 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 recomendada**.

```
 ⇑            ⇓
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; ◀◀◀
  scanf(" %d %d", &a, &b);
  int c = vabs(a);
  int d = vabs(b);
  int e = menor(c, d);
  printf("%d\n", 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**.
  - El **cuerpo** del procedimiento especifica la secuencia de **acciones** que resuelven un subproblema, utilizando los parámetros. Define sus propias **variables locales**.
  - **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)
{
  printf("Número: ");
  scanf(" %d", x);
}
                ⇓
void mostrar(int x)
{
  printf("Resultado: %d\n", x);
}
```

```
                    ⇕      ⇕
void ordenar(int* x, int* y)
{
    if (*x > *y) { // Atención
        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

- 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:** aparecen en la **definición** del subprograma (*parámetros formales*).
- **Argumentos:** aparecen en la **invocación** al subprograma (*parámetros actuales*).
- Los parámetros especifican:
  - **Tipo** que define el tipo del intercambio de la información.
  - **Nombre** con el que identificar a cada parámetro.
  - **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.

- Dependiendo del contexto en el que se utilice:
  - A veces, se usa **parámetro formal** como sinónimo de **parámetro** en la definición del subprograma.
  - A veces, se usa **parámetro actual** como sinónimo de **argumento** en la invocación del subprograma.

- En el lenguaje **C**, la transferencia de información entre subprogramas se implementa mediante el **paso por valor** y el **paso por valor** de **direcciones** (**punteros**).

Parámetros. 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)
{
   printf("Datos: ");
   scanf(" %d", 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) { // Atención
        int z = *x;
        *x = *y;
        *y = z;
    }
}

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

Paso de Parámetros por Valor y Punteros

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 actúa como una **nueva variable independiente**, inicializada con una **copia del valor** del argumento.
  - El parámetro **es una variable distinta** del argumento (aislamiento).
  - La modificación de la variable del parámetro **no afecta** al argumento.

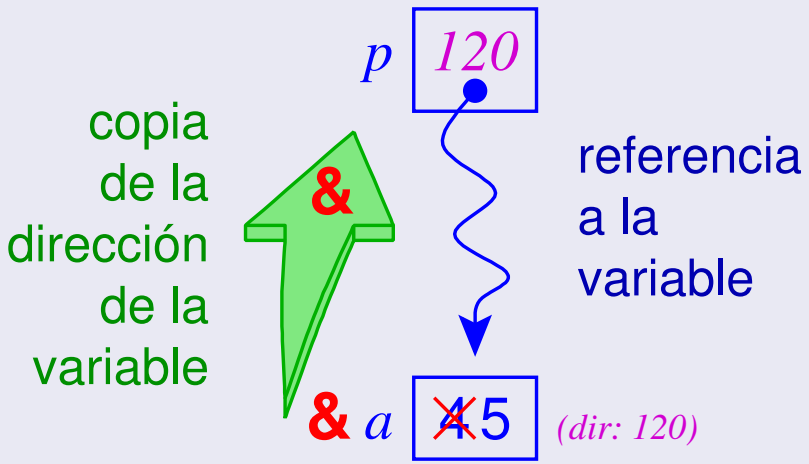
```
 ⇑           ⇓
int vabs(int x)
{
   if (x < 0) {
      x = -x;
   }
   return x;
}
int main()
{
   int a = -3;
   int b = vabs(a);
}
```

Paso de Parámetros por Valor y Punteros

Paso de Parámetros por Valor de Punteros (Direcciones)

- Parámetros de **Salida** y **Entrada/Salida**: se utiliza el **paso por valor** de **punteros** (**direcciones**).
- En el **paso por valor** de **punteros** (`int* x`) el parámetro se inicializa con la **dirección de la variable** situada como argumento, referenciando a la dirección de memoria de la variable.
  - El parámetro contiene la **dirección de la variable** del argumento.
  - La modificación del valor de la variable referenciada por el parámetro **también modifica** el valor de la variable del argumento.

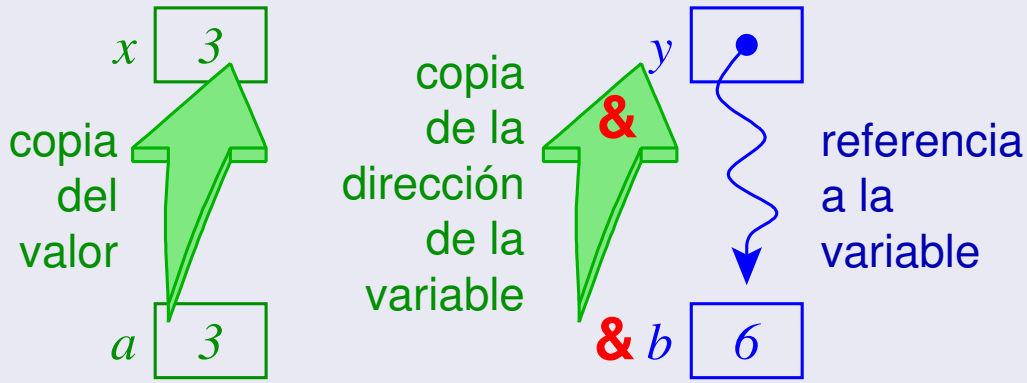
```
                          ⇕
void incremento(int* p)
{
    *p = *p + 1;
}
int main()
{
    int a = 4;
    incremento( &a );
}
```



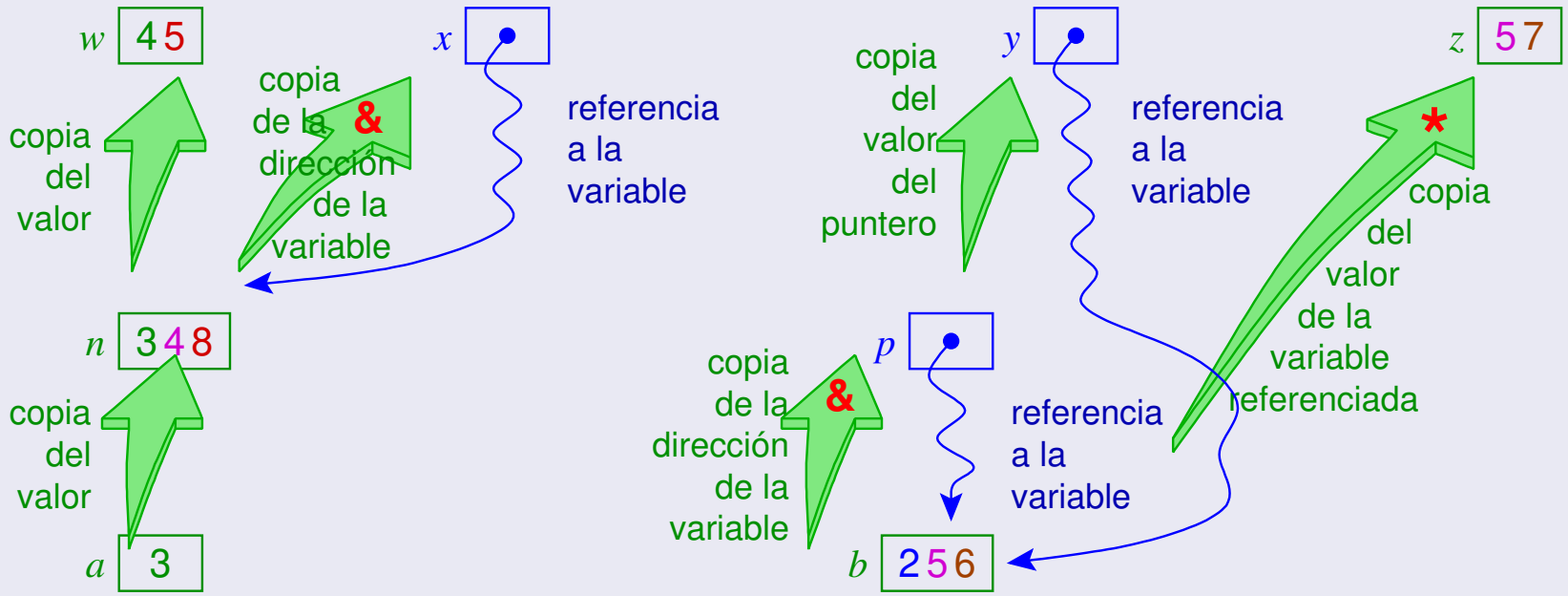
Paso de Parámetros por Valor y Punteros. Ejemplo

- 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 actúa como una **nueva variable independiente** inicializada con una **copia del valor** del argumento.
- El **paso por valor** de **punteros** permite realizar la transferencia de información de **salida** y **entrada/salida** de forma *eficiente*.
  - En el **paso por valor** de **punteros** el parámetro se inicializa con la **dirección de la variable** situada como argumento, referenciando a la dirección de memoria de la variable.

```
                          ⇓        ⇑
void subprograma(int x, int* y)
{
   *y = 2 * x;
}
int main()
{
   int a, b;
   a = 3;
   subprograma(a,  &b);
}
```



Paso de Parámetros por Valor y Punteros. Ejemplo




```
                    ⇓       ⇕       ⇕       ⇓
void proc_2(int w, int* x, int* y, int z)
{
   w = w + 1;
   *x = *x + 4;
   *y = *y + 1;
   z = z + 2;
}
```

```
                    ⇓       ⇕
void proc_1(int n, int* p)
{
   n = n + 1;
   *p = 3 + *p;
   proc_2(n,  &n, p, *p);
}
```

```
int main()
{
   int a = 3;
   int b = 2;
   proc_1(a, &b);
}
```

Paso de Parámetros por Valor y Punteros. Errores

- Cuando se trabaja con punteros, es muy facil cometer errores que son muy difíciles de detectar, ya que el compilador no avisa de ellos.
  - Uno de los errores más habituales consiste en utilizar los punteros, en asignaciones, comparaciones y operaciones, en lugar de las variables referenciadas.

```
// CORRECTO
                        ⇕
void incremento(int* p)
{
  *p = *p + 1;       // Atención
}
                    ⇕       ⇕
void ordenar(int* x, int* y)
{
    if (*x > *y) {  // Atención
        int z = *x;
        *x = *y;    // Atención
        *y = z;
    }
}
int main()
{
   int a = 5, b = 3;
   ordenar(&a, &b);
   incremento( &b );
}
```

```
// INCORRECTO
                        ⇕
void incremento(int* p)
{
  p = p + 1;         // Error. Incrementa el puntero
}
                    ⇕       ⇕
void ordenar(int* x, int* y)
{
    if (x > y) {    // Error. Compara los punteros
        int z = *x;
        x = y;      // Error. Asigna los punteros
        *y = z;
    }
}
int main()
{
   int a = 5, b = 3;
   ordenar(&a, &b);
   incremento( &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 **argumentos** (también denominados 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) { // Atención
        int z = *x;
        *x = *y;
        *y = z;
    }
}
```

```
int main()
{
    int a, b;
    scanf(" %d %d", &a, &b);
▶▶  int c = menor(a, b);
▶▶  ordenar(&a, &b);
    bool ok = (c == a);
}
```

Utilización de Subprogramas. Argumentos

- El **número de argumentos** en la invocación debe **coincidir** con el número de parámetros en la definición.
- Debe haber **correspondencia posicional** entre argumentos y parámetros.
- El **tipo** del argumento debe **coincidir** con el tipo del parámetro correspondiente.
- Un parámetro de **entrada** *(paso por valor)* requiere que el argumento sea una **variable**, **constante** o **expresión**.
- Un parámetro de **salida** o **entrada/salida** *(paso por valor de puntero)* requiere que el argumento sea un **puntero a una variable** (una dirección de una variable).

	Tipos Simples	
	(⇓)Ent	(⇑)Sal(⇕)E/S
Parámetro	P.Valor (int x)	P.Valor.Puntero (int* x)
Argumento	Constante Variable Expresión	Dirección de 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*.
   - **Copia de direcciones** de memoria en el *paso por valor de punteros*.
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) { // Atención
        int z = *x; ◀◀◀
        *x = *y;
        *y = z;
    }
}
```

```
int main()
{
    int a, b; ◀◀◀
    scanf(" %d %d", &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,    10,     x,      y, &c);
    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

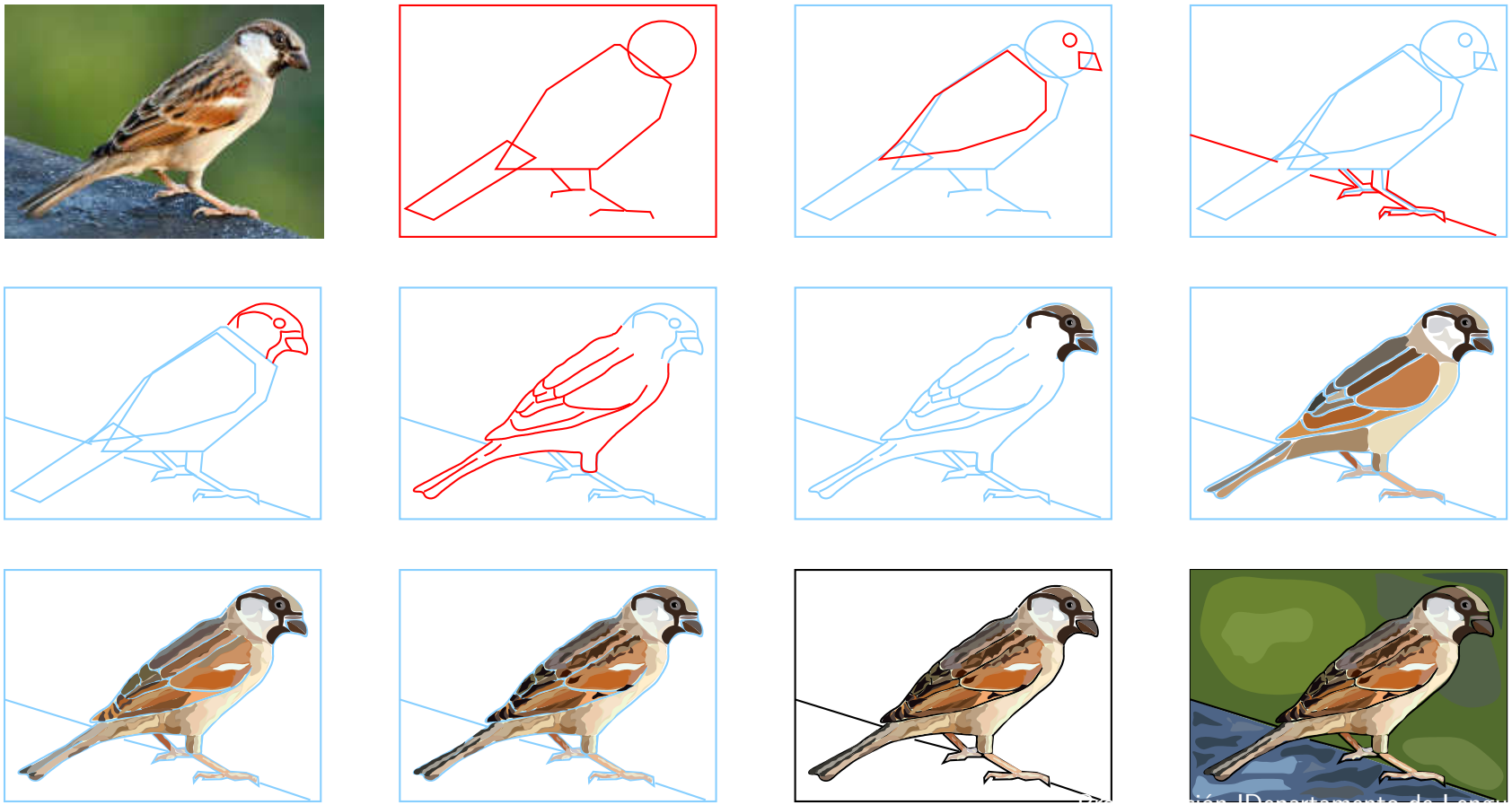
- La **definición** de un subprograma debe aparecer **antes** de su **invocación**.
- **Prototipo**: es posible declarar un determinado subprograma sin necesidad de definirlo explícitamente. **Aunque NO se recomienda**.
  - Define la **cabecera** del subprograma, terminada en **punto-y-coma**, y el cuerpo 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;
     scanf(" %d %d", &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) { // Atención
        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.



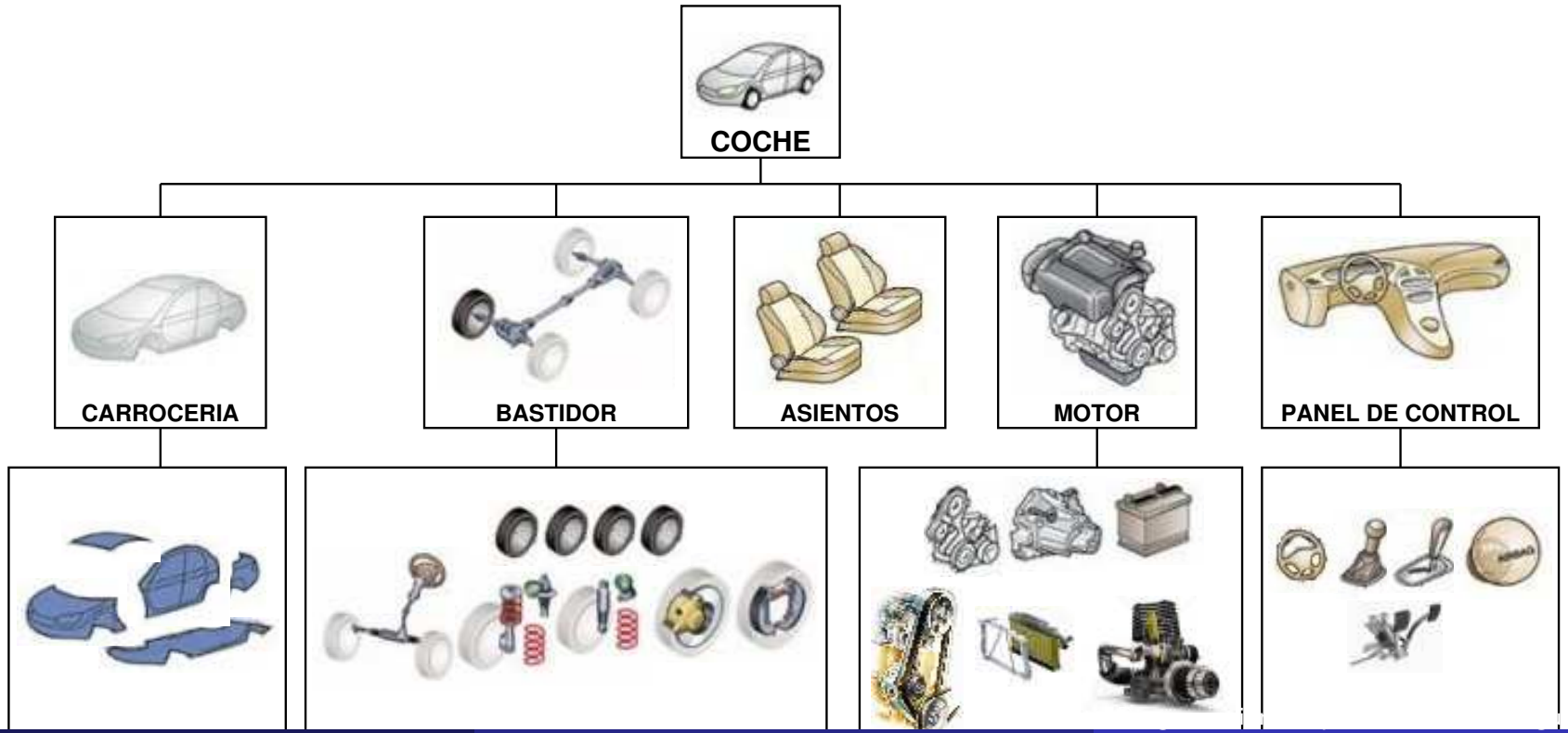
Diseño Descendente (II)

- 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.

Programación-IDepartamento de

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

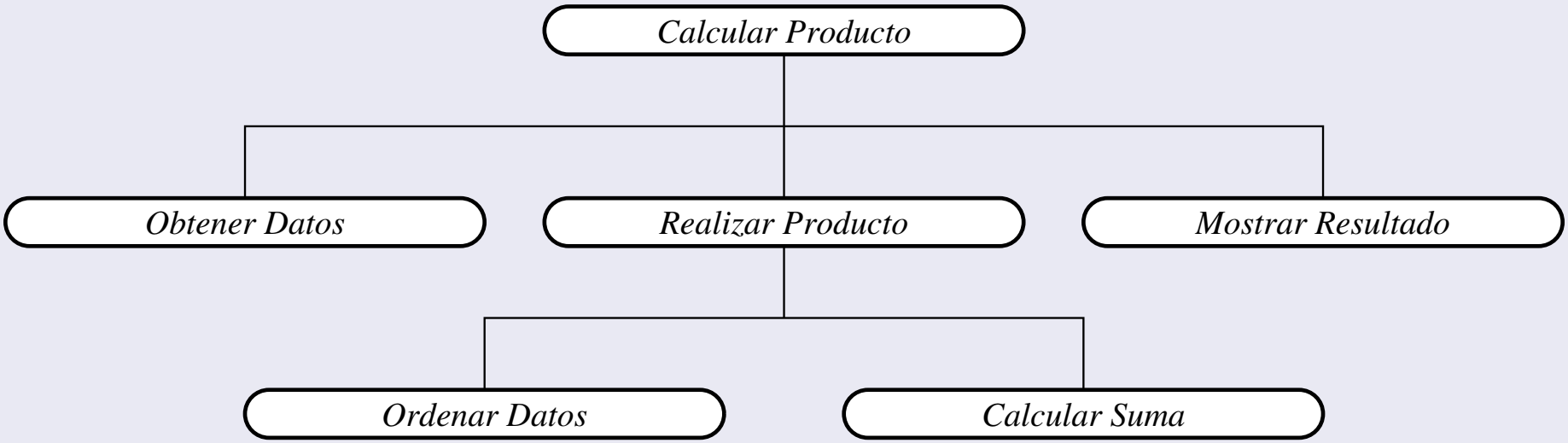
- Utilizamos **abstracción** y aplicamos **refinamientos sucesivos**:
  - Dividimos un **problema** en **subproblemas**.
    - Asocia un **subprograma** a cada subproblema.
  - Determinamos la **funcionalidad** del subprograma.
    - ¿ Que subproblema resuelve ?
  - Determinamos la **transferencia de 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 qué 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 producirá **menos errores** en su utilización. Además, será más fácil de comprobar y depurar.

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.

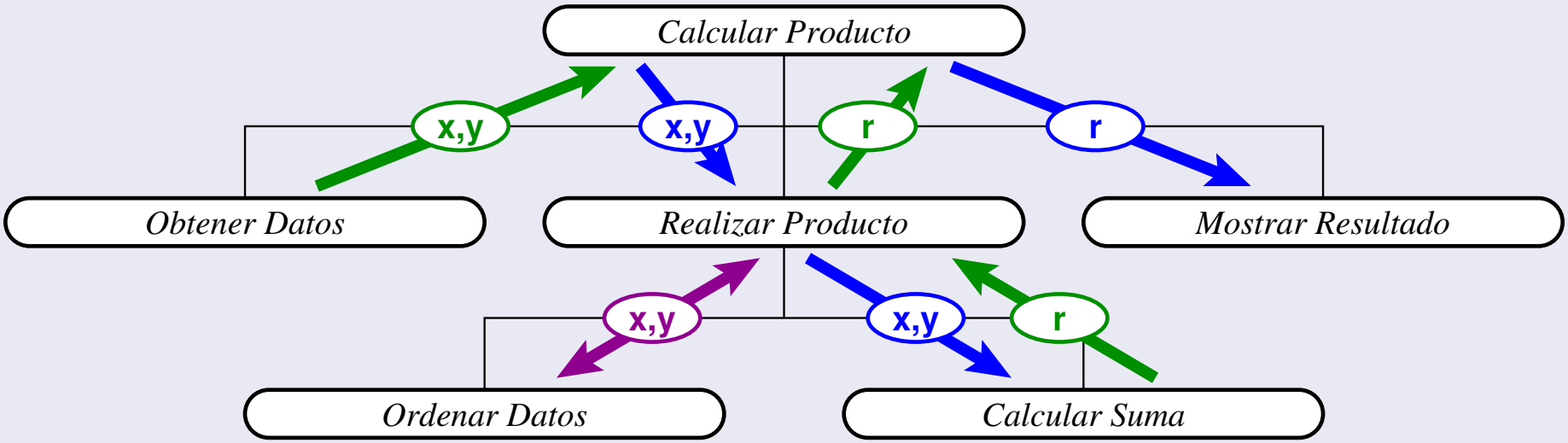
Diseño Descendente: Abstracción Procedimental. Ejemplo

- 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

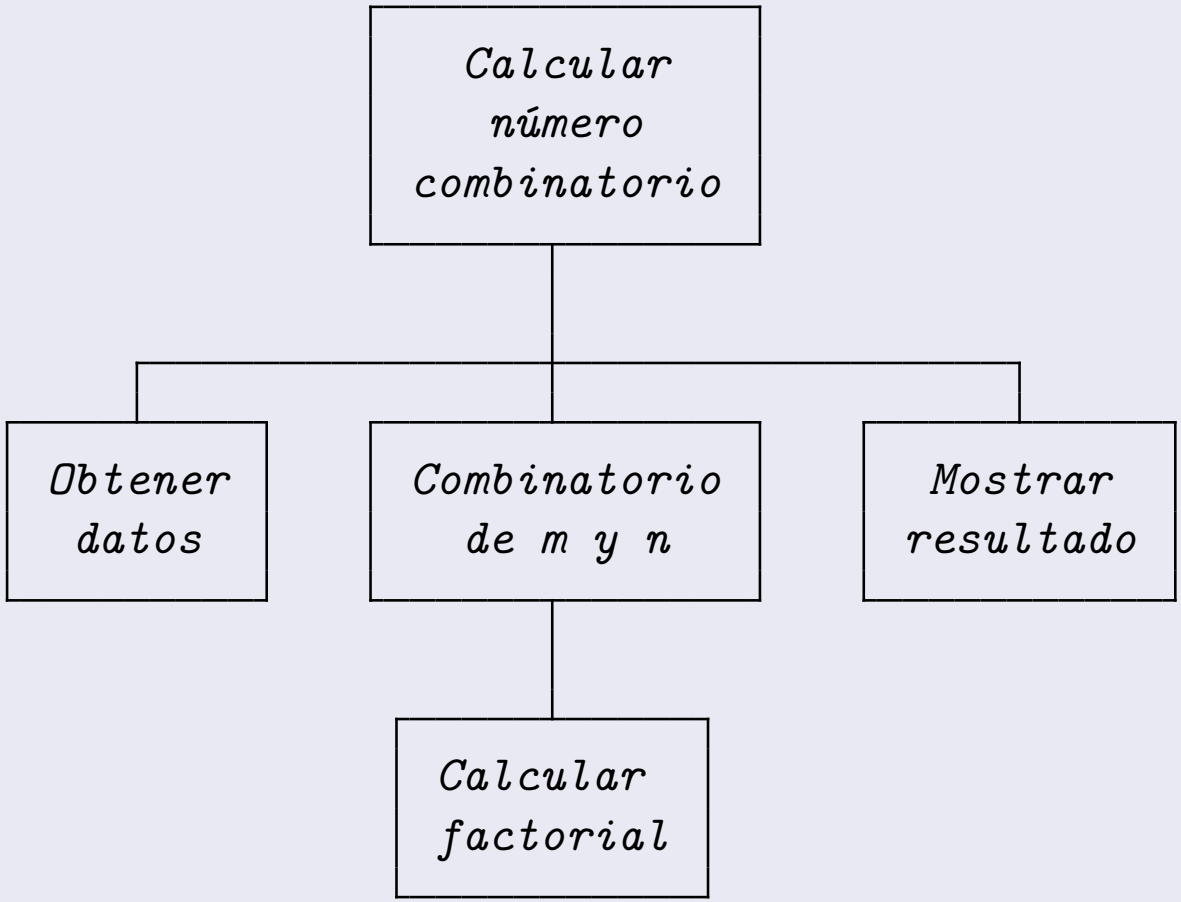
```c
#include <stdio.h>

                    ⇓       ⇓
void ordenar(int* x, int* y)
{
  if (*x > *y) { // Atención
    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
```

```c
 ⇑                              ⇓       ⇓
int realizarProducto(int x, int y)
{
  ordenar(&x, &y);
  return calcularSuma(x, y);
}
                      ⇑       ⇑
void leerDatos(int* x, int* y)
{
  printf("Introduce dos números ");
  scanf(" %d %d", x, y);  // Atención
}
                              ⇓
void mostrarResultado(int r)
{
  printf("Resultado: %d\n", r);
}
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

```c
#include <stdio.h>

void leerDatos(int* m, int* n) {
    printf("Introduce m y n (m >= n): ");
    scanf(" %d %d", m, n);
    while (*m < *n) { // Atención
        printf("Error. Introduce m y n (m >= n): ");
        scanf(" %d %d", m, n);
    }
}
void mostrarResultado(int m, int n, long res) {
    printf("El número combinatorio de %d sobre %d es %ld\n", m, n, res);
}
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

```c
#include <stdio.h>

void leer(int* num) {
    printf("Introduzca número: ");
    scanf(" %d", 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;
}
```

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

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) {
    printf("Introduce número: ");
    scanf(" %d", 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) {
        printf("Error\n");
    } else {
        printf("Valor: %d\n", fibonacci(n));
    }
}
```

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;
        }
    }
    printf("Resultado: %d\n", *cnt);
}
```

```
int main()
{
    int numero, cnt = 0;
    printf("Introduce número: ");
    scanf(" %d", &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.
- **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

```c
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)
{
    printf("Resultado: %d\n", cnt);
}
```

```c
int leer_num()
{
    int numero;
    printf("Introduce número: ");
    scanf(" %d", &numero);
    return numero;
}
int main()
{
    int numero = leer_num();
    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) {
        printf("Número perfecto: %d\n", *a);
    } else {
        ++ *a;
    }
}
int main()
{
    int a, b = 0;
    printf("Introduce valor inicial: ");
    scanf(" %d", &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) {
        printf("Número perfecto: %d\n", *a);
    } else {
        ++ *a;
    }
}
int main()
{
    int a, b = 0;
    printf("Introduce valor inicial: ");
    scanf(" %d", &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 DE PUNTEROS, 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;
}
```

```
int leer_num()
{
    int numero;
    printf("Introduce valor inicial: ");
    scanf(" %d", &numero);
    return numero;
}

void mostrar(int num)
{
    printf("Número perfecto: %d\n", num);
}

int main()
{
    int num = leer_num();
    mostrar( buscar_perfecto(num) );
}
```

BAJO ACOPLAMIENTO
Y
ALTA COHESIÓN

Asertos, Precondiciones y Postcondiciones

Asertos

- Se debe incluir la biblioteca `<assert.h>`.
- 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 <assert.h>
int main()
{
    ...
    assert(divisor != 0);
    int cociente = dividendo / divisor;
    int resto = dividendo % divisor;
    assert(dividendo == (divisor * cociente + resto));
    ...
}
```

Precondiciones y Postcondiciones

Precondiciones

- **Precondición** es una expresión lógica 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(...)`.
  - Ayudan a **detectar errores** de programación si el subprograma que invoca **no cumple las precondiciones** especificadas por el subprograma invocado.

Postcondiciones

- **Postcondición** es una expresión lógica 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)

Ejemplo con error de programación (la precondición no está asegurada)

```c
#include <stdio.h>
#include <assert.h>

void dividir(int dividendo, int divisor, int* cociente, int* resto)
{
    assert(divisor != 0);                                    // PRECOND
    *cociente = dividendo / divisor;
    *resto = dividendo % divisor;
    assert(dividendo == (divisor * *cociente + *resto)); // POSTCOND
}
int main()
{
    int dividendo, divisor, cociente, resto;
    printf("Introduce el dividendo y divisor: ");
    scanf(" %d %d", &dividendo, &divisor);
    // Error, antes de invocar a dividir hay que comprobar que divisor != 0
    dividir(dividendo, divisor, &cociente, &resto);
    printf("Cociente: %d Resto %d\n", cociente, resto);
}
```

Precondiciones y Postcondiciones. Ejemplo (v2)

Ejemplo con error corregido

```c
#include <stdio.h>
#include <assert.h>

void dividir(int dividendo, int divisor, int* cociente, int* resto)
{
    assert(divisor != 0);                                    // PRECOND
    *cociente = dividendo / divisor;
    *resto = dividendo % divisor;
    assert(dividendo == (divisor * *cociente + *resto)); // POSTCOND
}
int main()
{
    int dividendo, divisor, cociente, resto;
    printf("Introduce el dividendo y divisor: ");
    scanf(" %d %d", &dividendo, &divisor);
    if (divisor == 0) {
        printf("Error: división por cero\n");
    } else {
        dividir(dividendo, divisor, &cociente, &resto);
        printf("Cociente: %d Resto: %d\n", cociente, resto);
    }
}
```

Definición de Subprogramas "En-línea"

- 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**.
  - Es necesario especificar la opción -**O2** para poder compilar correctamente los subprogramas `inline`.

```
inline double media_3(int x, int y, int z)
{
    return (x + y + z) / 3.0;
}
inline double media_2(int x, int y)
{
    return (x + y) / 2.0;
}
```

```
double k = 5 * media_3(a, b, c) ;
double h = 5 * media_2(a, b) ;
```

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

Variables Locales Automáticas

- Las **variables locales automáticas** son:
  - Las variables **no estáticas** definidas dentro de un subprograma (incl. `main`).
  - Los parámetros de un subprograma.
- Una variable local automática se **crea** cuando el flujo de ejecución ejecuta la sentencia donde está definida.
- Una variable local sólo puede ser **accedida dentro** del subprograma (o bloque) donde está definida, a partir del punto donde ha sido definida (ámbito).
- Una variable local automática se **destruye** cuando el flujo de ejecución termina la ejecución del subprograma (o bloque) donde ha sido definida (ámbito).
- Cuando una variable local automática se define **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) { // Atención
        int z = *x; ◀◀◀
        *x = *y;
        *y = z;
    }
}
```

```
int main()
{
    int a, b; ◀◀◀
    scanf(" %d %d", &a, &b);
    int c = menor(a, b); ◀◀◀
    ordenar(&a, &b);
    bool ok = (c == a); ◀◀◀
}
```

Variables Estáticas

- Las **variables estáticas** se definen utilizando la palabra reservada `static`, y pueden ser de dos tipos:
  - **Variables estáticas locales**: definidas **dentro** de los subprogramas (incl. `main`), utilizando la palabra reservada `static`.
  - **Variables estáticas globales**: definidas **fuera** de los subprogramas (incl. `main`), utilizando la palabra reservada `static`.
- Todas las variables estáticas (locales y globales) se **crean** al comienzo del programa, y su tiempo de vida abarca toda la duración de la ejecución del programa.
  - Cuando termina la ejecución del programa, entonces se **destruyen** todas las variables estáticas.
- Si la variable estática es local, entonces sólo puede ser **accedida dentro** del subprograma (o bloque) donde está definida, a partir del punto donde ha sido definida (ámbito).
- Si la variable estática es global, entonces puede ser **accedida** desde cualquier código o subprograma que aparezca posterior a su definición.
- **Se recomienda que NO se DEFINAN ni UTILICEN VARIABLES ESTÁTICAS.**

```c
int incrementar()
{
    static int cuenta = 0; ◀◀◀
    ++cuenta;      // Efecto lateral
    return cuenta;
}
```

```c
int main()
{
    int a = incrementar();
    incrementar();
    printf("%d\n", incrementar());
}
```

Variables Globales y Efectos Laterales

- Las **variables globales**, tanto estáticas como no estáticas, son aquellas que se **definen fuera** de los subprogramas (incluida `main`).
- Las variables globales, tanto estáticas como no estáticas, pueden ser accedidas desde cualquier código o subprograma que aparezca posterior a su definición.
- En nuestro contexto, se denonima **efecto lateral** tanto al acceso como a la modificación de variables globales, tanto estáticas como no estáticas.
- Debido a los problemas asociados a las variables globales y efectos laterales:
- **Se recomienda que NO se DEFINAN ni UTILICEN VARIABLES GLOBALES**.
- **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

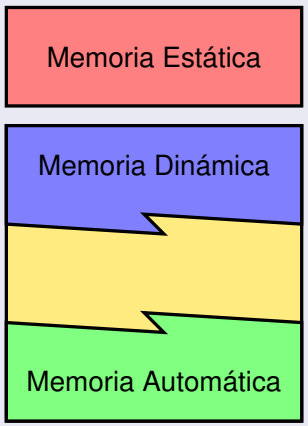
- **Se recomienda que NO se DEFINAN ni UTILICEN VARIABLES GLOBALES.**

```c
#include <stdio.h>

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();
    printf("%d\n", vble_global);    // EFECTO LATERAL (PROHIBIDO)
}
```

Áreas de Memoria

Áreas de Memoria durante la ejecución de un programa

- **Memoria estática** (*global*): almacena constantes y datos estáticos y globales, con un tiempo de vida que coincide con el tiempo de ejecución del programa.
- **Memoria automática** (*stack-pila de ejecución*): almacena los parámetros y variables locales automáticas que se crean y destruyen durante la invocación a subprogramas. Gestionada automáticamente por el compilador y el flujo de ejecución del programa.
- **Memoria dinámica** (*heap-montículo*): almacena datos cuyo tiempo de vida está gestionado dinámicamente por el programador, y cuyo acceso se realiza a través de punteros.

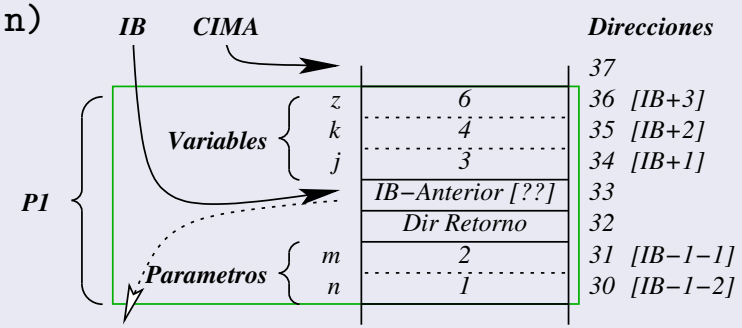


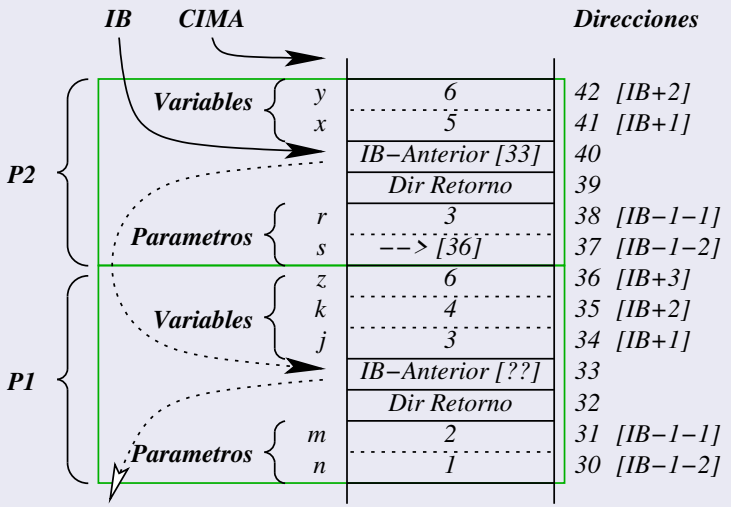
Memoria Automática: Ejemplo de la Pila de Ejecución

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.

```
void P2(int r, int* s)
{
    int x, y;
    x = 5;
    y = 6;
}
void P1(int m, int n)
{
    int j, k, z;
    j = 3;
    k = 4;
    z = 6;
    P2(j, &z);
}
```





Algunas Funciones de la Biblioteca <math.h>

Funciones de <math.h>	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, $\|x\|$
`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, $arctan(y/x)$

Es necesario especificar en las opciones de compilación que enlace con la biblioteca matemática (`-lm`).

Algunas Funciones de la Biblioteca <math.h>. Ejemplo

```c
#include <stdio.h>
#include <math.h>

void leer(double* x, double* y)
{
    printf("Introduce los valores de  los dos catetos: ");
    scanf(" %lg %lg", x, y);
}
void mostrar(double x)
{
    printf("Hipotenusa: %lg\n", x);
}
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);
}
```

Es necesario especificar en las opciones de compilación que enlace con la biblioteca matemática (`-lm`).

Introducción a la 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.

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.

Introducción a la 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 factorial_iterativo(int n)
{
    long fn = 1;
    for (int i = 2; i <= n; ++i) {
        fn = fn * i;
    }
    return fn;
}
```

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

```
  factorial_rec(4) ──────────┐
24<──────── 4 * factorial_rec(3) ──────────┐
              6<──────── 3 * factorial_rec(2) ──────────┐
                           2<──────── 2 * factorial_rec(1) ──────────┐
                                        1<──────── 1 * factorial_rec(0);
                                                     1<──────────┘
```

Introducción a la 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**.

Introducción a la Recursividad. Ejemplo MCD

```c
#include <stdio.h>
void leer(int* P, int* Q)
{
    printf("Introduzca dos numeros positivos: ");
    scanf(" %d %d", &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)) {
        printf("Error\n");
    } else {
        printf("El MCD es: %d\n", mcd(P, Q));
    }
}
```