UNIVERSIDAD DE MALAGA
DPTO. DE LENGUAJES Y C. DE LA COMPUTACION
E.T.S. DE INGENIERIA INFORMATICA

FUNDAMENTOS DE LA PROGRAMACIÓN

TEMA III

ABSTRACCIÓN PROCEDIMENTAL

III.1.- DISEÑO DESCENDENTE.

En la mayoría de los problemas reales, el algoritmo que los soluciona es demasiado largo y complejo para su implementación en bloque mediante un único texto (programa) con sentencias una tras otra. La razón es que no se comprende bien qué hace el programa debido a que se intenta abarcar toda la solución a la vez. Asimismo el programa se vuelve monolítico y difícil de modificar, ya que modificar cualquier instrucción del programa afecta a todo el código del mismo. Podemos resumir las desventajas que presentan este tipo de programas en:

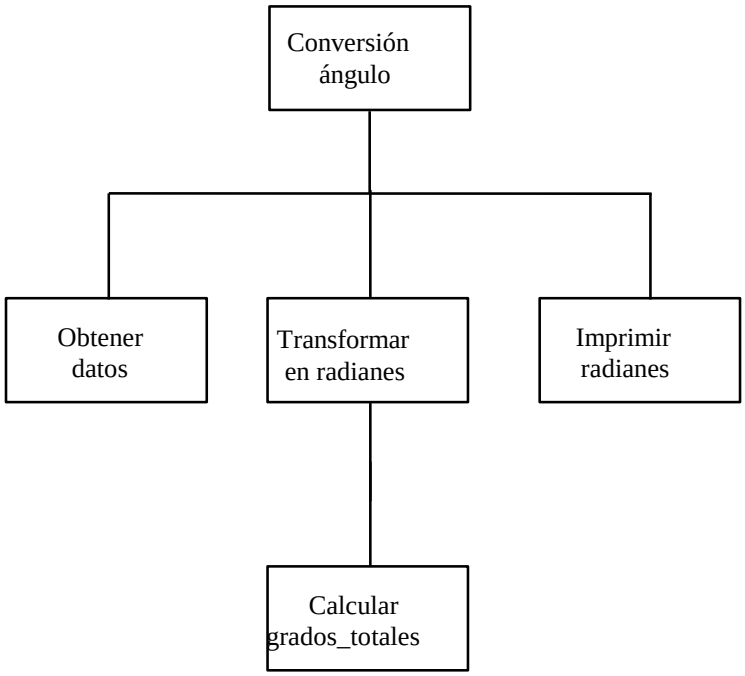
- Rigidez e inflexibilidad de los programas
- Pérdida excesiva de tiempo en corrección de errores
- Imposibilidad de reutilizar el programa o fragmentos suyos en proyectos futuros

Se hace pues necesaria la utilización de alguna metodología de diseño que evite estos inconvenientes y nos facilite la tarea de la programación. La metodología de **Diseño Descendente** (también llamada "de refinamientos sucesivos", "Top-Down" o "divide y vencerás") se ha mostrado como la más adecuada.

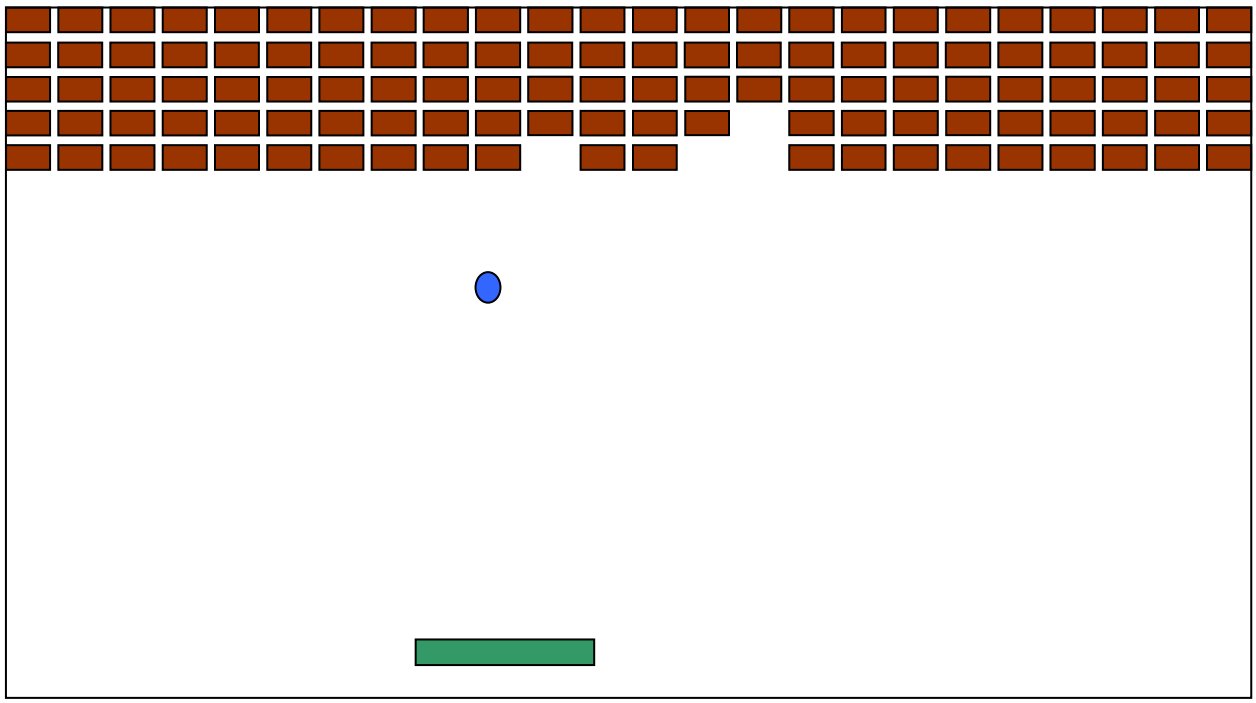
Según esta metodología, un **problema** se dividirá en dos o más subproblemas independientes, los cuales a su vez se dividirán en otros subproblemas hasta llegar a un nivel de **descomposición** que permita la solución sencilla de los diferentes **subproblemas**. La solución al problema inicial vendrá dada por la composición de cada una de las soluciones a los subproblemas en que se ha dividido.

Veamos dos ejemplos:

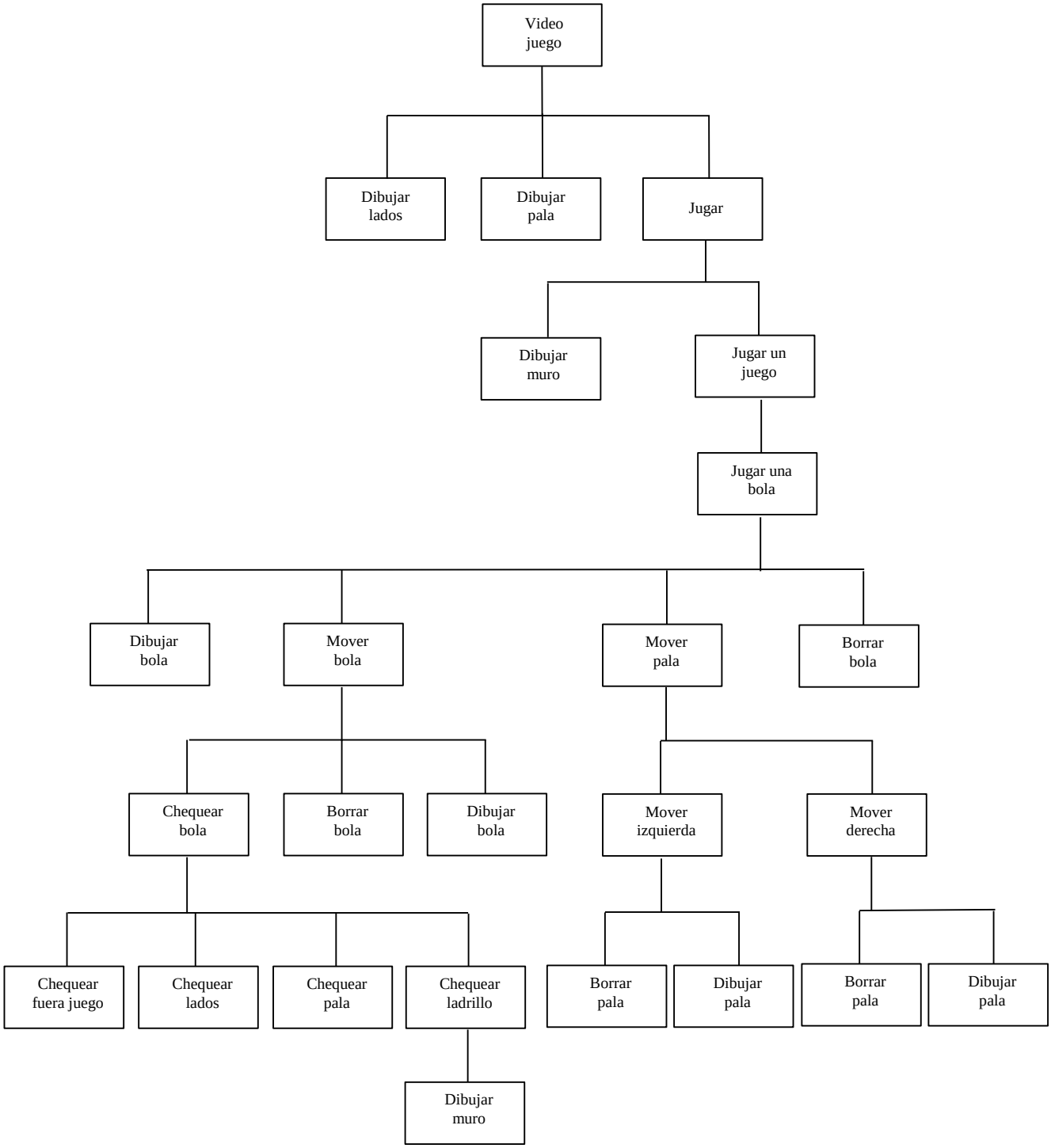
Ejemplo 1: Conversión de un ángulo expresado en grados, minutos y segundos al equivalente expresado en radianes. Se leen de teclado tres números naturales que representan los grados, minutos y segundos de un ángulo, se calcula el valor equivalente en radianes y se muestra el resultado por pantalla



Ejemplo 2: Juego del Arkanoid. Un juego clásico en el que el jugador dispone de un número determinado de vidas (bolas) para ir acumulando puntos que consigue conforme destruye los ladrillos de un muro situado en la parte superior de la pantalla. El jugador, con las teclas establecidas, controla el movimiento horizontal (izquierda y derecha) de una pala (que se encuentra en la parte inferior de la pantalla) con objeto de golpear la bola y que ésta de nuevo vuelva hacia arriba para golpear el muro. La bola puede seguir un movimiento vertival (arriba y abajo) y también puede desplazarse de izquierda a derecha o de derecha a izquierda en su trayectoria hacia arriba o hacia abajo. Si golpea uno de los muros laterales, cambia su sentido del movimiento (izquierda o derecha) aunque seguiría subiendo o banjando según la trayectoria que llevara. Si la bola golpea un ladrillo inicia movimiento de bajada. Si la pala es incapaz de golpear una bola, ésta se "pierde" por debajo de la pantalla, perdiendo el jugador una vida. Si el muro es destruido por completo, quedándole todavía vidas al jugador, se recontruye de nuevo.



Vidas: 4 Puntuación: 50


Video juego
Dibujar lados
Dibujar pala
Jugar
Dibujar muro
Jugar un juego
Jugar una bola
Dibujar bola
Mover bola
Mover pala
Borrar bola
Chequear bola
Borrar bola
Dibujar bola
Mover izquierda
Mover derecha
Chequear fuera juego
Chequear lados
Chequear pala
Chequear ladrillo
Borrar pala
Dibujar pala
Borrar pala
Dibujar pala
Dibujar muro

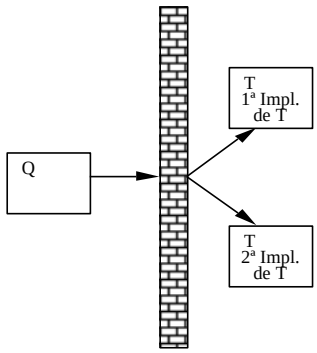
Este enfoque proporciona indudables ventajas:

- Simplificación del diseño de los programas
- Mejor comprensión y legibilidad de los programas
- Facilita la depuración de errores
- Programación aislada
- Posibilidad de reutilización de las soluciones a los subproblemas

Para aplicar esta metodología en el desarrollo de un programa usando un lenguaje programación de alto nivel, es necesario disponer de herramientas que permitan estructurar el **programa** (solución al problema inicial) como una **composición de subprogramas** o **módulos** (solución a los diferentes subproblemas en los que se ha dividido el problema inicial).

La utilización de subprogramas permite lo que se conoce como **Abstracción Procedimental**. Dicha abstracción se basa en el conocimiento del subproblema que resuelve un determinado subprograma, ignorando la forma de resolverlo. El programador, a la hora de diseñar un un subprograma, **no tiene que saber cómo** otro subprograma que necesita resuelve su subproblema, sino que **tan sólo tiene que saber qué** es lo que resuelve dicho subprograma.

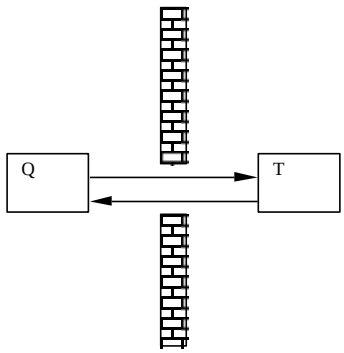
La abstracción procedimental **aísla** (**encapsula**) los diferentes subprogramas que componen un programa. De esta forma, si el método que utiliza el subprograma T para solucionar su subproblema cambia, el aislamiento evita que dicho cambio influya en cualquier otro subprograma Q que necesita a T.



Ejemplo. T realiza una ordenación de una colección de 100 números enteros y Q necesita dicha ordenación para resolver su problema. Si T cambia la forma de resolver la ordenación (p. ej. para que sea más eficiente), esto no afecta al diseño de Q.

Sin embargo, el **aislamiento** de los subprogramas **no puede ser total**. Aunque a un subprograma Q le baste saber qué resuelve un subprograma T y no cómo, hay ciertas condiciones (**precondiciones y postcondiciones**) que se deben cumplir para que T realice su tarea satisfactoriamente. Además existirá normalmente un **intercambio de información** entre ambos subprogramas.

Ejemplo. Q debe pasarle a T la colección a ordenar y además debe estar compuesta por 100 (y no 150, p. ej.) números enteros (y no reales, p. ej.). Por otro lado, T le devolverá a Q la colección inicial ordenada.



III.2.- PROCEDIMIENTOS Y FUNCIONES. PARÁMETROS

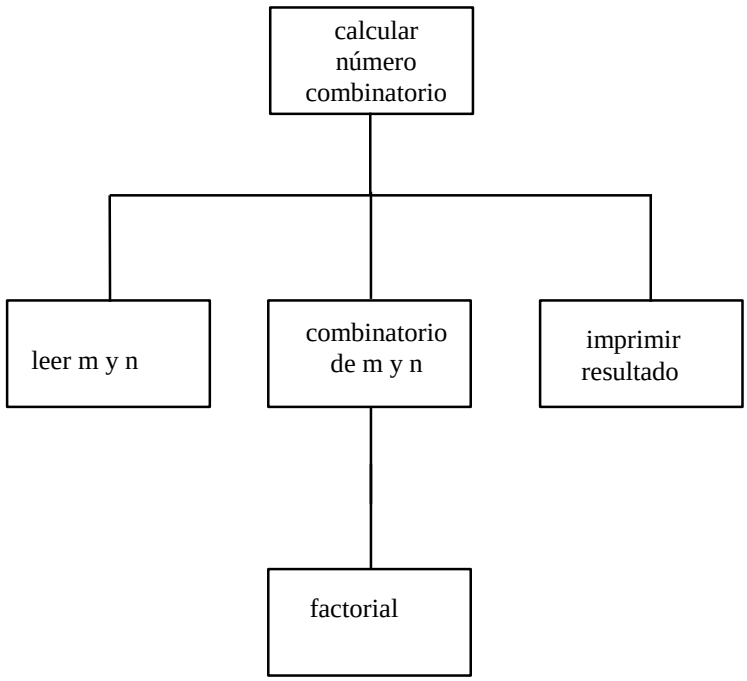
Los procedimientos y las funciones son las herramientas que ofrecen los lenguajes de programación de alto nivel para aplicar el diseño descendente y conseguir la abstracción procedimental introducidos en la sección anterior. Los procedimientos y las funciones son los subprogramas que se utilizarán para codificar las soluciones a los diferentes subproblemas en los que se ha dividido el problema inicial. La solución al problema inicial se codificará con la función `main`, que ha sido la única que hasta ahora se ha utilizado en la asignatura.

Vamos a introducir el uso de estas herramientas mediante un ejemplo. En él aparecerán conceptos (declaración, llamada, parámetros, …) que se tratarán más adelante de una forma más precisa y detallada.

III.2.1. Ejemplo.

Cálculo de un número combinatorio

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



Una vez realizada la descomposición del problema inicial en subproblemas mediante el diseño descendente, pasamos a codificar la función `main` que resuelve dicho problema:

```
int main() {
     unsigned m,n,comb;

     leerDatos(m,n);
     comb = combinatorio(m,n);
     imprimirResultado(m,n,comb);
}
```

Como puede observarse, para resolver el problema estamos haciendo uso de la abstracción procedimental: la función `main` resuelve el problema planteado sin necesidad de saber cómo se leen los datos, cómo se calcula el número combinatorio y cómo se imprime el resultado.

Posteriormente se codifican los procedimientos y funciones (subprogramas) utilizados por la función `main`:

```
void leerDatos(unsigned& m, unsigned& n) {
     do {
          cout << "Introduzca m y n (m >= n): ";
          cin >> m >> n;
     } while (m < n);
}


unsigned combinatorio(unsigned m, unsigned n) {
     return factorial(m) / (factorial(n) * factorial(m-n));
}

void imprimirResultado(unsigned m, unsigned n, unsigned res) {
     cout << "El numero combinatorio de "
          << m << " sobre " << n << " es: " << res << endl;
}
```

De nuevo, para la solución de `combinatorio` se utiliza abstracción procedimental: calculamos el número combinatorio sin necesidad de saber cómo se calcula el factorial de un número.

Finalmente, se codifica `factorial`, que ha sido utilizada por `combinatorio`:

```
unsigned factorial(unsigned x) {
     unsigned fact = 1;

     for (unsigned i = 2; i <= x; i++) {
          fact = fact * i;
     }
     return fact;
}
```

No siempre hay que asociar un procedimiento o función a cada una de los subproblemas en los que se ha dividido el problema inicial, sino que se puede codificar la solución al subproblema directamente en el lugar que se necesita. En nuestro ejemplo podríamos prescindir del procedimiento `imprimirResultado` y realizar dicha impresión directamente en la función `main`:

```
int main() {
     unsigned m,n,comb;

     leerDatos(m,n);
     comb = combinatorio(m,n);
     cout << "El numero combinatorio de "
          << m << " sobre " << n << " es: " << res << endl;
}
```

Con este ejemplo se pueden apreciar todas y cada una de las ventajas anteriormente mencionadas que se consiguen con el diseño descendente y la abstracción procedimental:

- Simplificación del diseño de los programas. Es más sencillo abordar problemas más pequeños que un problema global como un todo.
- Mejor comprensión y legibilidad de los programas. Cada subprograma es más corto y fácil de leer y comprender.
- Facilita la depuración de errores. Nos podemos centrar en un subprograma concreto sin necesidad de mirar el resto del código.
- Programación aislada. Distintas personas pueden abordar distintos procedimientos y funciones. Además, el cambio del método para

resolver un subproblema no afecta al que lo usa (por ejemplo `combinatorio` se puede resolver de otra forma más eficiente simplificando la expresión, pero la función `main` no cambia).

- Posibilidad de reutilización de las soluciones a los subproblemas. Por ejemplo `factorial` se puede reutilizar para otros problemas diferentes.

III.2.2. Definición y Declaración de Procedimientos y Funciones. Parámetros Formales.

El esquema general de la *definición* de un procedimiento es el siguiente:

```
void nombreProcedimiento(parámetros) {

     declaración de variables

     secuencia de sentencias
}
```

La definición de un procedimiento consta de una *cabecera* y un *cuerpo*. En la cabecera aparece la palabra reservada `void` seguida del nombre del procedimiento y, entre paréntesis, los posibles parámetros. Con ellos intercambia información (en ambos sentidos) con el procedimiento o función que lo llama (ya se verá más adelante cómo). El cuerpo contiene la posible declaración de variables utilizadas en él y la secuencia de sentencias a ejecutar cuando el procedimiento sea llamado. Ejemplos: `leerDatos`, `imprimirResultado`, de la sección III.2.1.

El esquema general de la *definición* de una función es el siguiente:

```
TipoResultado nombreFuncion(parámetros) {

     declaración de variables

     secuencia de sentencias
     (última sentencia "return" para devolver el resultado)
}
```

La definición de una función consta también de una *cabecera* y un *cuerpo*. En la cabecera aparece el tipo del resultado devuelto por la

función seguido del nombre de la función y, entre paréntesis, los posibles parámetros. Una función, al igual que un procedimiento, puede o no tener parámetros, aunque lo más normal es que sí los tenga. Éstos, aunque también pueden utilizarse para intercambiar información en ambos sentidos, se deben utilizar sólo para recibir información del procedimiento o función que la llama (ya se verá más adelante cómo). El cuerpo de la función contiene la posible declaración de variables utilizadas en él y la secuencia de sentencias a ejecutar cuando la función sea llamada. La última sentencia debe ser una sentencia `return` que sirve para devolver un valor del tipo especificado en la cabecera de la función delante del nombre. Es decir, que una función, tras su ejecución, devuelve de forma explícita un valor al procedimiento o función que la llamó. Ejemplos: `combinatorio`, `factorial`, de la sección III.2.1.

Parámetros Formales.

Los parámetros que aparecen en la definición de un procedimiento o una función se denominan Parámetros Formales y son declaraciones de variables (`TipoVariable nombreVariable`) separadas por comas. El tipo de la variable puede llevar o no un símbolo `&`, lo cual influirá en el sentido de la comunicación de los datos intercambiados. Ejemplos (de la sección III.2.1):

```
unsigned combinatorio(unsigned m, unsigned n)
```

```
void leerDatos(unsigned& m, unsigned& n)
```

En la sección III.2.4 se analizará con más detalle este aspecto.

Prototipos.

Hemos visto en la sección anterior (III.2.1) que primero codificamos la función `main` y después los procedimientos y funciones que ésta necesita. Este es el orden adecuado de hacerlo ya que encaja con la metodología del diseño descendente y la abstracción procedimental. Pero, al igual que ocurre con el resto de entidades en C++ (variables, constantes, …), los procedimientos y funciones deberían aparecer en el programa antes de ser

utilizados. Así, en nuestro ejemplo, todos los procedimentos y funciones aparecerían delante de la función `main` (siempre la última). Además, como `combinatorio` utiliza a `factorial`, ésta debería aparecer antes de `combinatorio`.

Por el contrario, también se puede optar por utilizar los denominados "prototipos". Un prototipo es una *declaración* de un procedimiento o una función, la cual consiste en especificar sólo la cabecera del procedimiento o función, poniendo al final un punto y coma.

El esquema general de un prototipo (*declaración*) de procedimiento es:

```
void nombreProcedimiento(parámetros);
```

El esquema general de un prototipo (*declaración*) de función es:

```
TipoResultado nombreFuncion(parámetros);
```

Ejemplos (de la sección III.2.1):

```
unsigned combinatorio(unsigned m, unsigned n);

void leerDatos(unsigned& m, unsigned& n);
```

El uso de prototipos nos permite *declarar* los diferentes procedimientos y funciones que se van a utilizar en el programa independientemente del orden en el que se vayan después a *definir*. De esta forma podemos empezar a definir la función `main` y por debajo los procedimientos y funciones que se utilicen en el orden que convenga conforme al esquema resultante del diseño descendente aplicado. Para que ello sea posible, no obstante, delante de la función main hay que especificar los prototipos de dichos procedimientos y funciones.

El único inconveniente del uso de prototipos es que la cabecera de los procedimientos o funciones aparecerán dos veces, una en el propio prototipo y otra en la definición del procedimiento o función.

III.2.3. Llamada a Procedimientos y Funciones. Parámetros Reales.

El esquema general de la **llamada a un procedimiento** es el siguiente:

```
nombreProcedimiento(parámetros);
```

Esta llamada **constituye por sí misma una sentencia**. Por ejemplo:

```
leerDatos(m,n);
```

El esquema general de la **llamada a una función** es el siguiente:

```
nombreFuncion(parámetros)
```

Esta llamada **NO constituye por sí misma una sentencia**. La llamada debe hacerse en un lugar adecuado donde el valor devuelto explícitamente por la función (mediante la sentencia `return`) sea utilizado por el procedimiento o función que la llama, en definitiva, en un lugar donde pudiera aparecer cualquier expresión del mismo tipo que el del valor devuelto por la función. Por ejemplo a la derecha de una sentencia de asignación, como en el ejemplo de la sección III.2.1:

```
comb = combinatorio(m,n);
```

Otros ejemplos:

```
res = 3 * combinatorio(m,n) + 1;
cout << combinatorio(m,n) << endl;
if (esPrimo(n)) { // esPrimo es una función que devuelve un bool
```

Parámetros Reales.

Los parámetros que aparecen en la llamada a un procedimiento o una función se denominan Parámetros Reales y son constantes, nombres de variable o de constante y expresiones en general separadas por comas.

Los parámetros han de someterse a unas reglas:

1) Número de parámetros formales = Número de parámetros reales.
2) El i-ésimo parámetro formal se corresponde con el i-ésimo parámetro real
3) El tipo del i-ésimo parámetro formal debe ser igual que el tipo del i-ésimo parámetro real.
4) Los parámetros de un procedimiento o una función pueden ser de cualquier tipo, al igual que cualquier variable.
5) Los nombres de un parámetro formal y su correspondiente real pueden o no ser iguales.
6) Un parámetro formal pasado por valor (sin `&`) permite variables, constantes y expresiones como parámetro real. En cambio, un parámetro formal pasado por referencia (con `&`) requiere una variable como parámetro real. (En la sección III.2.4 se verá el paso por valor y por referencia en detalle)

Ejercicio sobre paso de parámetros:

Supongamos que en la función `main` se declaran las siguientes variables:

```
double x,y;
int m;
char c;
```

y tenemos un procedimiento con la siguiente cabecera:

```
void prueba(int a, int b, double& c, double& d, char e)
```

averigua cuáles de las siguientes llamadas a `prueba` desde `main` son incorrectas y cuál es la razón:

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

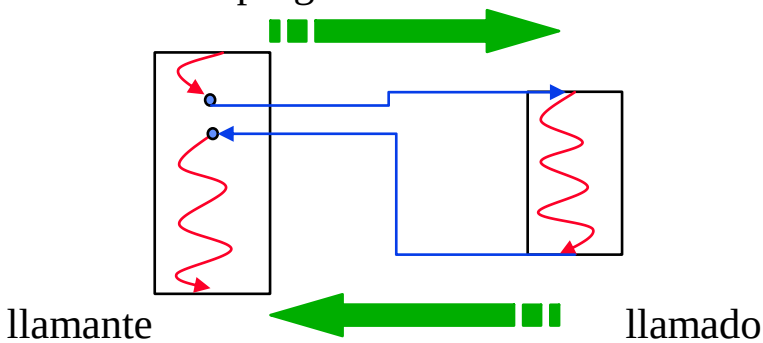
Flujo de Control.

Cuando se produce una llamada a un subprograma (procedimiento o una función):

- Se establecen las vías de comunicación entre el subprograma llamante y el llamado (a través de los parámetros)
- Se crean las variables declaradas en el llamado
- El flujo de control pasa a la primera instrucción del llamado

Cuando acaba el subprograma llamado:

- Se destruyen las variables creadas en el llamado
- El flujo de control continúa por la instrucción que sigue a la de llamada en el subprograma llamante



III.2.4. Paso de parámetros por valor y por referencia.

Ya hemos visto que los parámetros constituyen las vías de comunicación para intercambiar información entre subprogramas (procedimientos y funciones). Si la información fluye desde el llamante al llamado se denomina información de entrada. Si fluye desde el llamado al llamante se denomina información de salida. Si lo hace en ambos sentidos se denomina información de entrada/salida.

En la práctica, los lenguajes de programación normalmente implementan este flujo de información mediante los mecanismos conocidos como paso por valor y paso por referencia.

- El **paso por valor** consiste en almacenar en el parámetro formal una copia del valor del parámetro real. Como consecuencia, cualquier modificación sobre el parámetro formal no afecta al parámetro real. Los dos parámetros estan totalmente aislados. Sintácticamente el parámetro formal se declara indicando únicamente su tipo. Ejemplo:

```
void imprimirResultado(unsigned m, unsigned n, unsigned res) {
       cout << "El numero combinatorio de "
            << m << " sobre " << n << " es: " << res << endl;
}
```

- El **paso por referencia** consiste en almacenar en el parámetro formal una referencia a la variable situada como parámetro real. Como consecuencia, la variable especificada como parámetro formal pasa a ser la misma que la variable especificada en el parámetro real, por lo que cualquier modificación del parámetro formal implica en la práctica una modificación del parámetro real. Sintácticamente el parámetro formal se declara estableciendo su tipo seguido por el símbolo `&`. Ejemplo:

```
void leerDatos(unsigned& m, unsigned& n) {
  do {
       cout << "Introduzca m y n (m >= n): ";
       cin >> m >> n;
  } while (m < n);
}
```

Características de ambas formas de comunicación.

- Paso por valor:
  - Sólo permite comunicación de entrada, como por ejemplo en `imprimirResultado`.
  - Aisla.
  - Permite variables, constantes y expresiones como parámetro real.
  - Duplica memoria y realiza copia.
  - Utiliza más memoria en el caso de tipos estructurados.

- Paso por referencia:
  - Permite comunicación de salida si el parámetro formal se utiliza para comunicar un valor al llamante (p. ej. `leerDatos`) y de entrada/salida si se modifica el valor contenido en el parámetro formal. Por ej. la llamada a `intercambiar(m,n)`, donde `m` contiene 8 y `n` contiene 5, producirá una modificación en dichas variables de forma que `m` contendrá 5 y `n` contendrá 8).

```
void intercambiar(int& x, int& y) {
     int temp;

     temp = x;
     x = y;
     y = temp;
}
```

  - También permite comunicación de entrada si no hay modificación del parámetro formal. Normalmente se utiliza para los tipos de datos estructurados (se verá en el próximo tema).
  - No aisla.
  - Sólo permite una variable como parámetro real.
  - No duplica memoria y no realiza copia.
  - Utiliza menos memoria en el caso de tipos estructurados.

III.2.5. Interfaz.

- Término genérico aplicable a diferentes ramas: define la interaccion entre dos entidades.
- En el contexto de subprogramas se denomina interfaz a la forma en que se comunican y cooperan dos subprogramas.
- Los errores en el uso de subprogramas se presentan fundamentalmente debido a una interfaz incorrecta entre el llamante y el llamado.
- Para diseñar el interfaz debemos considerar:

- ¿Qué información del llamante necesita conocer el llamado para poder trabajar correctamente?
- ¿Qué información producirá el llamado que después sea necesitada en el llamante?
- ¿Bajo qué condiciones se realiza este intercambio de información?

- Atendiendo a la solución dada a las preguntas anteriores se definen parámetros capaces de comunicarla.

III.2.6. Criterios de modularización.

No existen algoritmos formales para determinar cómo descomponer un problema en subproblemas, es decir para determinar **cúantos subprogramas o módulos debemos establecer** para dar la solución a nuestro problema. Es una **labor subjetiva.** De cualquier forma se siguen algunos criterios que pueden guiarnos al modularizar.

1.- Acoplamiento.

Una de las ventajas frente a la programación convencional (sin ninguna técnica rigurosa de diseño) es que con el diseño descendente obtenemos software fácilmente modificable. La idea es que una modificación del sistema sólo requiera cambios en pocos módulos, con objeto de que podamos restringir nuestra atención a dicha porción del mismo.

Esta idea tendrá éxito si se puede suponer que cambios en determinados módulos no afectarán inadvertidamente a otros. Por tanto, un objetivo del diseño original deberá ser **maximizar la independencia entre módulos**.

Es obvio que en contra de dicho objetivo está el hecho antes mencionado de que debe haber alguna conexión entre módulos para formar un sistema coherente. Dicha conexión se conoce como acoplamiento.

Maximizar la independencia será **minimizar el acoplamiento**.

2.- Cohesión

Tan importante como minimizar el acoplamiento entre diferentes módulos es **maximizar las ligaduras internas** dentro de cada módulo individual.

Se usa el término cohesión para hacer referencia al grado de relación entre las diferentes partes internas a un módulo.

Si la cohesión es muy débil, la diversidad entre las distintas tareas realizadas dentro de un módulo es tal que posteriores modificaciones podrían resultar complicadas. Además dificulta la reutilización.

El diseñador de sotfware debe buscar un bajo acoplamiento entre los módulos y una alta cohesión dentro de cada uno.

III.2.7. Variables locales y globales. Efectos laterales.

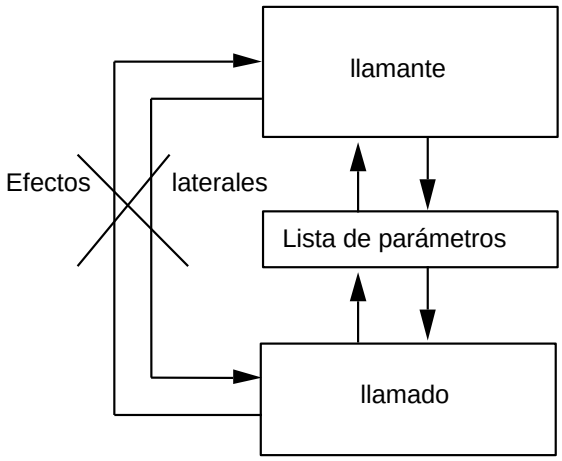
Las variables declaradas dentro de un procedimiento o una función (incluida la función `main`) se denominan **variables locales** a ese procedimiento o función. Los parámetros formales también son variables locales. Una variable local sólo puede ser accedida dentro del procedimiento y función que la declara y a partir del punto donde se declara. Una variable local se crea cuando el procedimiento o función es llamado y se destruye cuando la ejecución del mismo termina.

C++ permite también declarar variables fuera de cualquier procedimiento o función. Estas variables se denominan **variables globales** y son accesibles por cualquier procedimiento o función que aparezca a partir de su declaración y hasta el final del fichero que contiene el programa. Debido a los problemas que conlleva su utilización, a lo largo de esta asignatura **NO utilizaremos variables globales**.

Efectos laterales.

Es cualquier intercambio de información producida entre dos subprogramas o módulos realizado a través de variables globales.

La posibilidad de usar variables globales puede hacer pensar que no es necesario el uso de parámetros, ya que el subprograma llamado puede acceder globalmente a las variables del subprograma que lo llama. No obstante este uso en programación **está totalmente desaconsejado** (aumenta el acoplamiento, reduce la posibilidad de reutilización, aumenta la posibilidad de cometer errores que son difíciles de encontrar).



III.2.8. Precondiciones y Postcondiciones. Tratamiento de situaciones excepcionales.

Tal como se comentó en la sección III.1, el aislamiento de los subprogramas no puede ser total. Además del intercambio de información (a través de los parámetros tal como hemos visto en secciones anteriores), hay ciertas condiciones que se deben cumplir para que un subprograma realice su tarea satisfactoriamente. Estas condiciones son de dos tipos:

- *Precondiciones*: condiciones que deben cumplirse antes de que el subprograma se ejecute, con objeto de garantizar que se puede realizar la tarea.
- *Postcondiciones*: condiciones que el subprograma garantiza tras finalizar su ejecución, suponiendo que las precondiciones se cumplieron cuando el subprograma fue llamado.

Cuando se codifica un subprograma es buena práctica anteponer unos comentarios especificando claramente las precondiciones y postcondiciones del mismo. Por ejemplo, dada la función `combinatorio` que antes hemos diseñado, podemos establecer su precondición y postcondición así:

```
// precondicion: m >= n
// postcondicion: devuelve combinatorio de m sobre n
unsigned combinatorio(unsigned m, unsigned n) {
     return factorial(m) / (factorial(n) * factorial(m-n));
}
```

De cualquier forma esto no es suficiente, pues el procedimiento o función que utilice (llame) el subprograma diseñado, puede o no tener en cuenta las precondiciones de uso del mismo. Por ejemplo, un subprograma podría llamar a `combinatorio` con dos valores de `m` y `n` tales que `m < n`, saltándose la precondición de uso de `combinatorio` que establece que se debe cumplir `m >= n`. ¿Qué ocurriría? La operación `m-n` no será correcta, pues nos daría un número negativo cuando estamos trabajando con naturales (`unsigned`). En algún momento el sistema terminará el programa con un error de ejecución inesperado.

Ya vimos en el tema anterior cómo abordar situaciones de error mediante la utilización de excepciones. Cuando se diseñan procedimientos y funciones es fundamental tener en cuenta este tipo de situaciones, ya que estamos diseñando de forma separada un subprograma que será utilizado (llamado) por otro. Así, en nuestro ejemplo, se puede atacar esta situación diseñando la función combinatorio de la siguiente forma:

```
// precondicion: m >= n
// postcondicion: devuelve combinatorio de m sobre n
unsigned combinatorio(unsigned m, unsigned n) {
     if (m < n) {
          throw "Error: (m < n) en funcion combinatorio";
     }
     return factorial(m) / (factorial(n) * factorial(m-n));
}
```

Ahora, si se incumple la precondición, la ejecución de la función terminará tras lanzar la excepción correspondiente. Por otro lado, la excepción puede ser capturada por la función `main` que puede diseñarse para capturar excepciones inesperadas y mostrar información adecuada antes de terminar el programa. Como ya indicamos en el tema anterior, en esta asignatura no vamos a abordar el tratamiento (la captura) de las excepciones. Este aspecto será objeto de un tema completo en la asignatura del segundo cuatrimestre. En este primer cuatrimestre nos centraremos únicamente en el lanzamiento de excepciones cuando se produzcan situaciones excepcionales como la del ejemplo.

III.3.- RECURSIVIDAD.

III.3.1. Concepto de Recursividad.

La recursividad es una **técnica de programación** potente que ofrece una **alternativa** a las estructuras de control iterativas (**bucles**) para la resolución de procesos repetitivos.

Los algoritmos recursivos dan **soluciones elegantes y simples** a problemas de gran complejidad. Estas soluciones son, en general, bien estructuradas y modulares. De hecho, la forma mediante la que los módulos de una solución recursiva interactúan, es precisamente lo que hace de la recursión una herramienta de programación única y potente.

Por otro lado, la recursividad **también** tiene sus **desventajas** frente a la iteración, como ya veremos en la sección III.3.3.

¿En qué consiste realmente la recursividad?

Es una técnica que nos permite que un **subprograma (procedimento o función) se invoque a sí mismo** para resolver una "**versión más pequeña**" del problema para el que se ha diseñado.

III.3.2. Ejemplos.

1. Factorial

En matemáticas es frecuente definir un **concepto** en **función del proceso** usado para generarlo. Por ejemplo, una definición matemática de n! es:

$$n! = \begin{cases} 1 & si \quad n = 0 \\ n(n-1)(n-2)...1 & si \quad n > 0 \end{cases}$$

Con esta definición, fácilmente podemos diseñar una solución **iterativa**:

```
unsigned factorialIter(unsigned n) {
     unsigned fact;

     if (n == 0) {
          fact = 1;
     } else {
          fact = 1;
          for (unsigned i = 2; i <= n; i++) {
               fact = fact * i;
          }
     }
     return fact;
}
```

Aunque, como se vio en la sección III.2.1, la sentencia `if` no es necesaria:

```
unsigned factorialIter(unsigned n) {
     unsigned fact = 1;

     for (unsigned i = 2; i <= n; i++) {
          fact = fact * i;
     }
     return fact;
}
```

Por otro lado, en matemáticas también es común definir un **concepto** en **función del propio concepto** que se está definiendo (definición recursiva). Por ejemplo, una definición matemática de n! es:

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

A partir de esta definición, es fácil obtener una solución **recursiva**:

```
unsigned factorialRec(unsigned n) {
     unsigned fact;

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

De esta función podemos sacar varias **conclusiones**:

1.- La función **se invoca a sí misma** (esto es lo que la convierte en recursiva).

2.- Cada **llamada recursiva** se hace con un parámetro de menor valor que el de la anterior llamada. Así, cada vez se está invocando **a otro problema idéntico pero de menor tamaño**.

3.- Existe un caso especial en el que se actúa de forma diferente, esto es, ya no se utiliza la recursividad. Lo importante es que la forma en la que el tamaño del problema disminuye asegura que se llegará a este caso especial o **caso base**.

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

1.- ¿Existe una o varias salidas no recursivas o **casos base** del subprograma, y éste funciona correctamente para ellas? ¿Se alcanzan?

2.- ¿Cada llamada resursiva al subprograma se refiere a un **caso más pequeño** del problema original?

3.- Suponiendo que la(s) llamada(s) recursiva(s) funciona(n) correctamente, así como el caso base, ¿funciona correctamente todo el subprograma?

Si planteamos estas tres preguntas a la función `factorialRec`, podemos comprobar cómo las tres respuestas son afirmativas, con lo que podemos deducir que el algoritmo recursivo está bien diseñado.

2. Multiplicación de conejos.

Veamos un segundo ejemplo también sencillo.

Supongamos que partimos de una pareja de conejos recién nacidos, y queremos calcular cuántas parejas de conejos forman la familia al cabo de `n` meses si:

a) Los conejos nunca mueren.

b) Un conejo puede reproducirse al comienzo del tercer mes de vida.

c) Los conejos siempre nacen en parejas macho-hembra. Al comienzo de cada mes, cada pareja macho-hembra, sexualmente madura, se reproduce en exactamente un par de conejos macho-hembra.

Para un `n` pequeño, por ejemplo 6, la solución se puede obtener fácilmente a mano:

Mes 1: 1 pareja, la inicial.

Mes 2: 1 pareja, ya que todavía no es sexualmente madura.

Mes 3: 2 parejas; la original y una pareja de hijos suyos.

Mes 4: 3 parejas; la original, una pareja de hijos suyos nacidos ahora y la pareja de hijos suyos nacidos en el mes anterior.

Mes 5: 5 parejas; la original, una pareja de hijos suyos nacidos ahora, las dos parejas nacidas en los meses 3 y 4, y una pareja de hijos de la pareja nacida en el mes 3.

Mes 6: 8 parejas; las 5 del mes anterior, una pareja de hijos de la original, una pareja de hijos de la nacida en el mes 3 y una pareja de hijos nacida en el mes 4.

Si deseamos saber el número de parejas al cabo de n meses, para un n cualquiera, podemos construir un algoritmo recursivo fácilmente a partir de la siguiente relación:

$$Parejas(n) = \begin{cases} 1 & \text{si} \quad n \leq 2 \\ Parejas(n-1) + Parejas(n-2) & \text{si} \quad \text{otro caso} \end{cases}$$

En esta relación Parejas(n-1) son las parejas vivas en el mes n-1, y Parejas(n-2) son las parejas que nacen en el mes n a partir de las que había en el mes n-2.

La serie de números Parejas(1), Parejas(2), Parejas(3),... es conocida como la **Serie de Fibonacci**, la cual modela muchos fenómenos naturales.

La función recursiva quedaría:

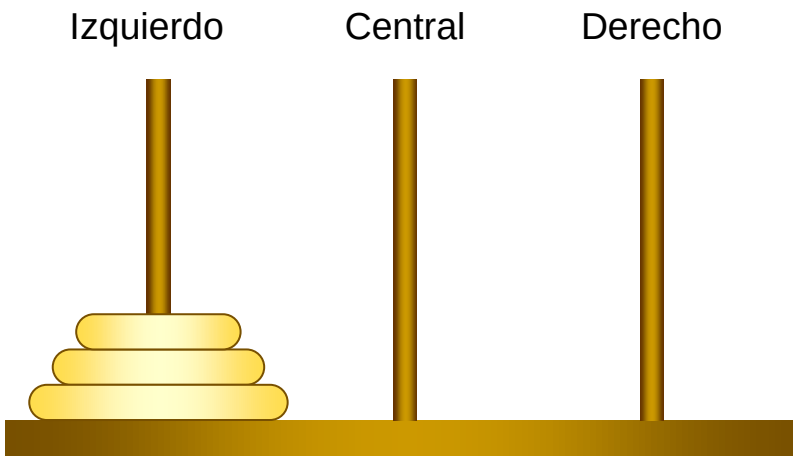
```
unsigned parejas(unsigned n) {
     unsigned res;

     if (n <= 2) {
          res = 1;
     } else {
          res = parejas(n-1) + parejas(n-2);
     }
     return res;
}
```

3. Torres de Hanoi.

Veamos un último ejemplo que es más complejo que los dos anteriores.

Las Torres de Hanoi es un juego cuya solución algorítmica se simplifica mucho si se piensa como un problema recursivo.

Se tienen 3 palos de madera, que llamaremos palo izquierdo, central y derecho. El palo izquierdo tiene ensartados un montón de discos concéntricos de tamaño decreciente, de manera que el disco mayor está abajo y el menor arriba.



El problema consiste en mover 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.

Se quiere diseñar un programa que lea por teclado un valor entero `n` y escriba la secuencia de pasos necesarios para resolver el problema de las Torres de Hanoi para `n` discos.

Solución.

Si `n=1`, el problema se resuelve inmediatamente sin más que mover el disco del palo `origen` al `destino`.

El problema de mover `n` discos (`n>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) se puede expresar en función del problema de mover `n-1` discos, descomponiendo la solución en 3 fases:

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.

Las fases 1 y 3 son idénticas al problema original, pero para un menor tamaño (un disco menos).

Planteamos un procedimiento recursivo `mueve` con cuatro parámetros:

- El número de discos a mover.
- El palo `origen` desde donde moverlos.
- El palo `destino` hacia el que moverlos.
- El palo `auxiliar`.

```
enum Palo {
     izquierdo,
     central,
     derecho
};

void leerNumeroDiscos(unsigned& discos) {
     do {
          cout << "Introduzca numero de discos (>=1): ";
          cin >> discos;
     } while (discos == 0);
}

void escribirPalo(Palo p) {
     switch (p){
          case izquierdo: cout << "izquierdo";
                          break;
          case central:  cout << "central";
                         break;
          case derecho:  cout << "derecho";
                         break;
     }
}

void mueveUno(Palo origen, Palo destino) {
     escribirPalo(origen);
     cout << " -> ";
     escribirPalo(destino);
     cout << endl;
}

void mueve(unsigned n, Palo origen, Palo auxiliar, Palo destino) {
     if (n == 1) {
          mueveUno(origen,destino);
     } else {
          mueve(n-1,origen,destino,auxiliar);
          mueveUno(origen,destino);
          mueve(n-1,auxiliar,origen,destino);
     }
}
```

```
int main() {
     unsigned discos;

     leerNumeroDiscos(discos);
     cout << "para " << discos
          << " discos se necesitan los movimientos: " << endl;
     mueve(discos,izquierdo,central,derecho);
}
```

III.3.3. Recursividad frente a Iteración.

Ya sabemos que la recursividad es una técnica potente de programación para resolver problemas que a menudo produce soluciones simples y claras, incluso para problemas muy complejos.

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.

Ya hemos visto que cuando se produce una llamada a un procedimiento o una función se deben establecer las vías de comunicación entre el subprograma llamante y el llamado (parámetros) y se deben crear las variables locales del llamado. Después, al finalizar la ejecución del llamado, se deben destruir todas esas variables locales y los parámetros utilizados. Todo esto conlleva una sobrecarga tanto en tiempo de ejecución como 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.

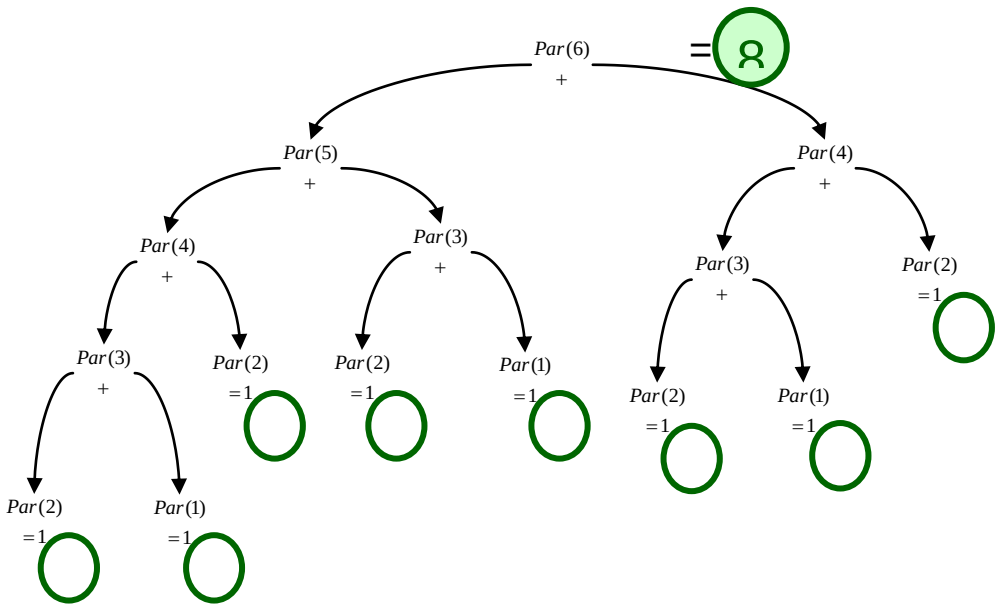
Pero en muchas circunstancias el uso de la recursividad permite a los programadores especificar soluciones naturales y sencillas que serían difíciles de resolver mediante técnicas iterativas. En estos casos la sencillez y naturalidad de la solución compensa la posible ineficiencia en la ejecución. Por ejemplo el problema de las Torres de Hanoi.

No obstante, no debemos usar la recursividad por el gusto de usarla, cuando podemos diseñar un algoritmo iterativo igual de sencillo. Por ejemplo, deberíamos usar la función iterativa `factorialIter` en lugar de su equivalente recursiva `factorialRec`. Aunque bien es cierto que muchos compiladores pueden realizar en "algunos casos concretos" (la denominada *recursividad de cola*, como por ej. en nuestro `factorialRec`) optimizaciones al traducir un subprograma recursivo de forma que la ejecución realmente es tan eficiente como la del equivalente iterativo.

2. La ineficiencia inherente de algunos algoritmos recursivos.

Este aspecto es muy diferente del anterior. Esta ineficiencia es debida al método empleado para resolver el problema concreto que se esté abordando.

Por ejemplo, en nuestra función `parejas` apreciamos un gran inconveniente: los mismos valores son calculados varias veces. Por ejemplo, para calcular `parejas(6)`, tenemos que calcular `parejas(4)` dos veces, `parejas(3)` tres veces, … Mientras más grande sea `n`, mayor ineficiencia.



Podemos diseñar un algoritmo más eficiente (`parejasEfic`) que evite este inconveniente, aunque requiere un poco más de esfuerzo y no es tan inmediato. En realidad, la función `parejasEfic` no es recursiva, sino que llama a otra función `parejasEficRec` que sí lo es. Los parámetros de `parejasEficRec` son: el mes `n` para el que queremos calcular cuántas parejas de conejos hay, un contador `cont` que marca el mes que se va intentando en cada paso (hasta llegar a `n`) y los valores de las parejas de conejos (`ant2` y `ant1`) que hay en los dos meses anteriores a `cont`.

```
unsigned parejasEficRec(unsigned ant2, unsigned ant1, unsigned cont, unsigned n) {
	unsigned res;

	if (cont == n) {
		res = ant2 + ant1;
	} else {
		res = parejasEficRec(ant1,ant2+ant1,cont+1,n);
	}
	return res;
}

unsigned parejasEfic(unsigned n) {
	unsigned res;

	if (n <= 2) {
		res = 1;
	} else {
		res = parejasEficRec(1,1,3,n);
	}
	return res;
}
```

En cualquier caso, también podemos diseñar un algoritmo iterativo sencillo:

```
unsigned parejasIter(unsigned n) {
	unsigned ant2,ant1,res;

	if (n <= 2) {
		res = 1;
	} else {
		ant2 = 1;
		ant1 = 1;
		for (unsigned cont = 3; cont<=n; cont++) {
			res = ant2 + ant1;
			ant2 = ant1;
			ant1 = res;
		}
	}
	return res;
}
```