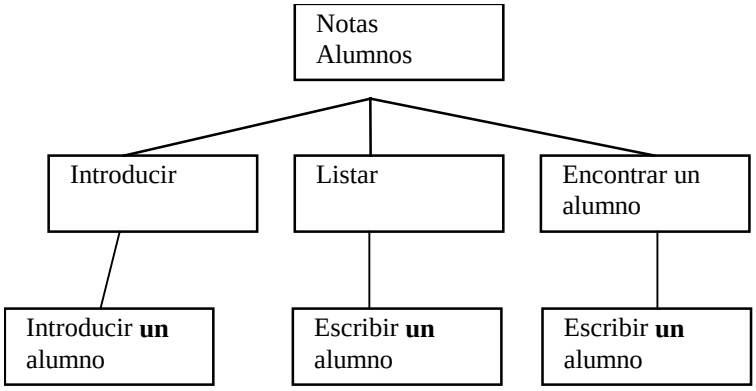
TEMA 5: Subprogramas, programación modular

5.1.-Definición de módulo. Programación modular

La programación modular está basada en la técnica de diseño descendente, que como ya vimos consiste en dividir el problema original en diversos subproblemas que se pueden resolver por separado, para después recomponer los resultados y obtener la solución al problema.

Ejemplo:



Ejemplo:

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

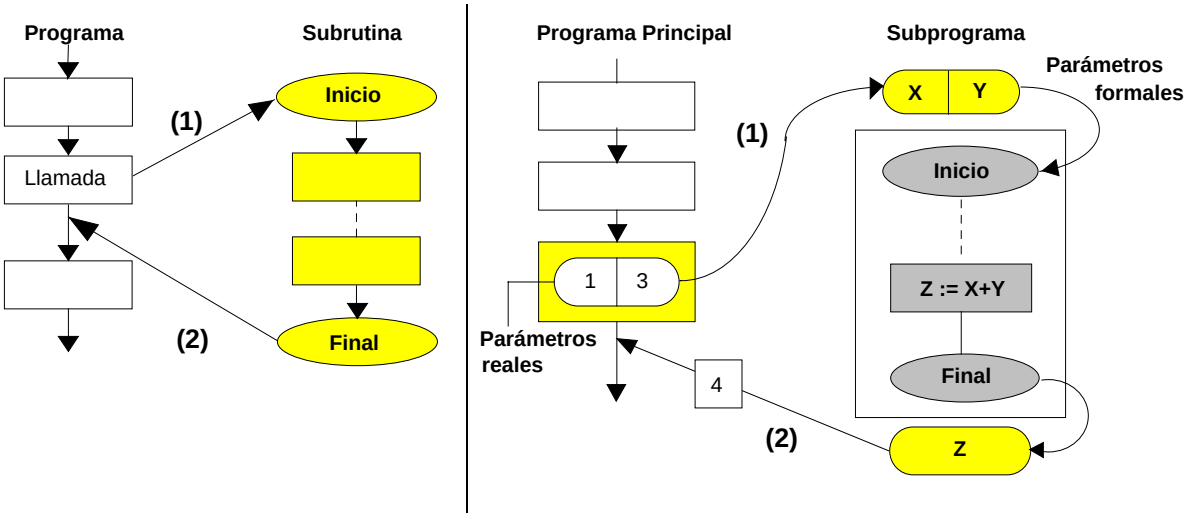
Un subproblema se denomina *módulo* y es una parte del problema que se puede resolver de manera **independiente**. Que un módulo sea independiente nos permite, por un lado, concentrarnos en su resolución olvidándonos del resto del problema, y por otro lado, permite reutilizar la solución obtenida para otra parte del programa u otro programa distinto.

Cada módulo se codifica dentro del programa como un *subprograma*, es decir, una sección de código independiente que realiza una tarea específica dentro del programa.

El concepto de subprograma es una evolución del antiguo concepto de subrutina, presente en lenguajes como ensamblador, Basic o primeras versiones de Fortran.

Una subrutina es una sección de código separada del programa principal que puede ser llamada en un momento dado (llamada a subrutina) y que una vez acabada su ejecución vuelve al punto donde se realizó la llamada.

Un subprograma hace el papel de un programa. Puede tener una sección de declaraciones (variables, constantes, etc…) y posee también unos datos de entrada y de salida. Esto permite, como ya veremos, que el subprograma sea totalmente independiente del programa principal.



5.2 Definición de subprogramas: funciones

En un subprograma hay que distinguir dos aspectos fundamentales:

- La definición del subprograma: Es la especificación de los parámetros de entrada y salida y las sentencias del subprograma.
- La llamada al subprograma: Es una sentencia que pasa el control del programa al subprograma. Cuando el subprograma acaba su ejecución, el control vuelve a la sentencia siguiente a la llamada.

Un subprograma puede necesitar o devolver datos. A estos datos se les denomina *parámetros*. Los parámetros pueden ser de *entrada* o de *salida*.

Los parámetros que se incluyen en la definición del subprograma se denominan *parámetros formales*. Los parámetros que se pasan al subprograma en la llamada se denominan *parámetros reales*.

5.2.1.-Definición de funciones en C++:

En C++ los únicos tipos de subprogramas existentes son las funciones. Una función es un subprograma que siempre tiene un parámetro de salida (Ej.: cos(x), pow(2,3)). Una función se define de la siguiente manera:

```
Tipo Nombre(lista de parámetros)    Cabecera de la función
  {
    Declaraciones

    Instrucciones                   Cuerpo de la función
    return Valor;
}
```

Donde *tipo* es el tipo del dato de salida, *nombre* es un identificador que representa el nombre de la función, *lista de parámetros* es una lista de parámetros separados por comas, donde cada parámetro se declara como en una declaración de variables normal.

Mediante la instrucción return se indica el valor que devolverá la función al acabar su ejecución.

Ejemplo:

```
int main()
{
    float x, y;

    x = triple(3) + 2;      →    x = 9 + 2
    y = triple(triple(2));  →    y = triple(6) →   y = 18
....
}

float triple(float x)
{
    return (3 * x);
}
```

Cuando hacemos una llamada a una función, lo primero que se realiza es una asignación de los parámetros reales a los parámetros formales y a continuación se ejecutan las instrucciones de la función.

Si queremos una función que no devuelva ningún valor, se declara de tipo *void*.

Ejemplo:

```
void EscribeSuma(int a, int b)
{
    cout << a + b;
    return;
}
```

Una función, al igual que cualquier otro identificador, sólo se puede utilizar a partir del momento en que lo declaramos. Para poder utilizar las funciones en cualquier punto del programa, lo que se hace es declararlas al principio del programa.

La declaración de una función únicamente necesita la cabecera y se denomina *prototipo.*

Ejemplo: *Prototipo para la función triple*

```
float triple(float x);
```

5.3.-Ámbito de identificadores

Ámbito de un identificador: Conjunto de sentencias donde puede utilizarse ese identificador.

Reglas para el cálculo del ámbito de un identificador:

1. Un identificador declarado en un bloque es accesible únicamente desde ese bloque y todos los bloques incluidos en él (se considera *local* a ese bloque). Un parámetro formal se considera también una declaración local al bloque de la función.

2. Los identificadores declarados fuera de cualquier bloque se consideran *globales* y pueden ser utilizados desde cualquier punto del fichero.

3. Cuando tenemos un bloque dentro de otro bloque y en ambos se declaran identificadores con el mismo nombre, el del bloque interno "oculta" al del bloque externo. (Ojo!! En C++ se admite la declaración de variables en cualquier bloque).

Ejemplo de ámbito

```
/************************************************/
/* Programa ejemplo de ámbito                   */
/************************************************/

#include <iostream.h>

int z;  // Global

int Sumar(int x, int y);

int main()
{
    int suma;  // Local a main

    z = 3;
    suma = Sumar(2, 3);
    cout << suma << endl << z << endl;
    return 0;
}

int Sumar(int x, int y)
{
    int z;  // Local a Sumar.
            // Oculta la z global

    z = x + y;
    return z;
}
```

En una función sólo se deben utilizar variables locales a la función o parámetros(que también son variables locales). Las variables globales no se deben utilizar nunca.

La razón es porque de esta manera la función es independiente del programa principal. La independencia permite que sea más fácil hacer cambios al programa, que la función pueda ser reutilizada en otros programas y que sea más fácil trabajar en equipo.

Además, si no se siguen estas reglas se pueden producir *efectos laterales*.

Ejemplo:

```
int z;

int Sumar(int x, int y);

int main()
{
    int suma;

    z = 3;
    suma = Sumar(2, 3);          -> 5
    cout << z;                   -> 5 (Efecto lateral)
}

int Sumar(int x, int y)
{
    z = x + y;

    return z;
}
```

En este ejemplo, la variable **z** ha cambiado "misteriosamente" de valor. Esto es lo que se denomina efecto lateral, que una función modifique el valor de variables que no han sido pasadas como parámetros.

Además, si cambiamos el nombre de la variable 'z', el procedimiento dejará de funcionar.

5.4.-Parámetros de un subprograma

Existen dos posibilidades para pasar parámetros a un subprograma: *por valor* y *por referencia*.

5.4.1.-Paso por valor

Se coge el valor del parámetro real y se asigna al parámetro formal. El parámetro formal es, por tanto, una copia del parámetro real.

Se utiliza exclusivamente para parámetros de entrada.

Ejemplo:

Todos los ejemplos que hemos visto hasta ahora utilizan paso de parámetros por valor.

```
void UnoMenos(int x)
{
    x = x - 1;
    cout << x;
    return;
}

...
n = 4;
UnoMenos(n);      → 3
cout << n;        → 4
```

'**n**' no cambia su valor.

5.4.2.-Paso por referencia

El paso de parámetros por referencia une la variable del parámetro formal con la variable del parámetro real y son tratadas como una sola variable. **Sólo** se pueden pasar por referencia variables.

Se utiliza para parámetros de entrada/salida.

Se diferencia del paso por valor poniendo el símbolo &.

Ejemplo:

```
void UnoMenos(int & x)
{
    x = x - 1;
    cout << x;
    return;
}
...
n = 4;
UnoMenos(n);                    → 3
cout << n;                      → 3
```

```
void intercambio(int & x, int & y)
{
    int z;

    z = x;
    x = y;
    y = z;
    return;
}
```

Las funciones son las únicas que tienen un parámetro exclusivamente de salida, y el paso de este parámetro es por valor.

5.4.3.-Gestión de la memoria

La memoria que utilizan las funciones se gestiona como una pila.

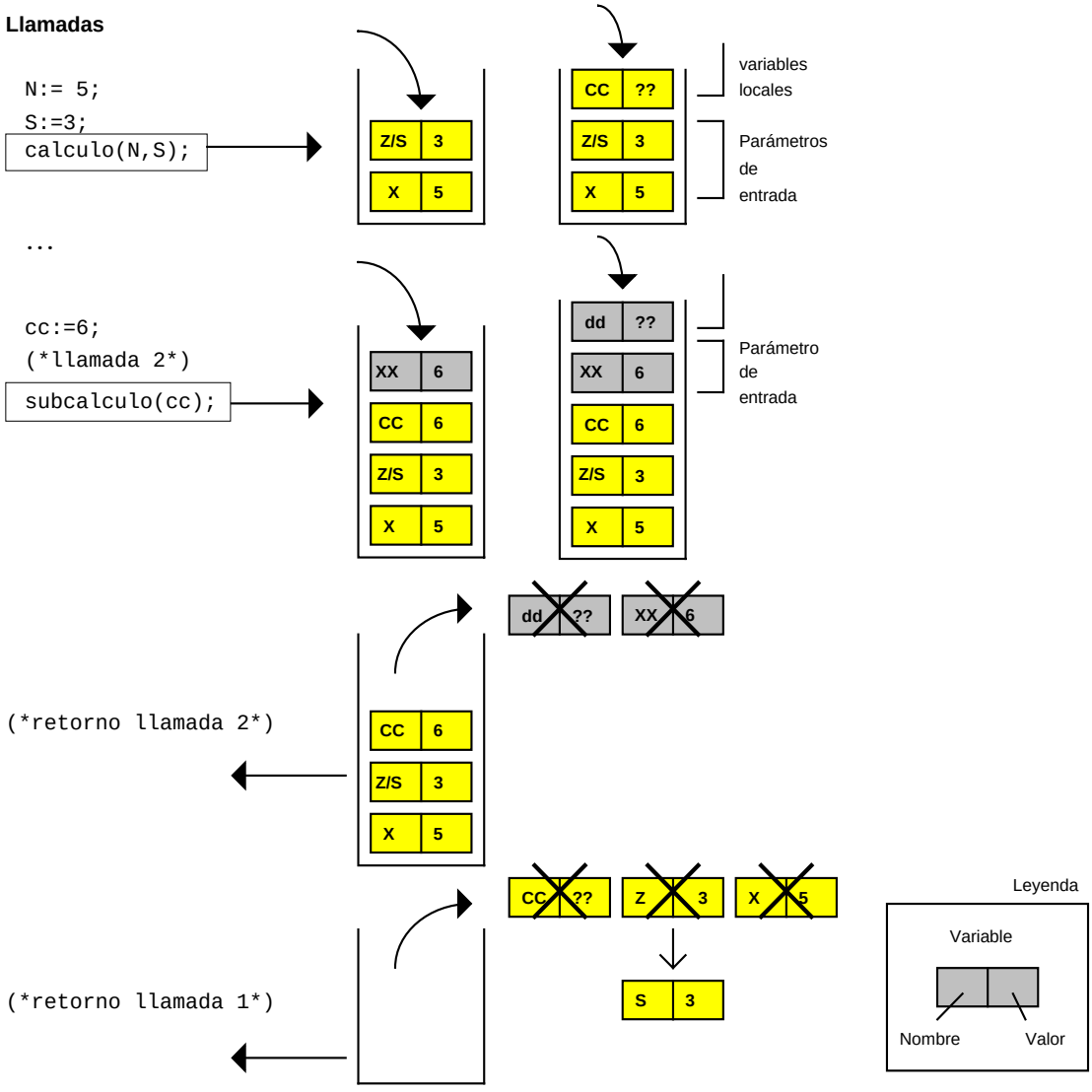
Cuando llamamos a un procedimiento, se reserva espacio para las variables locales "apilando" las variables. Cuando se acaba el procedimiento, las variables son "desapiladas" y por tanto dejan de existir.

En un lenguaje que sólo tenga gestión de memoria estática, realmente no existen subprogramas ni variables locales, únicamente subrutinas y variables globales.

Ejemplo:

```
int n, s;

void calculo(int x, int & z);
void subcalculo(int xx);

int main()
{
    n = 5; s = 3;
    calculo(n, s);
...
}

void calculo(int x, int & z)
{
    int cc;
```

```
        ...
        cc = 6;
        subcalculo(cc);
        ...
        return;
    }

    void subcalculo(int xx);
    {
        int dd;

        ...
        return;
    }
```



Traza de Programas:

Consiste en representar el valor de ciertas variables seleccionadas de nuestro programa después de ejecutar cada instrucción.

Ejemplo:

```
   /*****************************************/
   /* Ejercicio de Traza                    */
   /*****************************************/

   // Var globales
   int a, b;

   void P(int & c);

   int main()
   {
1)     a = 1;
2)     b = 3;
3)     P(a);
4)     cout << a << b << endl;
       return 0;
   }

   void P(int & c)
   {
       int b;

5)     b = 2;
6)     c = a + b + 2;
       return;
   }
```

Traza:

	a	b		
1)	1	?		
2)	1	3	c/a	b
3)	1	3	1	?
5)	1	3	1	2
6)	5	3	5	2
4)	5	3		

5.5.-Recursividad

Un subprograma se denomina *recursivo* cuando en el cuerpo del subprograma existe una llamada a sí mismo (Ej. N! = (N-1)! * N). Cuando únicamente existe una llamada, se denomina *recursividad lineal*, si existen varias es una *recursividad múltiple*.

Todo subprograma recursivo necesita una *condición de parada*, si no tendríamos una recursividad infinita. A la condición de parada se la denomina también *caso directo* y a la llamada recursiva *caso recursivo*.

Ejemplo:

```
long fact(int n)
{
    long f;

    if (n > 1)
    {
2)      f = fact(n-1);          -- Llamada recursiva
3)      f = f * n;
    }
    else
4)      f = 1;                  -- Caso base
    return f;
}
```

```
1)  F = fact (3)              →          F = 6
        fact (2) * n          →       2 * 3
            fact (1) * n      →   1 * 2
                1
```

	F						
	?	*n*	*f*				
1)	?	3	?	*n*	*f*		
2)	?	3	?	2	?	*n*	*f*
2)	?	3	?	2	?	1	?
4)	?	3	?	2	?	1	1
2')	?	3	?	2	1		
3)	?	3	?	2	2		
2')	?	3	2				
3)	?	3	6				
	6						

La recursividad es un método algorítmico alternativo a la iteración. Cualquier estructura iterativa es posible sustituirla por un subprograma recursivo y viceversa, aunque en ocasiones esta conversión puede ser bastante complicada.

La recursividad permite definir ciertos algoritmos de una manera más sencilla.

Normalmente un programa iterativo es más eficiente que uno recursivo, pero aún así, la versión recursiva puede ser más recomendable por claridad del código.

Cuando tenemos recursividad múltiple, existe el riesgo de que muchas de las llamadas se repitan, por lo que la versión recursiva del algoritmo no resulta recomendable.

Ejemplo: *Función de Fibonaci*

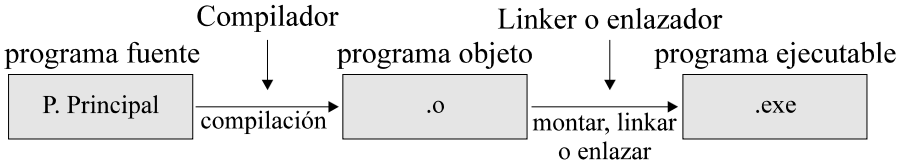
```
int fibonacci(int n)
{
    int fib;

    if (n = 0 || n = 1)
        fib = n;
    else
        fib = fibonacci(n - 1) + fibonacci(n - 2);
    return fib;
}
```

5.6.-Compilación separada

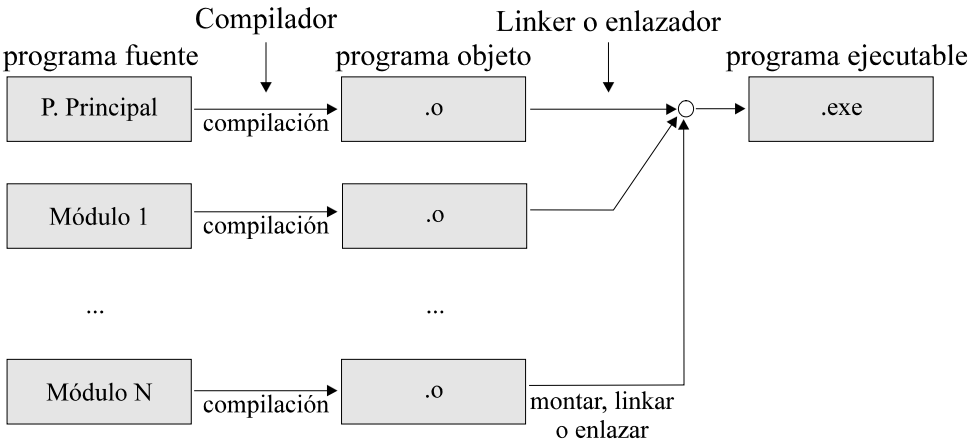
Cuando dividimos un problema en distintos módulos, no siempre ha de corresponder cada módulo a un subprograma. Puede corresponder a todo un conjunto de subprogramas que estarán agrupados en un fichero independiente. La principal utilidad de dividir el programa en ficheros distintos es que se pueden programar y compilar por separado. De esta manera no hace falta compilar cada vez todo el programa.

Esto se realiza normalmente para programas muy grandes o simplemente cuando queremos realizar una librería de subprogramas.

Compilación de un programa en un sólo fichero:



Compilación de un programa en varios ficheros:



En C++, cada módulo a compilar por separado consta de dos ficheros, el primero es un fichero que contiene todas las declaraciones y prototipos de funciones a utilizar en el programa principal y se denomina *fichero de cabecera*. Este fichero tiene extensión .h en lugar de la normal .cpp.

El segundo fichero contiene la definición de todas las funciones. Al inicio del fichero debe existir una directiva del compilador para incluir el fichero de cabecera:

```
#include "Fichero.h"
```

El programa principal debe incluir también todos los ficheros de cabecera de los módulos que quiera utilizar. Esto permite que se pueda compilar por separado del resto, pues en los ficheros de cabecera está toda la información necesaria para utilizar las funciones de los módulos.

Cada vez que se modifique algo del programa y halla que recompilar, sólo se compilarán los módulos que hayan sido modificados.

Ejemplo: Programa para el cálculo de $\binom{m}{n}$ mediante módulos.

fact.h:

```
/************************************************/
/* Fichero de cabecera para calculo de factorial */
/************************************************/

float fact(int n);
```

fact.cpp:

```
/************************************************/
/* Funciones para el calculo del factorial      */
/************************************************/

#include "fact.h"

// Calculo del factorial
float fact(int n)
{
    float f;
    int i;

    f = 1;
    for(i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

combinatorio.cpp:

```
/************************************************/
/* Programa para el calculo de m sobre n        */
/************************************************/

#include <iostream.h>
#include "fact.h"

int main()
{
    int m,n;
    float comb;

    cout << "Introduce los valores de m y n";
    cin >> m >> n;

    comb = fact(m) / (fact(n) * fact(m – n) );
    cout << "El resultado es:" << comb;
    return 0;
}
```