Programación modular.

Muchos programas se pueden dividir en pequeñas subtareas. Es una buena práctica de programación instrumentar cada una de esta subtareas como un módulo separado del programa. El diseño modular de los programas aumenta la corrección y claridad de éstos y facilita los posibles cambios futuros del programa.

Criterios de descomposición modular.

Cuando la tarea a realizar es suficientemente larga y costosa (y hay tareas innegablemente largas y costosas como, por ejemplo el diseño de un sistema operativo o de un compilador para un lenguaje de programación), o cuando parte del código del programa se va a utilizar varias veces y en distintas partes, o simplemente por mantener una unidad lógica en el algoritmo es interesante entresacar aquello que se repite.

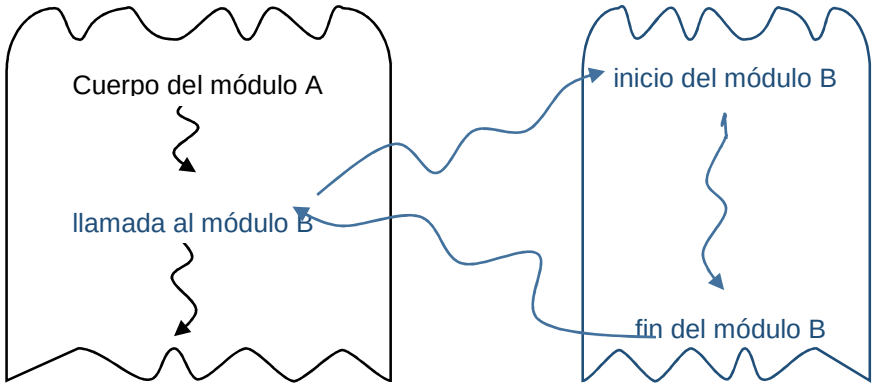
Uno de los métodos fundamentales para resolver un problema es dividirlo en problemas más pequeños, llamados *subproblemas*, pudiendo éstos, a su vez, ser subdivididos repetidamente en problemas más pequeños hasta llegar a obtener subprogramas lo suficientemente reducidos como para que resuelvan una única tarea y sean, por tanto, programados de forma más sencilla. Esto facilita, además, la detección de errores y la elaboración del programa, pues es más fácil pensar cómo resolver muchas tareas pequeñas, que pensar cómo resolver una sola más grande y compleja.

Esta técnica de dividir el problema principal en subproblemas se denomina frecuentemente *divide y vencerás* o *programación modular*. Además, si hay varias personas para hacer un programa, al dividir éste en fragmentos, hacemos posible que cada programador se pueda dedicar a uno o varios fragmentos, que posteriormente serán enlazados, reduciendo considerablemente el tiempo de progamación.

El método de diseño se denomina **diseño descendente** o ***top-down*** debido a que se comienza en la parte superior con un problema general y se diseñan soluciones específicas a sus subproblemas. Cada subproblema (*módulo* o *bloque*) será resuelto por medio de un **subprograma** y se corresponderá con una parte del problema global, que realizará una sola tarea y deberá ser programado con total independencia del resto de los módulos. Estos, podrán ser clasificados en *procedimientos* y *funciones*. Por otro lado, el **programa principal** será el encargado de controlar la secuencia de ejecución de los diferentes módulos de modo que se garantice el correcto funcionamiento del programa completo.

Un **subalgoritmo**, **subprograma** o **módulo** constituye una parte de un programa que realiza una tarea concreta mediante una serie de instrucciones.

Cuando un módulo A llama a otro módulo B, el flujo de ejecución pasa al módulo B. Cuando termina de ejecutarse el módulo B, el flujo de ejecución continúa en el módulo A a partir del punto en el que se llamó al módulo B.



El `main()` de un programa en C es un módulo que puede llamar a otros módulos pero no puede ser llamado por ningún módulo.

Funciones

Una **función**, en términos generales, es un módulo que devuelve un único resultado asociado a su nombre, al programa o subprograma que le llamó.

Se puede ver una función como una entidad de cálculo que recibe unos datos de entrada y después de procesarlos de manera adecuada, genera un resultado.

La sintaxis para definir una función es

```
TipoDatosResultado nombreFunción (TipoParametro1 param1, … TipoParametroN paramN){
    Secuencia de instrucciones
    return (expresión)
}
```

Para recibir los datos de entrada, la función emplea **parámetros** que son las variables definidas como `param1, … paramN.` El número de parámetros que recibe una función es variable: desde 0 hasta el número que se considere necesario. Sin embargo, una función obligatoriamente debe devolver un resultado. Para devolver el resultado se utiliza la instrucción:

```
return (expresión)
```

Una función finaliza su ejecución al terminar la secuencia de instrucciones que la componen o cuando ejecuta una instrucción `return`.

La expresión que devuelve la instrucción `return` debe ser del mismo tipo de datos que devuelve la función. Puede estar situada en cualquier punto de la secuencia de instrucciones aunque lo lógico es que sea la última instrucción ya que se encarga de devolver el resultado y el flujo de ejecución al punto desde donde se llamó a la función.

Ejemplo de función que recibe dos números y devuelve el más grande de los dos

```
int maximo(int a, int b)  {

   int  m;     //variable local a la función

   if (a > b)
      m = a;
   else
      m = b;
   return (m);
}
```

Llamando a una función.

Para ejecutar una función hay que realizar una llamada desde el `main` o desde otro módulo. Para llamar a una función, se emplea su nombre poniendo entre paréntesis los parámetros de entrada separados por comas. Si la función no tiene parámetros, se ponen los paréntesis vacíos. Dado que cuando se llama a una función se va a obtener un valor como resultado, es importante que la llamada a una función se incluya en algún tipo de expresión que permita aprovechar ese resultado. Usando la función máximo, algunos ejemplos son:

- En una condicional: `if (maximo(a,b)==a) . . .`

  En una asignación: `c = maximo(a,b);`

- En una instrucción de escritura: `cout << maximo(a,b);`

Ejemplo de programa que usa la función `maximo`

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

int main() {
   int n1, n2; //números introducidos por teclado
   int mayor; //el mayor número de los 2 introducidos

   cout << "Introduce dos números enteros:  ";
   cin >> n1 >> n2;
   mayor = maximo(n1, n2);
   cout << "El mayor número es:" << mayor;
}
```

Procedimientos

Un procedimiento es un módulo que realiza una tarea específica. Puede recibir cero o más valores del programa que lo llama y devolver cero o más valores a dicho programa llamador a través de la lista de parámetros.

En realidad, el lenguaje C no dispone de procedimientos. Para simular el uso de ellos se utilizan funciones que devuelven como resultado un dato de tipo `void`, lo que es equivalente a no devolver ningún dato, o lo que es lo mismo, devolver tipo vacío, nada.

Ejemplo de procedimiento que visualiza una fila de asteriscos

```
void estrellas (){
          cout << "***************";
}
```

Llamando a un procedimiento.

Para ejecutar un procedimiento hay que llamarlo mediante una sentencia formada por su nombre poniendo entre paréntesis los parámetros separados por comas. Si el procedimiento no tiene parámetros, se ponen los paréntesis vacíos.

Ejemplo de programa que llama al procedimiento estrellas

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

void estrellas();  // prototipo de la función

void main() {
   estrellas();
}
```

Estructura de un programa en C

Básicamente un programa en C está formado por un conjunto de módulos. En todo programa existe un módulo inicial que es el que se ejecuta cuando se ejecuta el programa. Se trata de la función `main()`. Este módulo se ocupa de llamar a los distintos módulos para que se realicen las tareas en el orden adecuado. A su vez, desde un módulo se puede llamar a otro y así sucesivamente. En programas complejos habrá muchos módulos que se llaman entre sí. Cuando desde un módulo se produce una llamada a otro módulo, resulta imprescindible que el compilador reconozca que ese identificador se corresponde con el nombre de un módulo que tiene unos determinados parámetros. Para ello, hay dos posibilidades:

- Tener escrito en el programa el código del módulo llamado antes que el código del módulo que llama. De esta forma, el compilar ya ha "visto" ese modulo y cuando se encuentra la llamada sabe a qué hace referencia.
- Declarar el módulo llamado al principio del programa. Lo que se conoce como escribir el prototipo de la función.

Para entender la diferencia entre ambas opciones, hay que distinguir entre **definición**, **declaración** y **llamada de un módulo**.

Declaración de un módulo: prototipos

Consiste en poner la cabecera o prototipo del módulo que incluye el nombre del módulo, el tipo de datos del resultado y el tipo de dato de los parámetros que tiene. El prototipo de la función máximo definida anteriormente se puede poner así:

```
int maximo(int a, int b);
```

o así:

```
int maximo(int, int);
```

La diferencia entre ambas radica en que en el segundo ejemplo no aparecen los nombres de los parámetros aunque sí su tipo de datos.

Es habitual, que si el código del programa esté en un fichero llamado `prog.c`, los prototipos de los módulos de ese programa estén en un fichero llamado `prog.h`.

Si no se emplean ficheros **.h** entonces los prototipos deben aparecer al principio del programa, debajo de la declaración de constante y tipos, caso de que los haya.

Definición de un módulo

Consiste en escribir el código del módulo. Por ejemplo, la definición de la función `maximo` es:

```
int maximo(int a, int b){
   int m;

   if (a > b)
      m = a;
   else
      m = b;
   return (m);
}
```

En la definición aparecen las instrucciones que componen el módulo.

Llamada a un módulo

Para ejecutar un módulo es necesario realizar una llamada o invocación desde el código de otro módulo. El caso del `main()`es diferente puesto que este módulo es llamado por el sistema cada vez que se ejecuta el programa.

Cuando se llama a un módulo se procede a ejecutarse su código. El primer paso consiste en asignar, según sea el modo de transferencia, los valores de los parámetros. Una vez asignado el valor de los parámetros, se procederá a ejecutar el cuerpo del módulo. Este código se ejecutará hasta que se alcance el final o se ejecute una instrucción `return`. A continuación el control vuelve al módulo que realizó la llamada y éste continuará ejecutándose.

Transferencia de información a/desde módulos: los parámetros.

Una de las características más importantes y diferenciadoras de los módulos es la posibilidad de comunicación entre el algoritmo principal y los subalgoritmos (o entre dos subalgoritmos). Esta comunicación se realiza a través de una *lista* de parámetros.

Un **parámetro** nos permite pasar información -valores a variables- desde el programa principal a un subprograma y viceversa.

Así pues, un **parámetro** puede ser considerado como una variable cuyo valor debe ser o bien proporcionado por el programa principal al procedimiento o ser devuelto desde el procedimiento hasta el programa principal. Por consiguiente, hay tres tipos de parámetros: **parámetros de entrada**, **parámetros de salida** y **parámetros de entrada/salida**.

- Los parámetros de entrada son parámetros cuyos valores deben ser proporcionados por el programa que llama;
- los parámetros de salida son parámetros cuyos valores se calcularán en el subprograma y se deben devolver al programa que ha llamado al módulo para su proceso posterior;
- los parámetros de entrada/salida además de proporcionar valores al subprograma serán susceptibles de devolver éstos modificados.

Ejemplo: **Parámetros de entrada**

El subalgoritmo *linea* dibuja el número de asteriscos indicados en el parámetro n.

```
void linea (int n){
   int j;

   for (j=1; j<=n; j++)
      cout << "*";
}
```

Ejemplo: **Parámetros de entrada y parámetros de salida**

El subalgoritmo *rectangulo* recibe el ancho (base) y alto (altura) de un rectángulo, calcula el área y el perímetro del mismo y devuelve los valores obtenidos al algoritmo principal.

```
void rectangulo (int ancho, int alto, int &ar, int &perim){
         ar = ancho * alto;
         perim = 2 * (ancho + alto);
}
```

Parámetros de entrada: *ancho*, *alto*

Parámetros de salida: *ar*, *perim*

Ejemplo: **Parámetros de entrada/salida**

El subalgoritmo *modifica* recibe el valor de una variable numérica alterándolo en función de que ésta sea positiva o negativa y devuelve el nuevo valor al algoritmo principal.

```
void modifica (int &n){
    if (n > 0){
        n = n * 2;
    else
        n = n * 3;
}
```

Parámetros actuales y parámetros formales.

Las sentencias de llamada a subalgoritmos constan de dos partes: *un nombre de procedimiento* y *una lista de parámetros* llamados **parámetros actuales o reales**. Por ejemplo, en la llamada a la función `maximo` definida anteriormente, n1 y n2 son los parámetros actuales. También se les llama argumentos.

```
...
mayor = maximo(n1, n2);
...
```

Los parámetros actuales tienen que tener unos valores que se pasan al subprograma **nombre_subalgoritmo**.

En la declaración de un subprograma, cuando se incluyen parámetros, éstos se denominan **parámetros formales o ficticios**. Estos sirven para contener los valores de los parámetros actuales cuando se invoca el subprograma. Por ejemplo, en la definición de la función `maximo`, a y b son los parámetros formales.

```
int maximo(int a, int b){
    ...
}
```

Los parámetros se emplean para almacenar el valor de los parámetros actuales cuando se llama al módulo. Un parámetro formal sólo existe mientras se está ejecutando el módulo.

Correspondencia de parámetros.

Los parámetros actuales en la invocación o llamada del subprograma deben coincidir en **número**, **orden** y **tipo** con los parámetros formales de la declaración del subalgoritmo. Es decir, debe existir una correspondencia entre parámetros actuales y formales.

Ejemplo (declarando el módulo al principio, es decir, escribiendo el prototipo de la función):

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

void rectangulo(int, int ,int &, int &);   //prototipo de la función


int main(){
   int base, altura, area, perimetro;

   cout << "Introduce la base:";
   cin >> base;
```

```
    cout << "Introduce la altura:";
    cin >> altura;
    rectangulo(base, altura, area, perimetro);
    cout << "El area es " << area;
    cout << " y el perimetro es " << perimetro;
}

void rectangulo (int ancho, int alto, int &ar, int &perim){
    ar = ancho * alto;
    perim = 2 * (ancho + alto);
}
```

Mismo ejemplo (escribiendo el módulo al principio):

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

void rectangulo (int ancho, int alto, int &ar, int &perim){
     ar = ancho * alto;
     perim = 2 * (ancho + alto);
}

int main(){
    int base, altura, area, perimetro;

    cout << "Introduce la base:";
    cin >> base;
    cout << "Introduce la altura:";
    cin >> altura;
    rectangulo(base, altura, area, perimetro);
    cout << "El area es " << area;
    cout << " y el perimetro es " << perimetro;
}
```

Los nombres de los parámetros actuales y formales pueden ser los mismos, aunque se recomienda que sean diferentes a efectos de legibilidad del programa. La correspondencia entre los parámetros del ejemplo anterior es:

Parámetros actuales (argumentos)		*Parámetros formales*
`base`	→	`ancho`
`altura`	→	`alto`
`area`	→	`ar`
`perimetro`	→	`perim`

Paso de parámetros por valor y por referencia.

Se establece una correspondencia entre cada argumento y su parámetro cada vez que se llama a un subalgoritmo. La sustitución de los parámetros formales por los actuales se puede realizar de dos formas:

- Transmisión por valor (paso por valor)
- Transmisión por referencia (paso por referencia)

Transferencia por valor.

Cuando un parámetro se pasa por valor, se está pasando una copia del valor del parámetro actual al parámetro formal, esta operación se denomina transmisión por valor. Son parámetros únicamente de entrada.

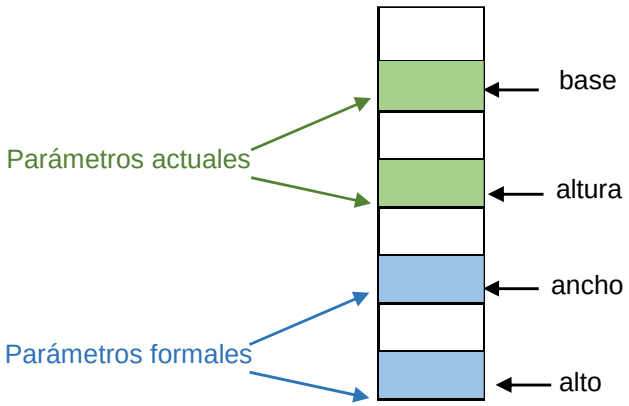
En la llamada al módulo, el valor del parámetro actual se pasa a la variable que represente el parámetro actual. Los parámetros actuales pueden ser cualquier expresión que se puede evaluar en ese momento, entendiendo como tal: constante, variable, función o combinación de los objetos anteriores con operadores. Este valor puede ser modificado dentro del programa, pero tal modificación no afecta al programa o subprograma llamador. Cuando la ejecución del módulo termina, el parámetro sigue conservando el mismo valor que tenía a la entrada del módulo.

Cuando se define un parámetro por valor, se crea una copia del mismo, de tal manera que cuando se llama al módulo se trabaja con esa copia y no con el parámetro real.

Se produce una asignación del valor del argumento a su parámetro correspondiente. El parámetro es, en efecto, una variable independiente de nueva creación, con su propia situación en memoria, que recibe el valor del argumento al comienzo de la ejecución del subalgoritmo. Puesto que argumento y parámetro son variables independientes, cualquier cambio en el parámetro que ocurra durante la ejecución del subalgoritmo no tiene ningún efecto en el valor del argumento original. Si un procedimiento o función contiene una sentencia que cambia un parámetro pasado por valor, sólo la copia del parámetro actual se cambia, no el original. Al terminar la ejecución del subalgoritmo la variable parámetro se destruye y el valor que contenía se pierde.

Ejemplo:

La siguiente figura refleja la memoria del ejemplo anterior (`rectangulo`), donde las celdas coloreadas en verde son los parámetros actuales y las coloreadas en azul son los parámetros formales del ejemplo anterior. Como se puede observar, parámetros correspondientes están en celdas distintas de la memoria, es decir `base` está en una celda y `ancho`, que es su parámetro correspondiente, está en otra celda distinta. Lo mismo sucede con `altura` y `alto`.



Transferencia por referencia.

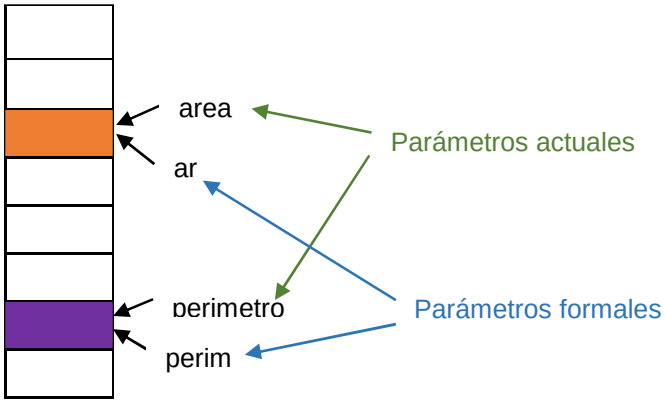
Los parámetros variables o por referencia se utilizan tanto para recibir como para transmitir valores desde un subprograma a otro, o entre el programa principal y un subprograma (módulo). Los parámetros actuales deben ser obligatoriamente variables. Cuando pasamos un parámetro por referencia en un módulo se pasa la dirección que apunta al parámetro actual. Todo cambio que se realice en el procedimiento afectará al parámetro real. Por esta razón no podemos pasar valores concretos.

No implica la creación de una posición nueva de memoria para el parámetro. El parámetro viene a ser otro nombre para la misma dirección de memoria, ya creada para el valor del argumento. Esta zona de memoria puede usarse para pasar información en ambos sentidos, pues cualquier referencia al parámetro correspondiente dentro del subalgoritmo es también una referencia al argumento mismo.

Un parámetro por referencia requiere como parámetro actual a una variable, pues puede ser modificado por el procedimiento, mientras que si se pasa por valor puede tener cualquier expresión que le suministre un valor del mismo tipo de datos que el parámetro formal.

Un parámetro por referencia puede actuar de parámetro de salida o de entrada/salida. Debe usarse cuando se necesita obtener el posible cambio que se haya producido en el parámetro tras la ejecución del módulo.

La siguiente figura refleja la memoria del ejemplo ya comentado. Como se puede observar, parámetros correspondientes están en las mismas celdas de la memoria, es decir `area` está en una celda y `ar`, que es su parámetro correspondiente, está en la misma celda. Lo mismo sucede con `perimetro` y `perim`. O lo que es lo mismo, apuntan a la misma dirección de memoria, tal y como se observa en el siguiente dibujo.



En lenguaje C, para indicar que un paso de parámetros es por referencia, en la definición del módulo se antepone el carácter & al nombre del parámetro formal.

En el siguiente ejemplo, `ar` y `perim` son parámetros por referencia, mientras que ancho y `alto` son por valor

```
void rectangulo (int ancho, int alto, int &ar, int &perim)
```

Cuándo utilizar parámetros por valor o por referencia.

Algunas reglas a seguir para utilizar uno u otro tipo de parámetro son:

- Si la información que se pasa al subprograma no tiene que ser devuelta fuera del subalgoritmo, el parámetro formal que representa la información puede ser un parámetro por valor (parámetro de entrada).
- Si se tiene que devolver información al programa llamador, el parámetro formal que representa esa información debe ser un parámetro por referencia (parámetro de salida).
- Si la información que se pasa al subalgoritmo puede ser modificada y se devuelve un nuevo valor, el parámetro formal que representa a esa información debe ser un parámetro variable (parámetro de entrada/salida).

Variables locales y globales.

Las variables pueden ser de dos clases: *variables locales* y *variables globales*.

Una **variable local** es una variable que está declarada dentro de un módulo, y se dice que es local al subprograma o que su ámbito de uso se restringe a dicho subprograma. Por tanto, una variable

local sólo está disponible durante el funcionamiento del mismo. Su valor se pierde una vez que el subprograma finaliza. Una variable local se destruye cuando finaliza la ejecución del módulo.

Las variables declaradas fuera de cualquier módulo se denominan **variables globales**. Al contrario que las variables locales, cuyos valores se pueden utilizar sólo dentro del subprograma en que están declaradas, las variables globales pueden ser utilizadas en todo el programa. Se destruyen cuando finaliza la ejecución del programa.

La comunicación entre módulos debe hacerse a través de parámetros y no de variables globales. Se debe evitar su utilización porque se pueden producir los denominados *efectos laterales*.

Efectos laterales.

Se podría definir un **efecto lateral** como cualquier efecto de un módulo sobre otro módulo que no es parte de la interfaz definida explícitamente entre ellos. Dicho de otro modo, la acción de modificar el valor de una variable global en un subprograma.

Si se desea transferir la información contenida en alguna variable global a un subprograma, es aconsejable hacerlo a través del uso de parámetros. Esta práctica no sólo evita los efectos laterales, sino que mejora la legibilidad de un subprograma, dado que como todos los parámetros se listan en la cabecera del subprograma, es fácil determinar las entradas y salidas del mismo, mientras que si no se utilizan parámetros, será necesario consultar el cuerpo del subprograma para averiguar que variables intervienen en el intercambio de información.

Ejemplo

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

int j;

void lateral(){
   for (j=1; j<=10; j++)
      cout << "j vale " << j << endl;
}

int main(){
   j=8;
   cout << "j= " << j << endl;
   lateral();
   cout << "j= " << j << endl;
}
```

En este ejemplo, el valor de la variable j se ve alterado tras la ejecución del módulo lateral, a pesar de que dicha variable no aparece como parámetro actual en la llamada al módulo y por tanto no debería sufrir ninguna modificación.

La comunicación entre un subprograma y el programa debe realizarse completamente a través de parámetros y no de variables globales.

Ámbito de un identificador.

En la descomposición de un algoritmo en subalgoritmos el algoritmo principal es la raíz y de éste penden muchas ramas (subprogramas). De estas ramas, a su vez, pueden pender otras ramas más pequeñas anidadas dentro del algoritmo principal y otros subalgoritmos. Es decir, que un programa puede contener diferentes bloques y un subprograma puede a su vez contener otros subprogramas, que se dice están anidados.

El *ámbito de un identificador* es la sección de un programa en la que un identificador es válido, y las reglas que definen el ámbito de un identificador se llaman *reglas de ámbito*.

Reglas de ámbito

1. El ámbito de un identificador es el dominio en que está declarado. Por consiguiente, un identificador declarado en un bloque P puede ser referenciado en el subprograma P y en todos los subprogramas encerrados en él. (Cualquier identificador se puede utilizar en cualquier parte del bloque en que está declarado o en cualquier bloque anidado).
2. Si un identificador j declarado en el subprograma P se redeclara en algún subprograma interno Q encerrado en P, entonces el subprograma Q y todos sus subprogramas encerrados se excluyen del ámbito de j declarado en P. Prevalece el "más interno" o bien, una variable local prevalece sobre cualquier variable que "le sea global".

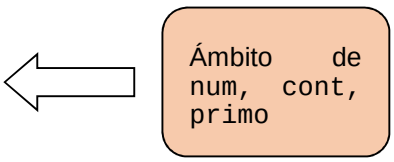
En el siguiente programa, hay una función llamada `es_primo` y el `main`. El parámetro formal `num` tiene un ámbito restringido solamente a la función `es_primo`, no puede ser referenciado en el `main`. Lo mismo se puede decir de las variables `cont` y `primo` (ya que estas son locales al módulo). De igual forma, la variable n sólo puede ser referenciada en el `main`.

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

```
// Este módulo comprueba si un número es primo o no
bool es_primo(int  num)
{
   int cont;  // contador  (dato auxiliar)
   bool primo; // es primo o no (dato de salida)

   primo = true;
   cont = 2;
   while ( (cont < num) && primo) {
      // comprobar si es divisible por otro número
      primo = ! (num % cont == 0);
      cont = cont + 1;
   }
   return (primo);
}

int main() {
   int n;  // número introducido por teclado  (dato de entrada)
   cout << "Introduce un número entero:  ";
   cin >> n;
   if (es_primo(n))
      cout << "El número es primo";
   else
      cout << "El número no es primo";
   cout << endl;
}
```



Diferencias entre procedimientos y funciones.

Las acciones y funciones que sirven para constituir subprogramas son similares, aunque presentan notables diferencias entre ellos:

1. Las funciones devuelven un solo valor asociado a su nombre a la unidad de programa que las invoca. Los procedimientos pueden devolver cero, uno o varios valores. Para "devolver" valores lo hacen utilizando parámetros de salida. En el caso de no devolver ningún valor, realizan alguna tarea tal como alguna operación de entrada y/o salida.
2. A un nombre de procedimiento no se le puede asignar un valor, y por consiguiente ningún tipo de datos está asociado a él.
3. Una función se referencia utilizando su nombre en una expresión, mientras que un procedimiento se referencia por una llamada o invocación al mismo.

Ventajas derivadas de la utilización de subprogramas.

A primera vista, los subalgoritmos parecen dificultar la escritura de un algoritmo. Sin embargo, no sólo no es así, sino que la organización de un programa en subalgoritmos lo hace más fácil de escribir y depurar.

Las ventajas más sobresalientes de utilizar módulos son:

1. El uso de módulos facilita el diseño descendente.
2. Los módulos se pueden ejecutar más de una vez en un programa y en diferentes programas, ahorrando en consecuencia tiempo de programación. Una vez que un módulo se ha escrito y comprobado, se puede utilizar en otros programas.
3. Un módulo sólo se escribe una vez y puede ser usado varias veces desde distintas partes del programa con lo que disminuye el tamaño total del programa.
4. El uso de módulos facilita la división de las tareas de programación entre un equipo de programadores.
5. Los módulos se pueden comprobar individualmente y estructurarse en librerías específicas.
6. Los programas son más fáciles de entender (más legibles).
7. Los programas son más fáciles de modificar.

Librerías propias de C/C++.

La mayoría de los lenguajes de programación proporcionan una colección de procedimientos y funciones de uso común.

En C/C++ para poder usar los módulos incluidos en una librería se utiliza la directiva del compilador `#include`. Precisamente, para poder usar las funciones `cin` y `cout` siempre se incluye la librería `iostream` en los programas en los que se quiere hacer entrada/salida de datos

```
#include <iostream>
```

Hay una gran variedad de librerías disponibles con módulos de distinto tipo:

- Funciones matemáticas.
- Manejo de caracteres y de cadenas de caracteres.
- Manejo de entrada/salida de datos.
- Manejo del tiempo (fecha, hora,…).
- etc.

Algunas funciones de uso común son:

Librería C++	Librería C	Función	Descripción
<math.h>	<math.h>	double exp(double x)	Devuelve e^x
		double fabs(double x)	Devuelve el valor absoluto de x
		double pow(double x, double y)	Devuelve x^y
		double round(double x)	Devuelve el valor de x redondeado
		double sqrt(double x)	Devuelve la raiz cuadrada de x
<iostream>	<ctype.h>	int isalnum(int c)	Devuelve verdadero si el parámetro es una letra o un dígito
		int isdigit(int c)	Devuelve verdadero si el parámetro es un dígito
		int toupper(int c)	Devuelve el carácter en mayúsculas
	<stdlib.h>	int rand(void)	Devuelve un número aleatorio entre 0 y RAND_MAX

Ejemplo: Programa que pide un número (entre 1 y 10) al usuario y comprueba si es igual que otro generado de forma aleatoria

```
#include <iostream>
#include <stdlib.h>
using namespace std;

//Función que pide al usuario un número entre 1 y 10
int pideNum(){
    int n;

    do{
        cout << "Introduce un número entre 1 y 10:";
        cin >> n;
        if (n<1 || n>10)
           cout << "Número no valido" << endl;
    }while (n<1 || n>10);
    return (n);
}

//Función que genera un número entre 1 y 10 de forma aleatoria
int generaAleat(){
    int n;

    srand(time(NULL));
    n = rand()%10+1;
    return (n);
}
```

```
int main(){
        int num, aleat;

        num=pideNum();
        aleat = generaAleat();
        if (num==aleat)
           cout << "Has acertado";
        else
           cout << "No has acertado";
}
```

En el programa anterior se hace uso de la función `rand()` para generar números aleatorios. Dado que se quiere obtener un número entre 1 y 10, una forma de sencilla de hacerlo es calcular el resto de la división entre 10 del número obtenido por la función `rand()`. De esta forma, sea cual sea el número generado por `rand()`, se obtiene un número entre 0 y 9 dependiendo del valor del resto. Si a este número se le suma un 1, seguro que el resultado se encuentra entre 1 y 10.

Para que la función `rand()` trabaje de forma adecuada, es importante inicializar el generador de números aleatorios, esto se hace con la función `srand()`. Es una práctica habitual inicializar el motor con la hora del sistema ya que éste es un valor que está en continuo cambio, de ahí el argumento `time(NULL)`.