Programación modular.

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

Descomposición modular.

Cuando hay que diseñar un programa para resolver un problema con un cierto nivel de complejidad, resulta muy útil tratar de descomponer el problema en trozos más pequeños que sean más fáciles de resolver. Esto puede suponer varias ventajas:

- Cada "trozo de programa" independiente será más fácil de programar, al realizar una función breve y concreta.
- El "programa principal" será más fácil de leer, porque no necesitará contener todos los detalles de cómo se hace cada cosa.
- Podremos repartir el trabajo, para que cada persona se encargue de realizar un "trozo de programa", y finalmente se integrara el trabajo individual de cada persona.

Estos "trozos" de programa son lo que se suele llamar módulos, subalgoritmos o subrutinas. Un **módulo**[1] constituye una parte de un programa que realiza una tarea concreta mediante una serie de instrucciones. En el lenguaje C, se les denomina **funciones**, en otros lenguajes son **funciones** o **procedimientos** dependiendo de sus características.

Por ejemplo, podríamos crear un módulo llamado `autor`, que escribiera varios mensajes en la pantalla.

```
void autor() {
    cout << "*****************************************" << endl;
    cout << "*   programa realizado por Programación 1  *" << endl;
    cout << "*****************************************" << endl;
}
```

Ahora, cada vez que queramos que se visualice este mensaje en nuestra pantalla tendremos que llamar a ese módulo.

```
int main() {
    autor();
}
```

[1] A partir de este momento se utilizarán de forma indistinta la palabra subalgoritmo, procedimiento, función, subrutina para hacer referencia a un módulo.

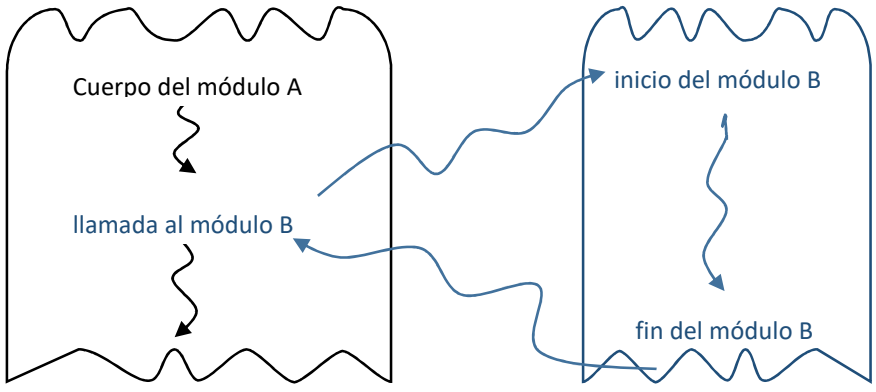
De modo, que el programa completo sería:

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

void autor() {
    cout << "*****************************************" << endl;
    cout << "*   programa realizado por Programación 1  *" << endl;
    cout << "*****************************************" << endl;
}

int main() {
    autor();
}
```

Y al compilar y ejecutar dicho código obtendríamos el siguiente resultado por pantalla

```
*****************************************
*   programa realizado por Programación 1  *
*****************************************
```

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 ni se debe llamar a sí mismo.

Funciones

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

- ✓ Realiza una función que devuelva el cuadrado de un número entero dado.

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

int  cuadrado(int n) {
  int result;

  result = n * n;

  return(result);
}

int main() {
   int num;

   cout << "Dame un número: ";
   cin >> num;
   cout << "El cuadrado de " << num << " es " << cuadrado(num) << endl;
}
```

- ✓ En el ejercicio anterior, modifica el módulo principal (main) para que almacene el resultado de "cuadrado" en una variable y posteriormente muestre ese valor.

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

   cout << "Dame un número: ";
   cin >> num;
   resultado = cuadrado(num);
   cout << "El cuadrado de " << num << " es " << resultado << endl;
}
```

Veamos un 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 escribe por pantalla y en orden de mayor a menor tres números que se le han pasado como parámetros:

```
void ordenados (int n1, int n2, int n3) {
  if (n1 >= n2 && n1 >= n3)
     if (n2 >= n3)
        cout << n1 << " " << n2 << " " << n3 << endl;
     else
        cout << n1 << " " << n3 << " " << n2 << endl;
  else  if (n2 >= n1 && n2 >= n3)
           if (n1 >= n3)
              cout << n2 << " " << n1 << " " << n3 << endl;
           else
              cout << n2 << " " << n3 << " " << n1 << endl;
```

```
        else   if (n1 >= n2)
                  cout << n3 << " " << n1 << " " << n2 << endl;
               else
                  cout << n3 << " " << n2 << " " << n1 << endl;
}
```

El main que llamará al módulo anterior será:

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 módulo principal (main) que llama al procedimiento **ordenados**.

```
int main(){
  int num1, num2, num3;

  cout << "Introduce tres números separados por espacio: ";
  cin >> num1 >> num2 >> num3;

  cout << "Los números ordenados de mayor a menor quedarían: " << endl;
  ordenados(num1, num2, num3);
  cout << endl;
}
```

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 `ordenados` definida anteriormente se puede poner así:

```
void ordenados(int n1, int n2, int n3);  //modo 1 prototipo procedimiento
```

o así:

```
void ordenados(int, int, int);  // modo 2 prototipo procedimientos
```

Si queremos mostrar el prototipo de la función máximo sería:

```
int maximo(int n1, int n2);    // modo 1 prototipo función
```

o así:

```
int maximo(int, int);    // modo 2 prototipo función
```

La diferencia entre los dos modos de escribir el prototipo radica en que en el modo 2 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` o `prog.cc`, 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 `ordenados` sería:

```
void ordenados (int n1, int n2, int n3) {
  if (n1 >= n2 && n1 >= n3)
     if (n2 >= n3)
        cout << n1 << " " << n2 << " " << n3 << endl;
     else
        cout << n1 << " " << n3 << " " << n2 << endl;
  else  if (n2 >= n1 && n2 >= n3)
           if (n1 >= n3)
              cout << n2 << " " << n1 << " " << n3 << endl;
           else
              cout << n2 << " " << n3 << " " << n1 << endl;
        else   if (n1 >= n2)
                  cout << n3 << " " << n1 << " " << n2 << endl;
               else
                  cout << n3 << " " << n2 << " " << n1 << endl;
}
```

Y la definición de la función que devuelve el mayor de dos números sería:

```
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 ejecuta 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 (en los procedimientos) o se ejecute una instrucción `return` (en las funciones). A continuación el control vuelve al módulo que realizó la llamada y éste continuará ejecutándose.

Veamos cómo quedaría el programa completo que utiliza un módulo que escribe por pantalla y de forma ordenada decreciente tres números.

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

void ordenados(int, int, int);  // prototipo de la función

int main(){
  int num1, num2, num3;

  cout << "Introduce tres números separados por espacio: ";
  cin >> num1 >> num2 >> num3;

  cout << "Los números ordenados de mayor a menor quedarían: " << endl;
  ordenados(num1, num2, num3);
  cout << endl;
}

void ordenados (int n1, int n2, int n3) {
  if (n1 >= n2 && n1 >= n3)
     if (n2 >= n3)
        cout << n1 << " " << n2 << " " << n3 << endl;
     else
        cout << n1 << " " << n3 << " " << n2 << endl;
  else  if (n2 >= n1 && n2 >= n3)
           if (n1 >= n3)
              cout << n2 << " " << n1 << " " << n3 << endl;
           else
              cout << n2 << " " << n3 << " " << n1 << endl;
        else   if (n1 >= n2)
                  cout << n3 << " " << n1 << " " << n2 << endl;
               else
                  cout << n3 << " " << n2 << " " << n1 << endl;
}
```

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**

Hemos visto anteriormente que para devolver un dato debemos escribir **`return(dato a devolver)`** en el módulo. A este tipo de módulos los denominamos funciones. Pero ¿y si queremos devolver más de un dato?

¿Sería posible escribir?

```
return (dato1, dato2);
```

ESTO ES INCORRECTO

o

```
return(dato1); return(dato2);
```

ESTO ES INCORRECTO

En ese caso debemos devolver los datos a través de los parámetros de salida. Estos parámetros a diferencia de los de entrada son parámetros pasados por referencia (paso por referencia explicado en pág 12), no por valor. Es decir, el parámetro tiene la dirección original de la variable que desea modificar, de ese modo, cuando se modifica el valor no se hace en la variable del módulo, sino en la variable original.

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**

En ocasiones puede que nos interese que una variable llegue al módulo con un valor y salga del módulo con otro valor. Entonces, ¿deberíamos hacer lo siguiente?

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

ES CORRECTO??????

Que la misma variable sea utilizada como parámetro de entrada y a la vez sea la variable que se da como salida en el `return` no es correcto con la excepción de que desde donde ha sido llamado queramos mantener ambos valores, el de entrada y el de salida en variables distintas. Si ese no es el caso, es incorrecto. Veamos dos maneras de uso:

```
int main(){
   int nent, nsal;

```

```
   cout << "Número: ";
   cin >> nent;

   nsal = modifica(nent);

   cout << "Tras modificar el valor " << nent << " tenemos " << nsal << endl;
}
```

En este ejemplo, hemos mantenido ambos valores porque nos interesa por alguna razón. Sin embargo, en

```
int main(){
   int num;

   cout << "Número: ";
   cin >> num;

   num = modifica(num);

   cout << "Tras modificar el valor tenemos " << num << endl;
}
```

No tiene sentido que la misma variable esté como parámetro y como variable que recoge el valor que devuelve la función.

Para este segundo caso, la mejor solución sería:

```
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 formales 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.

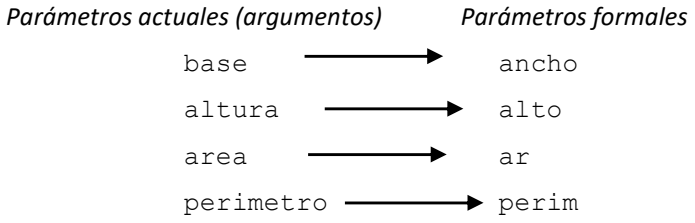
Ejemplo (declarando el módulo al principio, es decir, escribiendo el prototipo de la función):

```
#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:



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)

Paso 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.

Cuando los argumentos se pasan a un módulo por valor, se pasa una copia del valor del argumento y no el argumento en sí. Al pasar una copia, cualquier modificación que se realice sobre esta copia no tendrá efecto sobre el argumento original utilizado en la llamada del módulo. Es decir, los cambios realizados sobre esas variables dentro del módulo sólo tendrán efecto dentro del propio módulo. Al devolver el control al punto desde donde se llamó al módulo, las variables mantienen los valores que tenían antes de la llamada.

Resumiendo, 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.

Ejemplo:

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

void modificar(int);

int main(){
     int n;

     n = 1;
     cout <<"n = " << n << " antes de la llamada a modificar \n";
     modificar(n);
     cout <<"n = " << n << " posterior a la llamada a modificar \n";
}

void modificar(int var) {
     cout <<"  var = " << var << " dentro de modificar, antes de sumar \n";
     var = var + 5;
     cout <<"  var = " << var << " dentro de modificar, posterior a sumar
\n";
}
```

La salida por pantalla sería

```
n = 1 antes de la llamada a modificar
  var = 1 dentro de modificar, antes de sumar
  var = 6 dentro de modificar, posterior a sumar
n = 1 posterior a la llamada a modificar
```

¿Qué está ocurriendo? Desde main pasamos la variable n a modificar. Pero no pasamos la variable original, sino una copia de su contenido. Por esta razón, cuando la variable es modificada, modificamos la variable del módulo, no la variable del main, que sería la original.

Podemos pensar que eso ocurre porque las llamamos de forma diferente. Probemos qué ocurriría si tuviesen el mismo nombre. A continuación, adjuntamos el mismo código con esta modificación.

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

void modificar(int);

int main(){
     int n;

     n = 1;
     cout <<"n = " << n << " antes de la llamada a modificar \n";
     modificar(n);
     cout <<"n = " << n << " posterior a la llamada a modificar \n";
}

void modificar(int n) {
     cout <<"  n = " << n << " dentro de modificar, antes de sumar \n";
     n = n + 5;
     cout <<"  n = " << n << " dentro de modificar, posterior a sumar \n";

}
```

La salida obtenida sería la siguiente:

```
n = 1 antes de la llamada a modificar
  n = 1 dentro de modificar, antes de sumar
  n = 6 dentro de modificar, posterior a sumar
n = 1 posterior a la llamada a modificar
```

Con respecto al resultado de la ejecución anterior no ha cambiado nada, salvo el nombre de las variables. Esto ocurre porque el nombre que se le dé a una variable no es relevante, lo relevante es la dirección que ocupa y si accedemos a la dirección de esa variable o si realizamos una copia de ella. Evidentemente, en el caso en el que realizamos una copia, las modificaciones no afectarán a la original, sólo a la copia.

Paso por referencia.

Acabamos de ver que cuando se pasa un argumento por valor, realmente se pasa una copia de éste, y si esta copia se modifica, el argumento original no se ve alterado. Sin embargo, en ocasiones queremos que el módulo cambie los valores de los argumentos que pasamos. Para esto, debemos utilizar paso de argumentos por referencia.

Cuando pasamos argumentos por referencia, lo que estamos pasando son direcciones de memoria: las direcciones de memoria donde se encuentran las variables originales de los argumentos. Esta operación se consigue mediante el uso de punteros. De este modo, cuando llamamos a un módulo, lo que realmente le pasamos son los punteros a los argumentos que deseamos modificar.

A continuación, vamos a modificar el ejemplo anterior para que la variable original sea modificada dentro del módulo y, por tanto, sea alterada. Es decir, vamos a utilizar el paso de argumentos por referencia.

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

void modificar(int &);

int main(){
     int n;

     n = 1;
     cout <<"n = " << n << " antes de la llamada a modificar \n";
     modificar(n);
     cout <<"n = " << n << " posterior a la llamada a modificar \n";
}

void modificar(int &var) {
     cout <<"  var = " << var << " dentro de modificar, antes de sumar \n";
     var = var + 5;
     cout <<"  var = " << var << " dentro de modificar, posterior a sumar
\n";
}
```

La salida obtenida sería la siguiente:

```
n = 1 antes de la llamada a modificar
  var = 1 dentro de modificar, antes de sumar
  var = 6 dentro de modificar, posterior a sumar
n = 6 posterior a la llamada a modificar
```

Podemos observar que el valor de `n` ha cambiado, puesto que el módulo modificar ha utilizado la dirección de memoria de esta variable, es decir, ha trabajado con la variable original.

Vamos a analizar detenidamente este ejemplo. En C, todos los argumentos se pasan por valor. Pero en este caso, estamos pasando el valor de la dirección de memoria de la variable `n` y no el valor de su contenido (que sería 1 en este caso). Al finalizar la ejecución del módulo, el valor de dicha dirección permanece igual y lo que se ha modificado es el contenido de esa dirección de memoria. 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.

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 por referencia (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.

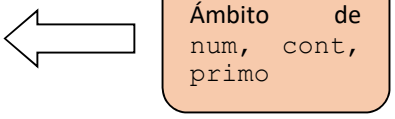
El *ámbito de un identificador* es la sección de un programa en la que un identificador es válido, y las reglas que definen el ámbito de un identificador se llaman *reglas de ámbito*.

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`.

```
// 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.

Los procedimientos 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)`.