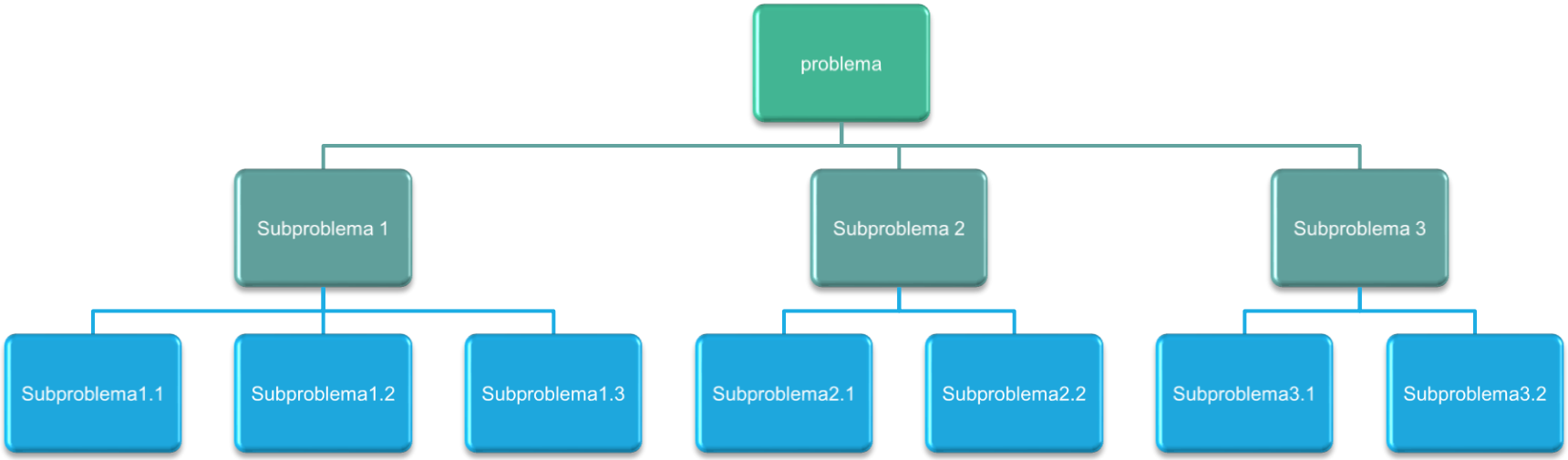
FUNCIONES EN C++

UNIDAD 2

Divide y Vencerás

- Un método para solucionar un problema complejo es dividirlo en subproblemas -problemas más sencillos- y a continuación dividir estos subproblemas en otros más simples, hasta que los problemas más pequeños sean fáciles de resolver.
- Esta técnica de **dividir el problema principal** en **subproblemas** se suele denominar “divide y vencerás”.

Diseño Descendente

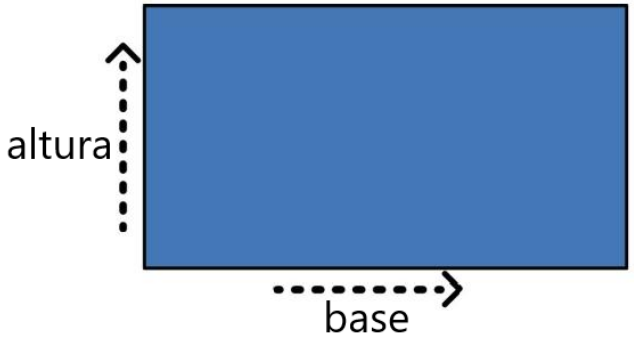


- Este método de diseñar la solución de un problema principal obteniendo las soluciones de sus subproblemas se conoce como diseño descendente (**top-down design**).
- Normalmente las partes en que se divide un programa deben poder desarrollarse **independientemente** entre sí.

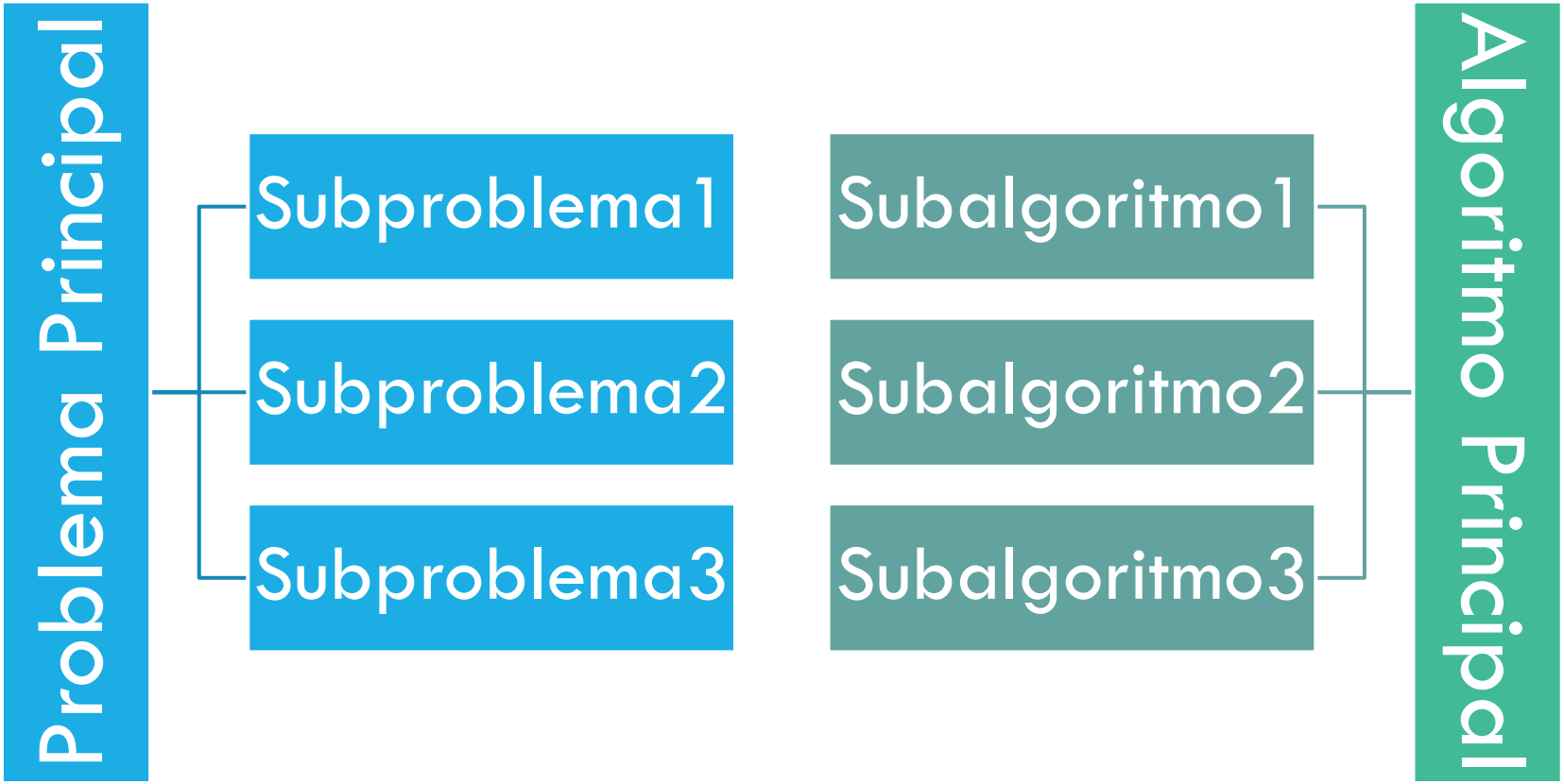
Ejemplo

- Sea el problema ***"cálculo de la superficie (área) de un rectángulo"***. Éste se puede dividir en tres subproblemas:
  - subproblema 1: entrada de datos de altura y de base.
  - subproblema 2: cálculo de la superficie.
  - subproblema 3: salida de resultados.

- El algoritmo que resuelve los tres subproblemas es:
  - leer (altura, base) //entrada de datos
  - area ← base * altura //cálculo de la superficie
  - escribir(base, altura, area) //salida de resultado



Ejemplo: Diseño descendente



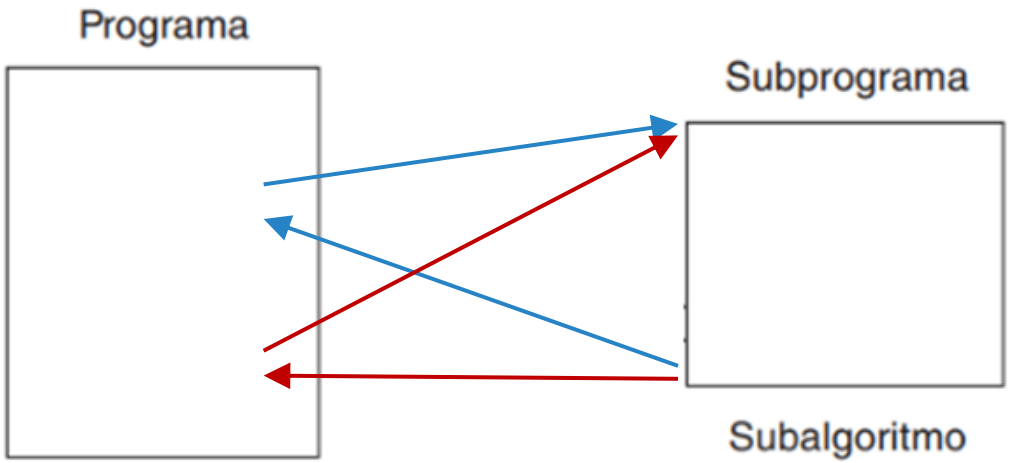
Programa y Subprogramas

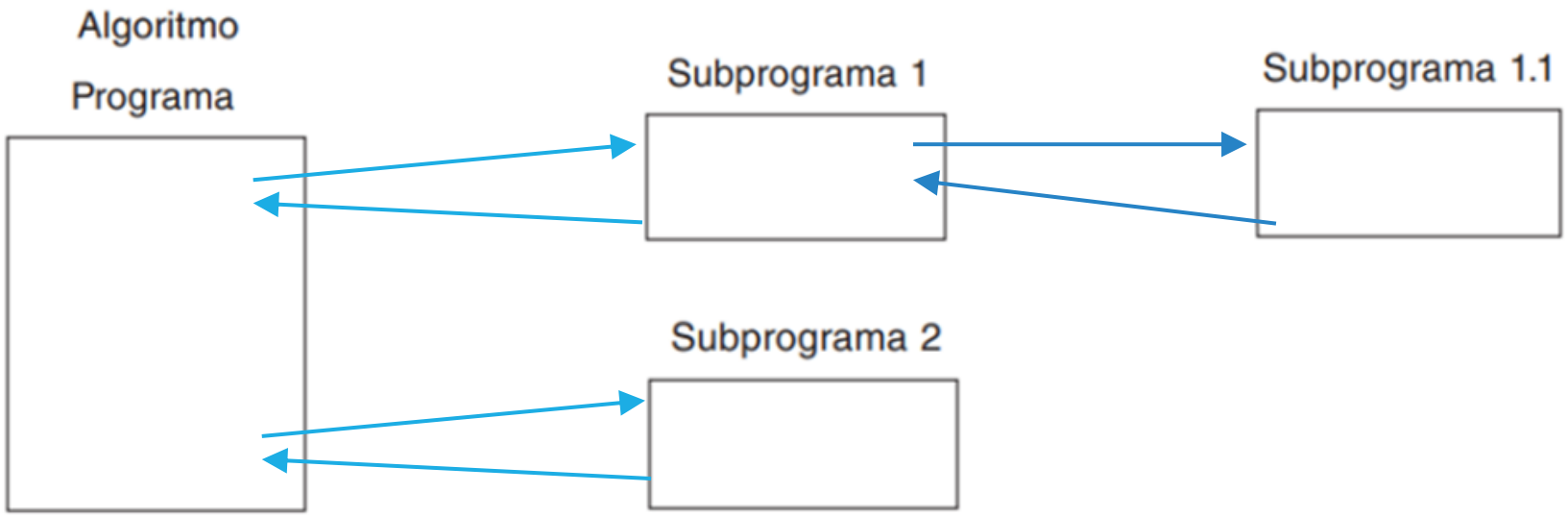
- El **problema principal** se soluciona por el correspondiente programa o algoritmo principal y la solución de los subproblemas mediante **subprogramas/subalgoritmos.**
- Un **subprograma** puede realizar las mismas acciones que un programa:
  - 1) aceptar datos
  - 2) realizar algunos cálculos y
  - 3) devolver resultados.

Programa y Subprogramas

- Un subprograma, **es utilizado por** el programa para un propósito específico.
- El subprograma **recibe** datos desde el programa y le **devuelve** resultados.
- El programa principal **llama o invoca** al subprograma.
- El subprograma ejecuta una tarea, y luego **devuelve** el control al programa. Esto puede suceder en diferentes lugares del programa.
- Cada vez que el subprograma es llamado, el control retorna al lugar desde donde fue hecha la llamada. Un subprograma puede llamar a su vez a sus propios subprogramas.
- Existen dos tipos importantes de subprogramas: ***funciones*** y ***procedimientos***

Programa y Subprogramas





Funciones, Métodos y Procedimientos

Funciones

- Una función es un conjunto de líneas de código que realiza una tarea específica y ***retorna un valor***

Métodos

- Los métodos y las funciones son funcionalmente idénticos, pero su diferencia radica en el contexto en el que existen.
- Un método también puede recibir valores, efectuar operaciones con estos y retornar valores, sin embargo, un método está asociado a un objeto, básicamente un método es una función que pertenece a un objeto o clase, mientras que una función existe por sí sola, sin necesidad de un objeto para ser usada.

Procedimientos

- Un procedimiento es un conjunto de instrucciones que se ejecutan sin retornar ningún valor. En el contexto de C++ un procedimiento es básicamente una función **void** que no nos obliga a utilizar una sentencia ***return***.

FUNCIONES

En C++

Funciones

- Todo programa en C/C++ se compone de una o más funciones, incluyendo la función **main**().

- El lenguaje C/C++ incluye muchas funciones de utilidad general (*funciones predefinidas*), las cuales están organizadas en **librerías.**

- También es posible definir ***nuevas funciones*** (*funciones definidas por el programador*)

Funciones predefinidas

- Son funciones que ya vienen incluidas en el lenguaje C++.
- Realizan tareas comunes y específicas.
- Están organizadas en bibliotecas (por ejemplo, cmath, cstring).
- Para utilizarlas debe incluirse la biblioteca correspondiente mediante la directiva **include**

Funciones predefinidas

- **Matemáticas:** sqrt(), pow(), sin(), cos(), tan(), log(), exp()
- **Cadenas:** strlen(), strcpy(), strcat(), strcmp()
- **Entrada/Salida:** cin, cout

FP-Ejemplo1.cpp

```cpp
#include <stdlib.h>
#include <iostream>
#include <time.h>

using namespace std;

int main ()
{
    // Se genera una semilla diferente cada vez (basada en el tiempo exacto de ejecución)
    // Se debe llamar siempre, para generar números realmente diferentes cada vez
    // La función time viene de la librería time.h
    srand (time(NULL));

    cout << ("Se va a generar un numero aleatorio ....\n");
    cout << ("El numero generado es : ");
    cout << rand(); //Se genera el número con rand y se muestra en pantalla
    return 0;
}

```

Ejemplo 1 de librerías en C++

En el siguiente ejemplo se observa el uso de la librería **stdlib.h** que posee una gran variedad de funcionalidades, para este ejemplo usaremos la función ***rand*** que nos permite generar un número aleatorio.

[*] FP-Ejemplo2.cpp

```cpp

#include <string.h>
#include <iostream>

using namespace std;

int main ()
{
    cout << ("Hola! Por favor ingrese su nombre ....\n");
    string cadena = "Hola "; //Se le da un valor inicial al string

    string nombre; //Esta cadena contendrá el nombre ingresado
    cin >> nombre; //Se lee el nombre

    cadena = cadena + nombre; //Se unen el saludo con el nombre usando el operador "+"
    cout << cadena; //Se muestra el resultado final.
    return 0;
}

```

Ejemplo 2 de librerías en C++

En el siguiente ejemplo se observa el uso de la librería **string.h** que nos permite básicamente crear y manipular muy fácilmente cadenas de caracteres.

Definición de funciones

- La primera línea de la definición recibe el nombre de ***encabezado*** y el resto, un bloque encerrado entre llaves, es el ***cuerpo*** de la función.

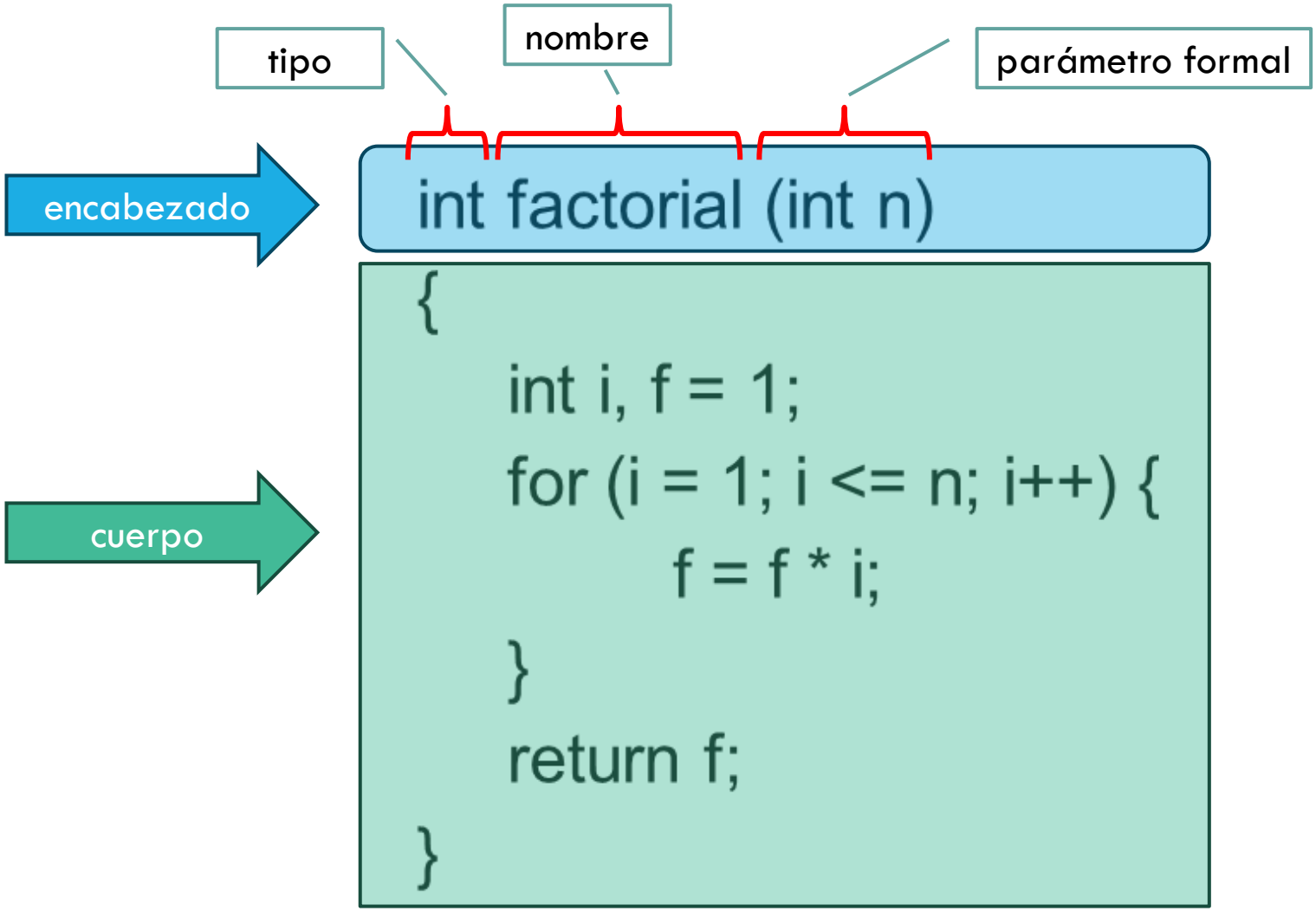
- La definición de una función se debe realizar en alguno de los archivos que forman parte del programa.

Definición de funciones

```
tipo nombre_funcion(tipo_1 arg_1, ..., tipo_n arg_n)
{
    sentencias;
    return expresion;  // optativo
}
```

- **tipo**: indica el tipo de valor (int, float, etc.) devuelto por la función.
- **nombre_funcion**: es el identificador usado para la función.
- la lista de argumentos es una secuencia de declaración de parámetros separados por comas y encerrados entre paréntesis. Son los llamados **parámetros formales** de la función.
- **return** expresion es un salto incondicional que permite evaluar una expresión y devolver su valor a la función llamante.

Ejemplo


tipo
nombre
parámetro formal
encabezado
cuerpo
int factorial (int n)
{
int i, f = 1;
for (i = 1; i <= n; i++) {
f = f * i;
}
return f;
}

Tipo de una función

- Una función devuelve, como mucho, un único valor a través de la sentencia **return**. Este valor tendrá el mismo tipo que el de la función.

- La palabra reservada **return** permite realizar un salto incondicional, de la misma forma que ***break***. En este caso, devuelve desde el interior de la función actual el control del programa a la función que la llamó.

- Una función puede tener varios puntos en los que se devuelve un valor con **return** y, lógicamente, finalizará en el primero que se ejecute.

- Al igual que con los ***break***, la legibilidad del código debe ser la decisión que determine que haya un único **return** (preferible) o varios.

Declaración de una función

- Al igual que ocurre con otros identificadores, en C++ no podemos llamar a una función en una sentencia sin que esté declarada **previamente.**
- Una declaración explícita de una función, también denominada **prototipo de la función,** tiene la expresión general:

```
tipo nombre_funcion(tipo_1, tipo_2, ..., tipo_n);
```

Llamada a la función

- La ***llamada a una función*** se hace especificando su nombre y, entre paréntesis (parámetros reales), las expresiones cuyos valores se van a enviar como argumentos de la función.

- Estos parámetros pueden ser identificadores o cualquier otra expresión válida.

- La llamada a una función se escribe de forma general como sigue:

```
salida = nombre_funcion(arg_1, arg_2, ..., arg_n);
```

Funciones con tipo. Ejemplo

[*] factorial.cpp

```
1   #include<iostream>
2
3   using namespace std;
4
5   int factorial(int n); //declaración de la función factorial
6
7   int main(){
8
9       int num,res;
10      cout<<"Ingrese un numero positivo entero: ";
11      cin>>num;
12      if (num>0){
13          res=factorial(num);//llamada a la función factorial
14          cout<<"El factorial de "<<num<<" es "<<res;
15      }
16  }
17  //definición de la función
18  int factorial(int n){
19      int i,f=1;
20      for(i=1;i<=n;i++) f=f*i;
21      return f;
22  }
23
```

Funciones void

- En ocasiones una función no necesita devolver ningún valor y, por tanto, no es obligatorio usar ***return***.

- Para ello, debemos definir la función con un tipo especial llamado **void**.

- Las funciones tipo void representan a los **procedimientos**

Función void. Ejemplo

mostrar.cpp

```
1   #include<iostream>
2
3   using namespace std;
4
5   void mostrar(int n); //declaración de la función mostrar
6
7   int main(){
8       int num;
9       cout<<"Ingrese un numero positivo entero: ";
10      cin>>num;
11      if (num>0) mostrar(num);//llamada a la función mostrar
12  }
13
14  //definición de la función
15  void mostrar(int n){
16      cout<<"El numero ingresado es "<<n;
17  }
18
```

Parámetros

- Cuando una función solicita los servicios de otra, debe facilitarle los datos que debe procesar. Esta información es enviada a la función a través de unos identificadores o expresiones denominados **argumentos** o **parámetros.**

- Los **argumentos** enviados en el momento de ser llamada la función se denominan **parámetros actuales o reales.**

- Los argumentos recibidos por la función se denominan **parámetros formales.**

- El paso de información se realiza estableciéndose un **emparejamiento posicional** entre los parámetros formales y los actuales.

Parámetros

- Existen dos alternativas para la transmisión de los parámetros a las funciones:

- **Paso por valor**
  - Los parámetros formales son **variables locales** a la función, es decir, solo accesibles en el ámbito de ésta. Reciben como valores iniciales los valores de los parámetros actuales. Posteriores modificaciones en la función de los parámetros formales, al ser locales, no afectan al valor de los parámetros actuales.

- **Paso por referencia**
  - Los parámetros formales no son variables locales a la función, sino **alias** de los propios parámetros actuales. **¡No se crea ninguna nueva variable!** Por tanto, cualquier modificación de los parámetros formales afectará a los actuales.

Parámetros por valor

- A los parámetros por valor solo se le antecede el **tipo de dato:**

```
void cambiar1(int z){
    z = 0;
}
```

- El parámetro **z** recibe una copia del parámetro real, cuando se realiza la llamada. Cuando se modifica z con el valor 0 el parámetro real no se modifica.

Parámetros por referencia

- En los parámetros por referencia se antecede al nombre del parámetro el carácter **&(ampersand)**:

```
void cambiar2(int &z) {
    z = 0;
}
```

- El parámetro **z** es un **alias** del parámetro real, cuando se realiza la llamada. Cuando se modifica z con el valor 0 se modifica el valor del parámetro real.

Parámetros por valor

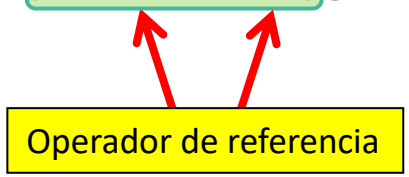
```
4
5  void intercambia(int a,int b){
6      int temp=a;
7      a=b;
8      b=temp;
9  }
10
11 int main(){
12     int num1=10,num2=20;
13     cout<<"Antes de llamar: num1="<<num1<<", num2="<<num2<<endl;
14     intercambia(num1,num2);
15     cout<<"Despues de la llamada: num1="<<num1<<", num2="<<num2<<endl;
16     return 0;
17 }
18
```

- Por lo general, las funciones reciben argumentos por valor. Esto significa que los parámetros son variables que solo existen dentro del contexto de la función.

Parámetros por referencia

- El ***paso por referencia*** asocia los parámetros de una función con variables que existen fuera del contexto de la función, lo cual permite modificar su valor.
- El paso por referencia es útil cuando se requiere que una función devuelva ***más de un resultado***.

```
4
5  void intercambia(int &a,int &b){
6      int temp=a;
7      a=b;
8      b=temp;
9  }
10
11 int main(){
12     int num1=10,num2=20;
13     cout<<"Antes de llamar: num1="<<num1<<", num2="<<num2<<endl;
14     intercambia(num1,num2);
15     cout<<"Despues de la llamada: num1="<<num1<<", num2="<<num2<<endl;
16     return 0;
17 }
18
```



Parámetros

Tener en cuenta:

- Una función o procedimiento pueden tener una ***cantidad*** cualquiera de parámetros, es decir pueden tener cero, uno, tres, diez, cien o más parámetros. Aunque habitualmente no suelen tener más de 4 o 5.

- Si una función tiene más de un parámetro cada uno de ellos debe ir separado por una ***coma***.

- Los parámetros de una función también tienen un ***tipo*** y un ***nombre*** que los identifica. El tipo del parámetro puede ser cualquiera y no tiene relación con el tipo de la función.

Práctica #1

a) Implemente una función f(x) que tome por parámetro dos números x, y (entero) y devuelva el resultado de x^y, no usar la función pow.

b) Implemente una función que verifique si un carácter introducido es un número o no.

Ninguna de las dos funciones anteriores debe pedir datos al usuario o imprimir información en la pantalla.

c) Dentro de la función main(), pida al usuario los valores que correspondan y llame a la función.

Variables Globales y Locales

- Una variable es **global** cuando el ámbito en el que dicha variable se conoce es el **programa completo**.

- Se consideran como **variables globales** aquellas que hayan sido declaradas en el **programa principal** y como **locales** las declaradas en el propio **subprograma**.

Variables Globales y Locales

- Toda variable que se utilice en un procedimiento debe ser **declarada** en el mismo. De esta forma todas las variables del procedimiento serán ***locales*** y la **comunicación** con el programa principal se realizará exclusivamente a través de los **parámetros.**

- Al declarar una variable en un procedimiento no importa que ya existiera otra con el mismo nombre en el programa principal; ambas serán ***distintas*** y, cuando nos encontremos en el procedimiento, sólo tendrá vigencia la declaración que se haya efectuado en él. Trabajando de esta forma se obtendrá la ***independencia*** de los módulos.

Alcance de variables y funciones

- El alcance de una variable o función es el ***contexto*** desde el cual puede accederse a la misma.

- En caso de ***ambigüedad*** (variables con mismo nombre, pero distinto alcance), el lenguaje C/C++ elegirá la variable declarada en el ***contexto más cercano***.

Ejemplo de alcance

¿Qué imprime este programa?

Alcance.cpp

```
1   #include<iostream>
2
3   using namespace std;
4
5   int x = 1;
6
7   int f() {
8       return x;
9   }
10
11  int main() {
12      int x = 2;
13      {
14          int x = 3;
15          cout << x << endl;
16      }
17      cout << x << endl;
18      cout << f() << endl;
19      return 0;
20  }
21
```

Práctica#2

- **Calculadora aritmética de enteros**
- **Descripción del problema**
- Implementar un programa que realice las siguientes tareas:
  1. Solicite dos operandos enteros al usuario
  2. Solicite el operador aritmético. Si el operador no es válido, se lanzará un mensaje de error y se volverá a solicitar hasta que el usuario introduzca un operador correcto.
  3. Mostrará por pantalla el resultado.
  4. Preguntará al usuario si desea realizar otro cálculo. En caso afirmativo se volverá al punto 1. En caso contrario, finalizará el programa.

FIN