Funciones en C++

Sitio: Centros - Cádiz
Curso: Programación
Libro: Funciones en C++

Imprimido por: Barroso López, Carlos
Día: martes, 21 de mayo de 2024, 23:26

Tabla de contenidos

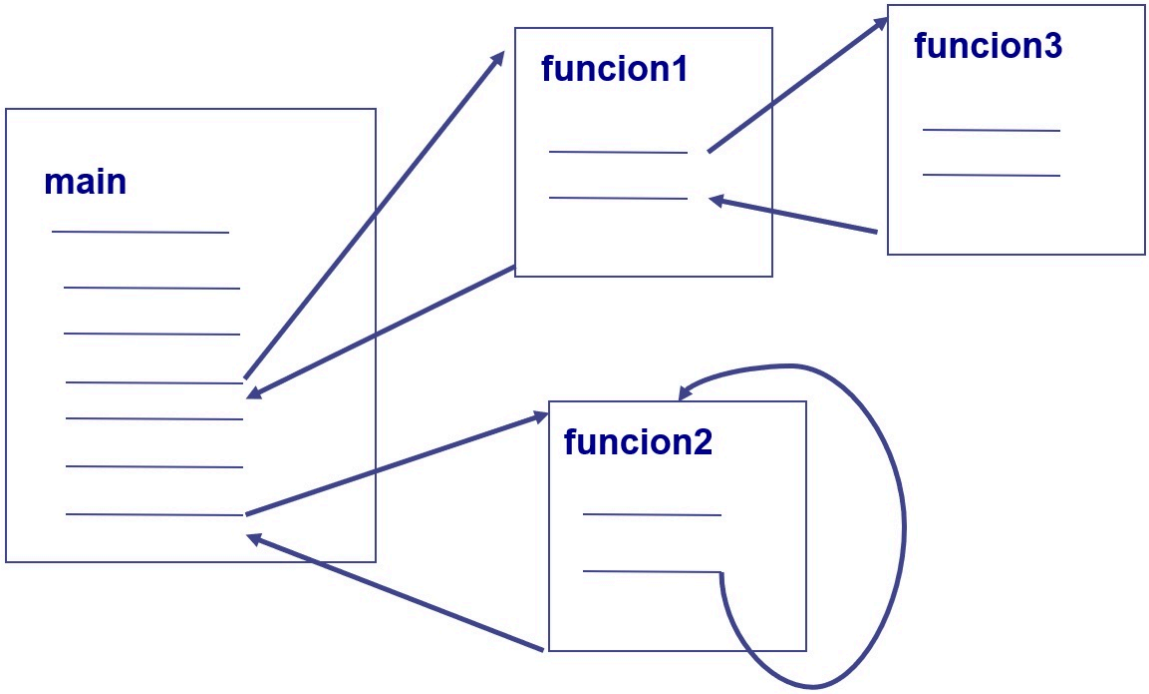
1. Programación modular

1. Definición

La **programación modular** se fundamenta en la resolución de problemas complejos mediante la división de éstos en subproblemas más pequeños que se solucionan independientemente. Si todavía resultan demasiado complejos se vuelven a dividir y así sucesivamente.

Este método de diseñar la solución de un problema, diseñando una jerarquía de módulos y buscando las soluciones de los subproblemas que lo integran es también conocido como **refinamiento descendente (top-down)**.

La codificación de los algoritmos que resuelven los subproblemas en lenguaje C++ se realiza mediante **funciones**.



2. Ventajas del uso de funciones

1. Claridad del código

Las funciones aportan **claridad lógica** al programa al descomponer éste en varias funciones concisas que resuelven partes concretas del problema global.

2. Reutilización del código

Muchos programas requieren la realización del mismo grupo de instrucciones desde distintas partes del programa. El uso de funciones **evita repetir líneas de código** de forma redundante.

Las funciones se pueden agrupar en **bibliotecas** (*libraries*), pudiéndose utilizar en cualquier otro programa que necesite hacer uso de alguna de ellas, permitiendo la **reutilización de código**.

3. Depuración del código

Facilita la depuración de errores. Una vez que estamos seguros de que una función hace su trabajo correctamente, podemos "olvidarnos" de ella a efectos de búsqueda de errores (**cajas negras**).

4. Trabajo en equipo

La subdivisión de un programa complejo en módulos más simples permite el desarrollo en equipo de un programa, repartiendo el trabajo.

3. Uso de bibliotecas

Para usar las funciones de una biblioteca, basta añadir al inicio del programa fuente el archivo de cabecera asociado a la biblioteca:

```
#include <cabecera_biblioteca>
```

Los **archivos de cabecera** son literalmente añadidos en el archivo fuente a través de la directiva `#include`. Esta metodología permite incorporar a las unidades de compilación, los archivos con extensión `*.cpp`, todos aquellos componentes de la biblioteca que permiten al compilador/enlazador generar el ejecutable.

2. Funciones en C++

Básicamente una **función o subrutina** puede realizar las mismas acciones que un programa:

- Recibir datos
- Realizar cálculos determinados
- Devolver resultados

Las funciones son **invocadas desde otras funciones**, con una excepción: la función `main()`, que tienen todos los programas en C++ y permite al compilador conocer donde está el punto inicial de un programa.

Para trabajar con funciones en C++ es necesario incluir para cada función:

- la **definición**
- la **declaración**
- la **llamada o invocación**

1. Definición de una función

Contiene el **código** que realiza las tareas para las que la función ha sido diseñada.

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

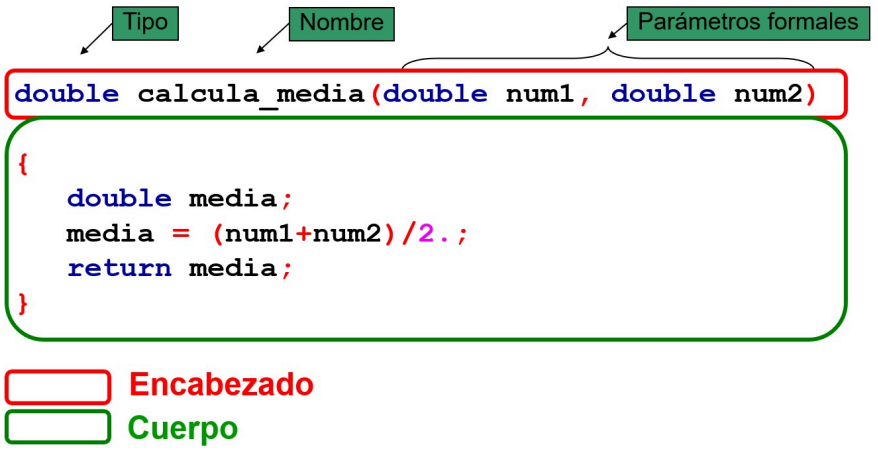
La primera línea de la definición recibe el nombre de **encabezamiento** y el resto, un bloque encerrado entre llaves, es el **cuerpo** de la función.

Elementos:

- `tipo`: indica el tipo de valor (`int`, `float`, etc.) devuelto por la función.
- `nombre_funcion`: es el identificador usado para la función.
- **Lista de argumentos**: secuencia de declaración de parámetros separados por comas y encerrados entre paréntesis, denominados **parámetros formales** de la función.
- `return expresion` es un **salto incondicional** que permite evaluar una expresión y devolver su valor a la función llamante. Este valor tendrá el mismo tipo que el de la función. También devuelve el control del programa a la función que la llamó (en el mismo punto de la llamada).

Ejemplo:

```
double calcula_media(double num1, double num2)
{
  double media;
  media = (num1 + num2)/2.;
  return media;
}
```



Uso de la sentencia *return*

Aunque es posible que una función tenga más de una sentencia *return* en su código, es recomendable que tenga sólo una al final del código, dado que es un salto incondicional y rompe la lógica estructurada del programa.

2. Declaración de una función

Al igual que ocurre con los identificadores de las variables, en C++ no se puede 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);
```

Ejemplo:

```
double calcula_media(double, double);
```

O con los identificadores de los parámetros (opcional):

```
double calcula_media(double x, double y);
```

La declaración permite al compilador realizar la reserva de memoria necesaria, y detectar posibles errores en tiempo de compilación (por ejemplo, errores en los tipos de datos pasados como parámetros).

3. Llamada o invocación a la función

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

Estos parámetros utilizados en la llamada, denominados **parámetros actuales**, pueden ser identificadores o cualquier otra expresión válida.

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

Ejemplo:

```
double resultado = calcula_media(numero1, numero2);
```

4. Declaración y definición simultáneas

C++ permite declarar y definir un función simultáneamente, si la definición se realiza antes de la función main().

Ejemplo completo:

```
// Cálculo de la media de dos números
#include <iostream>
using namespace std;

// Declaración y definición simultáneas
double calcula_media(double num1, double num2)
{
    double media;
    media = (num1+num2)/2.;
    return (media);  // return media es también válido;
}

int main()
{
    double numero1, numero2;
    cout << "Introduzca el primer número: ";
    cin >> numero1;
    cout << "Introduzca el segundo número: ";
    cin >> numero2;

    double resultado = calcula_media(numero1, numero2); // Llamada
    cout << "La media de "<< numero1 << " y " << numero2
        << " es " << resultado << ".\n";
}
```

5. Funciones void y sin argumentos

5.1. Funciones void

Son funciones que **no devuelven ningún valor**, y por tanto, no incluyen la sentencia return. Estas funciones se definen con el tipo especial void.

A este tipo de funciones se les denomina **procedimientos**.

Ejemplo:

```
void imprime_cadena(string cadena)
{
   cout << cadena;
}
```

5.2. Funciones sin argumentos

Funciones que no reciben ningún parámetro, por tanto, la lista de argumentos está vacía (paréntesis vacíos).

Ejemplo:

```
void imprime_mensaje_inicial()
{
   cout << "Este programa bla bla bla…\n";
}
```

El ejemplo anterior muestra una función sin argumentos de tipo void (no devuelve ningún valor).

6. Errores típicos

Función no declarada

```
error: use of undeclared identifier 'calcula_media'
```

o

```
error: 'calcula_media' was not declared in this scope
```

Función no definida

```
undefined reference to 'calcula_media(double, double)' error: linker command failed with exit code 1
```

En realidad, el compilador ha sido capaz de generar el código objeto del archivo. Es el **enlazador (linker)** el que ha dado la voz de alarma, no encuentra un archivo donde esté la definición de la función.

3. Ámbito de variables

En C++, las variables únicamente pueden ser utilizadas dentro de la estructura donde han sido declaradas, es lo que se conoce como **ámbito** de la variable.

Ejemplo:

```
for (int i=0; i<10; i++)
  cout << i;
```

"i" sólo es accesible dentro de la estructura for, si se intenta utilizar fuera dará error de compilación.

1. Variables locales

Son variables declaradas dentro de una estructura, y sólo son accesibles dentro de dicha estructura.

Ventajas:

i. **Uso eficiente de memoria**: una vez finalizada la ejecución de la estructura, la memoria reservada para las variables locales (a dicha estructura) se libera.

ii. Hace a las **funciones (subprogramas) independientes**: los identificadores declarados dentro de una función sólo afectan a dicha función, pudiéndose utilizar en cualquier otra parte del programa refiriéndose a variables distintas (ocupan posiciones de memoria diferentes).

Ejemplo:

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

double calcula_media(double num1, double num2)
{
  double media = (num1 + num2)/2.;
  return media;
}

int main()
{
  double num1, num2;
  cout << "Introduzca dos números reales: ";
  cin >> num1 >> num2;

  double resultado = calcula_media(num1, num2);

  cout << "La media es " << resultado << endl;
}
```

Las variables num1 y num2 de la función **son diferentes** a las variables num1 y num2 del main.

2. Variables globales

En C++, las variables declaradas fuera de la función main() y de cualquier otra función, **son accesibles desde cualquier parte del programa**, y se denominan variables globales.

2.1. Efectos secundarios

El uso de variables globales **conlleva un riesgo**, debido a esa deslocalización, una variable global puede ser modificada en cualquier parte del programa haciendo que dependa de ella, aumentando su complejidad.

Los efectos secundarios frecuentemente hacen que el comportamiento de un programa sea más difícil de predecir, o incluso causar efectos no deseados.

Sin embargo, en algunas ocasiones pueden resultar **útiles**, por ejemplo, se pueden usar para evitar tener que pasar variables usadas muy frecuentemente de forma continua entre diferentes subrutinas.

3. Conclusión

Es recomendable que toda comunicación entre el programa principal y las diferentes funciones se realice a través de parámetros, evitando el uso de variables globales.

4. Parámetros de una función

1. Parámetros formales y actuales

1.1. Parámetros formales

Son las variables que recibe la función, se crean en su definición. Su contenido lo recibe de los parámetro reales al realizar la llamada a la función.

Los parámetros formales son variables locales dentro de la función.

Ejemplo:

```
int CalcularMaximo(int num1,int num2) { ... }
```

1.2. Parámetros actuales o reales

Son la expresiones que se utilizan en la llamada de la función, y cuyos valores son enviados a los parámetros formales.

Ejemplos:

```
max = CalcularMaximo(5,6);
cout << CalcularMaximo(valor1,valor2);
```

2. Paso de parámetros

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

En C++ existen dos alternativas para la transmisión de los parámetros a las funciones:

2.1. Paso por valor

El parámetro formal recibe una **copia del valor** del parámetro actual.

Por tanto, cualquier modificación en la función del parámetro formal no afecta al valor del parámetro actual.

2.2. Paso por referencia

La **variable** pasada como parámetro actual es **compartida**, es decir, puede ser modificada por la función a través del parámetro formal.

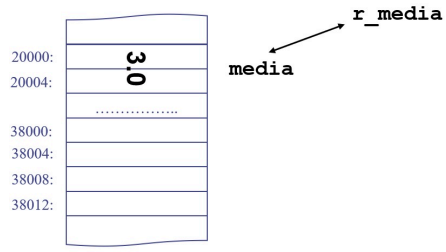
Para ello, C++ utiliza un tipo especial de variable, denominada **referencia,** que permite alterar el contenido de otra variable.

Referencias en C++

```
tipo_de_variable_referenciada& nombre_referencia{variable_referenciada};
```

Ejemplo:

```
double media = 2.;
double& r_media{media};
r_media = 3.;
```



Características:

- Las referencias **deben inicializarse** en el momento de su declaración, es decir, indicar a qué variable referencian.
- **No pueden modificarse**, es decir, cambiar la referencia a otra variable.
- A efectos prácticos, el compilador sustituye implícitamente la variable referencia por la variable referenciada.

Uso de referencias como parámetros formales

El uso de una referencia como parámetro formal de una función abre la puerta a poder modificar una variable local a una función desde una variable **alias** local a otra función. El fundamento es que ambas variables, la original y su referencia, están asociadas a la **misma dirección de memoria**.

```
tipo_funcion funcion(..., tipo_ref& r, ...);
```

En el momento en el que se produce la invocación a la función, el parámetro formal referencia se convierte en un alias de la variable real utilizada en la llamada (se inicializa).

Ejemplo:

```
#include <iostream>
using namespace std;
void incrementa(int& i) // Nótese el uso de una referencia
{
  i = i + 1;
  cout << "Valor incrementado en funcion: " << i << endl;
}
int main()
{
  int i = 5;
  incrementa(i);
  cout << "Valor incrementado en main: " << i << endl;
}
```

3. Parámetros de entrada y salida

3.1. Parámetros de entrada (E)

Permiten enviar datos desde el programa principal a la función.

En C++ se realiza mediante parámetros formales **por valor**.

3.2. Parámetros de salida (S)

Permiten recibir datos desde la función al programa principal.

En C++ se realiza mediante la **sentencia return**, si sólo se devuelve un valor, o mediante parámetros formales **por referencia (sin valor inicial)**, si es necesario devolver más de un valor.

3.3. Parámetros de entrada y salida (E/S)

Permiten enviar y recibir datos desde el programa principal y la función.

En C++ se realiza mediante parámetros formales **por referencia (con valor inicial)**.

Nota: El tipo de cada parámetro debe quedar bien especificado en la documentación de la función, así como en el modo que debe ser invocada.

5. Funciones recursivas

Un función recursiva es aquella que en su código **realiza una llamada o invocación a sí misma**.

Para evitar una secuencia de llamadas recursivas infinitas, el código debe incluir un **caso base**, el cual devuelve el control al programa principal.

Ejemplo:

```
#include <iostream>
using namespace std;
int CalcularFactorial(int num);  // Declaración

int main() {
        cout << "El factorial de 6 es " << CalcularFactorial(6) << endl;
        return 0;
}

int CalcularFactorial(int num)
{
        if (num==0 || num==1)  // Caso base
        {
                return 1;
        }
        return num * CalcularFactorial(num - 1);  // Llamada recursiva
}
```

6. Sobrecarga y parámetros opcionales

1. Sobrecarga de funciones

C++ permite especificar **más de una función con el mismo nombre** en el mismo ámbito. Estas funciones se denominan *funciones sobrecargadas*.

Las funciones sobrecargadas permiten proporcionar una semántica (comportamiento) diferente para una función, dependiendo de los tipos y el número de argumentos utilizados.

Nota: El tipo de dato devuelto no se tiene en cuenta para la sobrecarga.

Ejemplo:

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

void pinta() {
    cout << "Hola" << endl;
}
void pinta(string mensaje)
{
    cout << mensaje << endl;
}

main() {
    pinta();  // Escribe "Hola"
    pinta("Adiós");  // Escribe "Adiós"
}
```

El compilador selecciona la función sobrecargada que se va a invocar según la **mejor coincidencia** entre las declaraciones de la función y los argumentos proporcionados en la llamada.

Si no hay ningún candidato claro, la llamada genera un **error**.

2. Uso de parámetros opcionales

A veces puede interesar que ciertos parámetros que necesita una función no sea necesario proporcionarlos siempre en la llamada. Esto suele ocurrir cuando estos parámetros casi siempre se utilizan con un mismo valor.

C++ permite establecer como **opcionales** algunos o todos los parámetros definidos en la declaración de la función. En este caso, será necesario asignarles un **valor por defecto**.

En la llamada a la función, si se pasan valores a estos parámetros opcionales, se utilizarán dichos valores como cualquier otro parámetro. Por el contrario, si se omiten todos o algunos de estos parámetros opcionales, la función trabajará con los valores por defecto definidos.

Ejemplo:

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

void pinta(string ="Hola");  // Valor por defecto en la declaración
main()
{
    pinta();  // Escribe "Hola"
    pinta("Adiós");  // Escribe "Adiós"
}
void pinta(string mensaje) {  // Aquí NO debe ponerse el valor por defecto
    cout << mensaje << endl;
}
```

2.1. Limitaciones

- Sólo los **últimos parámetros** de la función pueden tener valores por defecto (ser opcionales).

```
void func(tipo, tipo, tipo=valor);  // Correcto
void func(tipo, tipo=valor, tipo);  // Error
```

- De estos, sólo los últimos pueden omitirse en la llamada.

```
void func(tipo, tipo=valor, tipo=valor);  // Función con 3 parámetros, 2 opcionales

...

// Posibles llamadas
func(par1);  // Con 1 parámetro (el primero)
func(par1, par2);  // Con 2 parámetros (el primero y el segundo)
func(par1, par2, par3);  // Con los 3 parámetros

func();  // Error
func(par1,  , par3);  // Error
```

- El valor por defecto debe definirse únicamente en la **declaración** de la función.

El valor por defecto debe definirse únicamente en la declaración de la función y NO en su definición, a menos que se haga la declaración y definición de forma conjunta.

Reiniciar tour para usuario en esta página