

Metodología de la Programación

Tema 3. Funciones (ampliación)

Departamento de Ciencias de la Computación e I.A.

Curso 2019-20



ugr

Universidad
de Granada





¡Esta es una licencia de Cultura
Libre!



Este obra cuyo autor es mgomez está bajo una
licencia de Reconocimiento-CompartirIgual 4.0
Internacional de Creative Commons.

1. Motivación y objetivos
2. La función main
3. La pila
4. Ámbito de un dato
5. Paso de parámetros y devolución
6. Uso de variables referencia
7. Parámetros y const
8. Parámetros con valor por defecto
9. Sobrecarga de funciones
10. Funciones inline
11. Variables locales static

El motivo de este tema radica en la necesidad de explicar algunos conceptos sobre funciones que no se vieron en la asignatura de **Fundamentos de la programación** ya que no se disponía entonces de conceptos que se han introducido en los temas previos de **Metodología de la programación**.

Se trata de **detalles** más que conceptos nuevos.

En concreto, se consideran los siguientes aspectos:

- la función **main**
- la pila
- concepto de ámbito de un dato
- resumen de formas de paso de parámetros
- uso de variables tipo referencia
- uso de **const** asociado a parámetros
- parámetros con valor por defecto
- sobrecarga de funciones
- funciones **inline**
- variables locales **static**

Al final del tema debéis ser capaces de:

- usar la función **main** de forma que un programa reciba y procese argumentos
- usar de forma adecuada las diferentes formas de paso de argumentos, optando siempre por la opción más adecuada
- estar familiarizados con el uso de variables tipo referencia
- saber determinar cuándo conviene usar **const** asociado a parámetros
- declarar funciones y métodos con parámetros con valor por defecto
- conocer las reglas que determinan en qué situaciones es posible usar sobrecarga
- determinar en qué situación conviene usar funciones **inline**
- comprender el sentido y la forma de manejo de variables locales **static**

1. Motivación y objetivos
- 2. La función main**
3. La pila
4. Ámbito de un dato
5. Paso de parámetros y devolución
6. Uso de variables referencia
7. Parámetros y const
8. Parámetros con valor por defecto
9. Sobrecarga de funciones
10. Funciones inline
11. Variables locales static

Ejecución de código en C++:

- un programa C++ comienza cuando el S.O. transfiere el control al método `main` y finaliza cuando este acaba
- hasta ahora, hemos usado la siguiente cabecera simple para `main` (que limita su funcionamiento):

```
int main()
```

Se observa que el tipo de retorno es `int`, aunque no lo hemos utilizado. ¿Para qué podría usarse?

- C++ ofrece una versión ampliada de la cabecera de `main` (que permite pasar argumentos al programa y hacer que su ejecución dependa de algún dato externo):

```
int main(int argc, char *argv[])
```

```
int main(int argc, char *argv[])
```

- valor de retorno: el entero devuelto por `main` informa al S.O. de posibles errores de ejecución del programa
 - 0: el programa terminó ok (valor por defecto)
 - otro valor: algún tipo de error

```
int main(int argc, char *argv[])
```

- los argumentos son:
 - `int argc`: número de argumentos usados al ejecutar el programa.
 - `char *argv[]`: array de cadenas con cada uno de los argumentos. Además:
 - `argv[0]`: nombre del ejecutable
 - `argv[1]`: primer argumento
 - ...

Gracias a este mecanismo podemos ejecutar programas especificando argumentos, como por ejemplo:

```
./comparaArchivos archivo1 archivo2
```

¿Cuál sería el valor de `argc`? ¿Cómo estarían almacenados los argumentos en `argv`?

Ejemplo de programa:

- mostrando la forma de uso del mecanismo de paso de argumentos a un ejecutable
- el programa recibe un determinado número de argumentos y los muestra
- en caso de error, informa al usuario sobre la forma de uso del programa
- se verá posteriormente otra versión en que los argumentos se trata como objetos de la clase **string** y no como **cadenas de caracteres estilo C**

La función main: Ejemplo

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
    if (argc<3){
        cerr << "Uso: "
            << " <Fichero1> <Fichero2> ..." << endl;
        return 1;
    }
    else{
        cout<<"Numero argumentos: " << argc << endl;
        for (int i=0; i<argc; ++i){
            cout<<argv[i]<<endl;
        }
    }
    return 0;
}
```

La función main

Si interesa pueden convertirse las cadenas estilo C al tipo `string` de forma directa (aquí se omite la comprobación sobre el número de argumentos):

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

int main(int argc, char *argv[]) {
    string par;
    cout<<"Argumentos: "<<endl;
    for (int i=0; i<argc; ++i) {
        par=argv[i];
        cout<<par<<endl;
    }
    return 0;
}
```

1. Motivación y objetivos
2. La función main
- 3. La pila**
4. Ámbito de un dato
5. Paso de parámetros y devolución
6. Uso de variables referencia
7. Parámetros y const
8. Parámetros con valor por defecto
9. Sobrecarga de funciones
10. Funciones inline
11. Variables locales static

La pila

La pila (**stack**) es una zona de memoria que almacena información sobre las **funciones activas** (en ejecución) de un programa.

¿Por qué el nombre de **pila**?

Al invocar una función (o método) ocurre lo siguiente:

- se crea en la pila un **entorno de programa** que almacena la información de la llamada:
 - dirección de memoria de retorno (dónde retomar la ejecución cuando finalice)
 - los parámetros formales
 - constantes y variables locales
 -
- al terminar la ejecución de la función se destruye su **entorno de programa** asociado

Ejecución de un programa en C++

- la ejecución de un programa en **C++** empieza creando un entorno de programa **en el fondo de la pila** para **main()**. Con respecto a esta función cabe destacar:
 1. es una función que debe aparecer en todo programa ejecutable escrito en C++
 2. presenta distintas versiones en cuanto a sus parámetros, como hemos visto
 3. devuelve un dato entero al sistema operativo
- un programa termina cuando se extrae de la pila el entorno de programa asociado a **main()**

Considerad la forma en que se comporta la pila al ejecutar el siguiente programa:

```
int main(){  
    int valor;  
    cout << "Introduce entero positivo: ";  
    cin >> valor;  
    imprimeFactorial(valor);  
}
```

Siendo el código de la función `imprimeFactorial`:

```
void imprimeFactorial(int n){  
    int result;  
    result=factorial(n);  
    cout << "Valor de factorial de " << n << " : "  
        << result;  
    cout << endl;  
}
```

Y la implementación de `factorial`:

```
int factorial(int n){  
    int i, valor=1;  
    for(i=2; i <= n; i++){  
        valor=valor*i;  
    }  
    return valor;  
}
```

Ejemplo

Al comenzarse la ejecución de **main**:

nombre de función	argumentos y variables locales
main	valor=?

En cuanto el usuario introduce un valor (imaginemos el valor 4):

nombre de función	argumentos y variables locales
main	valor=4

Se produce la llamada a `imprimeFactorial`

nombre de función	argumentos y variables locales
imprimeFactorial	n=4, result=?
main	valor=4

Desde `imprimeFactorial` se llama a `factorial`:

nombre de función	argumentos y variables locales
<code>factorial</code>	<code>n=4, i=?, valor=1</code>
<code>imprimeFactorial</code>	<code>n=4, result=?</code>
<code>main</code>	<code>valor=4</code>

Al final del bucle en **factorial**:

nombre de función	argumentos y variables locales
factorial	n=4, i=5, valor=24
imprimeFactorial	n=4, result=?
main	valor=4

Y **factorial** devuelve su valor a **imprimeFactorial**, con lo que se puede asignar valor a **result**:

nombre de función	argumentos y variables locales
imprimeFactorial	n=4, result=24
main	valor=4

Ejemplo

Con esto la función `imprimeFactorial` puede finalizar su trabajo, con lo que la pila quedaría con el espacio dedicado a `main`:

nombre de función	argumentos y variables locales
main	valor=4

A su vez pueden completarse todas las sentencias del programa principal, con lo que la pila quedaría vacía.

1. Motivación y objetivos
2. La función main
3. La pila
- 4. Ámbito de un dato**
5. Paso de parámetros y devolución
6. Uso de variables referencia
7. Parámetros y const
8. Parámetros con valor por defecto
9. Sobrecarga de funciones
10. Funciones inline
11. Variables locales static

El ámbito de un dato es el conjunto de todos aquellos módulos en que puede ser referenciado.
Importante: **está definido por las llaves que definen el bloque de código en que está declarado.**

La única excepción son los datos globales, que no deberían usarse.

Ámbito de un dato

¿Cuál es el ámbito de los datos que aparecen en esta función?

```
double f1(double x, double y){  
    double i, j;  
  
    for (double i=x; i<y; i++){  
        double z;  
        z=(i-x);  
        j=z/(y-x);  
        cout << j <<endl;  
    }  
}
```

- `x`, `y`, `i` (línea 2), `j`: visibles en todo el módulo
- `i` (línea 4), `z`: locales al cuerpo del bucle `for`

1. Motivación y objetivos
2. La función main
3. La pila
4. Ámbito de un dato
- 5. Paso de parámetros y devolución**
6. Uso de variables referencia
7. Parámetros y const
8. Parámetros con valor por defecto
9. Sobrecarga de funciones
10. Funciones inline
11. Variables locales static

Recordemos los diferentes tipos de paso de argumentos a usar en C++:

- **paso por valor**: es la forma de paso de argumentos por defecto (el parámetro actual se copia sobre el parámetro formal)
- **paso por referencia**: no hay copia; el argumento formal es un alias del parámetro actual
- **paso por puntero**: el argumento formal es una dirección de memoria que apuntará a la ubicación del parámetro actual

La información que aporta un argumento en el prototipo de la función nos indica para qué se usa:

- si se usa para obtener la solución, entonces hablamos de **parámetro de entrada**
- si se usa para almacenar la solución o parte de ella, se habla de **parámetro de salida**
- si se utiliza tanto para obtener la solución como para almacenarla y devolverla (de la función o método), entonces se denomina **parámetro de entrada - salida**

Ejemplo 1: determinar si un número es primo:

```
// argumento de entrada  
bool esPrimo(int);
```

Ejemplo 2: calcular el número de primos existentes en un intervalo de valores:

```
// argumentos de entrada  
int numeroPrimos(int, int);
```

Ejemplo 3: calcular el máximo y el mínimo de una secuencia de valores reales introducidos por la entrada estándar (los valores se leen dentro de la función):

```
// argumentos de salida  
void calcularMaximoYMinimo(double &, double &);
```

Ejemplo 4: cálculo de la suma de dos números complejos, representados como dos número reales:

```
// argumentos de salida rres y cres  
void sumarComplejos(double r1, double c1,  
                    double r2, double c2,  
                    double &rres, double &cres);
```

Ejemplo 5: cálculo de la derivada de un polinomio de grado 3:

```
// argumentos de salida ad, bd y cd  
void calcularDerivada(double a, double b,  
                     double c, double d,  
                     double &ad, double &bd,  
                     double &cd);
```

Cuando el paso de parámetros es por valor, el argumento actual puede ser una expresión, una constante o una variable.

Sin embargo, cuando el paso de parámetros es por referencia o puntero, el argumento actual debe ser obligatoriamente una variable.

1. Motivación y objetivos
2. La función main
3. La pila
4. Ámbito de un dato
5. Paso de parámetros y devolución
- 6. Uso de variables referencia**
7. Parámetros y const
8. Parámetros con valor por defecto
9. Sobrecarga de funciones
10. Funciones inline
11. Variables locales static

Se trata de un alias de otro dato u objeto. Normalmente se utiliza para el paso por referencia y en el valor de retorno de las funciones que devuelven por referencia. Sin embargo, podría usarse en otros ámbitos.

La sintaxis de declaración de una referencia tiene la siguiente forma:

```
tipo & identificador = iniciador;
```

Ejemplo 1:

```
int a=0;  
int &ref=a;  
ref=5;  
cout<<a<<endl;
```

Ejemplo 2:

```
int v[5]={1,2,3,4,5};  
int &ref=v[3];  
ref=0;  
cout<<v[3]<<endl;
```

Devolución por referencia

Una función puede devolver una referencia a un dato u objeto:

```
int& valor(int *v, int i){  
    return v[i];  
}
```

La referencia puede usarse en ambos lados de `=`

```
int main(){  
    int v[]={3,5,2,7,6};  
    int a=valor(v,3);  
    valor(v,3)=0;  
}
```

Devolución por referencia

Error típico: devolución de referencias a datos locales a una función (los datos locales se destruyen al terminar la función):

```
#include <iostream>
using namespace std;
int& funcion() {
    int x=3;
    return x; // Error: devolucion referencia a variable local
}

int main() {
    int y=funcion();
    cout << y << endl;
}
```

1. Motivación y objetivos
2. La función main
3. La pila
4. Ámbito de un dato
5. Paso de parámetros y devolución
6. Uso de variables referencia
- 7. Parámetros y const**
8. Parámetros con valor por defecto
9. Sobrecarga de funciones
10. Funciones inline
11. Variables locales static

Parámetros y const I

En el paso de parámetros (de cualquier tipo) podemos usar `const` para evitar que una función modifique el argumento.

```
int funcion1(const int a){
    a=3; // Error, a es const
    return a;
}

void funcion2(const int v[], int utilv){
    for(int i=0; i<utilv;++i){
        v[i]=0; // Error, v es const
    }
}

void funcion3(const int *v){
    *v=8; // Error, *v es const
}
```


Parámetros y const I

Una de las razones del uso de `const` con referencias consiste en querer tratar el dato como **entrada** únicamente (sin que pueda modificarse), pero evitando la copia del argumento (por razones de eficiencia).

```
struct Gigante{
    double x, y, z;
    string c1, c2, c3;
    int a, b, c;
    ...
};

void funcion(const Gigante &g){
    g.x=3.5; // Error: g es const
}
```

Parámetros y const I

Cuando una función devuelve una referencia, podemos hacer que ésta sea **const**.

```
const int &valor(const int *v, int i){  
    return v[i];  
}
```

```
int main(){  
    int v[]={0, 1, 2, 3, 4, 5};  
    v[2]=3*5; // Correcto  
    valor(v,2)=3*5; // Error, pues la referencia es const  
    int res=valor(v,2)*3; // Correcto  
}
```

Lo mismo ocurre cuando una función devuelve un puntero: podemos hacer que éste sea const.

```
const int *valor(int *v, int i){
    return v+i;
}

int main(){
    int v[]={0, 1, 2, 3, 4, 5};
    v[2]=3*5; // Correcto
    *(valor(v,2))=3*5; // Error, pues el puntero devuelto es const
    int res=*(valor(v,2))*3; // Correcto
}
```

1. Motivación y objetivos
2. La función main
3. La pila
4. Ámbito de un dato
5. Paso de parámetros y devolución
6. Uso de variables referencia
7. Parámetros y const
- 8. Parámetros con valor por defecto**
9. Sobrecarga de funciones
10. Funciones inline
11. Variables locales static

Parámetros con valor por defecto

Las funciones pueden tener parámetros con un valor por defecto. Su uso implica la siguiente restricción:

- deben aparecer en primer lugar aquellos argumentos que no toman valor por defecto

```
void funcion(char c, int i=7){  
    ...  
}  
int main(){  
    funcion('a',8);  
    funcion('z');  
}
```

Parámetros con valor por defecto: Ejemplo

```
#include <iostream>
using namespace std;
int volumenCaja(int largo=1, int ancho=1, int alto=1);

int main() {
    cout << "Volumen por defecto: " << volumenCaja() << endl;
    cout << "El volumen de una caja (10,1,1) es: " << volumenCaja(10) << endl;
    cout << "El volumen de una caja (10,5,1) es: " << volumenCaja(10,5) << endl;
    cout << "El volumen de una caja (10,5,2) es: " << volumenCaja(10,5,2) << endl;
    return 0;
}

int volumenCaja( int largo, int ancho, int alto ) {
    return largo * ancho * alto;
}
```

1. Motivación y objetivos
2. La función main
3. La pila
4. Ámbito de un dato
5. Paso de parámetros y devolución
6. Uso de variables referencia
7. Parámetros y const
8. Parámetros con valor por defecto
- 9. Sobrecarga de funciones**
10. Funciones inline
11. Variables locales static

Sobrecarga de funciones

C++ permite definir varias funciones en el mismo ámbito con el mismo nombre. C++ selecciona la función adecuada en base al número, tipo y orden de los argumentos.

```
void funcion(int x){  
    ...  
}  
void funcion(double x){  
    ...  
}  
void funcion(char *c){  
    ...  
}  
void funcion(int x, double y){  
    ...  
}
```

```
int main(){  
    char *c;  
    funcion(3);  
    funcion(4.5);  
    funcion(4,9.3);  
    funcion(c);  
}
```


C++ puede aplicar conversión implícita de tipos para buscar la función adecuada.

```
void funcion(double x){
    cout << "double" << x << endl;
}
void funcion(char *p){
    cout << "char *" << *p << endl;
}
int main(){
    funcion(4.5);
    funcion(3); // conversión implícita
}
```

El uso de la sobrecarga está regido por una serie de normas que debemos tener en cuenta y que iremos viendo a través de diferentes ejemplos.

Consideramos en primer lugar situaciones no admitidas:

- la única diferencia entre dos funciones de igual nombre es el tipo de retorno
- la única diferencia entre dos funciones de igual nombre consiste en que un parámetro por valor es constante en una versión y no en la otra

Única diferencia: tipo devuelto

```
int funcion(int x){  
    return x*2;  
}  
  
double funcion(int x){  
    return x/3.0;  
}  
  
int main(){  
    int x=funcion(3);  
    double f=funcion(5);  
}
```

cod.cpp: En la función 'double funcion(int)':

cod.cpp:4:8: error: ambiguating new declaration of 'double funcion(int)'

```
double funcion(int x){
```

```
~~~~~
```

codigo.cpp:1:5: nota: old declaration 'int funcion(int)'

```
int funcion(int x){
```

```
~~~~~
```

Sobrecarga de funciones

Única diferencia: un parámetro por valor es `const` en una versión y en la otra no

```
#include <iostream>
using namespace std;
void funcion(double x){
    cout << "funcion(double x): " << x <<endl;
}
void funcion(const double x){
    cout << "funcion(const double x): " << x <<endl;
}
int main(){
    double x=2;
    const double A=4.5;
    funcion(A);
    funcion(x);
}
```

cod.cpp: En la función 'void funcion(double)':

cod.cpp:6:6: error: redefinición de 'void funcion(double)'

```
void funcion(const double x){
```

```
~~~~~
```

cod.cpp:3:6: nota: 'void funcion(double)' previously defined here

```
void funcion(double x){
```

```
~~~~~
```

También puede haber problemas de ambigüedad en el caso de parámetros por defecto:

```
void funcion(char c, int i=7){  
    ...  
}  
void funcion(char c){  
    ...  
}  
int main(){  
    funcion('a',8);  
    funcion('z');  
}
```

```
sobrecarga10.cpp: En la función 'int main()':  
sobrecarga10.cpp:9:16: error: la llamada del 'funcion(char)' sobrecargado es ambigua  
    funcion('z');  
      ^  
sobrecarga10.cpp:1:6: nota: candidate: void funcion(char, int)  
void funcion(char c, int i=7){  
    ~~~~~~  
sobrecarga10.cpp:4:6: nota: candidate: void funcion(char)  
void funcion(char c){  
    ~~~~~~
```


Sobrecarga de funciones

También puede haber ambigüedad en situaciones en que la conversión de tipos puede optar por varias alternativas:

```
void funcion(int a, int b){
    ...
}
void funcion(double a, double b){
    ...
}
int main(){
    funcion(2,4);
    funcion(3.5,4.2);
    funcion(2,4.2); //Ambiguo
    funcion(3.5,4); //Ambiguo
    funcion(3.5,static_cast<double>(4));
}
```

Sobrecarga de funciones

```
sobrecarga6.cpp: En la función 'int main()':
sobrecarga6.cpp:17:18: error: la llamada del 'funcion(int, double)' sobrecargado es ambigua
    funcion(2,4.2); //Ambiguo
    ^
sobrecarga6.cpp:6:6: nota: candidate: void funcion(int, int)
void funcion(int a, int b){
    ~~~~~
sobrecarga6.cpp:10:6: nota: candidate: void funcion(double, double)
void funcion(double a, double b){
    ~~~~~
sobrecarga6.cpp:18:18: error: la llamada del 'funcion(double, int)' sobrecargado es ambigua
    funcion(3.5,4); //Ambiguo
    ^
sobrecarga6.cpp:6:6: nota: candidate: void funcion(int, int)
void funcion(int a, int b){
    ~~~~~
sobrecarga6.cpp:10:6: nota: candidate: void funcion(double, double)
void funcion(double a, double b){
    ~~~~~
```

Y consideramos ahora situaciones en que sí es posible la sobrecarga: la única diferencia radica en que un argumento tipo puntero o referencia aparece con **const** en una versión. Se verá un ejemplo de código con referencias y con punteros.

Sobrecarga de funciones

Diferencia: uso de const en parámetro por referencia

```
#include <iostream>
using namespace std;
void funcion(double &x){
    cout << "funcion(double &x): " << x <<endl;
}
void funcion(const double &x){
    cout << "funcion(const double &x): " << x <<endl;
}
int main(){
    double x=2;
    const double A=4.5;
    funcion(A);
    funcion(x);
}
```

Sobrecarga de funciones

Diferencia: uso de const en parámetro tipo puntero

```
#include <iostream>
using namespace std;
void funcion(double *p){
    cout << "funcion(double *p): " << *p << endl;
}
void funcion(const double *p){
    cout << "funcion(const double *p): " << *p << endl;
}
int main(){
    double x=2;
    const double A=4.5;
    funcion(&A);
    funcion(&x);
}
```

1. Motivación y objetivos
2. La función main
3. La pila
4. Ámbito de un dato
5. Paso de parámetros y devolución
6. Uso de variables referencia
7. Parámetros y const
8. Parámetros con valor por defecto
9. Sobrecarga de funciones
- 10. Funciones inline**
11. Variables locales static

Es una forma de declarar una función para que el compilador genere una copia de su código cada vez que es llamada (evita el uso de la pila y la ejecución es más rápida). A considerar:

- recomendado sólo funciones pequeñas y frecuentemente usadas
- ejecución más rápida en general, pero código generado de mayor tamaño

El uso práctico de `inline` debe hacerse teniendo en cuenta que:

- el compilador podría no hacer caso al calificador `inline`
- la definición se hace indicando `inline` antes del tipo de retorno
- el uso del calificador suele hacerse en archivos (`.h`), ya que el compilador necesita su definición para expandir las llamadas

Funciones inline: Ejemplo

```
#include <iostream>

inline bool numeroPar(const int n){
    return (n%2==0);
}

int main(){
    std::string parimpar;
    parimpar=numeroPar(25)?"par":"impar";
    std::cout<<"Es 25 par?: " << parimpar;
}
```

1. Motivación y objetivos
2. La función main
3. La pila
4. Ámbito de un dato
5. Paso de parámetros y devolución
6. Uso de variables referencia
7. Parámetros y const
8. Parámetros con valor por defecto
9. Sobrecarga de funciones
10. Funciones inline
- 11. Variables locales static**

Es una variable local de una función o método **que no se destruye al acabar la función y que mantendrá su valor entre llamadas:**

- se inicializan la primera vez que se llama a la función
- conserva el valor de una llamada hasta la siguiente
- es obligatorio asignarles un valor en su declaración

Variables locales static I

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

double cuadrado(double numero){
    static int contadorLlamadas=1;
    cout << "Llamadas a cuadrado: " << contadorLlamadas << endl;
    contadorLlamadas++;
    return numero*numero;
}

int main(){
    for(int i=0; i<5; ++i) {
        cout << i << "^2 = " << cuadrado(i) << endl;
```

```
}  
}
```

Salida obtenida:

Llamadas a `cuadrado`: 1

$0^2 = 0$

Llamadas a `cuadrado`: 2

$1^2 = 1$

Llamadas a `cuadrado`: 3

$2^2 = 4$

Llamadas a `cuadrado`: 4

$3^2 = 9$

Llamadas a `cuadrado`: 5

$4^2 = 16$