

Metodología de la Programación

Tema 2. Funciones (ampliación)

Departamento de Ciencias de la Computación e I.A.

Curso 2016-17





¡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.

Índice

1. Introducción
2. La función main
3. La pila
4. Ámbito de un dato
5. Referencias
6. Parámetros y const
7. Parámetros con valor por defecto
8. Sobrecarga de funciones
9. Funciones inline
10. Variables locales static

Declaración/Definición

La declaración de una función sirve al compilador para conocer:

1. su tipo (tipo de dato devuelto, o *void* si no devuelve nada),
2. su nombre
3. para cada argumento, su tipo.

La declaración de una función termina con un punto y coma.

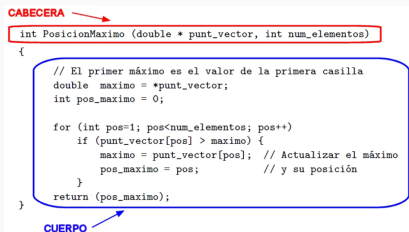
```
int PosicionMaximo(int *, int );
```

- Opcionalmente puede escribirse el nombre de cada argumento, el compilador no lo tiene en cuenta.
- No es obligatoria la declaración.

Declaración/Definición

En la definición de una función se especifica:

- La cabecera de la función. Obligatorio el nombre de los argumentos.
- Las instrucciones que va a ejecutar la función (cuerpo de la función).



The diagram shows a C++ function definition for `PosicionMaximo`. The function signature `int PosicionMaximo (double * punt_vector, int num_elementos)` is enclosed in a red box and labeled "CABECERA" with a red arrow. The function body is enclosed in a blue rounded rectangle and labeled "CUERPO" with a blue arrow. The body contains comments and code for finding the maximum value and its position in an array.

```
int PosicionMaximo (double * punt_vector, int num_elementos)
{
    // El primer máximo es el valor de la primera casilla
    double maximo = *punt_vector;
    int pos_maximo = 0;

    for (int pos=1; pos<num_elementos; pos++)
        if (punt_vector[pos] > maximo) {
            maximo = punt_vector[pos]; // Actualizar el máximo
            pos_maximo = pos;          // y su posición
        }
    return (pos_maximo);
}
```

Los parametros indicados en la cabecera de la función se conocen como parámetros formales.

Solo se conocen dentro de la función, y pueden usarse únicamente en el cuerpo de la función.

Declaración/Definición

- Una función podrá acceder a los datos locales y parametros formales especificados en su cabecera.
- Parámetros formales se inicializan con los valores indicados en la llamada a la función.
- Parámetros actuales son las expresiones que se emplean en la llamada a una función para inicializar los parametros formales.
- Parámetros actuales deben coincidir en número, tipo y orden con los parámetros formales.

```
int vector[TAM];  
int utilizados;  
.....  
int pos_max = PosicionMaximo(vector, utilizados);  
cout << "Maximo: " << vector[pos_max] << endl;
```

En la llamada a la función, *vector* y *utilizados* son los parámetros actuales.

Declaración/Definición

El ámbito de una función incluye:

- Todas las funciones que están definidas por debajo de su declaración,
- hasta el final del fichero.

Dos opciones:

1) Definir la función antes de usarse.

```
int PosicionMaximo(int *puntVector, int num_elementos){  
.....  
}  
  
int main (void){  
....PosicionMaximo(vector, utilizados);  
}
```

Fuerza a que el *main* sea la última función.

2) Declarar la función antes de usarse.

```
int PosicionMaximo(int *, int);

int main (void){
....PosicionMaximo(vector, utilizados);
}

int PosicionMaximo(int *puntVector, int num_elementos){
.....
}
```


- El orden en el que se definen las funciones no es importante.
- Permite escribir la función *main* primero.
- Permite agrupar las declaraciones en un fichero de cabecera:
 - Extensión .h
 - Sustituir las declaraciones por una línea *#include*
- Las definiciones de las funciones pueden agruparse:
 - Extensión .cpp
 - Añadir *#include*
 - Compilarse independientemente

- Paso por defecto
- Parámetro actual es un literal
- Parámetro formal es una copia del actual
- Si el parámetro actual es variable puntero
 - Se copia su contenido, dirección de memoria
 - Parámetro formal es otro puntero
 - Su valor no se modifica dentro de la función
 - Sin embargo, se puede modificar a lo que hace referencia

Índice

1. Introducción
2. La función main
3. La pila
4. Ámbito de un dato
5. Referencias
6. Parámetros y const
7. Parámetros con valor por defecto
8. Sobrecarga de funciones
9. Funciones inline
10. 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()
```

- 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[])
```

La función main

En esta declaración:

```
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
- 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
 - ...

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:

```
#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;
}
```

Índice

1. Introducción
2. La función main
- 3. La pila**
4. Ámbito de un dato
5. Referencias
6. Parámetros y const
7. Parámetros con valor por defecto
8. Sobrecarga de funciones
9. Funciones inline
10. Variables locales static

La pila (**stack**) es una zona de memoria que almacena información sobre las **funciones activas** (en ejecución) de un programa. Al invocar una función ocurre lo siguiente:

- Se crea en la pila un **entorno de programa** que almacena la información de la misma:
 - dirección de memoria de retorno (dónde retomar la ejecución cuando finalice)
 - constantes y variables locales
 - los parámetros formales
 -
- 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.

```
int main()
```

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

3. devuelve un dato entero al sistema operativo
- un programa termina cuando se extrae de la pila el entorno de programa asociado a **main()**.

Ejemplo

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

Ejemplo

Se produce la llamada a `imprimeFactorial`

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

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>

Ejemplo

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.

Índice

1. Introducción
2. La función main
3. La pila
- 4. Ámbito de un dato**
5. Referencias
6. Parámetros y const
7. Parámetros con valor por defecto
8. Sobrecarga de funciones
9. Funciones inline
10. Variables locales static

Ámbito de un dato

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

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` son globales a todo el módulo
- `i` (línea 4), `z` son locales al cuerpo del bucle `for`

Índice

1. Introducción
2. La función main
3. La pila
4. Ámbito de un dato
- 5. Referencias**
6. Parámetros y const
7. Parámetros con valor por defecto
8. Sobrecarga de funciones
9. Funciones inline
10. Variables locales static

Se trata de un alias (segundo nombre) para una variable declarada previamente.

La sintaxis de declaración tiene la siguiente forma:

```
tipo &alias = tipo de objeto;
```

La asociación de una referencia a un dato se especifica mediante el operador `=` y es obligatoria ya que no pueden definirse referencias sin inicializar.

Ejemplo:

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


La referencia es otra etiqueta asociada a una misma dirección de memoria. Podemos usar cualquiera de las dos etiquetas para acceder a dicha dirección de memoria.

Ejemplo:

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

Importante:

- Las variables de referencia deben ser inicializadas cuando se crean, antes de usarse.
- Hablamos de **asociación**, no de asignación.
- Una variable de referencia puede inicializarse a un valor constante.

```
const int CTE = 100;  
const int &ref_CTE = CTE;
```

Internamente una referencia se comporta como un puntero constante a un objeto, pero **no es un puntero**.

Diferencias fundamentales entre referencias y punteros:

- No existen referencias nulas. Una referencias siempre está asociada a un objeto.
- Una vez que una referencia es asociada a un objeto, ésta no puede ser asociada a otro objeto.
- Una referencia debe ser inicializada cuando es creada.

El uso habitual de las referencias:

- Paso de argumentos a funciones
- Devolución de referencias en las funciones

En estos dos casos, no se inicializan las variables de referencia.

Paso de argumentos por referencia

Cualquier modificación que se produzca sobre la referencia se efectúa realmente sobre la variable original.

Sintaxis para definir un parámetro formal por referencia:

```
tipo &alias
```

- No se inicializan las variables referencia
- Las asociaciones son implícitas, se producen cuando se llama a la función.

Paso de argumentos por referencia

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

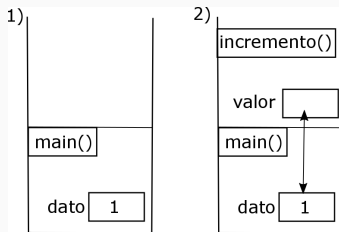
void Incremento (int &);

int main(){
    int dato = 1;
    Incremento(dato);
    cout << dato; //2
}

void Incremento(int &valor){
    valor++;
}
```

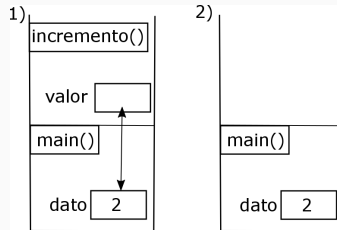
Paso de argumentos por referencia

- En la declaración: `void Incremento(int &);`
Se informa al compilador que sólo debe admitir el nombre de una variable *int*
- En la llamada: `Incremento(dato);`
Se indica el parámetro actual que se va a asociar a la referencia *valor*
- En la definición: `void Incremento(int &valor)`
Se da un nombre a la referencia *valor* y se asocia a la variable *dato*



Paso de argumentos por referencia

Los cambios que realice el cuerpo de la función sobre el parámetro formal (la referencia valor) afecta a la variable asociada (dato).

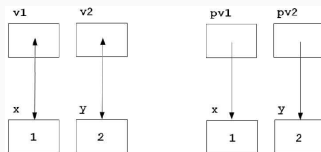


Paso de argumentos por referencia

```
// Paso por referencia real      // paso por referencia |"simulado"

void swap (int& v1, int& v2)    void swap (int * pv1, int * pv2)
{
    int tmp = v1;               {
    v1 = v2;                    int tmp = *pv1;
    v2 = tmp;                   *pv1 = *pv2;
                                *pv2 = tmp;
}                                }

int main()                     int main()
{
    int x, y;                   {
    ...                          int x, y;
    swap(x, y);                  ...
    ...                          swap(&x, &y);
}                                ...
}
```



Paso de argumentos por referencia

Las referencias son preferibles a los punteros siempre que no necesitemos reasignar la dirección del objeto, o cuando exista la posibilidad de introducir o devolver la dirección nula.

Paso de argumentos por referencia

Usaremos el paso por referencia en vez de por valor:

- Cuando queramos modificar el dato en la función
- Cuando el dato sea muy grande
 - Optimiza espacio en la pila
 - Reduce tiempo de ejecución

Una función puede devolver una referencia:

```
int& function(int valor);
```

Dos razones para devolver una referencia a un objeto:

- Si el objeto es grande, es preferible no usar devolución por valor, sino devolver una referencia.
- Cuando el tipo devuelto es una referencia puede ser usado como un l-value.

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 el lado derecho de una asignación.

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

Pero también en el lado izquierdo de la asignación.

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

Devolución por referencia

```
#include <iostream>
using namespace std;
int f1(int v[],int i){
    return v[i];
}
int& f2(int v[], int i){
    return v[i];
}

int main(){

    int v[]={3,44,35,70};

    cout << f1(v,2) << " " << f2(v,2) << endl;
    f2(v,2)=100;
    cout << f1(v,2) << " " << f2(v,2) << endl;
}
```

Devolución por referencia

La devolución de referencias a datos locales a una función es un error típico: 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;
}
```

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 obtenerla, entonces se denomina **parámetro de entrada - salida**

Ejemplos:

- determinar si un número es primo:

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

- calcular el número de primos existentes en un intervalo de valores:

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


- 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 &);
```

Ejemplos:

- cálculo de la suma de dos números complejos, representados como dos números reales:

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

- 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, el argumento actual debe ser obligatoriamente una variable.

Índice

1. Introducción
2. La función main
3. La pila
4. Ámbito de un dato
5. Referencias
- 6. Parámetros y const**
7. Parámetros con valor por defecto
8. Sobrecarga de funciones
9. Funciones inline
10. 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

También puede usarse con paso por referencia para evitar modificación y a la vez, evitar la copia del argumento (se trata de un parámetro de entrada que no desea copiarse 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  
}
```

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

Parámetros y const I

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


Índice

1. Introducción
2. La función main
3. La pila
4. Ámbito de un dato
5. Referencias
6. Parámetros y const
- 7. Parámetros con valor por defecto**
8. Sobrecarga de funciones
9. Funciones inline
10. Variables locales static

Parámetros con valor por defecto

Las funciones pueden tener parámetros con un valor por defecto, éstos son los valores que asumirán los parámetros formales que no tienen correspondencia en la llamada:

- deben ser los últimos en la definición de la función
- en la llamada a la función deben aparecer en primer lugar aquellos argumentos que no toman valor por defecto.

En la llamada a la función, si hay parámetros por defecto, la correspondencia se hace de izda a dcha.

```
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;
}
```

Parámetros con valor por defecto: Ejemplo

```
#include <iostream>
using namespace std;
int Potencia(long base, int exp=2);

int main() {
    cout << "Potencia(5): " << Potencia(5) << endl;
    cout << "Potencia(4,3): " << Potencia(4,3) << endl;
    return 0;
}

int Potencia(long b, int e) {
    long result;
    if (e==2)
        result = b*b;
    else{
        result = b;
        for(int i=1; i < e; i++)
            result *= b;
    }
    return result;
}
```

Índice

1. Introducción
2. La función main
3. La pila
4. Ámbito de un dato
5. Referencias
6. Parámetros y const
7. Parámetros con valor por defecto
- 8. Sobrecarga de funciones**
9. Funciones inline
10. 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); // conversion implicita
}
```

Sobrecarga de funciones

C++ no puede distinguir entre dos versiones de función que sólo se diferencian en el 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);  
}
```


Sobrecarga de funciones

C++ puede distinguir entre versiones en que un parámetro puntero o bien referencia 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);
}
```

Sobrecarga de funciones

C++ puede distinguir entre versiones en que un parámetro puntero o bien referencia es `const` en una versión y en la otra no.

```
#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);
}
```

Sobrecarga de funciones

Sin embargo, C++ no puede distinguir entre versiones en que 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);
}
```

Sobrecarga de funciones

A veces pueden darse errores de ambigüedad:

```
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));  
}
```

Índice

1. Introducción
2. La función main
3. La pila
4. Ámbito de un dato
5. Referencias
6. Parámetros y const
7. Parámetros con valor por defecto
8. Sobrecarga de funciones
- 9. Funciones inline**
10. 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. Esto evita llamadas a la función, lo que aumenta la velocidad de ejecución.

- se definen colocando **inline** antes del tipo de retorno, en la definición de la función.
- suelen ser funciones pequeñas y frecuentemente usadas.
- ejecución más rápida en general, pero código generado de mayor tamaño
- el compilador podría no hacer caso al calificador **inline**.
- suelen colocarse en ficheros de cabecera (**.h**), ya que el compilador necesita su definición para poder expandirlas

Funciones inline: Ejemplo

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

inline int max(int a, int b){
    return (a>b) ? a : b;
}

int main(){
    int maximo;
    maximo = max(2,5);
    cout<<"El maximo es: " << maximo << endl;
}
```

Tiene muchas opciones de ser traducida a:

```
maximo = (a > b) ? x : y;
```

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;
}
```


Índice

1. Introducción
2. La función main
3. La pila
4. Ámbito de un dato
5. Referencias
6. Parámetros y const
7. Parámetros con valor por defecto
8. Sobrecarga de funciones
9. Funciones inline
10. 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 anterior en sucesivas llamadas a la función
- 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<10; ++i) {
        cout << i << "^2 = " << cuadrado(i) << endl;
    }
}
```