

# Metodología de la Programación

## Tema 3. Funciones (ampliación)

Departamento de Ciencias de la Computación e I.A.



*ugr*

Universidad  
de Granada

ETSIIIT Universidad de Granada

Curso 2017-18

# Contenido del tema

- 1 La función main
- 2 La pila
- 3 Ámbito de un dato
- 4 Parámetros con valor por defecto
- 5 Sobrecarga de funciones
- 6 Funciones inline
- 7 Variables locales static

# Contenido del tema

- 1 La función main
- 2 La pila
- 3 Ámbito de un dato
- 4 Parámetros con valor por defecto
- 5 Sobrecarga de funciones
- 6 Funciones inline
- 7 Variables locales static

# La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:  
`int main()`

# La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:  
`int main()`
- C++ permite una versión ampliada de la cabecera de `main`:  
`int main(int argc, char *argv[])`

# La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:  
`int main()`
- C++ permite una versión ampliada de la cabecera de `main`:  
`int main(int argc, char *argv[])`
  - **Valor de retorno:** El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error

# La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:  
`int main()`
- C++ permite una versión ampliada de la cabecera de `main`:  
`int main(int argc, char *argv[])`
  - **Valor de retorno:** El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error
  - **Argumentos de `main`:**
    - `int argc`: Número de argumentos usados al ejecutar el programa.
    - `char *argv[]`: Array de cadenas con cada uno de los argumentos.  
`argv[0]`: Nombre del ejecutable  
`argv[1]`: Primer argumento

...

## La función main II: Ejemplo

```
1 #include <iostream>
2 using namespace std;
3 int main(int argc, char *argv[]){
4     if (argc<3){
5         cerr << "Uso: "
6             << " <Fichero1> <Fichero2> ..." << endl;
7         return 1;
8     }
9     else{
10        cout<<"Numero argumentos: " << argc << endl;
11        for (int i=0; i<argc; ++i){
12            cout<<argv[i] << endl;
13        }
14    }
15    return 0;
16 }
```



## La función main III

Podemos convertir las cadenas estilo C al tipo string

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main(int argc, char *argv[]){
6     string par;
7     cout<<"Argumentos: "<<endl;
8
9     for (int i=0; i<argc; ++i){
10         par=argv[i];
11         cout<<par<<endl;
12     }
13     return 0;
14 }
15
```

# Contenido del tema

- 1 La función main
- 2 La pila**
- 3 Ámbito de un dato
- 4 Parámetros con valor por defecto
- 5 Sobrecarga de funciones
- 6 Funciones inline
- 7 Variables locales static

# La pila

## La pila (stack)

Zona de memoria que almacena información sobre las **funciones activas** de un programa.

# La pila

## La pila (stack)

Zona de memoria que almacena información sobre las **funciones activas** de un programa.

## Funciones activas

Aquellas que han sido invocadas y aún no han terminado su ejecución.

# La pila

## La pila (stack)

Zona de memoria que almacena información sobre las **funciones activas** de un programa.

## Funciones activas

Aquellas que han sido invocadas y aún no han terminado su ejecución.

Cuando se llama a una función:

- se crea en la pila un entorno de programa que almacena la información del módulo:
  - la dirección de memoria de retorno,
  - las constantes y variables locales,
  - los parámetros formales, ...

# La pila

## La pila (stack)

Zona de memoria que almacena información sobre las **funciones activas** de un programa.

## Funciones activas

Aquellas que han sido invocadas y aún no han terminado su ejecución.

Cuando se llama a una función:

- se crea en la pila un entorno de programa que almacena la información del módulo:
  - la dirección de memoria de retorno,
  - las constantes y variables locales,
  - los parámetros formales, ...
- Al terminar la ejecución del módulo 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()`.

# 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()`.
- `main()`
  - 1 es una función que debe aparecer en todo programa ejecutable escrito en C++.



# 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()`.

- `main()`

- 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[]);
```

# 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()`.
- `main()`
  - 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.

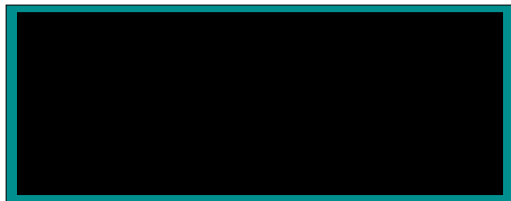
# Ejecución de un programa en C++

- La ejecución de un programa en C++ empieza creando un entorno de programa en el fondo de la pila para `main()`.
- `main()`
  - 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 desapila el entorno de programa asociado a `main()` de la pila.

# Ejemplo

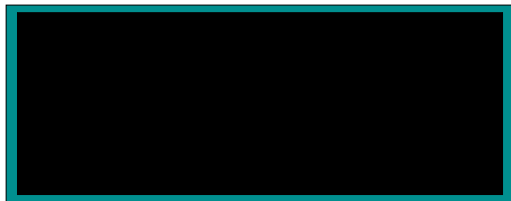
```
1 int main(){
2     int valor;
3     cout << "\nIntroduce "
4     << "un entero positivo: ";
5     cin >> valor;
6     ImprimeFactorial (valor);
7     Pausa();
8 }
```



PILA

# Ejemplo

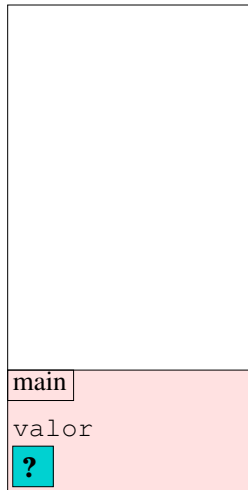
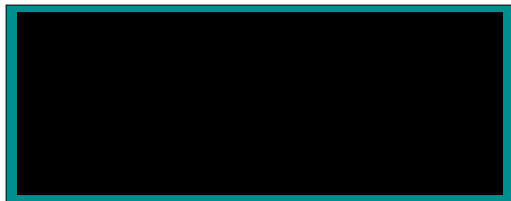
```
1 ▶ int main(){  
2   int valor;  
3   cout << "\nIntroduce "  
4   << "un entero positivo: ";  
5   cin >> valor;  
6   ImprimeFactorial (valor);  
7   Pausa();  
8 }
```



PILA

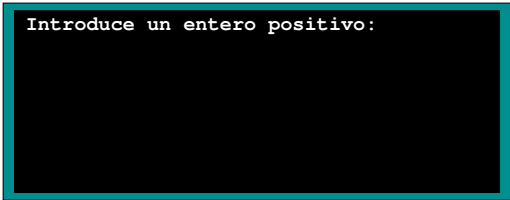
# Ejemplo

```
1 int main(){
2     int valor;
3     cout << "\nIntroduce "
4     << "un entero positivo: ";
5     cin >> valor;
6     ImprimeFactorial (valor);
7     Pausa();
8 }
```



# Ejemplo

```
1 int main(){
2     int valor;
3     cout << "\nIntroduce "
4     << "un entero positivo: ";
5     cin >> valor;
6     ImprimeFactorial (valor);
7     Pausa();
8 }
```



Introduce un entero positivo:



main

valor

?

# Ejemplo

```
1 int main(){
2     int valor;
3     cout << "\nIntroduce "
4     << "un entero positivo: ";
5     cin >> valor;
6     ImprimeFactorial (valor);
7     Pausa();
8 }
```

Introduce un entero positivo: 3

main

valor

3



# Ejemplo

ImprimeFactorial(valor)

```
1 void ImprimeFactorial (int n){  
2     int resul;  
3  
4     resul = Factorial(n);  
5     cout << "\nEl factorial de "  
6         << n << " es " << resul  
7         << endl;  
8 }
```

Introduce un entero positivo: 3

main

valor

3

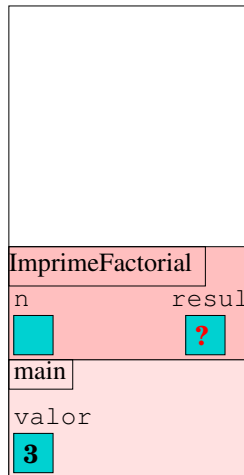
# Ejemplo

ImprimeFactorial(valor)

```

1 void ImprimeFactorial (int n){
2     int resul;
3
4     resul = Factorial(n);
5     cout << "\nEl factorial de "
6         << n << " es " << resul
7         << endl;
8 }
```

Introduce un entero positivo: 3



# Ejemplo

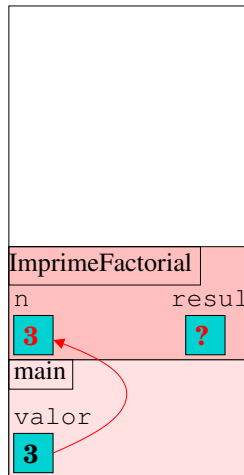
ImprimeFactorial(valor)

```

1 void ImprimeFactorial (int n){
2     int resul;
3
4     resul = Factorial(n);
5     cout << "\nEl factorial de "
6         << n << " es " << resul
7         << endl;
8 }

```

Introduce un entero positivo: 3



# Ejemplo

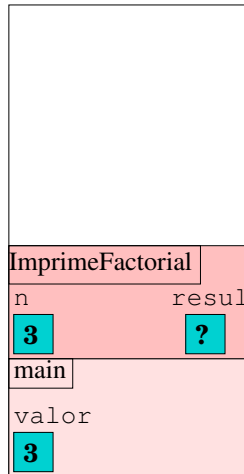
ImprimeFactorial(valor)

```

1 void ImprimeFactorial (int n){
2     int resul;
3
4     resul = Factorial(n);
5     cout << "\nEl factorial de "
6         << n << " es " << resul
7         << endl;
8 }

```

Introduce un entero positivo: 3

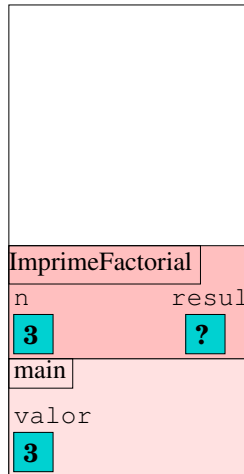


# Ejemplo

```
result = Factorial (n)
```

```
1 int Factorial (int n){
2     int i, valor=1;
3
4     for (i=2; i<=n; i++)
5         valor=valor*i;
6
7     return valor;
8 }
```

Introduce un entero positivo: 3

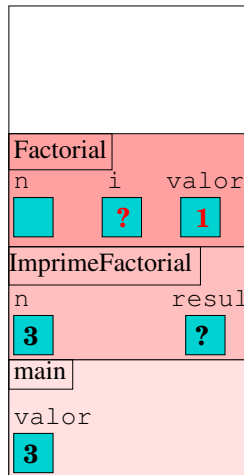


# Ejemplo

```
result = Factorial (n)
```

```
1 int Factorial (int n){
2     int i, valor=1;
3
4     for (i=2; i<=n; i++)
5         valor=valor*i;
6
7     return valor;
8 }
```

Introduce un entero positivo: 3

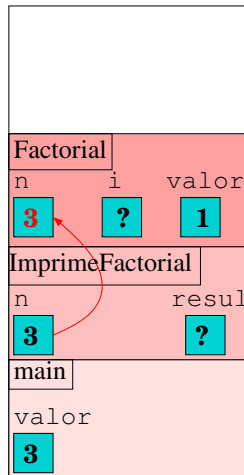


# Ejemplo

```
result = Factorial (n)
```

```
1 int Factorial (int n){
2   int i, valor=1;
3
4   for (i=2; i<=n; i++)
5     valor=valor*i;
6
7   return valor;
8 }
```

Introduce un entero positivo: 3

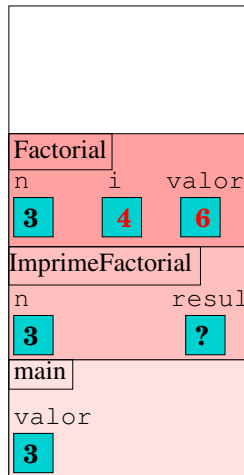


# Ejemplo

```
result = Factorial (n)
```

```
1 int Factorial (int n){
2     int i, valor=1;
3
4     for (i=2; i<=n; i++)
5         valor=valor*i;
6
7     return valor;
8 }
```

Introduce un entero positivo: 3



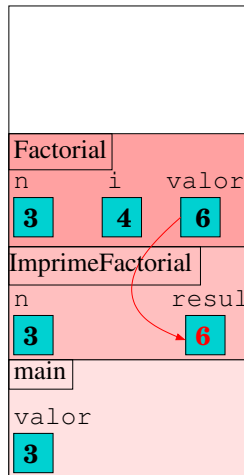


# Ejemplo

```
result = Factorial (n)
```

```
1 int Factorial (int n){
2   int i, valor=1;
3
4   for (i=2; i<=n; i++)
5     valor=valor*i;
6
7   return valor;
8 }
▶
```

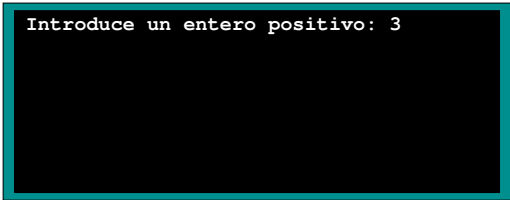
Introduce un entero positivo: 3



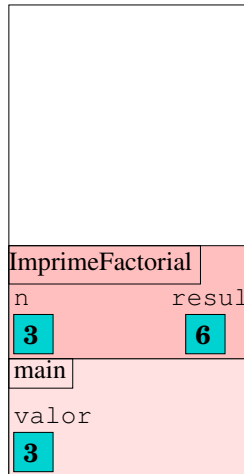
# Ejemplo

```
result = Factorial (n)
```

```
1 int Factorial (int n){
2     int i, valor=1;
3
4     for (i=2; i<=n; i++)
5         valor=valor*i;
6
7     return valor;
8 }
```



```
Introduce un entero positivo: 3
```



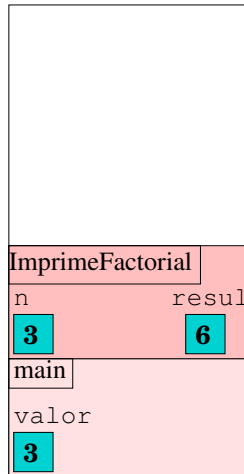
# Ejemplo

ImprimeFactorial(valor)

```

1 void ImprimeFactorial (int n){
2     int resul;
3
4     resul = Factorial(n);
5     cout << "\nEl factorial de "
6         << n << " es " << resul
7         << endl;
8 }
```

Introduce un entero positivo: 3



# Ejemplo

ImprimeFactorial(valor)

```

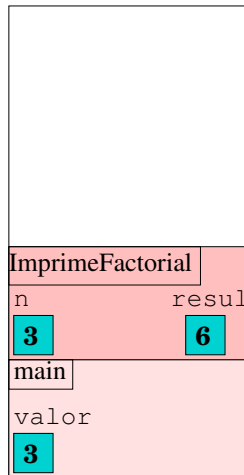
1 void ImprimeFactorial (int n){
2     int resul;
3
4     resul = Factorial(n);
5     cout << "\nEl factorial de "
6         << n << " es " << resul
7         << endl;
8 }

```

```

Introduce un entero positivo: 3
El factorial de 3 es 6

```



# Ejemplo

ImprimeFactorial(valor)

```
1 void ImprimeFactorial (int n){  
2     int resul;  
3  
4     resul = Factorial(n);  
5     cout << "\nEl factorial de "  
6         << n << " es " << resul  
7         << endl;  
8 }  ▶
```

```
Introduce un entero positivo: 3  
El factorial de 3 es 6
```

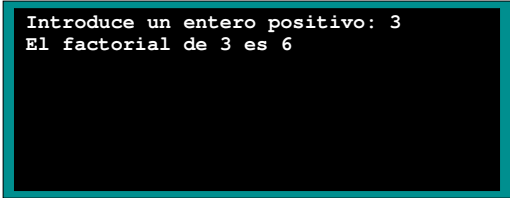
main

valor

3

# Ejemplo

```
1 int main(){
2     int valor;
3     cout << "\nIntroduce "
4     << "un entero positivo: ";
5     cin >> valor;
6     ImprimeFactorial (valor);
7     Pausa();
8 }
```



```
Introduce un entero positivo: 3
El factorial de 3 es 6
```



The diagram shows a stack frame for the `main` function. It is a vertical rectangle with a light pink background. At the bottom, there is a cyan box containing the number `3`, which is the value of the `valor` variable. Above this box, the variable name `valor` is written in black text. The top of the frame is labeled `main` in a small box.

main

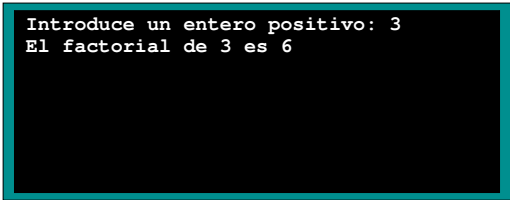
valor

3

# Ejemplo

## Pausa()

```
1 void Pausa(){  
2     char tecla;  
3  
4     cout << "Pulsa una tecla "  
5         << "para continuar: ";  
6     cin >> tecla;  
7  
8 }
```



```
Introduce un entero positivo: 3  
El factorial de 3 es 6
```



main

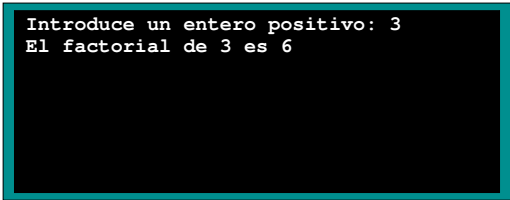
valor

3

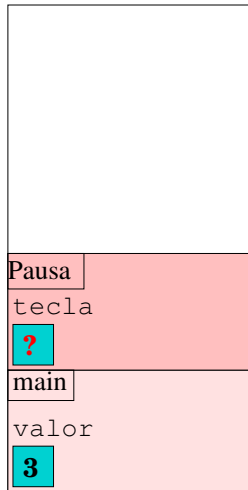
# Ejemplo

## Pausa()

```
1 void Pausa(){  
2     char tecla;  
3  
4     cout <<"Pulsa una tecla "  
5         <<"para continuar: ";  
6     cin >> tecla;  
7  
8 }
```



```
Introduce un entero positivo: 3  
El factorial de 3 es 6
```

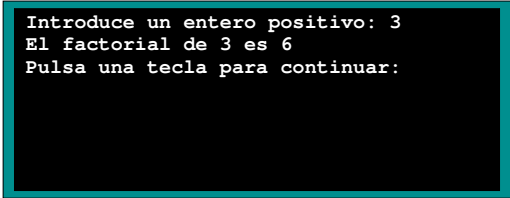




# Ejemplo

## Pausa()

```
1 void Pausa(){  
2     char tecla;  
3  
4     cout <<"Pulsa una tecla "  
5         <<"para continuar: ";  
6     cin >> tecla;  
7  
8 }
```



```
Introduce un entero positivo: 3  
El factorial de 3 es 6  
Pulsa una tecla para continuar:
```

Pausa

tecla

?

main

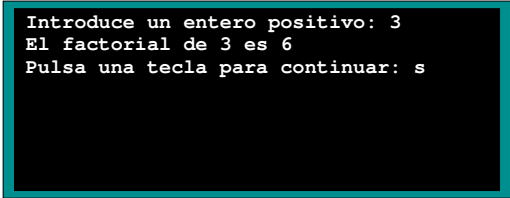
valor

3

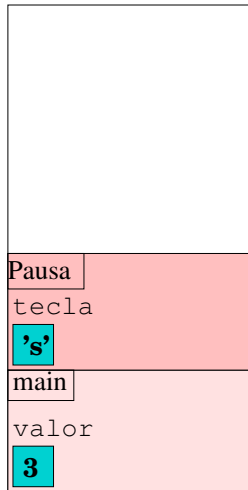
# Ejemplo

## Pausa()

```
1 void Pausa(){
2     char tecla;
3
4     cout <<"Pulsa una tecla "
5         <<"para continuar: ";
6     cin >> tecla;
7
8 }
```



```
Introduce un entero positivo: 3
El factorial de 3 es 6
Pulsa una tecla para continuar: s
```



# Ejemplo

## Pausa()

```
1 void Pausa(){
2     char tecla;
3
4     cout <<"Pulsa una tecla "
5           <<"para continuar: ";
6     cin >> tecla;
7
8 }
```

```
Introduce un entero positivo: 3
El factorial de 3 es 6
Pulsa una tecla para continuar: s
```

main

valor

3

# Ejemplo

```
1 int main(){
2     int valor;
3     cout << "\nIntroduce "
4     << "un entero positivo: ";
5     cin >> valor;
6     ImprimeFactorial (valor);
7     Pausa();
8 }
```

```
Introduce un entero positivo: 3
El factorial de 3 es 6
Pulsa una tecla para continuar: s
```

main

valor

3

# Ejemplo

```
1 int main(){
2     int valor;
3     cout << "\nIntroduce "
4     << "un entero positivo: ";
5     cin >> valor;
6     ImprimeFactorial (valor);
7     Pausa();
8 }
```

```
Introduce un entero positivo: 3
El factorial de 3 es 6
Pulsa una tecla para continuar: s
```

PILA

# Contenido del tema

- 1 La función main
- 2 La pila
- 3 Ámbito de un dato**
- 4 Parámetros con valor por defecto
- 5 Sobrecarga de funciones
- 6 Funciones inline
- 7 Variables locales static

# Ámbito de un dato

## El ámbito de un dato

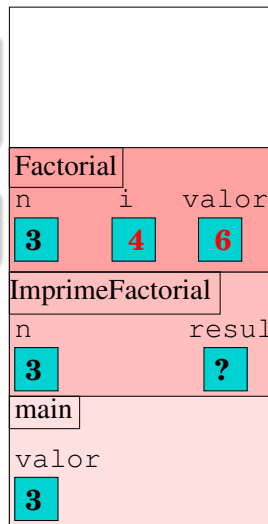
es el conjunto de todos aquellos módulos que lo pueden referenciar.

# Ámbito de un dato

## El ámbito de un dato

es el conjunto de todos aquellos módulos que lo pueden referenciar.

¿Cuál es el ámbito de cada uno de los datos que aparecen en la figura?





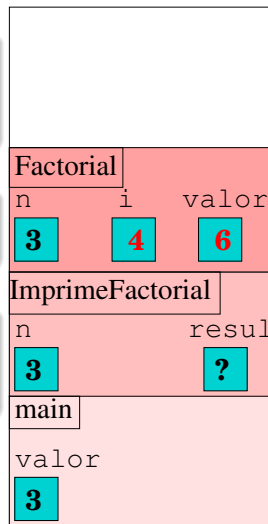
# Ámbito de un dato

## El ámbito de un dato

es el conjunto de todos aquellos módulos que lo pueden referenciar.

¿Cuál es el ámbito de cada uno de los datos que aparecen en la figura?

El ámbito de un dato está definido por el par de llaves que definen el bloque de código dónde se ha declarado el dato



# Ámbito de un dato

## El ámbito de un dato

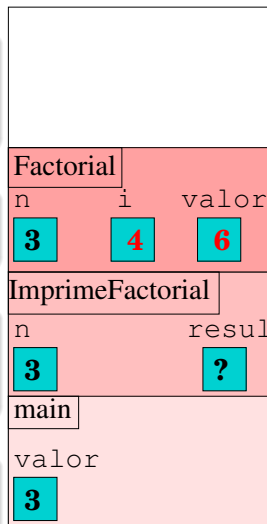
es el conjunto de todos aquellos módulos que lo pueden referenciar.

¿Cuál es el ámbito de cada uno de los datos que aparecen en la figura?

El ámbito de un dato está definido por el par de llaves que definen el bloque de código dónde se ha declarado el dato

## Excepción

Datos globales



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

```
1 double f1(double x, double y){  
2     double i, j;  
3  
4     for (double i=x; i<y; i++){  
5         double z;  
6         z=(i-x);  
7         j=z/(y-x);  
8         cout << j <<endl;  
9     }  
10 }
```

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

```
1 double f1(double x, double y){  
2     double i, j;  
3  
4     for (double i=x; i<y; i++){  
5         double z;  
6         z=(i-x);  
7         j=z/(y-x);  
8         cout << j <<endl;  
9     }  
10 }
```

## Solución

- x, y, i(línea 2), j son globales a todo el módulo.
- i(línea 4), z son locales al cuerpo del **for**.

# Referencias

## Referencia

Es una especie de 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.

## Sintaxis

`<tipo> & <identificador> = <iniciador> ;`

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

# Devolución por referencia I

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 II

## Devolución de referencias a datos locales

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;
}
```

## Parámetros y const I

En el paso de parámetros por valor 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 II

También lo podemos hacer con el paso por referencia para evitar modificación y a la vez, evitar copia del argumento.

```
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 III

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 IV

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
}
```

# Contenido del tema

- 1 La función main
- 2 La pila
- 3 Ámbito de un dato
- 4 Parámetros con valor por defecto**
- 5 Sobrecarga de funciones
- 6 Funciones inline
- 7 Variables locales static

# Parámetros con valor por defecto

Una función o método puede tener parámetros con un valor por defecto

- Deben ser los últimos de la función.
- En la llamada a la función, si sólo se especifican un subconjunto de parámetros actuales, deben ser los primeros.

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

# Parámetros con valor por defecto: Ejemplo

```
1 #include <iostream>
2 using namespace std;
3 int volumenCaja(int largo=1, int ancho=1, int alto=1);
4
5 int main(){
6     cout << "Volumen por defecto: " << volumenCaja() << endl;
7     cout << "El volumen de una caja (10,1,1) es: "
8         << volumenCaja(10) << endl;
9     cout << "El volumen de una caja (10,5,1) es: "
10        << volumenCaja(10,5) << endl;
11     cout << "El volumen de una caja (10,5,2) es: "
12        << volumenCaja(10,5,2) << endl;
13     return 0;
14 }
15
16 int volumenCaja( int largo, int ancho, int alto ){
17     return largo * ancho * alto;
18 }
```

# Contenido del tema

- 1 La función main
- 2 La pila
- 3 Ámbito de un dato
- 4 Parámetros con valor por defecto
- 5 Sobrecarga de funciones**
- 6 Funciones inline
- 7 Variables locales static

# Sobrecarga de funciones

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

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



# Sobrecarga de funciones

## Conversión implícita de tipos

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

```
void function(double x){  
    cout << "double" << x << endl;  
}  
  
void function(char *p){  
    cout << "char *" << *p << endl;  
}  
  
int main(){  
    function(4.5);  
    function(3); // conversion implicita  
}
```

# Sobrecarga de funciones

## Distinción por el tipo devuelto

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

## Uso de const en punteros y referencias

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

```
1 #include <iostream>
2 using namespace std;
3 void function(double &x){
4     cout << "function(double &x): " << x << endl;
5 }
6 void function(const double &x){
7     cout << "function(const double &x): " << x << endl;
8 }
9 int main(){
10     double x=2;
11     const double A=4.5;
12     function(A);
13     function(x);
14 }
```

# Sobrecarga de funciones

## Uso de const en punteros y referencias

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

```
1 #include <iostream>
2 using namespace std;
3 void funcion(double *p){
4     cout << "funcion(double *p): " << *p << endl;
5 }
6 void funcion(const double *p){
7     cout << "funcion(const double *p): " << *p << endl;
8 }
9 int main(){
10     double x=2;
11     const double A=4.5;
12     funcion(&A);
13     funcion(&x);
14 }
```

# Sobrecarga de funciones

## Uso de const en parámetros por valor

Sin embargo, C++ no puede distinguir entre versiones en que un parámetro por valor es const en una versión y en la otra no.

```
1 #include <iostream>
2 using namespace std;
3 void function(double x){
4     cout << "function(double x): " << x << endl;
5 }
6 void function(const double x){
7     cout << "function(const double x): " << x << endl;
8 }
9 int main(){
10     double x=2;
11     const double A=4.5;
12     function(A);
13     function(x);
14 }
```

# Sobrecarga de funciones

## Ambigüedad

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));  
}
```

# Sobrecarga de funciones

## Otro ejemplo de ambigüedad

En este caso al usar funciones con parámetros por defecto

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

# Contenido del tema

- 1 La función main
- 2 La pila
- 3 Ámbito de un dato
- 4 Parámetros con valor por defecto
- 5 Sobrecarga de funciones
- 6 Funciones inline**
- 7 Variables locales static



# Funciones inline

## Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.

# Funciones inline

## Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.

# Funciones inline

## Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).

# Funciones inline

## Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.

# Funciones inline

## Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.
- Código generado de mayor tamaño.

# Funciones inline

## Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de los macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.
- Código generado de mayor tamaño.
- El compilador puede que no haga caso al calificador `inline`.

# Funciones inline

## Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.
- Código generado de mayor tamaño.
- El compilador puede que no haga caso al calificador `inline`.
- Suelen colocarse en ficheros de cabecera (`.h`) ya que el compilador necesita su definición para poder expandirlas.

# Funciones inline: Ejemplo

```
1 #include <iostream>
2
3 inline bool numeroPar(const int n){
4     return (n%2==0);
5 }
6
7 int main(){
8     std::string parimpar;
9     parimpar=numeroPar(25)?"par":"impar";
10    std::cout<<"Es 25 par?: " << parimpar;
11 }
```



# Contenido del tema

- 1 La función main
- 2 La pila
- 3 Ámbito de un dato
- 4 Parámetros con valor por defecto
- 5 Sobrecarga de funciones
- 6 Funciones inline
- 7 Variables locales static**

# Variables locales static

## Variable local static

Es una variable local de una función o método que no se destruye al acabar la función, y que mantiene su valor entre llamadas.

- Se inicializa 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.

```

1  #include<iostream>
2  using namespace std;
3  double cuadrado(double numero){
4      static int contadorLlamadas=1;
5      cout<<"Llamadas a cuadrado: " <<contadorLlamadas<<endl;
6      contadorLlamadas++;
7      return numero*numero;
8  }
9  int main(){
10     for(int i=0; i<10; ++i)
11         cout<<i << "^2 = " << cuadrado(i) << endl;
12 }
```