

## 4

## La abstracción procedimental

Grados en Ingeniería Informática, Ingeniería del Software e Ingeniería de Computadores

Alberto de la Encina Vara  
(adaptadas del original de Luis Hernández Yáñez)



Facultad de Informática  
Universidad Complutense



## Índice

---

Subprogramas	2
Parámetros	7
Argumentos	11
Parámetros por referencia constante	22
Resultado y vuelta de la función	26
Notificación de errores	33
Subprogramas y declaraciones	39
Prototipos	45
Sobrecarga de funciones	49
Funciones de operador	52
Argumentos implícitos	56
La función main()	64



## Subprogramas



## Abstracción procedimental

---

### *Subprogramas*

*Pequeños programas dentro de otro programa.*

- ✓ Unidades de ejecución independientes.
- ✓ Encapsulan código y datos.
- ✓ Pueden comunicarse con otros subprogramas (parámetros).

*Subrutinas, procedimientos, funciones, acciones, ...*

- ✓ Realizan tareas individuales del programa.
- ✓ Funcionalidad concreta, identificable y coherente (diseño).
- ✓ Se ejecutan de principio a fin cuando se llaman (*invocan*).
- ✓ Terminar devolviendo el control al punto de llamada.



Aumentan el nivel de abstracción del programa.  
Facilitan la prueba, la depuración y el mantenimiento.



# Comunicación con el exterior

---

Datos de entrada, datos de salida y datos de entrada/salida

Datos de entrada: aceptados



Datos de salida: devueltos



Datos de entrada/salida:  
aceptados y modificados



# Comunicación con el exterior

---

```
typedef enum {lunes, martes,..., domingo}tDiaSemana;  
const int NumDias = 7;          // Tema 3
```

✓ Subprograma que dado un día de la semana lo muestra

```
void mostrar(tDiaSemana ddls);  
int main(){ ... mostrar(lunes); ...}
```



✓ Subprograma que lee y devuelve un día de la semana

```
tDiaSemana leerDdls();  
int main(){... tDiaSemana dia = leerDsIs();  
    mostrar(dia);  
    ...}
```



# Comunicación con el exterior

- ✓ Subprograma que dado un día de la semana devuelve el siguiente día

```
tDiaSemana siguiente(tDiaDemana ddls);  
int main(){... tDiaSemana dia = siguiente(lunes); ...}
```



- ✓ Subprograma que dada una variable la modifica con el siguiente día

```
void siguiente(tDiaDemana & ddls);  
int main(){... tDiaSemana dia = lunes; siguiente(dia); ...}
```



## Parámetros en C++

### *Declaración de parámetros*

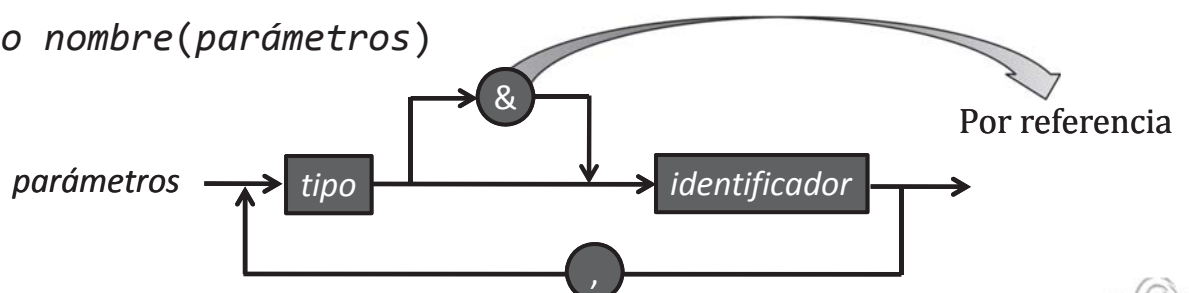
Se distinguen dos clases de parámetros:

- Por valor (se copia el valor): Sólo de entrada
- Por variable: Sólo salida o entrada/salida

### *Lista de parámetros formales*

Entre los paréntesis de la cabecera del subprograma.

*tipo nombre(parámetros)*



# Parámetros en C++

---

## *Parámetros por valor*

```
int cuadrado(int num);  
double potencia(double base, int exp);  
void mostrar(tDiaSemana dia);  
void mostrar(double dia);
```

## *Argumentos*

```
mostrar(potencia(cuadrado(3*2+1), 3));  
mostrar(potencia(cuadrado(7), 3));  
mostrar(potencia(49, 3));  
mostrar(117649);
```



# Parámetros en C++

---

## *Parámetros por referencia o variable*

&

```
void incr(int & num);  
void leer(tDiaSemana & dia);  
void intercambia(double & x, double & y);
```

## *Argumentos*

```
int n = 45; double v = 2.3, y = 33; tDiaSemana ddls;  
incr(n); // ahora n == 46  
leer(ddls); // supongamos martes  
siguiente(ddls); // ahora ddls == miercoles  
intercambia(v, y); // ahora v == 33 e y == 2.3
```



# Parámetros en C++

---



Puede haber tanto parámetros por valor como por referencia

¡Atención! Los arrays y los flujos NO se pueden pasar por valor

- ✓ Los arrays se pasan por defecto (sin poner &) por referencia

Parámetros de salida o E/S: no poner nada

```
void agregar(char cstr[], char ch);
```

Parámetros de entrada: declarar const

```
double media(const TArray lista);
```

- ✓ Los flujos (stream) hay que pasarlos por referencia (poner &)

```
void escribir(ostream & flujo, tipo dato);
```

```
void leer(istream & flujo, tipo & dato);
```

Aunque el flujo sea de lectura (input), el cursor debe moverse.



## Argumentos

---

*Llamada a un subprograma*

*nombre(Argumentos)*

- Tantos argumentos como parámetros y en el mismo orden
- Concordancia de tipos entre argumentos y parámetros
- Por valor: expresiones válidas (se pasa el resultado).
- Por referencia: ¡sólo variables!

Se copian los valores de las expresiones pasadas por valor  
en los correspondientes parámetros

Se hacen corresponder los argumentos pasados por referencia  
(variables) con sus correspondientes parámetros.



# Argumentos

## *Llamada a una función con argumentos pasados por valor*

Expresiones válidas con concordancia de tipo:

```
void proc(int x, double a);
```

→ `proc (23 * 4 / 7, 13.5);`

→ `double d = 3;  
proc (12, d);`

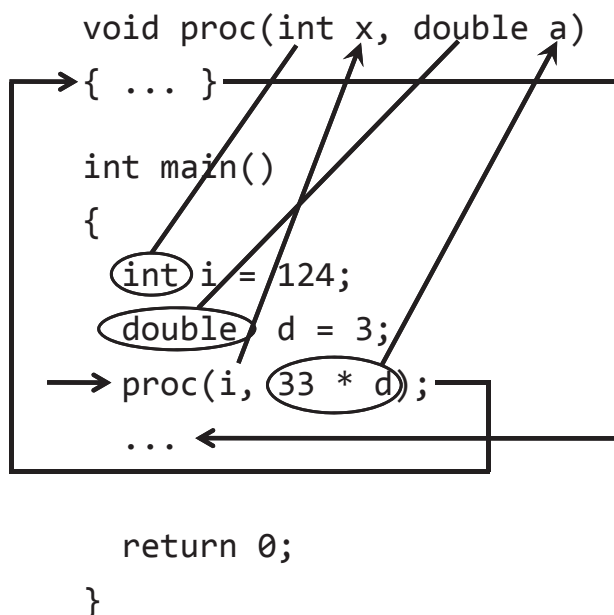
→ `double d = 3;  
int i = 124;  
proc (i, 33 * d);`

→ `double d = 3;  
int i = 124;  
proc (sqrt(20) * 34 + i, i * d);`



## Paso de argumentos (parámetros)

### *Argumentos pasados por valor*

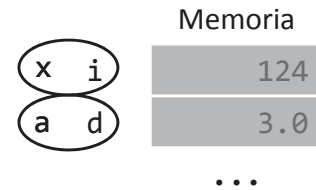
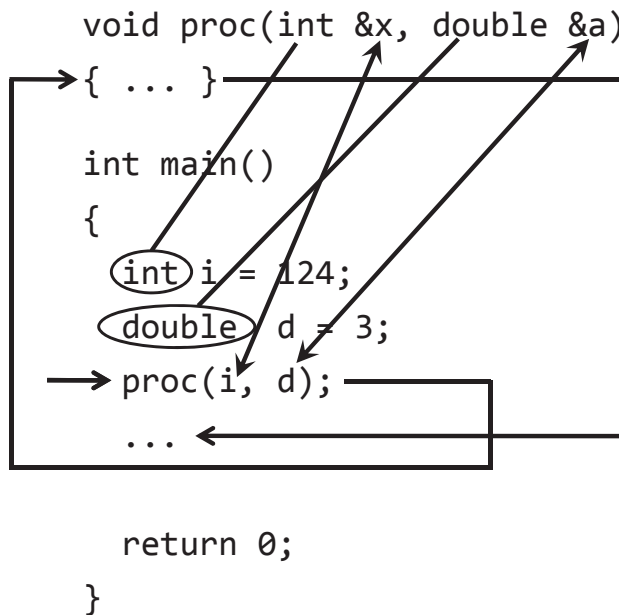


Memoria	
i	124
d	3.0
...	
...	
x	124
a	99.0
...	



# Paso de argumentos (parámetros)

## Argumentos pasados por referencia



## Argumentos

Dadas las siguientes declaraciones:

```
int i;
double d;
void proc(int x, double & a);
```

¿Qué llamadas son correctas?

- |                    |   |   |
|--------------------|---|---|
| proc(3, i, d);     | ✗ | Nº de argumentos ≠ Nº de parámetros           |
| proc(i, d);        | ✓ |   |
| proc(3*i + 12, d); | ✓ |   |
| proc(i, 23);       | ✗ | Parámetro por referencia → argumento variable |
| proc(d, i);        | ✗ | ¡Argumento double para parámetro int!         |
| proc(3.5, d);      | ✗ | ¡Argumento double para parámetro int!         |
| proc(i);           | ✗ | Nº de argumentos ≠ Nº de parámetros           |





# Paso de argumentos (parámetros)

---

```
...
void divide(int op1, int op2, int &div, int &rem) {
// Divide op1 entre op2 y devuelve el cociente y el resto
    div = op1 / op2;
    rem = op1 % op2;
}

int main() {
    int cociente, resto;
    int num1 = 10; int num2 = 2;
    divide(num1, num2, cociente, resto);
    ...

    return 0;
}
```



# Paso de argumentos (parámetros)

---

```
...
void divide(int op1, int op2, int &div, int &rem){
// Divide op1 entre op2 y devuelve el cociente y el resto
    div = op1 / op2;
    rem = op1 % op2;
}

int main() {
    int cociente, resto;
    int num1 = 10; int num2 = 2;
→ divide(num1, num2, cociente, resto);
    ...

    return 0;
}
```

Memoria	
cociente	?
resto	?
num1	10
num2	2
...	



# Paso de argumentos (parámetros)

```
...
void divide(int op1, int op2, int &div, int &rem) {
    // Divide op1 entre op2 y devuelve el cociente y el resto
    div = op1 / op2;
    rem = op1 % op2;
}

int main() {
    int cociente, resto;
    int num1 = 10; int num2 = 2;
    → divide(num1, num2, cociente, resto);
    ...

    return 0;
}
```

		Memoria
div	cociente	?
rem	resto	?
num1		10
num2		2
		...
op1		10
op2		2
		...



# Paso de argumentos (parámetros)

```
...
void divide(int op1, int op2, int &div, int &rem) {
    // Divide op1 entre op2 y devuelve el cociente y el resto
    div = op1 / op2;
    rem = op1 % op2;
}

int main() {
    int cociente, resto;
    int num1 = 10; int num2 = 2;
    → divide(num1, num2, cociente, resto);
    ...

    return 0;
}
```

		Memoria
div	cociente	5
rem	resto	0
num1		10
num2		2
		...
op1		10
op2		2
		...



# Paso de argumentos (parámetros)

```
...
void divide(int op1, int op2, int &div, int &rem) {
    // Divide op1 entre op2 y devuelve el cociente y el resto
    div = op1 / op2;
    rem = op1 % op2;
}

int main() {
    int cociente, resto;
    int num1 = 10; int num2 = 2;
    divide(num1, num2, cociente, resto);
    ...

    return 0;
}
```

	Memoria
cociente	5
resto	0
num1	10
num2	2
...	...



# Paso de argumentos (parámetros)

```
...
void intercambia(double &valor1, double &valor2) {
    // Intercambia los valores de las dos variables
    double tmp; // Variable local (temporal)
    tmp = valor1;
    valor1 = valor2;
    valor2 = tmp;
}

int main() {
    double num1, num2;
    cout << "Valor 1: "; cin >> num1;
    cout << "Valor 2: "; cin >> num2;
    if (num1 < num2)
        intercambia(num1, num2);
    cout << "Mayor " << num1
        << " y menor " << num2 << endl;

    return 0;
}
```

Memoria temporal  
de la función

tmp	?
...	...

Memoria de main()

valor1 num1	13.6
valor2 num2	317.14
...	...



# Parámetros por referencia constante

---

*Para evitar copias en el paso de parámetros por valor, los parámetros de entrada se pueden pasar por referencia constante.*

`const tipo_Parámetro & nombre_Parámetro`

Ejemplo:

```
void func(const string & nomb);
```

*En el cuerpo de la función, el parámetro nomb, es una constante local. En las llamadas, el argumento tiene que ser una variable. Se pasa la referencia de la variable.*

*Recomendado cuando el tipo del parámetro no es básico, y no se necesita una variable local.*



# Parámetros por referencia constante

---

```
bool suma(const string & nombre, double & suma){
    bool error;
    ifstream archivo;
    archivo.open(nombre);
    error = !archivo.is_open();
    if (!error) {
        double dato;
        suma = 0; arch >> dato;
        while (!arch.fail()) {
            suma += dato;
            arch >> dato;
        }
        archivo.close();
    }
    return error;
}

int main() {
    string nombArch; double sum;
    cout << "Nombre del archivo: "; cin >> nombArch;
    if(suma(nombArch, sum)) // se pasan los argumentos
        cout << "Suma = " << sum << endl;
    return 0;
}
```



# Más ejemplos

```
#include <iostream>
#include <fstream>
using namespace std;
double suma(istream & flujo);
```

```
int main() {
    double resultado;
    ifstream archivo;
    archivo.open("datos.txt");
    if (!archivo.is_open()) cout << "ERROR DE APERTURA" << endl;
    else {
        cout << "Suma = " << suma(archivo) << endl;
        archivo.close();
    }
    return 0;
}

double suma(istream & flujo) {
    double dato, sum = 0; flujo >> dato;
    while (!flujo.fail()) {
        sum += dato; flujo >> dato; }
    return sum;
}
```



Los archivos siempre se pasan por referencia.



## Parámetros y argumentos

- ✓ Parámetros: por valor o por referencia (&)

Son variables locales de la función.

Reciben sus valores iniciales de los argumentos al ejecutarse una llamada (paso de parámetros).

Se destruyen al terminar la ejecución de la función, junto con todos los datos locales.

- ✓ Argumentos: del tipo del parámetro (posibles promociones )

Parámetros por valor: expresiones del tipo del parámetro

Parámetros por referencia: variables del tipo del parámetro

- ✓ Paso de parámetros:

Parámetros por valor: se copia el valor resultante de la expresión

Parámetros por referencia: se copia la referencia de la variable



# Resultado de la función

## *Una función ha de devolver un resultado*

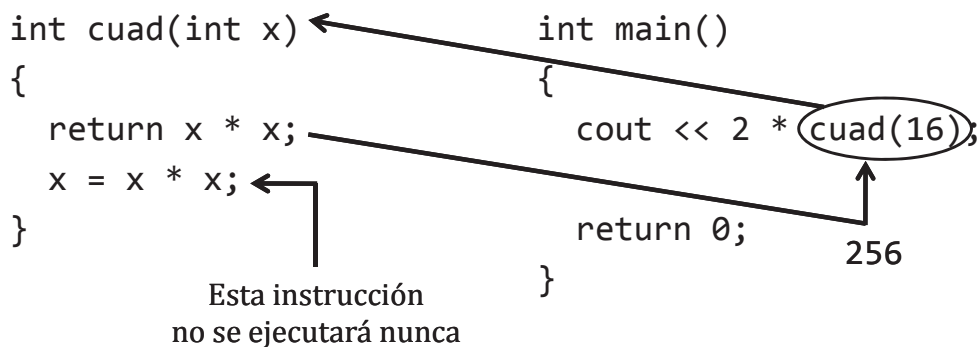
Subprograma de tipo distinto de void.

La función ha de terminar su ejecución devolviendo el resultado.

*La instrucción return*

- Devuelve el dato que se indique a continuación como resultado.
- Termina la ejecución de la función.

El dato devuelto por la función sustituye a la llamada en la expresión.



# Resultado de la función

## *Cálculo del factorial*

Factorial (N) = 1 x 2 x 3 x ... x (N-2) x (N-1) x N

```
long long int factorial(int n) {
    // Calcula y devuelve el factorial del número dado
    long long int fact = 1;
    if (n < 0) fact = 0;
    else
        for (int i = 2; i <= n; i++)
            fact *= i;
    return fact;
}

int main() {
    int num;
    cout << "Num: "; cin >> num;
    cout << "El factorial de "<< num << " es "
        << factorial(num) << endl;

    return 0;
}
```



# Vuelta de la función

---

## ¿Cuándo termina la función?

Subprograma de tipo `void` (*procedimientos*):

- Al encontrar la llave de cierre que termina el subprograma, o
- Al encontrar una instrucción `return` (sin dato).

Subprograma de tipo distinto de `void` (*funciones*):

- Al encontrar una instrucción `return` (con dato).

Nuestros subprogramas siempre terminarán al final:

- ✓ No usaremos `return` en los procedimientos
- ✓ Funciones: sólo un `return` y estará al final



Para facilitar la depuración y el mantenimiento, codifica los subprogramas con un único punto de salida.



# Vuelta de la función

---

## Un único punto de salida

```
int compara(int val1, int val2) {  
  // -1 si val1 < val2, 0 si iguales, +1 si val1 > val2  
  if (val1 == val2)  
    return 0;  
  else if (val1 < val2)  
    return -1;  
  else  
    return 1;  
}
```

¡3 puntos de salida!

```
int compara(int val1, int val2) {  
  int resultado;  
  if (val1 == val2) resultado = 0;  
  else if (val1 < val2) resultado = -1;  
  else resultado = 1;  
  return resultado;  
}
```

→ Punto de salida único



# Vuelta de la función

---

## ¿Varios puntos de salida de la función?

- ✓ Buen diseño.- Todos los caminos del flujo de ejecución terminan en un mismo punto: el punto de salida de la función, la última instrucción.
- ✓ Buen diseño.- Si al remplazar los puntos de salida por:
  - una variable del tipo de la función *tipoFun* resultado;
  - una última instrucción `return resultado`;
  - y una asignación por cada punto de salidaSe obtiene un código equivalente.



# Vuelta de la función

---

## ¿Varios puntos de salida de la función?

```
bool divide(int op1, int op2, int &div, int &rem){  
    if (op2 == 0) return false; // divisor 0  
    else {  
        div = op1 / op2;  
        rem = op1 % op2;  
        return true;  
    }  
}
```

Buen diseño.- Al remplazar los puntos de salida por una variable del tipo de ..., se obtiene un código correcto equivalente.

```
bool divide(int op1, int op2, int &div, int &rem){  
    bool ok;  
    ok = op2 != 0;  
    if (ok) { // divisor distinto de 0  
        div = op1 / op2;  
        rem = op1 % op2;  
    }  
    return ok;  
}
```





# Vuelta de la función

---

*¿Varios puntos de salida de la función?*

```
int compara(int val1, int val2) {  
    // -1 si val1 < val2, 0 si iguales, +1 si val1 > val2  
    if (val1 == val2)  
        return 0;  
    if (val1 < val2)  
        return -1;  
    return 1;  
}
```

MAL DISEÑO.- Al remplazar los puntos de salida por una variable del tipo de ..., se obtiene un código INCORRECTO.

```
int compara(int val1, int val2) {  
    int resultado;  
    if (val1 == val2) resultado = 0;  
    if (val1 < val2) resultado = -1;  
    resultado = 1;  
    return resultado;  
}
```



## Fundamentos de la programación

---

# Notificación de errores



# Notificación de errores

En los subprogramas se pueden detectar errores

Errores que impiden realizar los cálculos:

```
void divide(int op1, int op2, int &div, int &rem) {  
    // Divide op1 entre op2 y devuelve el cociente y el resto  
    → { if (op2 == 0) // división por 0  
        cout << "Error: El divisor es cero" << endl;  
        else {  
            div = op1 / op2;  
            rem = op1 % op2;  
        }  
    }  
}
```

¿Debe el subprograma notificar al usuario o al programa?

→ Mejor notificarlo al punto de llamada y allí decidir qué hacer



# Notificación de errores

```
...  
void divide(int op1, int op2, int &div, int &rem) {  
    // Divide op1 entre op2 y devuelve el cociente y el resto  
    if (op2 == 0) // división por 0  
        cout << "Error: El divisor es cero" << endl;  
    else {  
        div = op1 / op2;  
        rem = op1 % op2;  
    }  
}
```

```
int main() {  
    int cociente, resto;  
    int num1 = 10; int num2 = 0;  
    divide(num1, num2, cociente, resto);  
    cout << num1 << " entre " << num2  
        << " da un cociente de "  
        << cociente << " y un resto de " << resto << endl;  
    return 0;  
}
```

↓  
Si op2 es cero  
los parámetros  
se quedan sin inicializar

↓  
main() mostrará  
datos *basura*



# Notificación de errores

## *Uso del resultado de la función para informar del éxito o fallo*

```
...
void divide(int op1, int op2, int &div, int &rem) {
// Divide op1 entre op2 y devuelve el cociente y el resto
    if (op2 != 0) {
        div = op1 / op2;
        rem = op1 % op2;
    }
    else cout << "Error: El divisor es cero" << endl;
}
```

En lugar de mostrar un mensaje de error,  
informar del fallo al punto de llamada.

Allí se decidirá qué hacer, si mostrar un mensaje  
o realizar alguna otra opción.

*El mensaje en la función nos condena a verlo.*



# Notificación de errores

## *Uso del resultado de la función para informar del éxito o fallo*

```
→ bool divide(int op1, int op2, int &div, int &rem) {
// Divide op1 entre op2 y devuelve el cociente y el resto
    bool ok;
    ok = op2 != 0;
    if (ok) {
        div = op1 / op2;
        rem = op1 % op2;
    }
    → return ok;
}
```

```
int main() {
    int cociente, resto;
    int num1 = 10; int num2 = 0;
    if (divide(num1, num2, cociente, resto))
        cout << num1 << " entre " << num2 << " da un cociente de "
            << cociente << " y un resto de " << resto << endl;
    else cout << "Error: El divisor es cero" << endl;
    return 0;
}
```

Ahora en main() podemos saber  
si todo ha ido bien o no.



# Notificación de errores

---

*Uso del resultado de la función para informar de errores*

```
typedef enum{ Correcto, Error1, ... } tError;
```

```
tError func(...) {  
    if (caso error1) return Error1;  
    else if (caso error2) return Error2;  
    else if ...  
    else { // no hay errores  
        ...  
        return Correcto;  
    }  
}
```



## Fundamentos de la programación

---

# Subprogramas y declaraciones



# Declaraciones locales y globales

---

## *Declaraciones locales*

### *De uso exclusivo del subprograma*

```
tipo nombre(parámetros) // Cabecera
{
    Declaraciones Locales // Cuerpo
}
```

- ✓ Declaraciones locales de tipos, constantes y variables  
Dentro del cuerpo del subprograma
- ✓ Parámetros declarados en la cabecera del subprograma  
Comunicación del subprograma con otros subprogramas



# Declaraciones locales y globales

---

## *Declaraciones en los programas*

- ✓ Globales: declarados fuera de los subprogramas.
- ✓ Locales: declarados en algún subprograma  
No se pueden declarar funciones locales

## *Ámbito y visibilidad de los identificadores Tema 3*

Cada función crea un nuevo bloque dentro del programa

- Globales: resto del programa  
Se conocen dentro de los subprogramas que siguen
- Locales: resto del subprograma  
No se conocen fuera del subprograma.
- Visibilidad de los identificadores:  
Los locales ocultan los externos homónimos.



# Declaraciones locales y globales

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

```
const int MAX = 100;
const double IVA = 21; } Globales
```

op de proc()  
es distinta  
de op de main()

```
...
void proc() {
    int op;
    const double IVA = 18; } Locales a proc()
    ... ← Se conocen MAX (global), op (local)
        e IVA (local que oculta la global)
}

int main() {
    int op;
    ... ← Locales a main()
    return 0; Se conocen MAX e IVA (globales)
            y op (local)
}
```



# Declaraciones locales y globales

## *Sobre el uso de datos globales en las funciones*

NO SE DEBEN USAR datos globales en subprogramas

✓ *¿Necesidad de datos externos?*

Define parámetros en el subprograma

Los datos externos se pasan como argumentos en la llamada

✓ Uso de datos globales en los subprogramas:

Riesgo de *efectos laterales*

Modificación inadvertida de esos datos afectando otros sitios

Excepciones:

✓ Constantes globales (valores inalterables)



# Declaraciones locales y globales

*El uso de datos globales en las funciones aumenta el riesgo de sufrir efectos laterales*

```
const int MAX = 100;
double ingresos;
...
void proc() {
    int op;
    ingresos = 0;
    ...
}

void main() {
    ingresos = 100.55;
    proc();
    ...
}
```

**Datos globales**  
Existen durante toda la ejecución del programa.

**Datos locales a proc()**  
Existen sólo durante la ejecución del subprograma  
**ERROR: Se olvida declarar la VARIABLE ingresos**

Compila porque se conocen MAX (global), op (local) e ingresos (global).

La variable ingresos cambia de valor de forma imprevista



# Fundamentos de la programación

## Prototipos



# Prototipos de las funciones

## ¿Qué funciones hay en el programa?

¿Podemos colocar las funciones en cualquier lugar del archivo:  
Antes de `main()`, después de `main()`?

El compilador necesita saber qué funciones se han declarado:

¿Son correctas las llamadas a funciones en el programa?

- ¿Existe la función?
- ¿Concuerdan los argumentos con los parámetros?

→ Incluir el prototipo de la función al principio del archivo.

Prototipo: Cabecera de la función terminada en ;

```
void dibujarCirculo();
void mostrarM();
void proc(double &a, int x);
int cuad(int x);
...
```



`main()` es la única función  
que no hay que prototipar.



# Prototipos de las funciones

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

bool divide(int op1, int op2, int &div, int &rem); // Prototipo
// Divide op1 entre op2 y devuelve el cociente y el resto.
// Devuelve false si el divisor es 0
int main() {
    int cociente, resto;
    if ( divide(22, 3, cociente, resto) )
        cout << 22 << " entre " << 3 << " da un cociente de "
            << cociente << " y un resto de " << resto << endl;
    }
    return 0;
}
```

Documenta los prototipos  
con un comentario

```
bool divide(int op1, int op2, int &div, int &rem){
    bool ok;
    ok = op2 != 0;
    if (ok) {
        div = op1 / op2;
        rem = op1 % op2;
    }
    return ok;
}
```

Si se ponen los prototipos  
No importa el orden  
en el que se coloquen  
las funciones.





# Prototipos de las funciones

---

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

void intercambia(double &valor1, double &valor2); // Prototipo
// Intercambia los valores de las dos variables.

void intercambia(double &valor1, double &valor2){
    double tmp; // Variable local (temporal)
    tmp = valor1;
    valor1 = valor2;
    valor2 = tmp;
}

int main() {
    double num1, num2;
    cout << "Valor 1: "; cin >> num1;
    cout << "Valor 2: "; cin >> num2;
    intercambia(num1, num2);
    cout << "Ahora el valor 1 es " << num1
         << " y el valor 2 es " << num2 << endl;
    return 0;
}
```

Documenta los prototipos  
con un comentario

Si se ponen los prototipos  
No importa el orden  
en el que se coloquen  
las funciones.



# Fundamentos de la programación

---

## Sobrecarga de funciones



# Sobrecarga de funciones

---

## *Igual nombre, distinto número o tipo de parámetros*

Podemos tener varias funciones con igual nombre pero con distinta lista de parámetros.

```
int abs(int n);
double abs(double n);
long int abs(long int n);
```

Se ejecutará la función que corresponda al tipo de argumento:

```
f(int(13)); // argumento int --> primera función
f(-2.3);    // argumento double --> segunda función
long int n = 3;
f(n);       // argumento long int --> tercera función
```



# Sobrecarga de funciones

---

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

void intercambia(int &x, int &y);
void intercambia(double &x, double &y);
void intercambia(char &x, char &y);

void intercambia(int &x, int &y)
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}

void intercambia(double &x, double &y)
{
    double tmp;
    tmp = x;
    x = y;
    y = tmp;
}

void intercambia(char &x, char &y) {
    char tmp;
    tmp = x;
    x = y;
    y = tmp;
}

int main() {
    int i1 = 3, i2 = 7;
    double d1 = 12.5, d2 = 35.9;
    char c1 = 'a', c2 = 'b';
    ...
    intercambia(i1, i2);
    intercambia(d1, d2);
    intercambia(c1, c2);
    ...
    return 0;
}
```



## Funciones de operador



## Funciones de operador

---

### *Notación infija (de operador)*

*operandoIzquierdo operador operandoDerecho*

*a + b*

Se ejecuta el operador + tomando como argumentos los operandos.

Los operadores se implementan como funciones:

*tipo operadorSímbolo(parámetros);*

Si es un operador monario sólo habrá un parámetro.

Si es binario habrá dos parámetros.

*Símbolo* es un símbolo de operador (uno o dos caracteres):

*+, -, \*, /, --, <<, %, ...*



# Funciones de operador

---

## *Ejemplo <<*

```
void escribir(ostream & arch, tDato dato){
    ...
}
void operator<<(ostream & arch, tDato dato){
    escribir(arch, dato);
}
...
ofstream sal; tdato dat;
...
escribir(sal, dat);
escribir(cout, dat);
cout << dat;
sal << dat;
cout << dat << endl; // ERROR !!
```

¡La implementación de estas dos funciones será exactamente la misma!



# Funciones de operador

---

## *Ejemplo <<*

```
ofstream & operator<<(ostream & arch, tdato dato){
    escribir(arch, dato);
    return arch;
}
ofstream sal; tdato dat;
...
escribir(sal, dat);
escribir(cout, dat);
sal << dat;
cout << dat;
sal << dat << endl; // CORRECTO !!
```

¡Hay que añadir una última instrucción para devolver el archivo!



## Argumentos implícitos



## Argumentos implícitos

---

### *Valores predeterminados para los parámetros por valor*

El valor por defecto para un parámetro se pone tras un = a continuación del nombre del parámetro:

```
void proc(int i = 1); // sólo se usa en compilación
```

Ahora, si no proporcionamos argumento en la llamada, el parámetro toma ese valor predeterminado.

```
proc(12);
```

```
proc(); —————> el compilador completa a proc(1);
```

Todos los parámetros que se declaren con argumentos implícitos deben encontrarse al final de la lista de parámetros:

```
void f(int i, int j = 2, int k = 3); // CORRECTO
```

```
void f(int i = 1, int j, int k = 3); // INCORRECTO
```



ERROR DE COMPILACIÓN



# Argumentos implícitos

---

```
void f(int i, int j = 2, int k = 3);
```

...

```
f(13);    // el compilador completa a f(13, 2, 3)
f(5, 7);  // el compilador completa a f(5, 7, 3)
f(3, 9, 12);
```



Declara los argumentos implícitos en el prototipo, y documenta, con un comentario, su funcionalidad



## Fundamentos de la programación

---

### La función main()



# Lo que devuelve main()

---

## *Uso del resultado de la función para informar de errores*

La función `main()` devuelve al sistema operativo un código de terminación de programa.

- 0: *Todo OK*
- Distinto de 0: *¡Ha habido un error!*

Si la ejecución llega al final de la función, todo OK:

```
int main(int argc, char *argv[]){  
    ...  
    return 0; // Fin del programa  
}
```



# Parámetros de main()

---

## *Comunicación con el sistema operativo*

No es obligatorio declarar los parámetros de la función `main()` si no se van a utilizar.

```
int main(int argc, char *argv[]){ ... return 0;}
```

Permiten obtener datos proporcionados al ejecutar el programa:

```
C:\>prueba cad1 cad2 cad3
```

Ejecuta `prueba.exe` con tres argumentos (cadenas de caracteres).

Parámetros de `main()`:

- `argc`: número de argumentos que se proporcionan (4 en el ejemplo; el primero (`argv[0]`) siempre es el nombre del programa con su ruta).
- `argv`: array con las cadenas que se proporcionan como argumentos.








## *Licencia CC (Creative Commons)*

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:

-  Reconocimiento (*Attribution*):  
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.
-  No comercial (*Non commercial*):  
La explotación de la obra queda limitada a usos no comerciales.
-  Compartir igual (*Share alike*):  
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Pulsa en la imagen de arriba a la derecha para saber más.



Fundamentos de la programación: Más sobre tipos e instrucciones

Página 62



## Fundamentos de la programación

# Precondiciones Poscondiciones



Fundamentos de la programación: La abstracción procedimental

Página 63





# Precondiciones y postcondiciones

---

## *Integridad de los subprogramas*

Un subprograma puede establecer una serie de condiciones que se deben dar antes de comenzar su ejecución → Precondiciones

- ✓ Será responsabilidad del que llame al subprograma garantizar que se satisfacen las precondiciones.

El subprograma puede, además, garantizar otra serie de condiciones que se darán cuando termine su ejecución → Postcondiciones

- ✓ En el punto de llamada se pueden dar por garantizadas las postcondiciones.

Aserciones: Condiciones que si no se cumplen interrumpen la ejecución.

`assert()`



# Precondiciones y postcondiciones

---

## *Aserciones como precondiciones*

Por ejemplo, no realizaremos conversiones de valores negativos:

```
double pulgACm(double pulg)
{
    assert(pulg > 0);
    double cmPorPulg = 2.54;
    return pulg * cmPorPulg;
}
```

La función tiene una precondición: `pulg` debe ser positivo.

`assert(pulg > 0);` interrumpirá la ejecución si la condición no es cierta.



# Precondiciones y postcondiciones

---

## Aserciones como precondiciones

Será responsabilidad del punto de llamada garantizar la precondición:

```
int main() {
    double valor;
    int op;
    do {
        op = menu();
        switch (op) {
            case 1:
                cout << "Pulgadas: "; cin >> valor;
                if (valor < 0)
                    cout << "¡Valor no válido! (Ha de ser positivo.) " << endl;
                else // Se cumple la precondición.
                    cout << "Equivale a " << pulgACm(valor) << " cm." << endl;
                break;
            ...
        }
    } while (op != 0);
}
```

Si no: error de programación.



# Precondiciones y postcondiciones

---

## Aserciones como postcondiciones

Una función puede garantizar ciertas condiciones al terminar:

```
int menu()
{
    cout << "1 - Pulgadas a Cm." << endl;
    cout << "2 - Libras a Gr." << endl;
    cout << "3 - Fahrenheit a °C" << endl;
    cout << "4 - Euros a Pts." << endl;
    cout << "0 - Salir" << endl;
    int op;
    do { cout << "Elige: "; cin >> op; } while ((op < 0) || (op > 4));
    assert ((op >= 0) && (op <= 4));
    return op;
}
```

La función debe asegurarse de que se cumpla la condición.

Si no: error de programación.



## Diseño descendente (un ejemplo)



## Diseño descendente

---

### *Refinamientos sucesivos*

Especificación inicial (Paso 0).-

*Desarrollar un programa que haga operaciones de conversión de cantidades hasta que el usuario decida que no quiere hacer más.*

Análisis y diseño aumentando el nivel de detalle en cada paso.

*¿Qué operaciones de conversión?*

- ★ *Pulgadas a centímetros*
- ★ *Libras a gramos*
- ★ *Grados Fahrenheit a centígrados*
- ★ *Galones a litros*

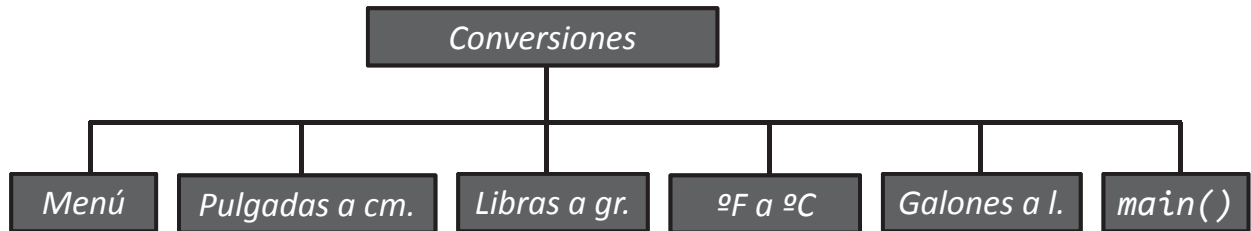


# Diseño descendente

---

## Refinamientos sucesivos

### Paso 1.- Diseño



# Diseño descendente

---

## Refinamientos sucesivos

### Paso 2.- Especificación de cada subprograma

- ★ *Menú*: `int menu()`  
Mostrar las cuatro opciones más una para salir.  
Validar la entrada y devolver la elegida.
- ★ *Pulgadas a centímetros*: `double pulgACm(double pulg)`  
Devolver el equivalente en centímetros del valor en pulgadas.
- ★ *Libras a gramos*: `double lbgAgr(double libras)`  
Devolver el equivalente en gramos del valor en libras.
- ★ *Grados Fahrenheit a centígrados*: `double grFAGrC(double grF)`  
Devolver el equivalente en centígrados del valor en Fahrenheit.
- ★ *Galones a litros*: `double galALtr(double galones)`  
Devolver el equivalente en pesetas del valor en euros.
- ★ *Programa principal* (`int main()`)



# Diseño descendente

---

## Refinamientos sucesivos

### Paso 3.- Algoritmos detallados de cada función → Programación

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

// Prototipos
int menu();
double pulgACm(double pulg);
double lbAGr(double libras);
double grFAgrC(double grF);
double galALtr(double galones);

int menu() {
    cout << "1 - Pulgadas a Cm." << endl;
    cout << "2 - Libras a Gr." << endl;
    cout << "3 - Fahrenheit a °C" << endl;
    cout << "4 - Galones a L." << endl;
    cout << "0 - Salir" << endl;
    int op;
    cout << "Elige: "; cin >> op;
    return op;
}
```



# Diseño descendente

---

```
int main() {
    double valor;
    int op;
    do {
        op = menu();
        switch (op) {
            case 1:
                cout << "Pulgadas: "; cin >> valor;
                cout << "Son " << pulgACm(valor) << " cm." << endl; break;
            case 2:
                cout << "Libras: "; cin >> valor;
                cout << "Son " << lbAGr(valor) << " gr." << endl; break;
            case 3:
                cout << "Grados Fahrenheit: "; cin >> valor;
                cout << "Son " << grFAgrC(valor) << " °C" << endl; break;
            case 4:
                cout << "Galones: "; cin >> valor;
                cout << "Son " << eurosAPts(valor) << " ltr." << endl; break;
        }
    } while (op != 0);
    return 0;
}
```



# Diseño descendente

---

```
double pulgACm(double pulg) {  
    const double cmPorPulg = 2.54;  
    return pulg * cmPorPulg;  
}  
  
double lbAGr(double libras) {  
  
    const double grPorLb = 453.6;  
    return libras * grPorLb;  
}  
  
double grFAgrC(double grF) {  
    return ((grF - 32) * 5 / 9);  
}  
  
double galALtr(double galones) {  
    const double ltrPorGal = 4.54609;  
    return galones * ltrPorGal;  
}
```

