

TEMA 2.

DISSENY

DESCENDENT.

TEORIA.

FONAMENTS DE PROGRAMACIÓ II

CURS: 2024-25

GRAUS: GEI, GEI-Biotec

UNIVERSITAT ROVIRA I VIRGILI

Any fool can write code that a computer can understand.

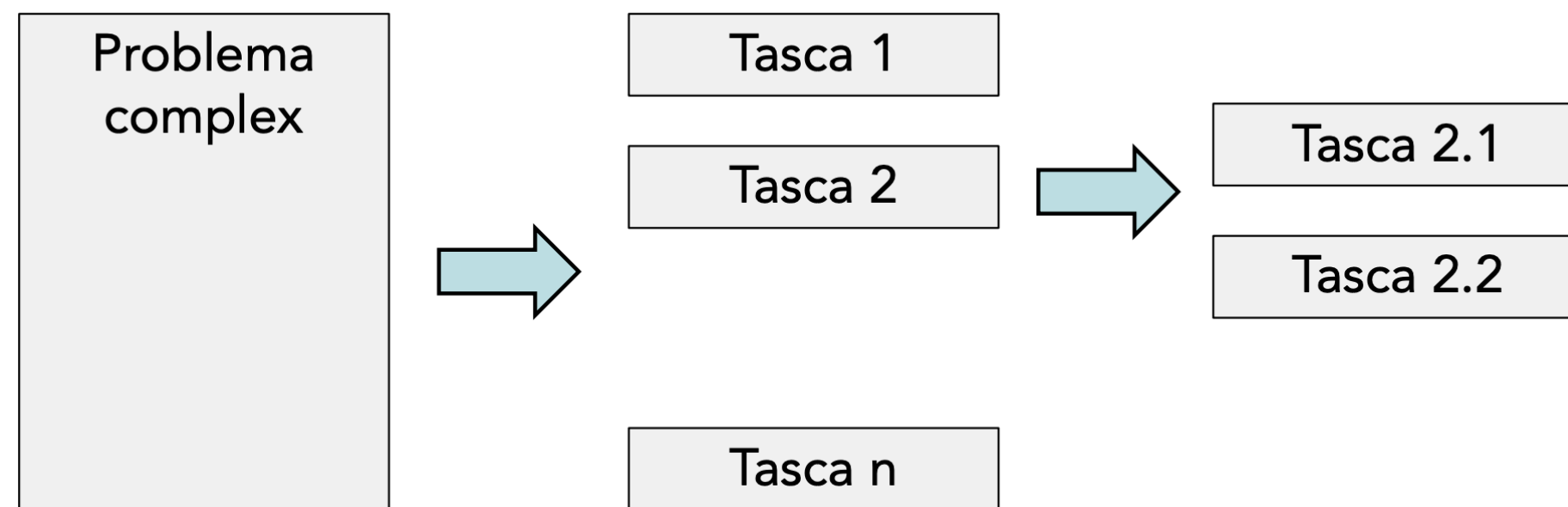
Good programmers write code that humans can understand.

Martin Fowler

COMENCEM DES DEL COMENÇAMENT

La necessitat dels procediments

- Fins ara hem resolt problemes simples.
- Si volem resoldre problemes més complexos, generarem molt de codi, i llavors és necessari l'ús de procediments.
- A l'hora de resoldre problemes, farem servir el **disseny descendent**



- Al resoldre problemes complexos, les instruccions que resolen parts del problema general les podem **encapsular** en **procediments**.

AVANTATGES D'USAR PROCEDIMENTS

Estructurarem millor els programes, per facilitar-ne el seguiment i la seva comprensió (**llegibilitat**).

Reaprofitarem el codi que ja està implementat i testejat (**re-usabilitat**).

Localitzarem i esmenarem errors amb més facilitat (**depurabilitat**).

Accions vs. funcions

Funcions

Procediments que retornen un resultat

Hem d'assignar el resultat de la funció a una variable del mateix tipus que el seu retorn

Definició:

```
funció suma (a: enter, b: enter)
retorna enter és
var
    s: enter;
fvar
inici
    s := a + b;
    retorna s;
ffunció
```

Crida:

```
x := suma(3,5);
```

Accions

Procediments que **no** retornen un resultat

Són procediments que escriuen a pantalla, a fitxer, o bé que modifiquen els propis paràmetres d'entrada que reben

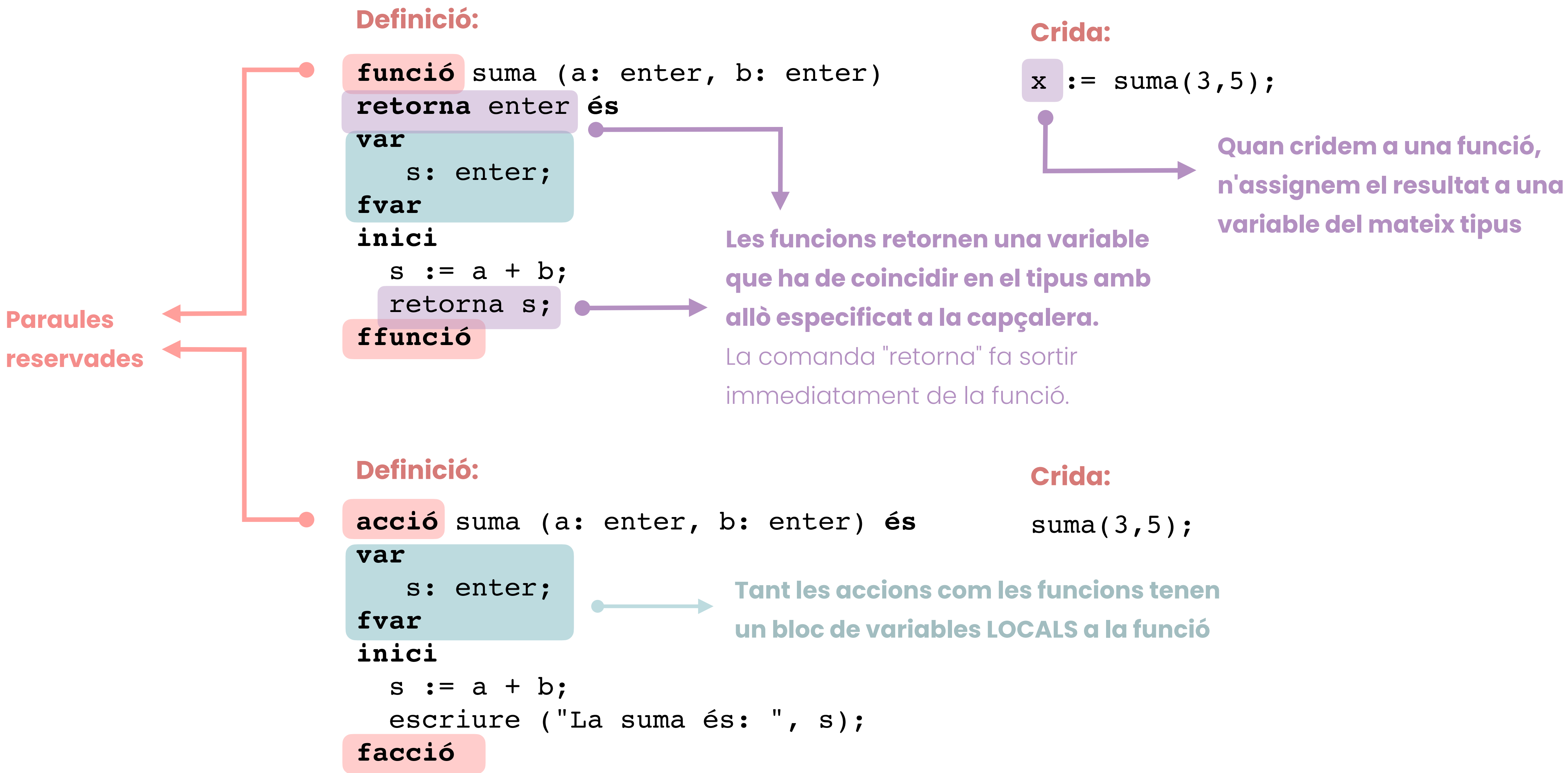
Definició:

```
acció suma (a: enter, b: enter) és
var
    s: enter;
fvar
inici
    s := a + b;
    escriure ("La suma és: ", s);
facció
```

Crida:

```
suma(3,5);
```

Accions vs. funcions: recordatori de pseudocodi



Pas de paràmetres

Pas per valor vs. pas per referència


- Distinció **molt important**. En programació, distingim entre dos mecanismes de pas de paràmetres:

Pas per valor

Quan passem una variable, el procediment rep **el valor** d'aquella variable.

Els paràmetres **no** són modificables

Qualsevol canvi que fem dins el procediment no tindrà efecte quan sortim del procediment



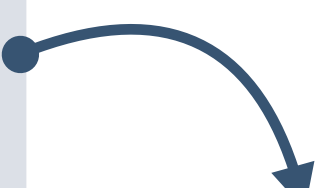
```
x := 1;  
y := 2;  
...  
intercanvia (x, y);  
...  
$ x val 1  
$ y val 2
```

Pas per referència

Quan passem una variable, el procediment rep **la referència** (adreça de memòria) d'aquella variable.

Els paràmetres són modificables

Com que el procediment té accés a l'adreça "on viu" la variable, pot modificar-la. Els canvis que hi fem es mantenen al sortir del procediment.



```
x := 1;  
y := 2;  
...  
intercanvia (x, y);  
...  
$ x val 2  
$ y val 1
```


Pas de paràmetres

Pas per valor vs. pas per referència

- Distinció **molt important**. En programació, distingim entre dos mecanismes de pas de paràmetres:

@5000	1	x
@5004	2	y

Memòria de 32 bits

Pas per valor

En un llenguatge que fa servir pas per valor, el procediment *intercanvia* rep els valors de les variables *x* i *y*, és a dir:

1
2

```
x := 1;  
y := 2;  
...  
intercanvia (x, y);  
...  
$ x val 1  
$ y val 2
```

Pas per referència

En un llenguatge que fa servir pas per referència, el procediment *intercanvia* rep les referències (adreces) de les variables *x* i *y*, és a dir:

5000
5004

```
x := 1;  
y := 2;  
...  
intercanvia (x, y);  
...  
$ x val 2  
$ y val 1
```


Pas de paràmetres

Pas per valor vs. pas per referència

- Distinció **molt important**. En programació, distingim entre dos mecanismes de pas de paràmetres:

Pas per valor

pass by value



fillCup()

Pas per referència

pass by reference



fillCup()

Pas de paràmetres

Pas per valor vs. pas per referència

Cada llenguatge de programació gestiona el pas de paràmetres de manera diferent

En llenguatge C...

... es fa servir pas per valor

En C **sempre es fa servir pas per valor**

¿



?

Per simplificar, a FPI se us ha dit que en C els paràmetres es passen per valor, excepte en algunes ocasions, com les taules.

Això és una sobresimplificació, i és poc acurat.
Anem a entendre què passa realment.

En C **sempre** es fa servir pas per valor

- En qualsevol procediment (acció o funció), els paràmetres d'entrada són còpies del valor que tinguin en el moment de la crida.
- Això significa que qualsevol canvi que es faci sobre aquests paràmetres dins de la funció no afecta les variables originals fora de la funció (pas per valor).

```
double mitjana(double v[], int n) {  
    double suma = 0.0;  
  
    for(int i = 0; i < n; i++) {  
        suma = suma + v[i];  
    }  
  
    return suma / n;  
}
```

FUNCIÓ "MITJANA"

Si dins d'aquesta funció jo modifico la variable "n", aquest canvi no es veurà reflectit al programa principal.

Per què? Doncs perquè per modificar la variable "n", he de saber "on viu", és a dir, la seva adreça de memòria, i no la sé.

Només sé el seu valor (e.g. 5).

@1000

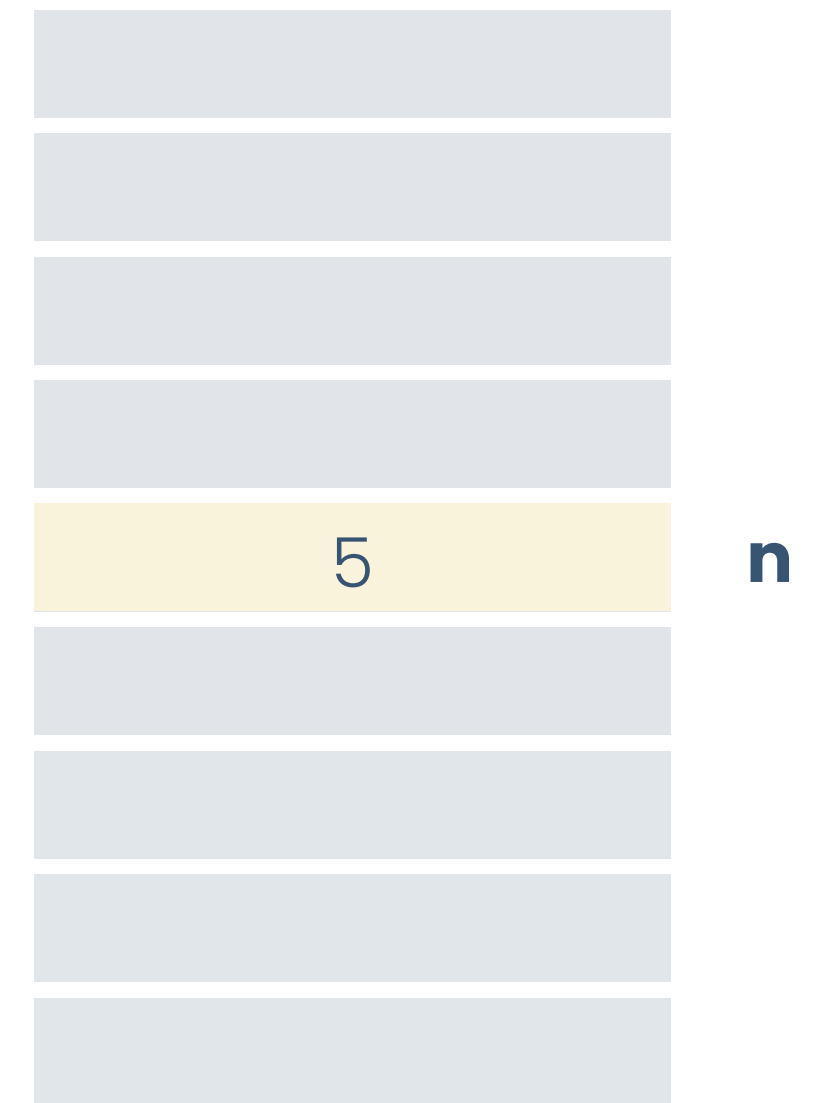
5

n

En C **sempre** es fa servir pas per valor

- En qualsevol procediment (acció o funció), els paràmetres d'entrada són còpies del valor que tinguin en el moment de la crida.
- Això significa que qualsevol canvi que es faci sobre aquests paràmetres dins de la funció no afecta les variables originals fora de la funció (pas per valor).

@1000



```
double mitjana(double v[], int n) {  
    double suma = 0.0;  
    printf("Què és v? %p\n", v);  
    for(int i = 0; i < n; i++) {  
        suma = suma + v[i];  
    }  
  
    return suma / n;  
}
```

En canvi, si modifico la variable **v**, que és una taula, aquests canvis sí que es veuran al programa principal.

Per què? No havíem quedat que en C només es feia pas per valor?

Què estic rebent però, quan em passen una taula?

Què és v? 0x7fff62337140

Estic rebent la seva adreça de memòria **PER VALOR!!!**
Ara bé, com que tinc l'adreça, puc "anar allà on viu", i modificar-la

FUNCIÓ "MITJANA"

En C **sempre** es fa servir pas per valor

```
double mitjana(double v[], int n) {  
    double suma = 0.0;  
    printf("Què és v? %p\n", v);  
    for(int i = 0; i < n; i++) {  
        suma = suma + v[i];  
    }  
  
    return suma / n;  
}
```

FUNCIÓ "MITJANA"

```
#include <stdio.h>  
int main() {  
    double valors[] = {1.5, 2.0, 3.5, 4.0, 5.5};  
    int n = 5;  
    double resultat;  
  
    resultat = mitjana(valors, n);  
  
    printf("La mitjana és: %.2f\n", resultat);  
    return 0;  
}
```

PROGRAMA PRINCIPAL

En C **sempre** es fa servir pas per valor

```
resultat = mitjana(valors, n);
```

=

```
resultat = mitjana(0x7fff62337140, 5);
```

Tant l'adreça 0x7fff62337140 com el número 5 es passen **PER VALOR**. Ara bé, com que tinc l'adreça de la variable v, puc canviar-ne els continguts, ***emulant un pas de paràmetres per referència***.

```
double mitjana(double v[], int n) {  
    double suma = 0.0;  
  
    for(int i = 0; i < n; i++) {  
        suma = suma + v[i];  
    }  
  
    return suma / n;  
}
```

FUNCIÓ "MITJANA"

```
#include <stdio.h>  
int main() {  
    double valors[] = {1.5, 2.0, 3.5, 4.0, 5.5};  
    int n = 5;  
    double resultat;  
  
    resultat = mitjana(valors, n);  
  
    printf("La mitjana és: %.2f\n", resultat);  
    return 0;  
}
```

PROGRAMA PRINCIPAL

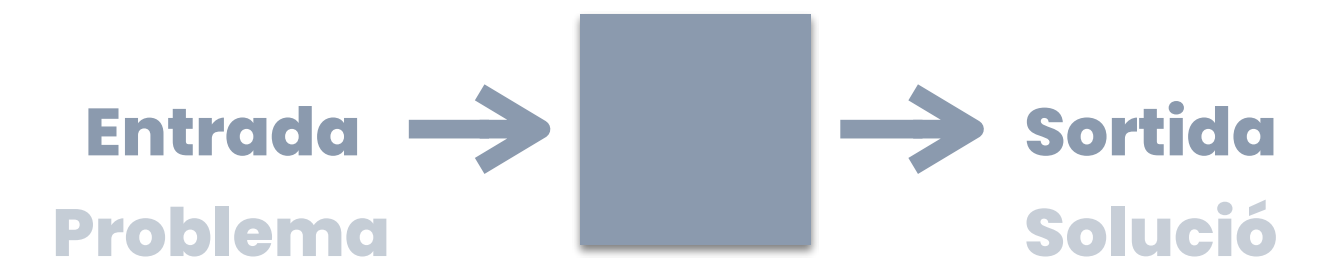
ESPECIFICACIÓ I DISSENY DE PROCEDIMENTS

Especificació de procediments: precondition i postcondició

Precondició

- A l'hora de dissenyar un procediment, hem de decidir quins són els paràmetres d'entrada i els paràmetres de sortida.

Exemple: funció que calcula l'àrea d'un rectangle



Entrada: dimensions (base i altura) del rectangle en centímetres

Sortida: L'àrea del rectangle en centímetres quadrats calculat com: $\text{Àrea} = \text{base} * \text{alçada}$

- A més, necessitem reflexionar sobre quines condicions han de complir els paràmetres d'entrada perquè el procediment funcioni correctament. Aquestes condicions s'anomenen **precondicions**.

En l'exemple anterior:

Precondició: els paràmetres d'entrada han de ser reals positius o zero, és a dir, $\text{base} \geq 0$ i $\text{alçada} \geq 0$

Especificació de procediments: preconditionió i postcondició

Postcondició

- De la mateixa manera, la condició que expressa les propietats dels resultats de l'algorisme o procediment s'anomena **postcondició**. No cal que especifiquem com s'aconsegueixen aquests resultats.

En l'exemple anterior:

Postcondició: *el paràmetre de sortida serà un nombre real positiu resultant de calcular $base * alçada$.*

Si $base$ o $alçada$ són zero, el resultat ha de ser zero.

Especificació Pre/post

- El parell format per la preconditionió i la postcondició s'anomena **especificació pre/post**
- Donar la preconditionió i la postcondició conjuntament especifica per complet què fa el programa o funció
- És convenient incloure a la documentació del procediment quina és la especificació pre/post

Especificació de procediments: precondition i postcondició

Exemples

- Quina és la precondition i postcondició dels procediments següents?

Funció que calcula el factorial d'un nombre

Precondició: El nombre $x \geq 0$

Postcondició: Retorna **$x!$** . Si x era zero, es retorna 1.

Funció que comprova si una lletra és vocal

Precondició: El paràmetre d'entrada és una lletra majúscula o minúscula en el rang 'A' ... 'Z' o 'a' ... 'z'.

Postcondició: Retorna cert si és una vocal o fals en cas contrari

Funció que inverteix un vector d'enters

Precondició: Té dos paràmetres d'entrada, el vector (que no pot ser NULL), i longitud, un enter que ha de ser positiu o zero

Postcondició: Després de l'execució, els elements de la matriu han d'estar invertits en ordre. Si longitud és 1 o 0, la matriu no canviarà. El vector ha de contenir els mateixos elements després de la funció, però en ordre invers.

Especificació de procediments: precondition i postcondició

Què passa si no respectem la precondition?

- A l'exemple del factorial, què passa si cridem al procediment amb valors no permesos?

Funció que calcula el factorial d'un nombre n

Precondició: El nombre $n \geq 0$

Postcondició: Retorna el factorial de n

```
factorial(0);  
factorial(5);  
factorial(-4);
```

- Les crides primera i segona tenen un resultat definit. Però i la tercera?
- Quan cridem a una funció amb paràmetres que no compleixen la precondition, la funció pot fallar o donar un resultat incorrecte.



De qui és la responsabilitat de
comprovar la precondition?

Especificació de procediments: preconditionió i postcondició

De qui és responsabilitat de comprovar la preconditionió?

- Essent estrictes amb el terme de "precondició", se n'hauria d'encarregar el programa principal. La funció només s'ha d'encarregar de donar el resultat correcte donades unes precondicions d'entrada.
- A l'hora de la veritat, tenim dues possibilitats:
 - O bé la comprova qui crida la funció (ja sigui el programa principal o una altra funció)
 - O bé se'n fa càrrec la pròpia funció.



La funció no comprova res,
només fa els càlculs

OPCIÓ 1: El programa principal
comprova la precondició

```
long factorial(int n) {  
    long resultat = 1;  
    for (int i = 1; i <= n; i++) {  
        resultat *= i;  
    }  
    return resultat;  
}
```

```
int main() {  
    int nombre = -5;  
  
    // El programa principal comprova la precondició  
    if (nombre < 0) {  
        printf("Error: No es pot calcular el factorial d'un nombre negatiu.\n");  
    } else {  
        printf("El factorial de %d és %lu\n", nombre, factorial(nombre));  
    }  
    return 0;  
}
```

El programa principal només
crida a la funció si sap que es
compleix la precondició

Abans de fer els càlculs, la funció
comprova la precondition

OPCIÓ 2: La funció comprova la
precondició

```
long factorial(int n) {  
    if (n < 0) {  
        printf("Error: No es pot calcular el factorial d'un nombre negatiu.\n");  
        return 0; // Codi d'error  
    }  
  
    long resultat = 1;  
    for (int i = 1; i <= n; i++) {  
        resultat *= i;  
    }  
    return resultat;  
}
```

El programa principal crida la
funció sense comprovació prèvia

```
int main() {  
    int nombre = -5;  
  
    // El programa principal només crida la funció  
    long resultat = factorial(nombre);  
    // Comprova si la funció ha retornat un valor vàlid  
    if (resultat != 0) {  
        printf("El factorial de %d és %lu\n", nombre, resultat);  
    }  
    return 0;  
}
```

El programa principal avalua el valor de
retorn de la funció, per si és un codi d'error

Especificació de procediments: precondition i postcondició

De qui és responsabilitat de comprovar la precondition?

- Essent estrictes amb el terme de "precondition", se n'hauria d'encarregar el programa principal. La funció només s'ha d'encarregar de donar el resultat correcte donada unes precondicions d'entrada.
- A l'hora de la veritat, tenim dues possibilitats, o bé la comprova el programa principal, o bé se'n fa càrrec la funció.

La funció és la responsable de comprovar la precondition

- En el nostre cas, per fer funcions més robustes i autocontingudes, serà **la funció la que tindrà la responsabilitat d'assegurar-se que els paràmetres compleixen la precondition**.
- Si **la funció** detecta que no es compleix, **haurà de retornar un codi d'error**.
- **El programa principal** se n'haurà d'encarregar de **comprovar que el que retorna la funció no sigui un codi d'error**.



➡ Veurem com gestionar els casos d'error al laboratori

Unit testing

La importància del testeig

- Un avantatge dels procediments és que permeten testejar una part del codi de forma aïllada del programa complet.
- El Unit Testing és un mètode per verificar el funcionament correcte d'unitats individuals de codi, generalment funcions, de manera aïllada del sistema. L'objectiu és assegurar que cada funció es comporti segons l'esperat en diferents escenaris i condicions.

Unit testing

Quines proves he de fer a la funció?

Test*	Significat	Exemple: En una funció que inverteix un string...
Casos base	Les proves de fum. Són els casos típics o normals per als quals la funció ha estat dissenyada	Comprovar que si li passem un string { 'H' , 'O' , 'L' , 'A' , '\0' }, el resultat és { 'A' , 'L' , 'O' , 'H' , '\0' }
Casos límit	Són els casos on les entrades es troben en els límits extrems de l'interval d'operació de la funció	Comprovar que si li passem una cadena amb un sol caràcter { 'A' , '\0' }, la funció no canvia la cadena (el resultat hauria de ser el mateix).
Casos nuls	Són els casos en què les entrades són nul·les o buides.	Comprovar què passa si la cadena està buida, és a dir, només conté el caràcter de terminació ({ '\0' }).
Casos invàlids	Són els casos en què les entrades no són vàlides o estan fora del rang permès	Comprovar què passa si rebem una cadena que no té el caràcter de terminació '\0'

**No tots aquests tests són aplicables a totes les funcions*

Resumint

Passos per dissenyar un procediment

1. Entendre el problema

- Defineix clarament què ha de fer el procediment i quines dades necessita (inputs) i retorna (outputs).

2. Especificar precondicions i postcondicions

- Precondicions: Quines condicions han de complir-se abans d'executar la funció?
- Postcondicions: Què ha de garantir la funció després de la seva execució?

3. Dissenyar i implementar el procediment

- Escriu el codi que fa que la funció faci el que ha de fer.

5. Unit testing

- Fes proves per als casos base (inputs normals), casos límit (inputs al límit del rang d'operació), casos nuls (inputs buits o nuls), i casos invàlids.
- Comprova que el teu programa no falla en cap d'aquests casos, i que si de cas, retorna un codi d'error quan hi ha un error.
- Si el procediment no passa alguna de les proves, torna al pas (4) i modifica'l

6. Integració

- Integra el teu procediment dins del programa principal i comprova que funciona dins d'aquest.

DISSENY D'UN PROGRAMA MODULAR

Disseny de programes seguint el Top-Down design

Com estructurar un programa tenint en compte el disseny descendent?

- Una cosa és saber com dissenyar i implementar una funció, i l'altra és saber com dividir un problema molt gran en altres problemes més petits.

Passos:

1. Identificar l'objectiu global
2. Dividir l'objectiu global en objectius més petits (que seran les nostres funcions)
3. Dissenyar aquestes funcions (i testejar-les per separat!)
4. Unir totes les peces





Disseny de programes seguint el Top-Down design

Exemple 1: Un programa que implementi el joc pedra-paper-tisora

1. Identificar l'objectiu global

- Primer de tot, hem de saber què ha de fer el programa. Volem que el programa demani a l'usuari que triï entre pedra, paper i tisora i jugui contra l'ordinador. Per tant, l'usuari triarà una opció, l'ordinador en triarà una altra, i després decidirem qui ha guanyat.

2. Dividir-ho en objectius més petits

- Necessitem un procés que demani a l'usuari quina opció tria (pedra, paper o tisora). I que comprovi que l'opció triada és vàlida.  **get_opcio_usuari**
- Necessitem un procés que determini amb quina opció juga l'ordinador.  **get_opcio_ordinador**
- Necessitem un procés que determini qui dels dos ha guanyat.  **determina_guanyador**
- Necessitem un procés que gestioni tot això.  **jugar_partida**

Disseny de programes seguint el Top-Down design

Exemple 1: Un programa que implementi el joc pedra-paper-tisora

3. Dissenyar les funcions

get_opcio_usuari

- En aquesta funció imprimirem un menú d'opcions on demanarem a l'usuari que triï entre pedra, paper o tisora. Si l'usuari tria una opció no correcta, tornarem a demanar-li*.
- Paràmetres d'entrada: cap.
- Paràmetres de sortida: l'opció que ha triat.

**Una altra decisió de disseny possible seria sortir del programa si l'usuari tria una opció incorrecta.*

get_opcio_ordinador

- En aquesta funció generarem un nombre aleatori entre 1 i 3 que representi {pedra, paper, tisora}
- Paràmetres d'entrada: cap
- Paràmetres de sortida: la tria de l'ordinador

determina_guanyador

- Aquesta funció rebrà l'opció de l'usuari i l'opció de l'ordinador, i farà servir la lògica de les regles del joc per decidir qui ha guanyat, i ho imprimirà per pantalla.
- Paràmetres d'entrada: opció d'usuari, opció ordinador
- Paràmetres de sortida: cap (escriu per pantalla)

jugar_partida

- Aquesta funció cridarà a totes les anteriors. Cada cop que la cridem, jugarem a una nova partida.

Disseny de programes seguint el Top-Down design

Exemple 1: Un programa que implementi el joc pedra-paper-tisora

3. Dissenyar les funcions

```
int get_opcio_usuari() {  
    int op;  
  
    printf("Tria una opció: \n");  
    printf("1 - Pedra\n");  
    printf("2 - Paper\n");  
    printf("3 - Tisores\n");  
    scanf("%d", &choice);  
  
    while (op < 1 || op > NUM_OPCIONS) {  
        printf("Opció invàlida. Torna a triar: ");  
        scanf("%d", &op);  
    }  
    return op;  
}
```

```
int get_opcio_ordinador() {  
    // Més 1 per estar en rang [1..3]  
    return rand() % NUM_OPCIONS + 1;  
}
```

↓
Generem un nombre aleatori entre 1 i 3

→ Demanem a l'usuari que introdueixi un nombre de l'1 al 3, que codifica les opcions

Disseny de programes seguint el Top-Down design

Exemple 1: Un programa que implementi el joc pedra-paper-tisora

3. Dissenyar les funcions

```
#define PEDRA 1
#define PAPER 2
#define TISORES 3
#define NUM_OPCIONS 3
```

↓

Per no referir-me a cada opció com a 1, 2, i 3, defineixo una macro per poder fer servir el nom PEDRA, PAPER i TISORES per referir-me a l'1, 2 i 3.

```
char* get_nom_opcio(int codi_opcio){
    switch(codi_opcio){
        case PEDRA:
            return "Pedra";
            break;
        case PAPER:
            return "Paper";
            break;
        case TISORES:
            return "Tisores";
            break;
        default:
            return "Unknown option";
    }
}
```

→

Decideixo crear una nova funció auxiliar, que donat un codi numèric (1, 2 o 3) em retorni l'string de l'opció. Ho faig per facilitar la impressió per pantalla.

Disseny de programes seguint el Top-Down design

Exemple 1: Un programa que implementi el joc pedra-paper-tisora

3. Dissenyar les funcions

```
void determina_guanyador(int usuari, int ordinador) {  
    printf("Jugador: %s, Ordinador: %s\n", get_nom_opcio(usuari), get_nom_opcio(ordinador));  
  
    if (usuari == ordinador) {  
        printf("Empat!\n");  
    } else if ((usuari == PEDRA && ordinador == TISORES) ||  
               (usuari == PAPER && ordinador == PEDRA) ||  
               (usuari == TISORES && ordinador == PAPER)) {  
        printf("L'usuari guanya!\n");  
    } else {  
        printf("L'usuari perd!\n");  
    }  
}
```

Crido la funció anterior per poder escriure el nom de l'opció

Aquí s'implementa la lògica del joc. Gràcies a definir-ho com macros aconseguixo codi més legible.

Disseny de programes seguint el Top-Down design

Exemple 1: Un programa que implementi el joc pedra-paper-tisora

3. Dissenyar les funcions

```
void juga_partida(){
    int opcio_usuari = get_opcio_usuari();
    printf("*** L'usuari ha triat: %s ***\n", get_nom_opcio(opcio_usuari));
    int opcio_ordinador = get_opcio_ordinador();
    printf("*** L'ordinador ha triat: %s ***\n", get_nom_opcio(opcio_ordinador));
    determina_guanyador(opcio_usuari, opcio_ordinador);
}
```

La funció juga_partida crida a les dues funcions anteriors i a determina_jugador

```
int main() {
    srand(time(NULL));
    juga_partida();
    return 0;
}
```

Crido a la funció juga_partida des del main. Podria fins i tot fer-ho diverses vegades, demanant a l'usuari quantes partides vol jugar...etc

4. Unir totes les peces



➡ Trobareu l'exemple complet al moodle: pedra_paper_tisores.c
Al laboratori practicarem amb un altre exemple.

COM *NO* FER SERVIR PROCEDIMENTS

Fer servir les funcions com a calaixos de sastre

El vostre codi sovint és un codi desordenat, amb repeticions, i codi innecesari:



METÀFORA DEL VOSTRE CODI

- A l'hora de fer servir funcions, molts repartiu el codi que abans estava al programa principal, en funcions, de manera arbitrària



METÀFORA D'UNA FUNCIÓ

Com fer servir les funcions correctament?

Una funció ha de fer una sola cosa.

- No barregeu diferents tipus de coses dins una mateixa funció. Una funció ha de tenir una única responsabilitat clarament definida. Si una funció fa més d'una cosa, és un senyal que cal dividir-la en funcions més petites i especialitzades.



METÀFORA D'UNA FUNCIO

```
#include <stdio.h>

int calcula_area_rectangle() {
    int ample, alt;
    printf("Introdueix l'ample: ");
    scanf("%d", &ample);
    printf("Introdueix l'alt: ");
    scanf("%d", &alt);
    return ample * alt; // Calcula l'àrea
}
```

Aquesta funció fa dues coses:
demanar les mides a l'usuari, i
calcular-ne el resultat.

A més, les dues coses que fa són de naturalesa diferent: demanar input a l'usuari, i fer un càlcul

Com fer servir les funcions correctament?

Una funció ha de fer una sola cosa.

- No barregeu diferents tipus de coses dins una mateixa funció. Una funció ha de tenir una única responsabilitat clarament definida. Si una funció fa més d'una cosa, és un senyal que cal dividir-la en funcions més petites i especialitzades.



METÀFORA D'UNA FUNCIO

```
int obtenir_valor(const char* missatge) {
    int valor;
    printf("%s", missatge);
    scanf("%d", &valor);
    return valor;
}

int calcula_area(int ample, int alt) {
    return ample * alt;
}
```

Les dues accions que fèiem abans ara són en dues funcions diferents. Cadascuna només s'encarrega d'una cosa:

- `obtenir_valor()` només gestiona l'entrada de dades.
- `calcula_area()` només s'encarrega de calcular l'àrea.

Com fer servir les funcions correctament?

Una funció ha de fer una sola cosa.

- No barregeu diferents tipus de coses dins una mateixa funció. Una funció ha de tenir una única responsabilitat clarament definida. Si una funció fa més d'una cosa, és un senyal que cal dividir-la en funcions més petites i especialitzades.



METÀFORA D'UNA FUNCió



```
#include <stdio.h>

int main() {
    int ample = obtenir_valor("Introdueix l'ample: ");
    int alt = obtenir_valor("Introdueix l'alt: ");
    int area = calcula_area(ample, alt);
    printf("L'àrea del rectangle és: %d\n", area);
    return 0;
}
```

Les dues accions que fèiem abans ara són en dues funcions diferents. Cadascuna només s'encarrega d'una cosa:

- `obtenir_valor()` només gestiona l'entrada de dades.
- `calcula_area()` només s'encarrega de calcular l'àrea.

Com fer servir les funcions correctament?

Una funció ha de fer una sola cosa.

- No barregeu diferents tipus de coses dins una mateixa funció. Una funció ha de tenir una única responsabilitat clarament definida. Si una funció fa més d'una cosa, és un senyal que cal dividir-la en funcions més petites i especialitzades.



METÀFORA D'UNA FUNCIO

Altres símptomes que la funció que has escrit no fa una sola cosa...

- **La funció és massa llarga.** Una funció massa llarga és indicatiu que probablement està fent més d'una cosa i que l'hauries de dividir.
- **La funció rep massa arguments.** Si la teva funció rep més de 2 o 3 arguments, pot ser símptoma de que estiguis fent massa coses dins de la funció.
- **La funció té un nom poc específic.** Si una funció té un nom genèric o ambigu, com ara `processa()` o `gestiona()`, això sovint indica que està fent diverses coses. Els noms de les funcions haurien de reflectir clarament la seva única responsabilitat.

Com fer servir les funcions correctament?


Les funcions han de ser reutilitzables

- No té sentit crear dues funcions independents quan podem crear una funció més general i una de més específica, on la funció específica cridi a la general.

Aquí hem definit dues funcions `genera_rectangle` i `genera_quadrat` que són independents (creant repetició de codi) quan realment una és un cas particular de l'altra.



METÀFORA D'UNA FUNCIO



```
void genera_quadrat(int costat) {  
    for (int i = 0; i < costat; i++) {  
        for (int j = 0; j < costat; j++) {  
            printf("*");  
        }  
        printf("\n");  
    }  
}  
  
void genera_rectangle(int alçada, int amplada) {  
    for (int i = 0; i < alçada; i++) {  
        for (int j = 0; j < amplada; j++) {  
            printf("*");  
        }  
        printf("\n");  
    }  
}
```

Com fer servir les funcions correctament?

Les funcions han de ser reutilitzables


- No té sentit crear dues funcions independents quan podem crear una funció més general i una de més específica, on la funció específica cridi a la general.

Com que `genera_quadrat` és un cas particular de `genera_rectangle`, el que fem és cridar a la funció general.

Evitem repetició de codi!



METÀFORA D'UNA FUNCIÓ



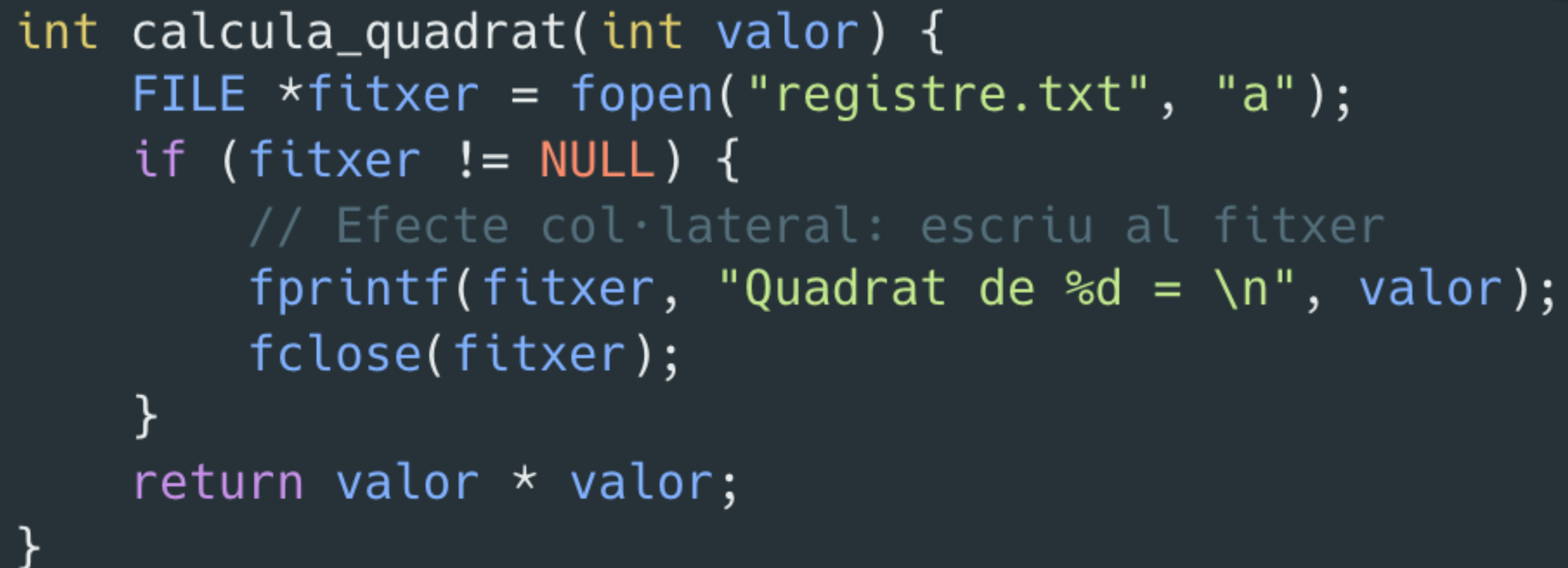
```
void genera_rectangle(int alçada, int amplada) {
    for (int i = 0; i < alçada; i++) {
        for (int j = 0; j < amplada; j++) {
            printf("*");
        }
        printf("\n");
    }
}

void genera_quadrat(int costat) {
    // Crida a la funció general
    genera_rectangle(costat, costat);
}
```


Altres errors al fer servir funcions: efectes col·laterals

Crear funcions amb efectes col·laterals

- Els efectes col·laterals són enganys perquè una funció promet fer una cosa, però en realitat en fa d'altres amagades. Això pot modificar variables externes o globals, o els paràmetres d'entrada sense que sigui evident, causant dependències en l'ordre d'execució i errors inesperats.



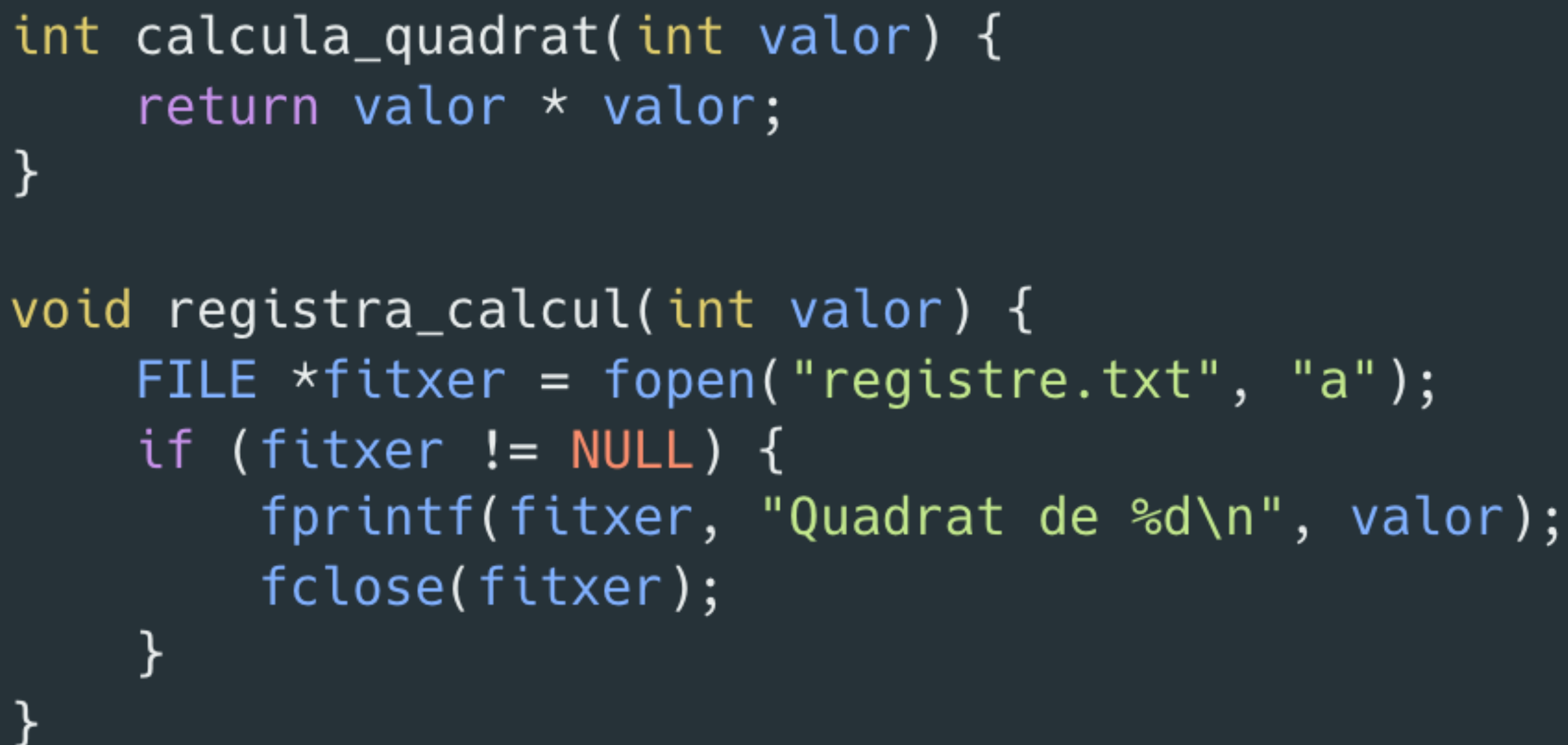
```
int calcula_quadrat(int valor) {  
    FILE *fitxer = fopen("registre.txt", "a");  
    if (fitxer != NULL) {  
        // Efecte col·lateral: escriu al fitxer  
        fprintf(fitxer, "Quadrat de %d = \n", valor);  
        fclose(fitxer);  
    }  
    return valor * valor;  
}
```

Aquesta funció promet "calcular un quadrat". Però a l'hora de la veritat, també crea un arxiu i en guarda el valor! Aquest "efecte secundari" no ens l'esperàvem!

Altres errors al fer servir funcions: efectes col·laterals

Crear funcions amb efectes col·laterals

- Els efectes col·laterals són enganys perquè una funció promet fer una cosa, però en realitat en fa d'altres amagades. Això pot modificar variables externes o globals, o els paràmetres d'entrada sense que sigui evident, causant dependències en l'ordre d'execució i errors inesperats.



```
int calcula_quadrat(int valor) {  
    return valor * valor;  
}  
  
void registra_calcul(int valor) {  
    FILE *fitxer = fopen("registre.txt", "a");  
    if (fitxer != NULL) {  
        fprintf(fitxer, "Quadrat de %d\n", valor);  
        fclose(fitxer);  
    }  
}
```

Separar la funció en dues funcions que clarament especifiquen què fan és una manera d'evitar els efectes col·laterals inesperats.

A més, complim que una funció faci una sola cosa!

Altres errors al fer servir funcions: efectes col·laterals

Crear funcions amb efectes col·laterals

- Els efectes col·laterals són enganys perquè una funció promet fer una cosa, però en realitat en fa d'altres amagades. Això pot modificar variables externes o globals, o els paràmetres d'entrada sense que sigui evident, causant dependències en l'ordre d'execució i errors inesperats.

Un exemple d'efecte col·lateral molt amagat

- Imagina una funció que rep un conjunt de lletres, anomenat `lletres`, i una paraula:

```
lletres = "TAPGQDARAT"  
paraula = "PATATA"
```
- I hem de fer una funció, anomenada, `busca_paraula`, que retorni cert si amb les lletres del conjunt es pot formar la paraula.
- Si quan ho implementem, marquem cada lletra amb un caràcter especial (e.g. `' - '`) per denotar que ja les hem fet servir, estaríem modificant el conjunt de lletres original, que al cridar la funció seria `"TAPGQDARAT"`, i al sortir seria `"---GQD-R--"`
- Això és un efecte col·lateral amagat que s'ha d'evitar!

Altres errors al fer servir funcions: noms poc descriptius

No posar-hi noms descriptius

"You know that you are working on clean code when each routine turns out to be pretty much what you expected" -- Ward Cunningham

- Triar un bon nom per a una funció sembla una tasca secundària, però és molt important.

```
void proc(int a, int b);           // Nom poc descriptiu  
void processPayment(int amount, int discount); // Nom clar i descriptiu
```

- El nom ha d'indicar què fa la funció de manera unívoca.

```
// Nom ambigu  
void calculate(int x);  
  
// Nom que indica clarament què fa la funció  
void calculateTotalPrice(int quantity, float unitPrice);
```

Altres errors al fer servir funcions: noms poc descriptius

No posar-hi noms descriptius

"You know that you are working on clean code when each routine turns out to be pretty much what you expected" -- Ward Cunningham

- No tinguis por de triar un nom llarg. Un nom llarg i descriptiu és millor que un nom curt i enigmàtic. Un nom llarg i descriptiu és millor que un comentari llarg i descriptiu.

```
int cp(int x);    // Nom curt i enigmàtic
int calculateProductPriceWithTax(int basePrice); // Nom llarg i descriptiu
```

- A vegades el nom de la funció ens pot servir fins i tot per recordar l'ordre dels paràmetres:

```
// Hem de consultar la documentació per recordar l'ordre
bool compare(char * expected, char * actual);

// La funció ens recorda l'ordre
bool compareExpectedActual(char * expected, char * actual);
```