

Estructuras de datos

Clases teóricas

por Pablo E. “Fidel” Martínez López

9. Modelo imperativo

Modelos de computación

Modelo de computación

- ❏ Modelo denotacional
 - ❏ También llamado *funcional*
 - ❏ Se evalúan expresiones que describen valores
 - ❏ Funciones para abstraer (combinaciones de) expresiones
 - ❏ Las funciones transforman valores
 - ❏ El programa es una expresión y su resultado es el valor de esa expresión
 - ❏ Las estructuras de datos se dan mediante expresiones

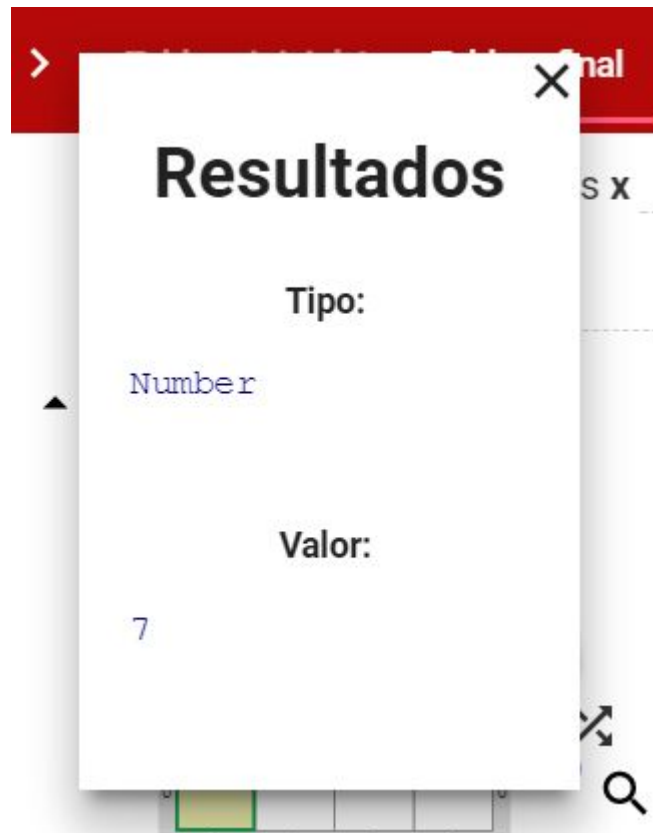
Modelo de computación

- ❑ Modelo destructivo
 - ❑ También llamado *imperativo*
 - ❑ Se ejecutan comandos que producen efectos
 - ❑ Procedimientos para abstraer (secuencias de) comandos
 - ❑ Los procedimientos producen efectos sobre el estado
 - ❑ El programa va de un estado inicial a un estado final
 - ❑ ¿Qué compone el estado?
 - ❑ Tablero, en Gobstones
 - ❑ Memoria, en general

Modelo de computación

Modelo denotacional, ejemplo

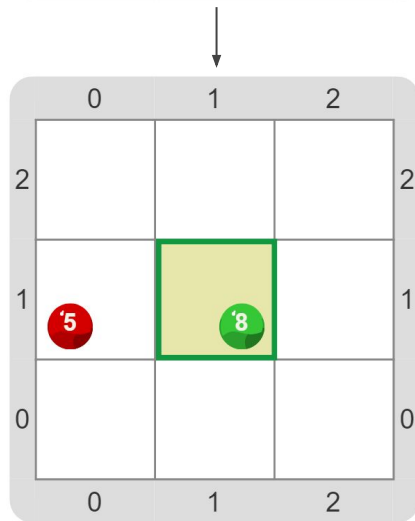
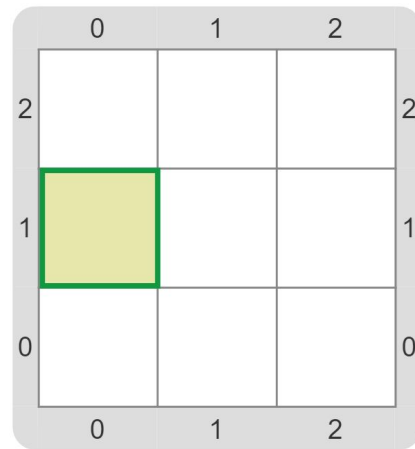
```
program {  
  return (sumar (dos () , sumar (dos () , 3) ) )  
}  
  
function sumar(n,m) {  
  return (n+m)  
}  
  
function dos() {  
  return (2)  
}
```



Modelo de computación

Modelo destructivo, ejemplo

```
program {  
  Poner__Veces(Rojo, 5)  
  Mover(Este)  
  Poner__Veces(Verde, 8)  
}  
  
procedure Poner__Veces(cant,color) {  
  repeat (cant) { Poner(color) }  
}
```



Modelo de computación

- En ambos modelos se transforma información
 - En el destructivo, se transforma un estado inicial en final
 - En el denotacional, argumentos en un resultado
- ¿Por qué usar el modelo destructivo?
 - Muchas cosas se pueden hacer más eficientes
 - El mundo físico sigue este modelo
 - Las computadoras en particular, se implementan sobre él
- Lenguajes que trabajan sobre el modelo destructivo
 - C, C++, Java, Python, Ruby, Javascript, etc.

Memoria

Memoria

- ❏ El modelo destructivo modifica un estado
- ❏ Usualmente ese estado es una ***memoria***
- ❏ ¿Qué es una memoria? Intento 1

Memoria

- ❏ El modelo destructivo modifica un estado
- ❏ Usualmente ese estado es una **memoria**
- ❏ ¿Qué es una memoria? Intento 1
 - ❏ En muy bajo nivel, es un arreglo de celdas
 - ❏ Cada celda se compone de uno o más bytes

Memoria

- ❑ El modelo destructivo modifica un estado
- ❑ Usualmente ese estado es una **memoria**
- ❑ ¿Qué es una memoria? Intento 1
 - ❑ En muy bajo nivel, es un arreglo de celdas
 - ❑ Cada celda se compone de uno o más bytes
 - ❑ Las celdas representan números
 - ❑ Los números pueden representar otras cosas

Memoria

- ❑ El modelo destructivo modifica un estado
- ❑ Usualmente ese estado es una **memoria**
- ❑ ¿Qué es una memoria? Intento 1
 - ❑ En muy bajo nivel, es un arreglo de celdas
 - ❑ Cada celda se compone de uno o más bytes
 - ❑ Las celdas representan números
 - ❑ Los números pueden representar otras cosas
 - ❑ ¿Se puede programar en bajo nivel con comodidad?

Memoria

- ❏ ¿Qué es una memoria? Intento 1
 - ❏ En muy bajo nivel, es un arreglo de celdas
 - ❏ Requiere conocer la representación usada

01101101
00100000
00101100
01100001
01101100
01101111
01001000

Memoria,
bajo nivel

109
32
44
97
108
111
72

Representación,
como números

'm'
''
','
'a'
'l'
'o'
'H'

Representación,
como ASCII

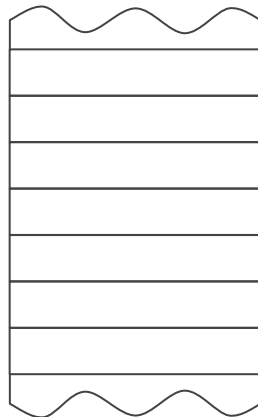
C y la memoria

C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a ciertas celdas de memoria
- ❑ Su contenido se codifica en formato binario

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Antes de ejecutar el programa



Datos vistos
en alto nivel



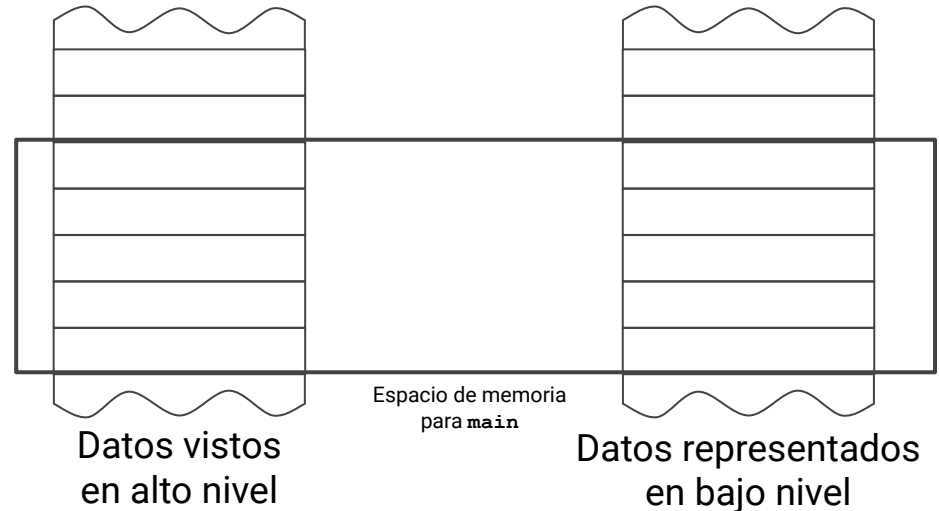
Datos representados
en bajo nivel

C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a ciertas celdas de memoria
- ❑ Su contenido se codifica en formato binario

```
➡ int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Se calcula cuánta memoria
usará el procedimiento `main` y
se asigna para su uso

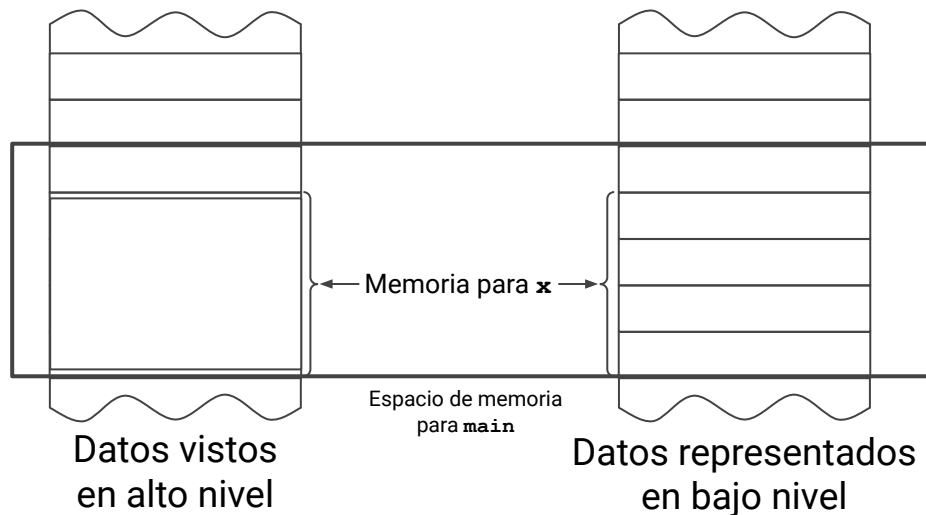


C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a ciertas celdas de memoria
- ❑ Su contenido se codifica en formato binario

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Se reservan celdas de memoria
para **x** (dependiendo del tipo)

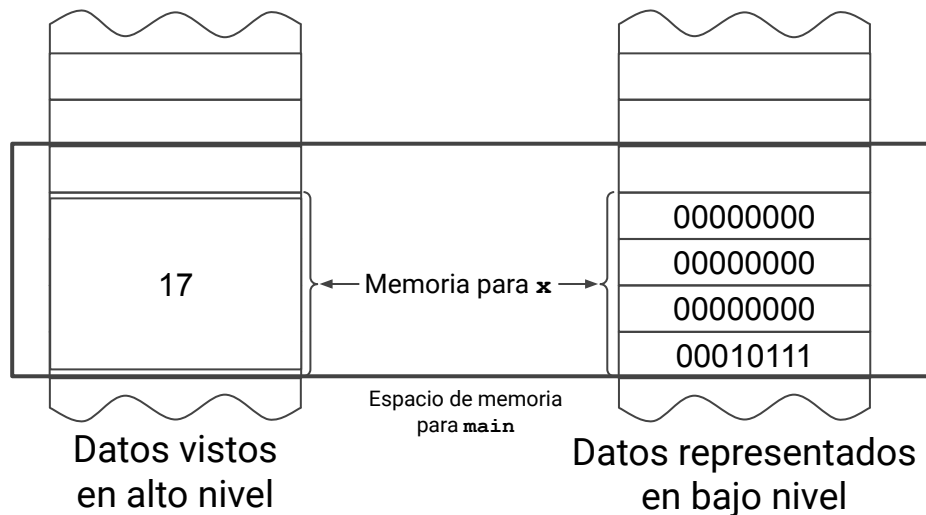


C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a ciertas celdas de memoria
- ❑ Su contenido se codifica en formato binario

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Se asigna la representación correspondiente al valor usado

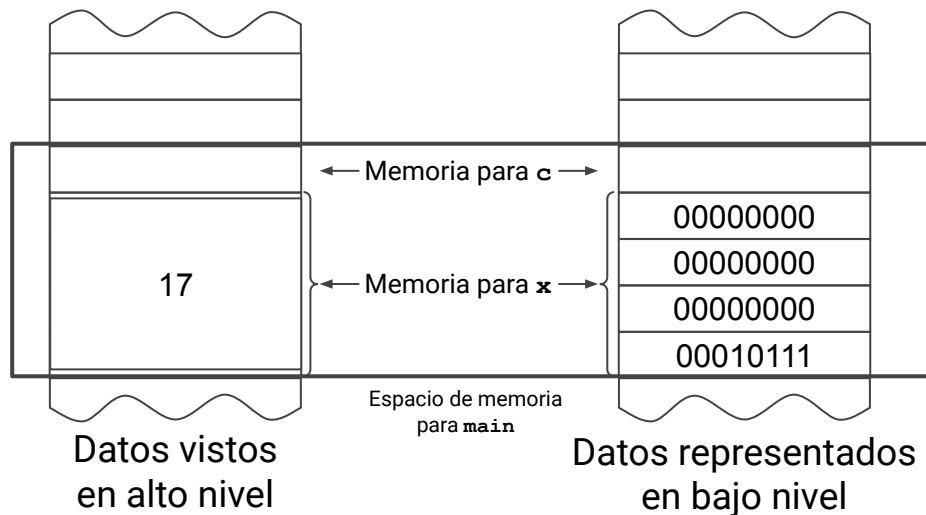


C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a ciertas celdas de memoria
- ❑ Su contenido se codifica en formato binario

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Se reservan celdas de memoria
para c (dependiendo del tipo)

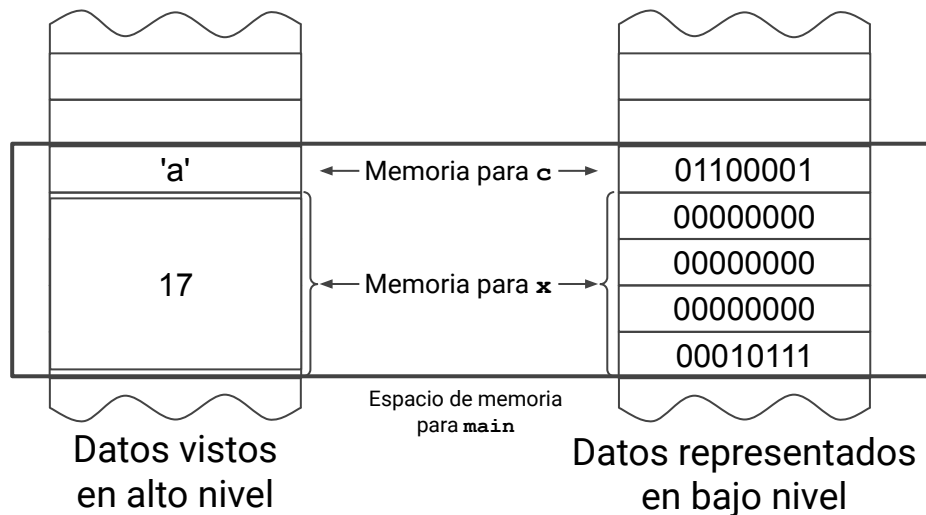


C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a ciertas celdas de memoria
- ❑ Su contenido se codifica en formato binario

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Se asigna la representación correspondiente al valor usado



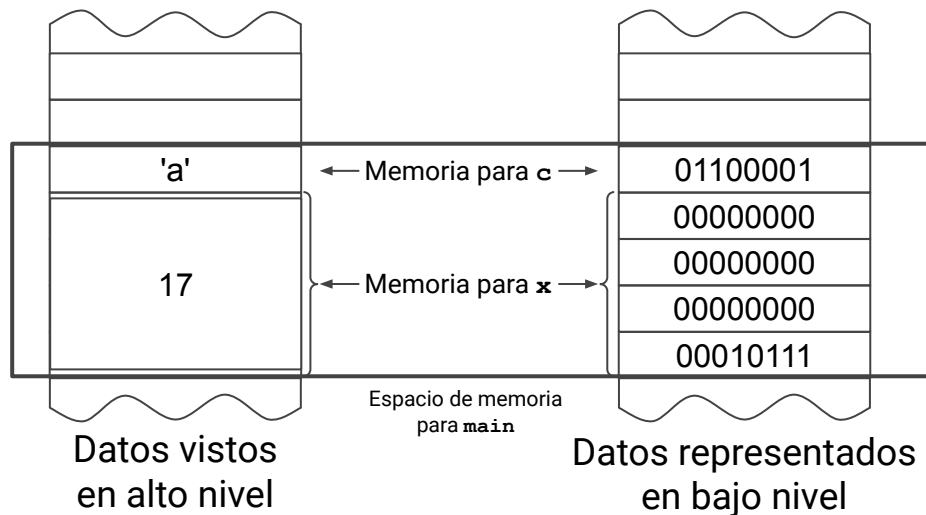
C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a ciertas celdas de memoria
- ❑ Su contenido se codifica en formato binario

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```



Al usar una variable, se buscan sus celdas y se decodifica la representación de la misma



C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
 - ❑ Cada variable se asigna a ciertas celdas de memoria
 - ❑ Su contenido se codifica en formato binario
- ❑ ¿Resulta adecuado pensar en este bajo nivel?
- ❑ ¿Cómo podemos abstraer la representación binaria?
 - ❑ Poner foco solamente con la memoria como abstracción
 - ❑ No tener en cuenta los detalles de representación
(cuántas celdas ocupa cada dato, cuál es el código binario, etc.)

C y la memoria: variables

- ❏ ¿Cómo trabaja un programa en C/C++?
- ❏ Cada variable se asigna a cierto espacio de memoria
- ❏ Al representarlo, no necesitamos pensar en bajo nivel

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Antes de ejecutar el programa

C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a cierto espacio de memoria
- ❑ Al representarlo, no necesitamos pensar en bajo nivel

```
➡ int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Se asigna un espacio de memoria para **main**

main



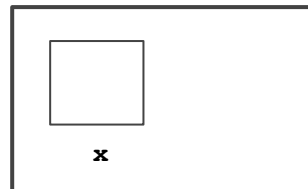
C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a cierto espacio de memoria
- ❑ Al representarlo, no necesitamos pensar en bajo nivel

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Se reserva memoria para **x**

main



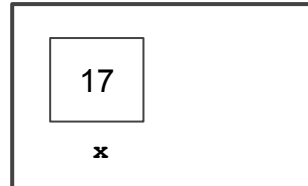
C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a cierto espacio de memoria
- ❑ Al representarlo, no necesitamos pensar en bajo nivel

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Se asigna el valor usado

main

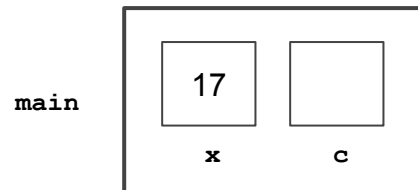


C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a cierto espacio de memoria
- ❑ Al representarlo, no necesitamos pensar en bajo nivel

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```

Se reserva memoria para c



C y la memoria: variables

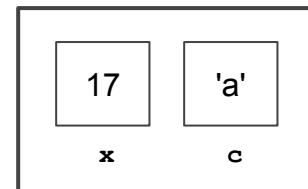
- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a cierto espacio de memoria
- ❑ Al representarlo, no necesitamos pensar en bajo nivel

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```



Se asigna el valor usado

main



C y la memoria: variables

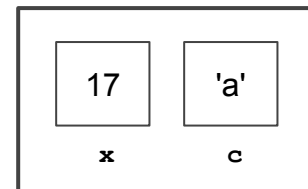
- ❑ ¿Cómo trabaja un programa en C/C++?
- ❑ Cada variable se asigna a cierto espacio de memoria
- ❑ Al representarlo, no necesitamos pensar en bajo nivel

```
int main() {  
    int x = 17;  
    char c = 'a';  
    cout << x << endl;  
    cout << c << endl;  
}
```



Al usar una variable, se leen los valores de la memoria

main



C y la memoria: variables

- ❑ ¿Cómo trabaja un programa en C/C++?
 - ❑ Cada variable se asigna a cierto espacio de memoria
 - ❑ Al lugar donde se ubican estos espacios se lo conoce como “*frame*” (marco)
 - ❑ Las variables locales de un procedimiento se alojan en el *frame* que le corresponde a ese procedimiento
 - ❑ Al representarlo, no necesitamos pensar en bajo nivel
 - ❑ No importan cuántas celdas ocupa cada variable
 - ❑ No importa en qué direcciones están las celdas

Más sobre el manejo de memoria

C y la memoria: funciones

- ❑ ¿Cómo son los procedimientos/funciones en C/C++?
- ❑ Solamente hay *funciones*, con efectos permanentes
- ❑ Si no devuelven nada (*void*), son procedimientos

```
int succ(int n) {  
    return(n+1);  
}
```

Arranca la ejecución

```
→ int main() {  
    int x = 17;  
    int y = succ(x);  
    cout << x << "+1=" << y << endl;  
}
```

main



C y la memoria: funciones

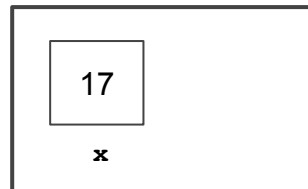
- ❑ ¿Cómo son los procedimientos/funciones en C/C++?
- ❑ Solamente hay *funciones*, con efectos permanentes
- ❑ Si no devuelven nada (*void*), son procedimientos

```
int succ(int n) {  
    return(n+1);  
}
```

Se crea el espacio para **x**
y se asigna

```
int main() {  
    int x = 17;  
    int y = succ(x);  
    cout << x << "+1=" << y << endl;  
}
```

main



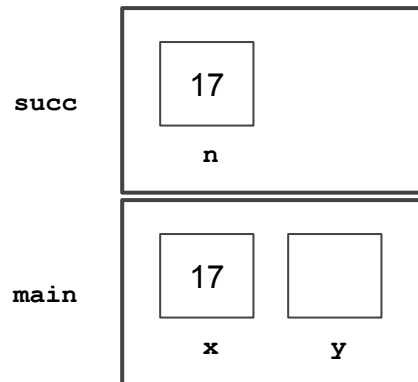
C y la memoria: funciones

- ❑ ¿Cómo son los procedimientos/funciones en C/C++?
- ❑ Solamente hay *funciones*, con efectos permanentes
- ❑ Si no devuelven nada (*void*), son procedimientos

```
➡ int succ(int n) {  
    return(n+1);  
}
```

Cada procedimiento tiene su propio *frame*; los parámetros también van en celdas

```
➡ int main() {  
    int x = 17;  
    int y = succ(x);  
    cout << x << "+1=" << y << endl;  
}
```

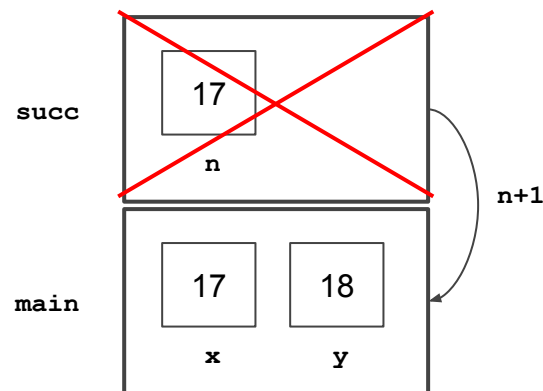


C y la memoria: funciones

- ❑ ¿Cómo son los procedimientos/funciones en C/C++?
- ❑ Solamente hay *funciones*, con efectos permanentes
- ❑ Si no devuelven nada (*void*), son procedimientos

```
➡ int succ(int n) {  
    return(n+1);  
}  
  
➡ int main() {  
    int x = 17;  
    int y = succ(x);  
    cout << x << "+1=" << y << endl;  
}
```

Cada procedimiento tiene su propio *frame*; los parámetros también van en celdas



C y la memoria: funciones

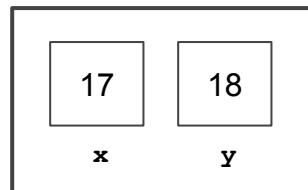
- ❑ ¿Cómo son los procedimientos/funciones en C/C++?
- ❑ Solamente hay *funciones*, con efectos permanentes
- ❑ Si no devuelven nada (*void*), son procedimientos

```
int succ(int n) {  
    return(n+1);  
}
```

El frame de un procedimiento
desaparece al terminar el mismo

```
int main() {  
    int x = 17;  
    int y = succ(x);  
    cout << x << "+1=" << y << endl;  
}
```

main



C y la memoria: funciones

- ❑ ¿Cómo son los procedimientos/funciones en C/C++?
 - ❑ Solamente hay *funciones*, con efectos permanentes
 - ❑ Cada función tiene su propio *frame*
 - ❑ Al anidar funciones, los *frames* se apilan
 - ❑ Por eso esta memoria de C se conoce como **Stack** y a cada *frame* como **stack frame**
 - ❑ Los parámetros también tienen espacio de memoria
 - ❑ ¡Y son como cualquier otra variable!
 - ❑ Se la conoce como **memoria estática**

C y la memoria: recursión

- ❏ ¿Cómo funciona la recursión en C/C++?
- ❏ Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}
```

➡

```
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Empieza el programa

main



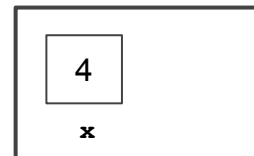
C y la memoria: recursión

- ❏ ¿Cómo funciona la recursión en C/C++?
- ❏ Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}  
  
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se crea el espacio
para **x** y se asigna

main



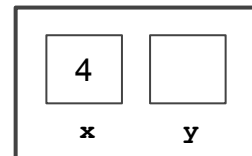
C y la memoria: recursión

- ❏ ¿Cómo funciona la recursión en C/C++?
- ❏ Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}  
  
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se crea el espacio
para y...

main



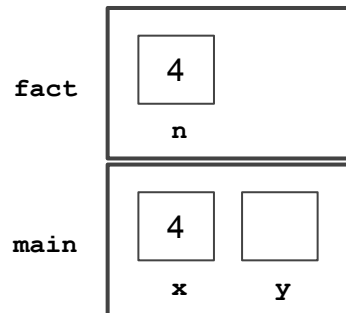
C y la memoria: recursión

- ❏ ¿Cómo funciona la recursión en C/C++?
- ❏ Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}
```

...y se invoca fact con 4

```
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```



C y la memoria: recursión

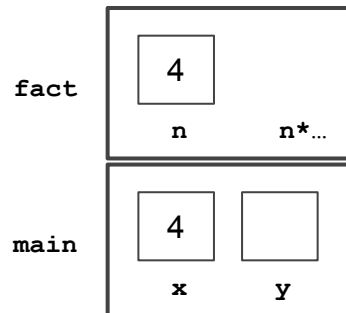
- ❏ ¿Cómo funciona la recursión en C/C++?
- ❏ Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}
```



Como no es cero

```
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

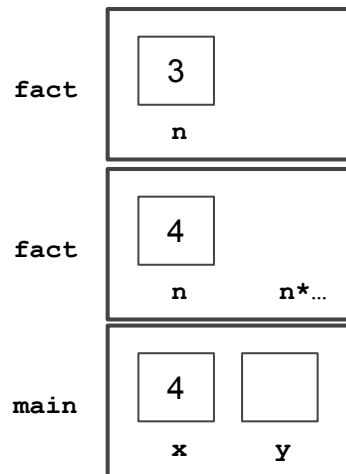


C y la memoria: recursión

- ❏ ¿Cómo funciona la recursión en C/C++?
- ❏ Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}  
  
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se invoca fact con 3



C y la memoria: recursión

- ❏ ¿Cómo funciona la recursión en C/C++?
- ❏ Los *frames* de memoria se anidan...

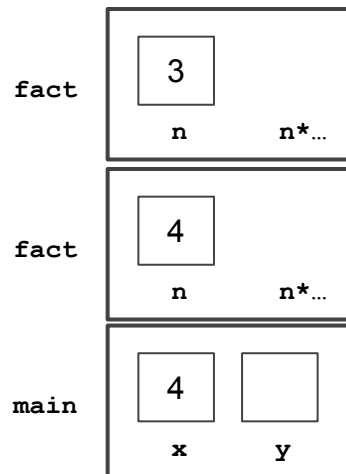
```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}
```



```
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```



Como no es cero



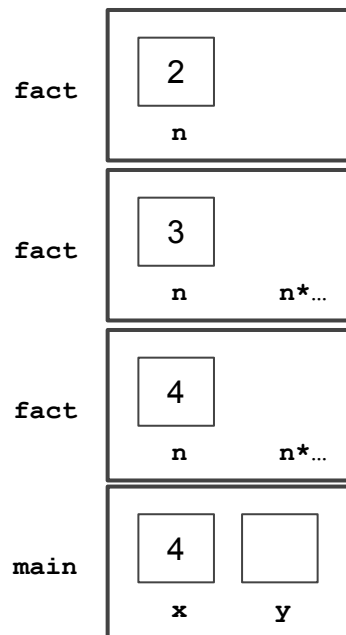
C y la memoria: recursión

❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

```
→ int fact(int n) {  
    if (n==0)  
        { return(1); }  
→   else { return(n*fact(n-1)); }  
    }  
  
int main() {  
    int x = 4;  
→   int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se invoca fact con 2



C y la memoria: recursión

❏ ¿Cómo funciona la recursión en C/C++?

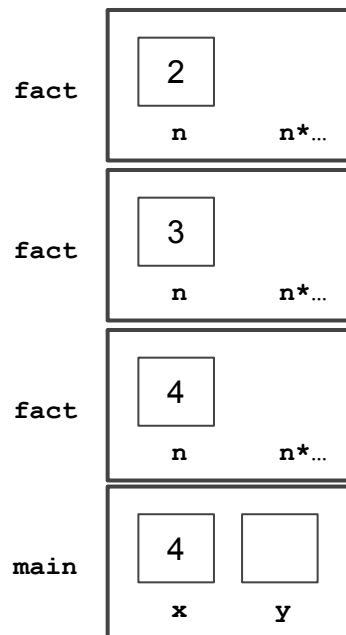
❏ Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}
```



Como no es cero

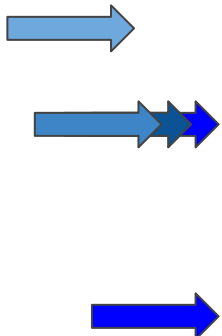
```
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```



C y la memoria: recursión

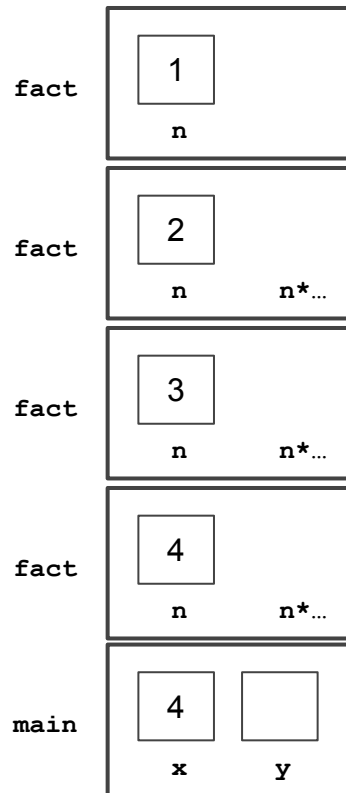
❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...



```
int fact(int n) {  
    if (n==0)  
        { return(1);  
        }  
    else { return(n*fact(n-1)); }  
}  
  
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se invoca fact con 1

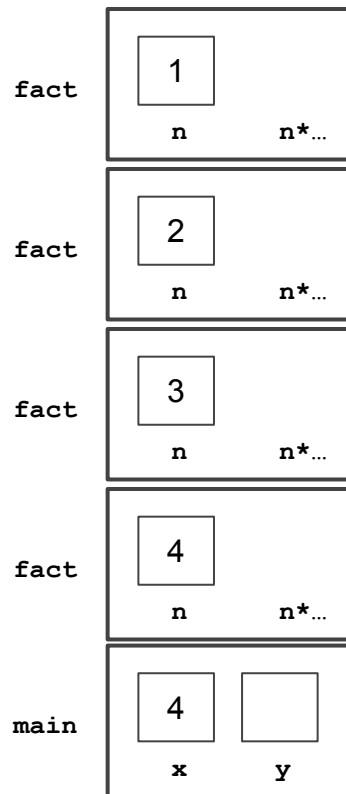


C y la memoria: recursión

- ¿Cómo funciona la recursión en C/C++?
- Los *frames* de memoria se anidan...

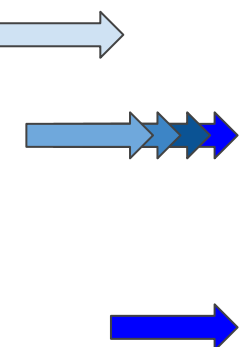
```
int fact(int n) {  
    if (n==0)  
        { return(1);  
        }  
    else { return(n*fact(n-1)); }  
}  
  
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Como no es cero



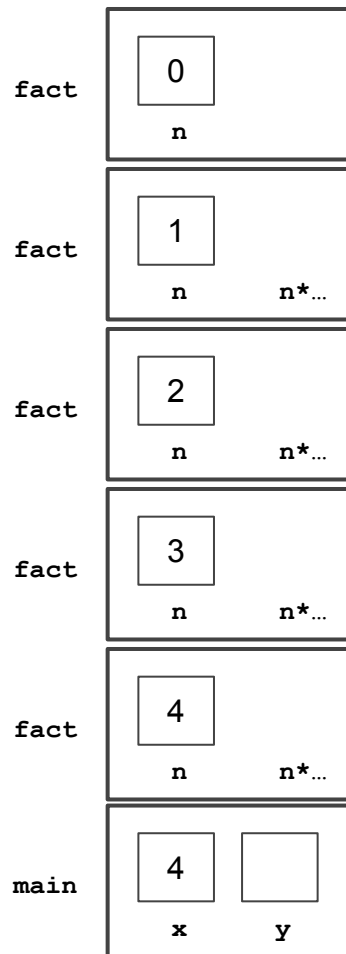
C y la memoria: recursión

- ¿Cómo funciona la recursión en C/C++?
- Los *frames* de memoria se anidan...



```
int fact(int n) {  
    if (n==0)  
        { return(1);  
        }  
    else { return(n*fact(n-1)); }  
}  
  
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se invoca fact con 0

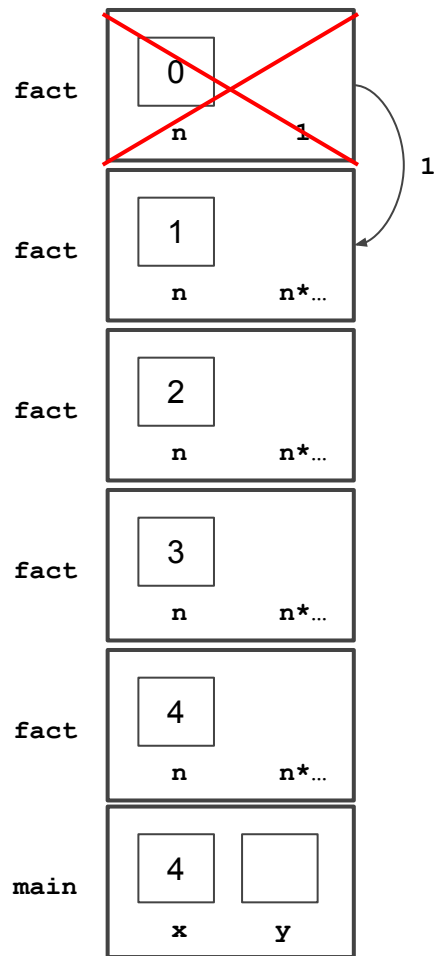


C y la memoria: recursión

- ¿Cómo funciona la recursión en C/C++?
- Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}  
  
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Como es cero,
se retorna 1



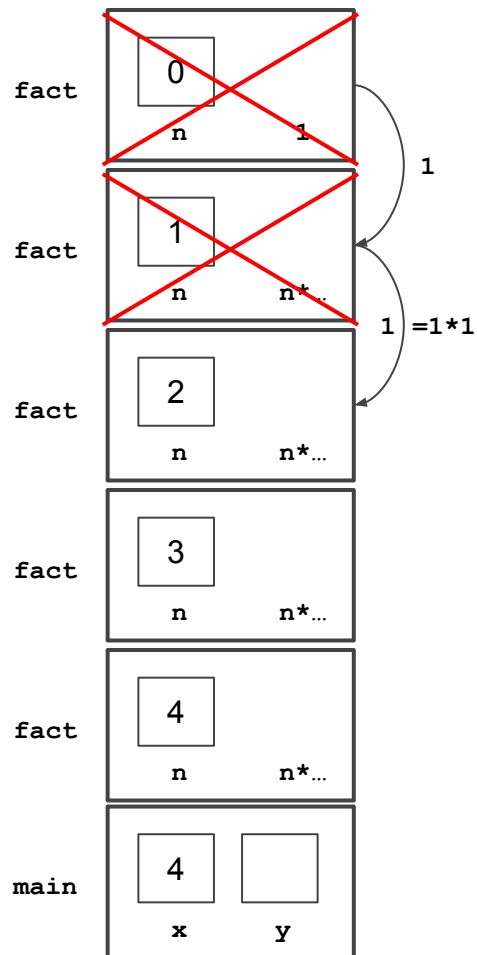
C y la memoria: recursión

- ¿Cómo funciona la recursión en C/C++?
- Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}
```

```
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se retorna de `fact(1)`



C y la memoria: recursión

❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

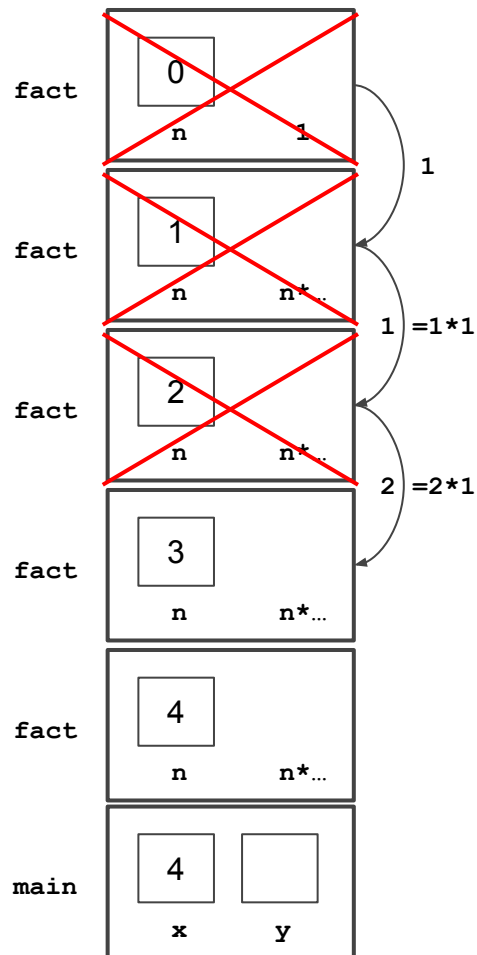
```
int fact(int n) {  
    if (n==0)  
        { return(1);  
        }  
    else { return(n*fact(n-1)); }  
}
```



```
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```



Se retorna de fact(2)



C y la memoria: recursión

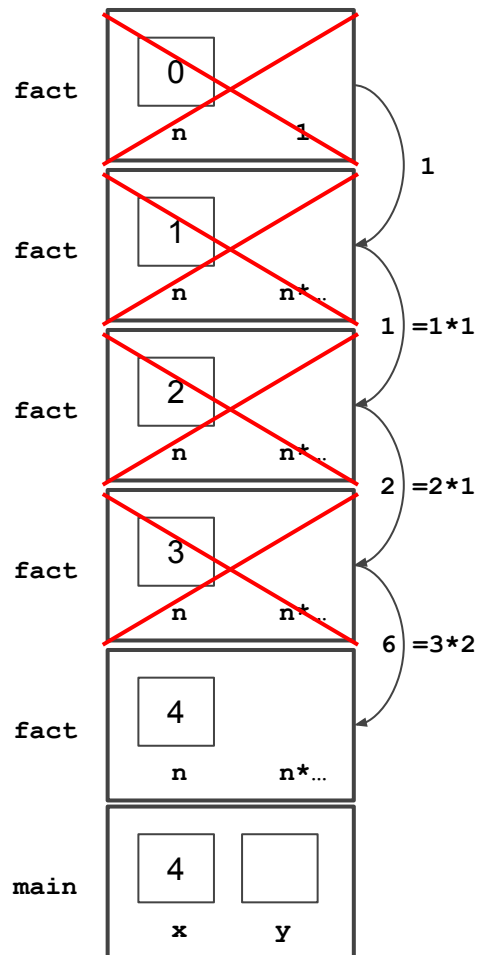
❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1);  
        }  
    else { return(n*fact(n-1)); }  
}
```

```
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se retorna de fact(3)



C y la memoria: recursión

❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

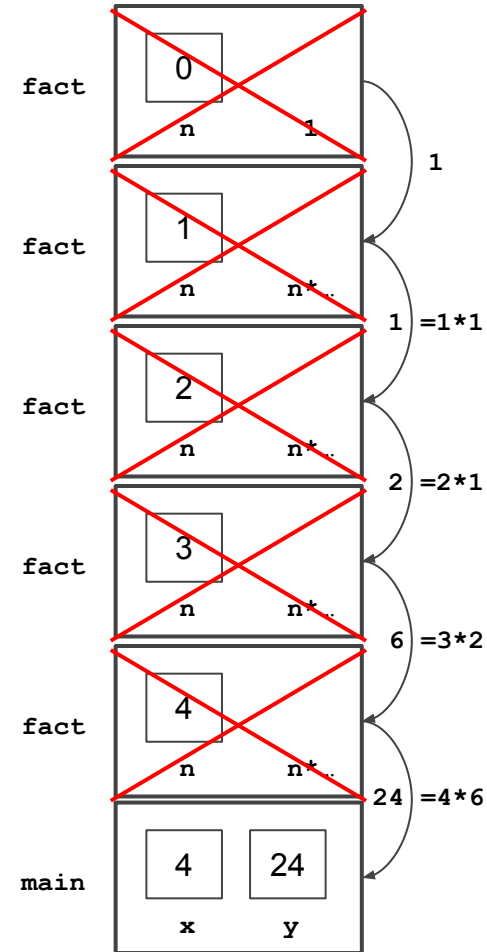
```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}
```



```
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```



Se retorna de fact(4)



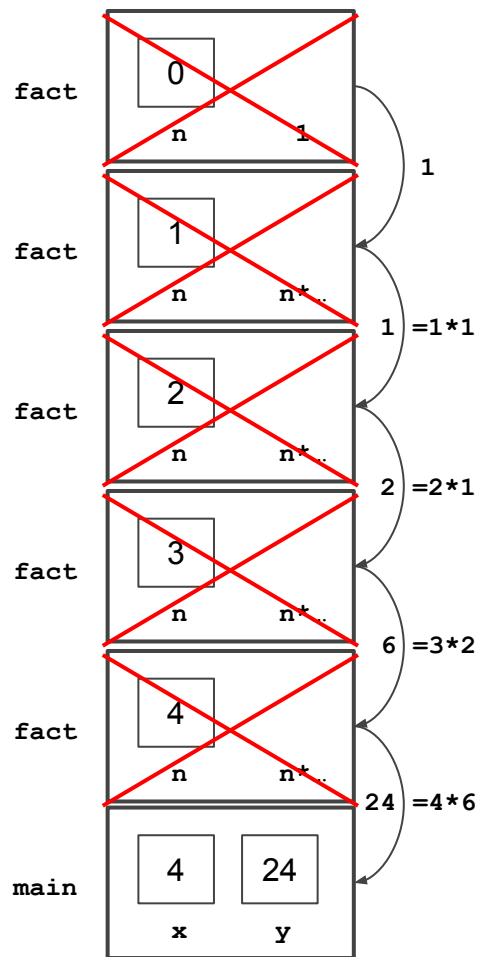
C y la memoria: recursión

❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1);  
        }  
    else { return(n*fact(n-1)); }  
}  
  
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Cada llamado recursivo
tiene su propio *frame*
(O(x) de memoria)



C y la memoria: recursión

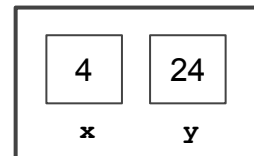
- ❏ ¿Cómo funciona la recursión en C/C++?
- ❏ Los *frames* de memoria se anidan...

```
int fact(int n) {  
    if (n==0)  
        { return(1); }  
    else { return(n*fact(n-1)); }  
}  
  
int main() {  
    int x = 4;  
    int y = fact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se completa la función y
se continúa el código



main



C y la memoria: funciones

- ❏ ¿Cómo son los procedimientos/funciones en C/C++?
 - ❏ Las funciones recursivas ocupan una cantidad lineal de memoria respecto de los llamados
 - ❏ Solamente se justifican para árboles
 - ❏ Si no, se debe buscar resolver con una iteración

C y la memoria: iteración

❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

❏ ... por eso se prefieren iteraciones

```
int ifact(int n) {  
    int f = 1;  
    while (n>0)  
        { f = f*n; n = n-1; }  
    return(f);  
}
```

Comienza el
programa

➡

```
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

main



C y la memoria: iteración

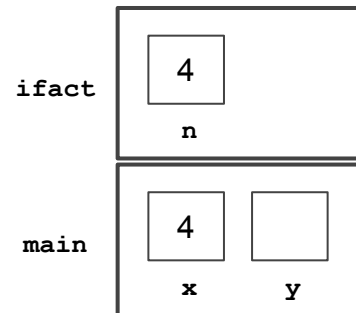
❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

❏ ... por eso se prefieren iteraciones

```
➡ int ifact(int n) {  
    int f = 1;  
    while (n>0)  
        { f = f*n; n = n-1; }  
    return(f);  
}  
  
➡ int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se reserva
memoria y se
invoca a ifact



C y la memoria: iteración

❏ ¿Cómo funciona la recursión en C/C++?

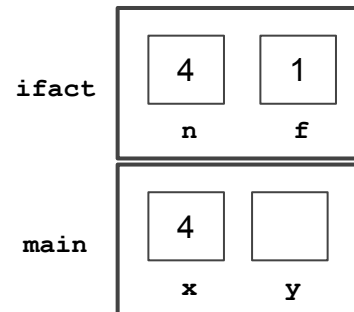
❏ Los *frames* de memoria se anidan...

❏ ... por eso se prefieren iteraciones

```
int ifact(int n) {  
    int f = 1;  
    while (n>0)  
        { f = f*n; n = n-1; }  
    return(f);  
}
```

```
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se reserva memoria
para la variable local
de ifact



C y la memoria: iteración

❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

❏ ... por eso se prefieren iteraciones

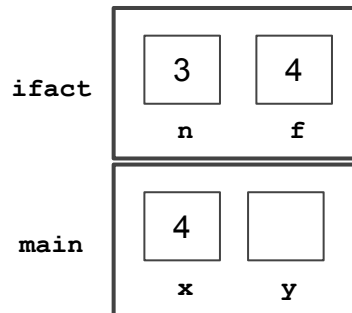
```
int ifact(int n) {  
    int f = 1;  
    while (n>0)  
        { f = f*n; n = n-1; }  
    return(f);  
}
```



```
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```



¡La memoria del
parámetro se puede
asignar!



C y la memoria: iteración

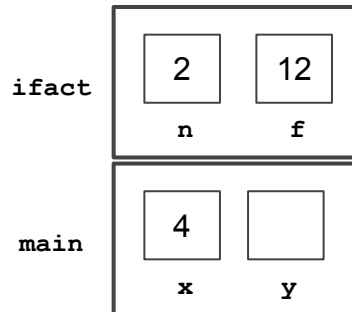
❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

❏ ... por eso se prefieren iteraciones

```
int ifact(int n) {  
    int f = 1;  
    while (n>0)  
        { f = f*n; n = n-1; }  
    return(f);  
}  
  
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

¡La memoria del
parámetro se puede
asignar!



C y la memoria: iteración

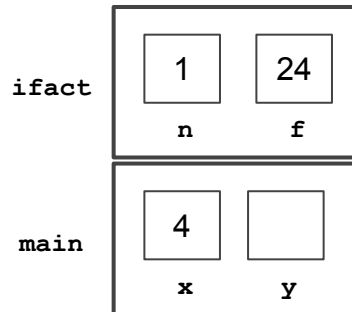
❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

❏ ... por eso se prefieren iteraciones

```
int ifact(int n) {  
    int f = 1;  
    while (n>0)  
        { f = f*n; n = n-1; }  
    return(f);  
}  
  
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

¡La memoria del
parámetro se puede
asignar!



C y la memoria: iteración

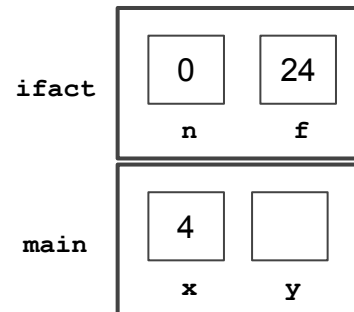
❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

❏ ... por eso se prefieren iteraciones

```
int ifact(int n) {  
    int f = 1;  
    while (n>0)  
        { f = f*n; n = n-1; }  
    return(f);  
}  
  
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

¡La memoria del
parámetro se puede
asignar!



C y la memoria: recursión

❏ ¿Cómo funciona la recursión en C/C++?

❏ Los *frames* de memoria se anidan...

❏ ... por eso se prefieren iteraciones

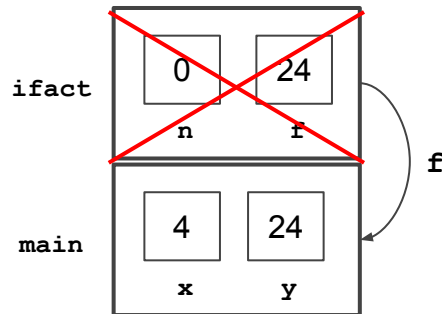
```
int ifact(int n) {  
    int f = 1;  
    while (n>0)  
        { f = f*n; n = n-1; }  
    return(f);  
}
```



```
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```



¡La memoria del
parámetro se puede
asignar!



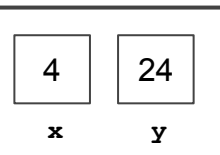
C y la memoria: iteración

- ❏ ¿Cómo funciona la recursión en C/C++?
- ❏ Los *frames* de memoria se anidan...
- ❏ ... por eso se prefieren iteraciones

```
int ifact(int n) {  
    int f = 1;  
    while (n>0)  
        { f = f*n; n = n-1; }  
    return(f);  
}  
  
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

Se completa el
programa

main



C y la memoria: funciones

- ❏ ¿Cómo son los procedimientos/funciones en C/C++?
 - ❏ Las funciones iterativas ocupan memoria constante
 - ❏ Aunque el costo en tiempo sigue siendo lineal, es mucho menor que pedir y liberar memoria

C y la memoria: iteración

❏ ¿Cómo funciona la recursión en C/C++?

❏ ¿Qué pasa si no quiero asignar el parámetro?

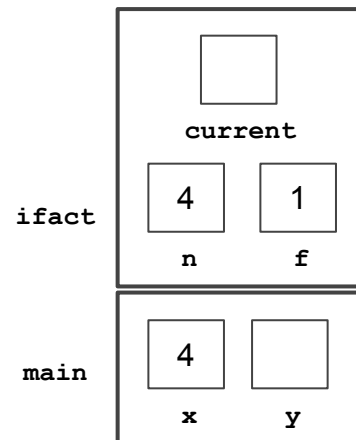
```
int ifact(int n) {  
    int f = 1;  
    int current = n;  
    while (current>0)  
    { f = f*current;  
      current = current-1; }  
    return(f);  
}
```



```
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```



¡La memoria del parámetro se puede asignar!



C y la memoria: iteración

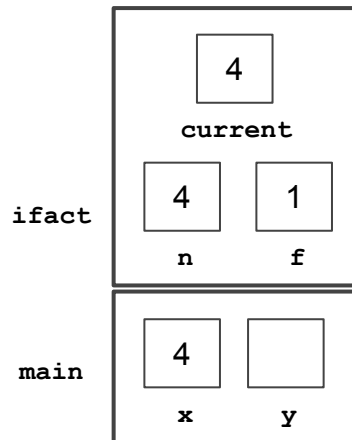
❏ ¿Cómo funciona la recursión en C/C++?

❏ ¿Qué pasa si no quiero asignar el parámetro?

```
int ifact(int n) {  
    int f = 1;  
    int current = n;  
    while (current>0)  
    { f = f*current;  
      current = current-1; }  
    return(f);  
}
```

```
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

¡La memoria del parámetro se puede asignar!



C y la memoria: iteración

❏ ¿Cómo funciona la recursión en C/C++?

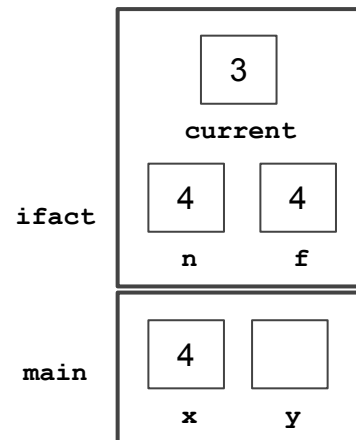
❏ ¿Qué pasa si no quiero asignar el parámetro?

```
int ifact(int n) {  
    int f = 1;  
    int current = n;  
    while (current>0)  
    { f = f*current;  
      current = current-1; }  
    return(f);  
}
```



```
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

¡La memoria del parámetro se puede asignar!



C y la memoria: iteración

❏ ¿Cómo funciona la recursión en C/C++?

❏ ¿Qué pasa si no quiero asignar el parámetro?

➡

```
int ifact(int n) {  
    int f = 1;  
    int current = n;  
    while (current>0)  
    { f = f*current;  
      current = current-1; }  
    return(f);  
}
```

➡

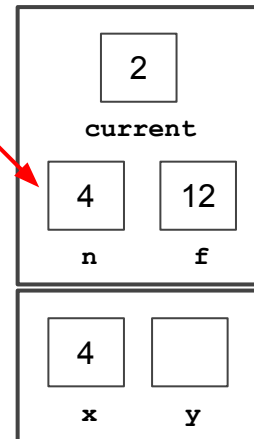
```
int main() {  
    int x = 4; int y = ifact(x);  
    cout << "fact(" << x << ")=";  
    cout << y << endl;  
}
```

WAT

¡La memoria del
parámetro se puede
asignar!

ifact

main



C y la memoria: funciones

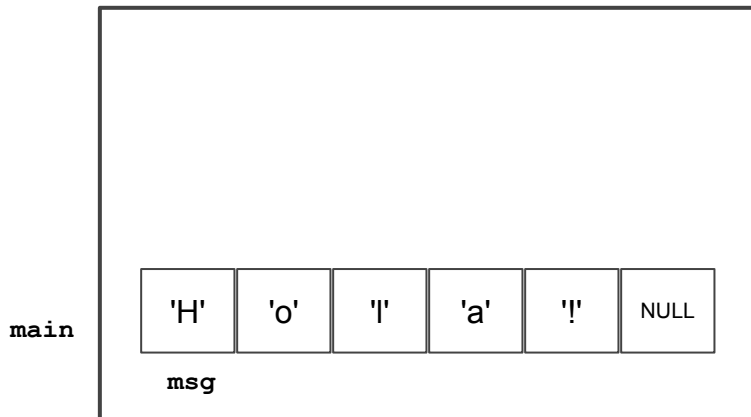
- ❏ ¿Cómo son los procedimientos/funciones en C/C++?
 - ❏ Las funciones iterativas ocupan memoria constante
 - ❏ Aunque el costo en tiempo sigue siendo lineal, es mucho menor que pedir y liberar memoria
 - ❏ ¿Y si quiero dejar el parámetro como en Gobstones?
 - ❏ Gasto memoria de más

C y la memoria: strings

- ❑ ¿Cómo son los strings en C/C++?
- ❑ Se usa una celda por caracter
- ❑ Siempre termina con una celda con 0 (char NULL)

```
int main() {  
    string msg = "Hola!";  
    cout << msg << endl;  
}
```

Es un poco de bajo nivel
mirarlo de esta forma...



C y la memoria: strings

- ❑ ¿Cómo son los strings en C/C++?
 - ❑ Se usa una celda por caracter
 - ❑ Siempre termina con una celda con 0 (char NULL)

```
int main() {  
    string msg = "Hola!";  
    cout << msg << endl;  
}
```

... por lo que usaremos una
forma de más alto nivel



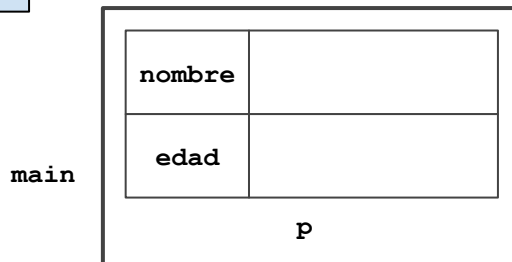
C y la memoria: structs

- ❑ ¿Cómo son los registros en C/C++?
- ❑ Son similares a Gobstones, pero se llaman **structs**
- ❑ Hay que considerar cómo se representan en memoria

```
struct Persona {  
    string nombre;  
    int edad;  
};
```

El espacio de un struct se compone de los espacios de todos sus campos

```
int main() {  
    struct Persona p;  
    p.nombre = "Alejandro";  
    p.edad = 36;  
    cout << p.nombre << endl;  
}
```



C y la memoria: structs

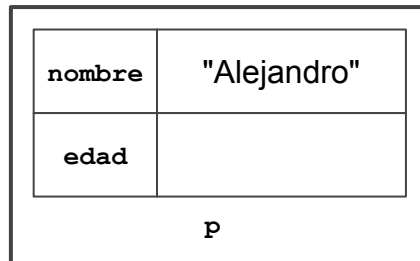
- ❑ ¿Cómo son los registros en C/C++?
- ❑ Son similares a Gobstones, pero se llaman **structs**
- ❑ Hay que considerar cómo se representan en memoria

```
struct Persona {  
    string nombre;  
    int edad;  
};
```

El espacio de un struct se compone de los espacios de todos sus campos

```
int main() {  
    struct Persona p;  
    p.nombre = "Alejandro";  
    p.edad = 36;  
    cout << p.nombre << endl;  
}
```

main



C y la memoria: structs

- ❑ ¿Cómo son los registros en C/C++?
- ❑ Son similares a Gobstones, pero se llaman **structs**
- ❑ Hay que considerar cómo se representan en memoria

```
struct Persona {  
    string nombre;  
    int edad;  
};
```

El espacio de un struct se compone de los espacios de todos sus campos

```
int main() {  
    struct Persona p;  
    p.nombre = "Alejandro";  
    p.edad = 36;  
    cout << p.nombre << endl;  
}
```

main

nombre	"Alejandro"
edad	36

p

Memoria y bajo nivel

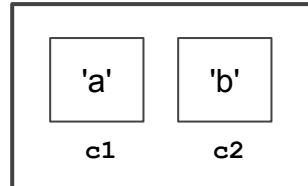
C y la memoria: bajo nivel

- ❑ ¿Cuándo es necesario mirar el bajo nivel en C/C++?
- ❑ Cuando no se respeta el tipo
 - ❑ Tratar a un char como su fuese un número...

```
int main() {  
    char c1 = 'a';  
    char c2 = c1 + 1;  
    cout << c1 << " ";  
    cout << c2 << endl;  
}
```

¿Se le puede sumar 1 a un char?
¿Qué va a dar como resultado?

main



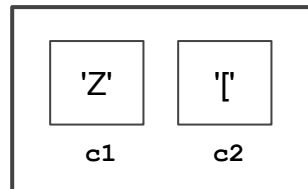
C y la memoria: bajo nivel

- ❏ ¿Cuándo es necesario mirar el bajo nivel en C/C++?
- ❏ Cuando no se respeta el tipo
 - ❏ Tratar a un char como su fuese un número...

```
int main() {  
    char c1 = 'Z';  
    char c2 = c1 + 1;  
    cout << c1 << " ";  
    cout << c2 << endl;  
}
```

¿Se le puede sumar 1 a un char?
¿Qué va a dar como resultado?

main



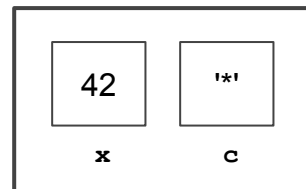
C y la memoria: bajo nivel

- ❏ ¿Cuándo es necesario mirar el bajo nivel en C/C++?
- ❏ Cuando no se respeta el tipo
 - ❏ ... o tratar a un número como su fuese un char

```
int main() {  
    int x = 42;  
    char c = x;  
    cout << x << " ";  
    cout << c << endl;  
}
```

¿Por qué 42 es '*'?
¿Cómo se convierte?

main



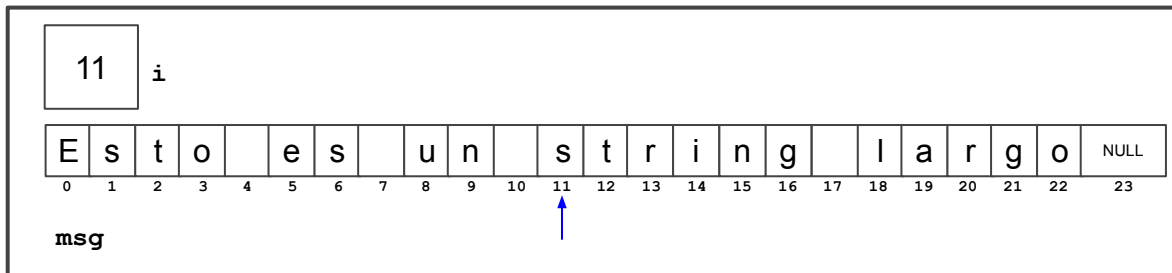
C y la memoria: bajo nivel

- ¿Cuándo es necesario mirar el bajo nivel en C/C++?
- Cuando se desea acceder a las partes de un string
 - Se empiezan a numerar desde 0

```
int main() {  
    string msg = "Esto es un string largo";  
    for(int i=11; i<17; i++) {  
        cout << msg[i];  
    }  
    cout << endl;  
}
```

main

¿Qué significa `msg[i]`?
¡Se interpreta al string
según sus celdas!



Memoria y TADs

C y la memoria: TADs

- ❑ ¿Se pueden hacer TADs en C/C++?
 - ❑ La representación y la interfaz van en un archivo `.h`
 - ❑ La implementación va en un archivo `.cpp` (del mismo nombre)
 - ❑ Se utiliza `#include` para incluir la interfaz
 - ❑ Ej. si el TAD es `Persona`, en el archivo que lo use debe ponerse como primera línea
`#include "Persona.h"`

C y la memoria: TADs

- ❑ ¿Se pueden hacer TADs en C/C++?
 - ❑ Primera aproximación: memoria estática
 - ❑ Definiciones de representación (en **Persona.h**)

```
struct RegistroDeP {  
    string nombre;  
    int edad;  
};
```

typedef sirve para definir alias de tipos

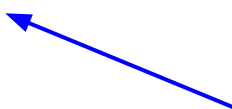
```
typedef struct RegistroDeP Persona;
```

- ❑ Las zonas rojas NO son parte de la interfaz
 - ❑ ¿Por qué necesito que estén acá?

C y la memoria: TADs

- ❏ ¿Se pueden hacer TADs en C/C++?
- ❏ Primera aproximación: memoria estática
- ❏ Definiciones de interfaz (en **Persona.h**)

```
Persona nacer(string n);  
Persona cumplirAnios(Persona p);  
string nombre(Persona p);  
int      edad(Persona p);  
void     ShowPersona(Persona p);
```



Los *prototipos* (resultado, nombre y parámetros) de las funciones constituyen la interfaz

C y la memoria: TADs

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Operaciones (parte 1, en `Persona.cpp`)

```
Persona nacer(string n) {  
    Persona p;  
    p.nombre = n; p.edad = 0;  
    return(p);  
}  
  
Persona cumplirAnios(Persona p) {  
    p.edad++;  
    return(p);  
}
```

C y la memoria: TADs

- ❏ ¿Se pueden hacer TADs en C/C++?
- ❏ Primera aproximación: memoria estática
- ❏ Operaciones (parte 2, en `Persona.cpp`)

```
string nombre(Persona p) {  
    return (p.nombre) ;  
}  
  
int edad(Persona p) {  
    return (p.edad) ;  
}
```


C y la memoria: TADs

- ❏ ¿Se pueden hacer TADs en C/C++?
- ❏ Primera aproximación: memoria estática
- ❏ Operaciones (parte 3, en `Persona.cpp`)

```
void ShowPersona(Persona p) {  
    cout << "Persona (";  
    cout << "nombre <- \"" << p.nombre << "\", "  
    cout << "edad <- " << p.edad;  
    cout << ")" << endl;  
}
```

C y la memoria: TADs

- ❏ ¿Se pueden hacer TADs en C/C++?
- ❏ Primera aproximación: memoria estática
- ❏ Uso de la interfaz

```
#include "Persona.h"

int main() {
    Persona carlos = nacer("Carlitos");
    carlos = cumplirAnios(carlos);
    ShowPersona(carlos);
}
```

C y la memoria: TADs

- ❑ ¿Se pueden hacer TADs en C/C++?
 - ❑ Primera aproximación: memoria estática
 - ❑ Uso de la interfaz

```
#include "Persona.h"
```


```
➡ int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = cumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se comienza la ejecución

main



C y la memoria: TADs

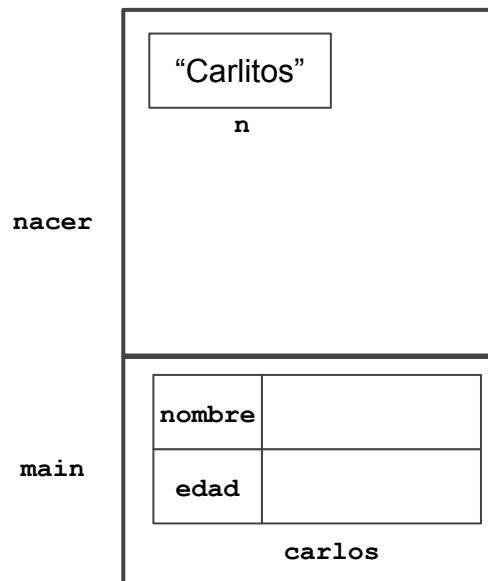


```
Persona nacer(string n) {  
    Persona p;  
    p.nombre = n; p.edad = 0;  
    return(p);  
}
```


- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = cumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se crea carlos, y se invoca a nacer



C y la memoria: TADs

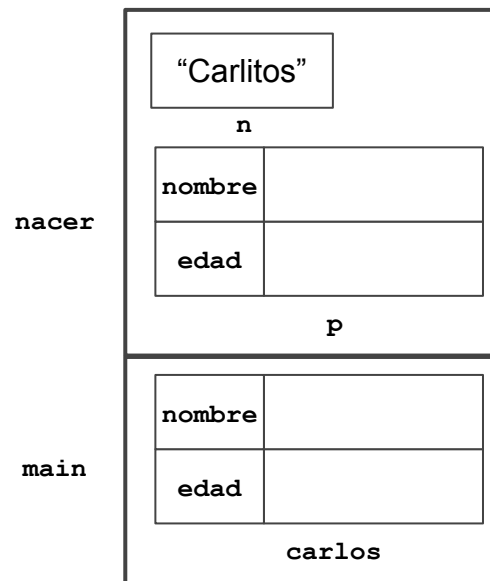


```
Persona nacer(string n) {  
    Persona p;  
    p.nombre = n; p.edad = 0;  
    return(p);  
}
```


- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = cumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se crea la variable local **p** de **nacer**



C y la memoria: TADs

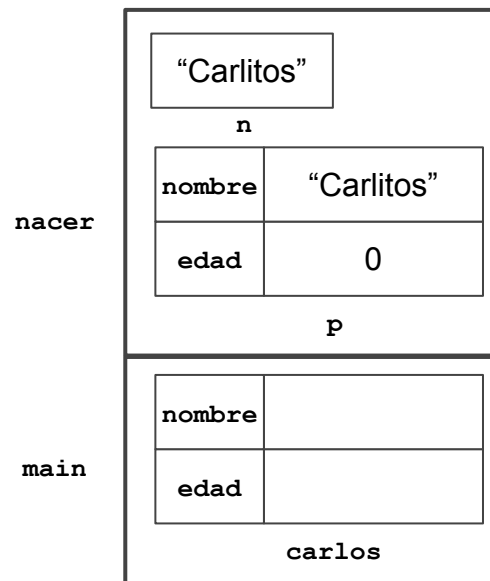


```
Persona nacer(string n) {  
    Persona p;  
    p.nombre = n; p.edad = 0;  
    return(p);  
}
```


- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = cumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se inicializan los campos




C y la memoria: TADs



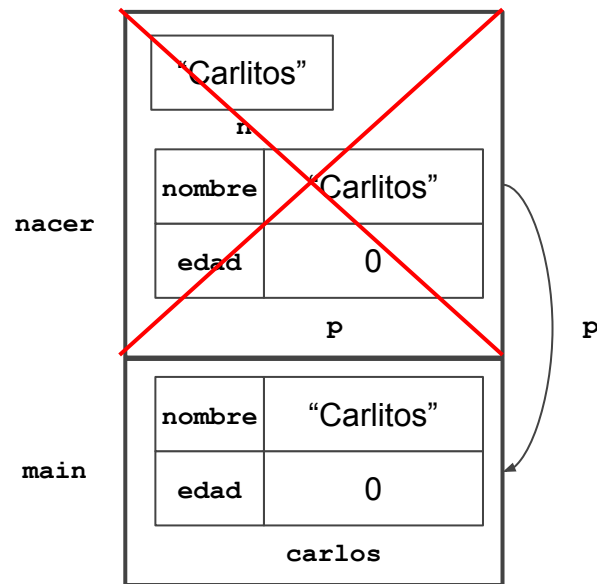
```
Persona nacer(string n) {  
    Persona p;  
    p.nombre = n; p.edad = 0;  
    return(p);  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz



```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = cumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se retorna p



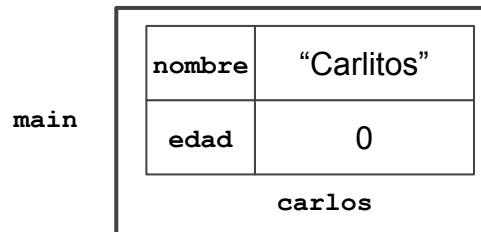
C y la memoria: TADs

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"

int main() {
    Persona carlos = nacer("Carlitos");
    carlos = cumplirAnios(carlos);
    ShowPersona(carlos);
}
```

Se completa la ejecución de **nacer**



C y la memoria: TADs



```
Persona cumplirAnios(Persona p) {  
    p.edad++;  
    return(p);  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"
```

```
int main() {
```

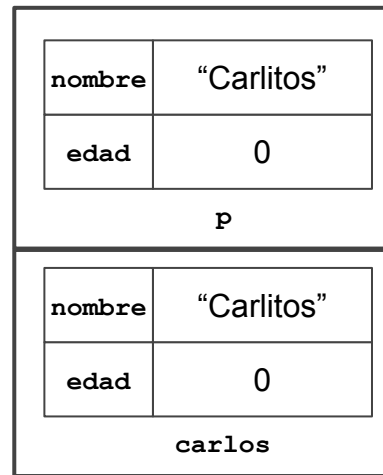
```
    Persona carlos = nacer("Carlitos");
```

```
    carlos = cumplirAnios(carlos);
```

```
    ShowPersona(carlos);
```

```
}
```

Se invoca a `cumplirAnios`



C y la memoria: TADs



```
Persona cumplirAnios(Persona p) {  
    p.edad++;  
    return(p);  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = cumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```



Se incrementa la edad

cumplirAnios

main

nombre	"Carlitos"
edad	1
p	

nombre	"Carlitos"
edad	0
carlos	

C y la memoria: TADs



```
Persona cumplirAnios(Persona p) {  
    p.edad++;  
    return(p);  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"
```

```
int main() {
```

```
    Persona carlos = nacer("Carlitos");
```

```
    carlos = cumplirAnios(carlos);
```

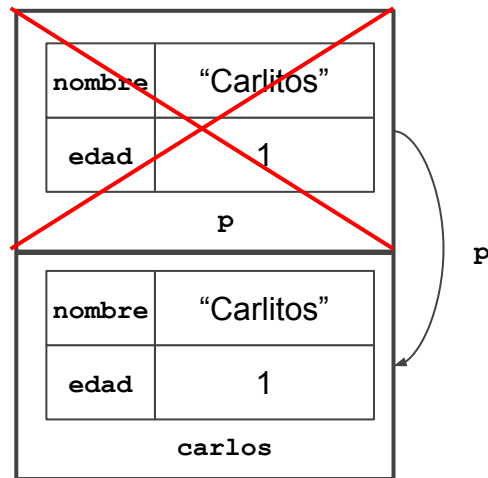
```
    ShowPersona(carlos);
```

```
}
```

Se retorna p

cumplirAnios

main



C y la memoria: TADs

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"

int main() {
    Persona carlos = nacer("Carlitos");
    carlos = cumplirAnios(carlos);
    ShowPersona(carlos);
}
```



Se completa la ejecución
de `cumplirAnios`

main

nombre	"Carlitos"
edad	1

carlos

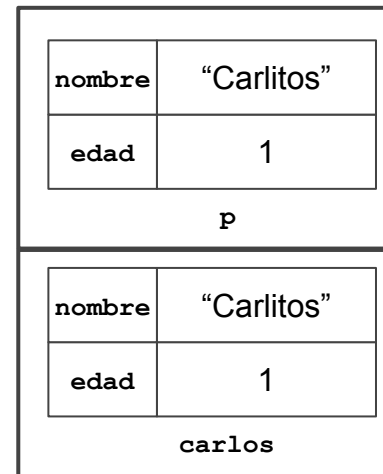
C y la memoria: TADs

```
void ShowPersona(Persona p) {  
    cout << "Persona (";  
    cout << "nombre <- \"" << p.nombre << "\", "  
    cout << "edad <- " << p.edad;  
    cout << ")" << endl;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = cumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se ejecuta ShowPersona



C y la memoria: TADs

- ❑ ¿Se pueden hacer TADs en C/C++?
 - ❑ Dificultades con el uso de memoria estática
 - ❑ Las variables se copian de ida y vuelta
 - ❑ Esto es un ineficiente uso de memoria (y de tiempo)
 - ❑ Se pueden duplicar datos erróneamente
 - ❑ ¿Qué hace la siguiente variante del código anterior?

```
int main() {  
    Persona carlos = nacer("Carlitos");  
    Persona clonCarlos = cumplirAnios(carlos);  
    ShowPersona(carlos);  
    ShowPersona(clonCarlos);  
}
```

Resumen

Resumen

- ❑ Para programar en C hay que saber usar la memoria
- ❑ No hace falta conocer el más bajo nivel (salvo algunos casos especiales)
- ❑ Para eficiencia debe considerarse el uso de la memoria
 - ❑ Pueden producirse pérdidas importantes si no
- ❑ La memoria estática NO alcanza
 - ❑ Falta un mecanismo adicional