Repaso de indirección múltiple
Recursividad
Creando nuestra propia biblitoeca

Sofía Beatriz Pérez
Daniel Agustín Rosso

sperez@iua.edu.ar
drosso@iua.edu.ar

Centro Regional Univesitario Córdoba
Instituto Univeristario Areonáutico

Clase número 10 - Ciclo lectivo 2023

Agenda

Disclaimer

Los siguientes slides tienen el objetivo de dar soporte al dictado de la asignatura. De ninguna manera pueden sustituir los apuntes tomados en clases y/o la asistencia a las mismas.
Es importante mencionar que todos este material se encuentra en un proceso de mejora continua.
Si encuentra bugs, errores de ortografía o redacción, por favor repórtelo a sperez@iua.edu.ar y/o drosso@iua.edu.ar. También puede abrir issues en el repositorio de este link: ▸ infoI_IUA_GitLab

Análisis de un caso escalar I

- Definir e inicializar una variable entera
- Crear una función capaz de sumarle uno a una variable recibida por referencia
- Crear una variable puntero a un entero y asignar la posición de memoria de la variable definida en el punto 1
- Realizar el diagrama de las cajas
- Llamar a la función de todas las maneras posibles
- Analizar que cambios puede realizar la función

Análisis de un caso escalar II

```
1  #include <stdio.h>
2
3  void acum_by_reference(int *p);
4
5  int main()
6  {    int dato = 3;
7       int *pointer_to_int = NULL;
8       acum_by_reference(&dato);
9       printf("Valor: %d\n", dato);
10      pointer_to_int = &dato;
11      acum_by_reference(pointer_to_int);
12      printf("Valor: %d\n", dato);
13      return 0;
14 }
15
```

Análisis de un caso escalar III

```
16  void acum_by_reference(int *p)
17  {
18      *(p) = *(p) + 1;
19  }
```

Análisis de un caso escalar I

Cree un programa en C que explore el uso de punteros y funciones para modificar dinámicamente los datos a los que apuntan los punteros.

1. Declare dos variables enteras llamadas dato1 y dato2 e iniciarlas con valores conocidos
2. Declare un puntero a un entero llamado pint y establece su valor inicial como NULL.
3. Asigne la dirección de memoria de dato2 al puntero pint
4. Imprima las direcciones de memorias involucradas y realice el *diagrama de las cajas*.

Análisis de un caso escalar II

5. Imprima el valor al que apunta pint después de esta asignación.
6. Defina una función llamada change_pointer capaz de cambiar la dirección de memoria a donde apunta pint. Esta función debe recibir una dirección de memoria de una variable puntero y la dirección de memoria de una variable entera
   - Proponga un prototipo para esta función
   - Analice cada una de las posibilidades
7. Llame a change_pointer tres veces pasando la dirección de pint y las direcciones de memoria de dato2, dato1 y nuevamente dato2, respectivamente.
8. Ejecutar el programa paso a paso y analizar los cambios en las variables

Análisis de un caso escalar III

```c
#include <stdio.h>

void change_pointer(int **p, int *new_data);

int main()
{
  int dato1 = 3;
  int dato2 = 2;
  int *pint = NULL;
  pint = &dato2;
  printf("Direccion dato1: %p\n", &dato1);
  printf("Direccion dato2: %p\n", &dato2);
  printf("Direccion de pint: %p\n", &pint);
  printf("Valor de pint: %p\n", pint);
  printf("Valor apuntado por pint: %d\n", *(pint));
```

Análisis de un caso escalar IV

```
16    change_pointer(&pint, &dato2);
17    printf("Direccion dato1: %p\n", &dato1);
18    printf("Direccion dato2: %p\n", &dato2);
19    printf("Direccion de pint: %p\n", &pint);
20    printf("Valor de pint: %p\n", pint);
21    printf("Valor apuntado por pint: %d\n", *(pint));
22    change_pointer(&pint, &dato1);
23    printf("Direccion dato1: %p\n", &dato1);
24    printf("Direccion dato2: %p\n", &dato2);
25    printf("Direccion de pint: %p\n", &pint);
26    printf("Valor de pint: %p\n", pint);
27    printf("Valor apuntado por pint: %d\n", *(pint));
28    change_pointer(&pint, &dato2);
29    printf("Direccion dato1: %p\n", &dato1);
30    printf("Direccion dato2: %p\n", &dato2);
```

Análisis de un caso escalar V

```
31      printf("Direccion de pint: %p\n", &pint);
32      printf("Valor de pint: %p\n", pint);
33      printf("Valor apuntado por pint: %d\n", *(pint));
34      return 0;
35  }
36
37  void change_pointer(int **p, int *new_data)
38  {
39      *(p) = new_data;
40  }
```

Análisis de un caso escalar VI

Realizar un análisis comparativo con las estructuras dinámicas. ¿Por qué necesitamos utilizar indirección múltiple para hacer push en una LIFO/FIFO?.

Introducción a la recursión I

Definición

Una función recursiva es una función que se llama a sí misma de manera directa o indirecta a través de otra función.

- La función recursiva en realidad sólo sabe cómo resolver el problema para el caso más sencillo, conocido como **caso base**.
- Si se invoca a la función desde el caso base, ésta simplemente devuelve un resultado.
- Si se llama a la función desde un problema más complejo, la función divide el problema en dos partes conceptuales. Una parte que la función sabe cómo resolver y una parte que la función no sabe cómo resolver.

Introducción a la recursión II

- La parte que la función no sabe resolver, es en general una versión mas simple del problema actual y hay un camino directo hasta llegar al caso base. Es por esto que una nueva llamada a la misma función, podría ser de ayuda para reducir aún más el problema.
- Debido a que este problema se parece al problema original, la función lanza (llama) a una nueva copia de sí misma para que trabaje con el problema más pequeño, a esto se le denomina llamada recursiva o también paso recursivo.

Ejemplos de algoritmos recursivos I

El factorial de un entero no negativo n, se escribe $n!$ y se calcula como $n * (n-1) * (n-2)$ donde por definición $1! = 1$ y $0! = 1$

```
1  #include<stdio.h>
2  int factorial(int n);
3  int main() {
4      int n=0,resultado;
5      printf("Ingrese un numero positivo\n");
6      scanf("%d",&n);
7      resultado=factorial(n);
8      printf("Factorial de %d = %d\n", n, resultado);
9      return 0;
10 }
11
12
13
```

Ejemplos de algoritmos recursivos II

```
14  int factorial(int n)
15  {
16      if (n>=1)
17          //Llamada recursiva
18          return n*  factorial(n-1);
19      else
20          //Caso base
21          return 1;
22  }
```

Recursión y uso de la stack I

Recursión y uso de la stack

En C, la recursión se implementa utilizando la pila (stack) de llamadas (call stack). La pila es una LIFO, lo que significa que la última función que se llamó es la primera que debe completarse antes de que el programa vuelva a la función que la llamó. Cuando una función es llamada recursivamente, la información sobre la llamada actual se almacena en la pila de llamadas.

Recursión y uso de la stack II

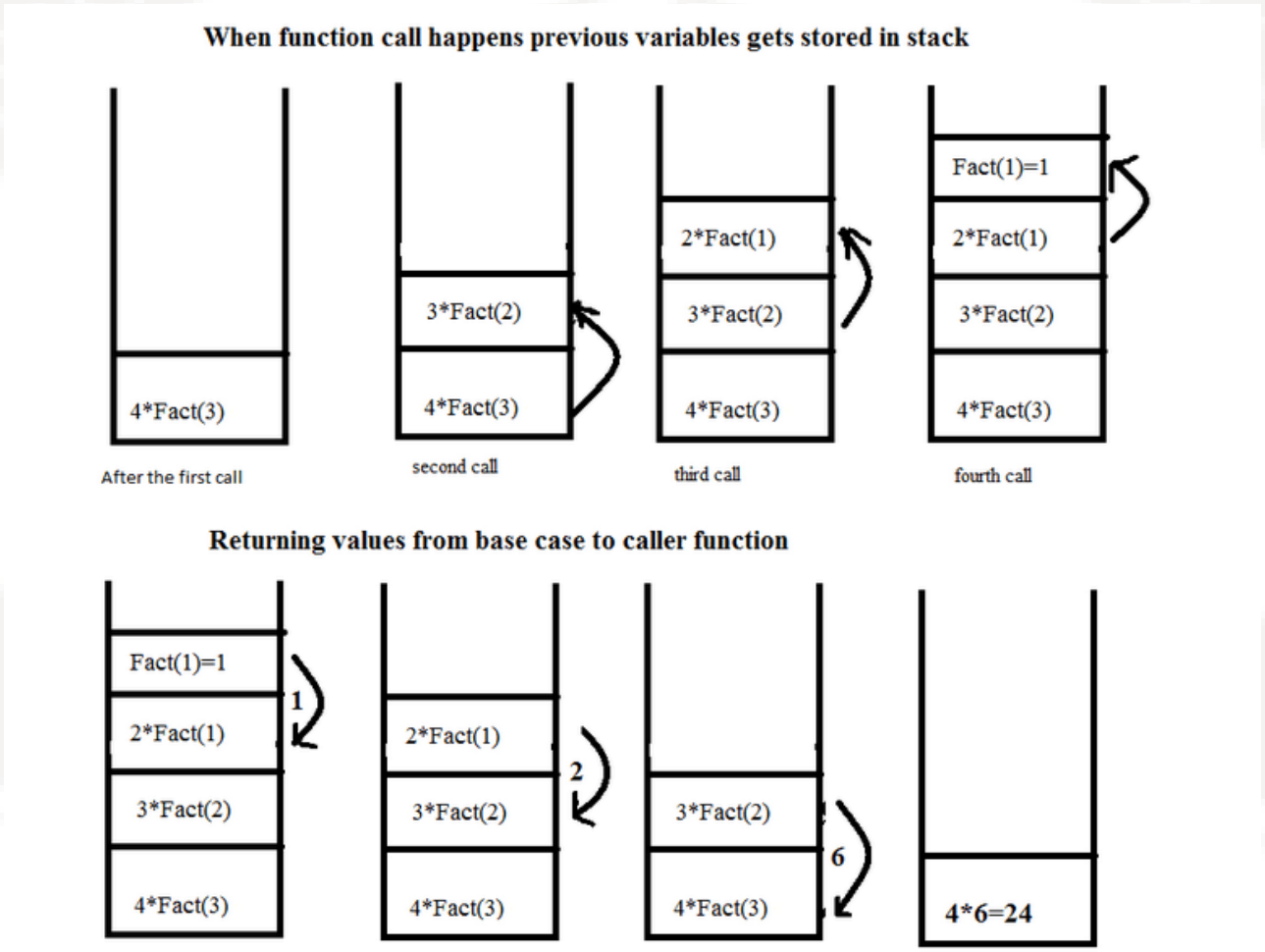
Haciendo un análisis muy sencillo, cuando una función es llamada en C, se almacenan en la pila de llamadas los siguientes elementos:

- Dirección de retorno: la dirección de memoria a la que la ejecución debe regresar después de que la función haya terminado
- Variables locales: Las variables locales de la función, es decir, las variables definidas dentro de la función, se almacenan en la pila
- Parámetros formales: Los parámetros de la función también se almacenan en la pila
- Cuando una función recursiva se llama a sí misma, se crea un nuevo marco de pila con los mismos elementos mencionados anteriormente

Recursión y uso de la stack III

- Cada llamada recursiva crea un nuevo marco de pila
- Cuando la condición de terminación de la recursión se cumple (es decir, cuando la función alcanza un caso base y no se llama a sí misma nuevamente), los marcos de pila comienzan a desapilarse
- Este proceso de apilamiento y desapilamiento de marcos de pila es fundamental para el funcionamiento de las funciones recursivas
- Cada llamada recursiva agrega un nuevo marco a la pila, y cuando la recursión alcanza el caso base, los marcos de pila se desapilan en orden inverso (del último al primero) para completar las llamadas y devolver los valores apropiados

Recursión y uso de la stack IV



Recursión y uso de la stack V

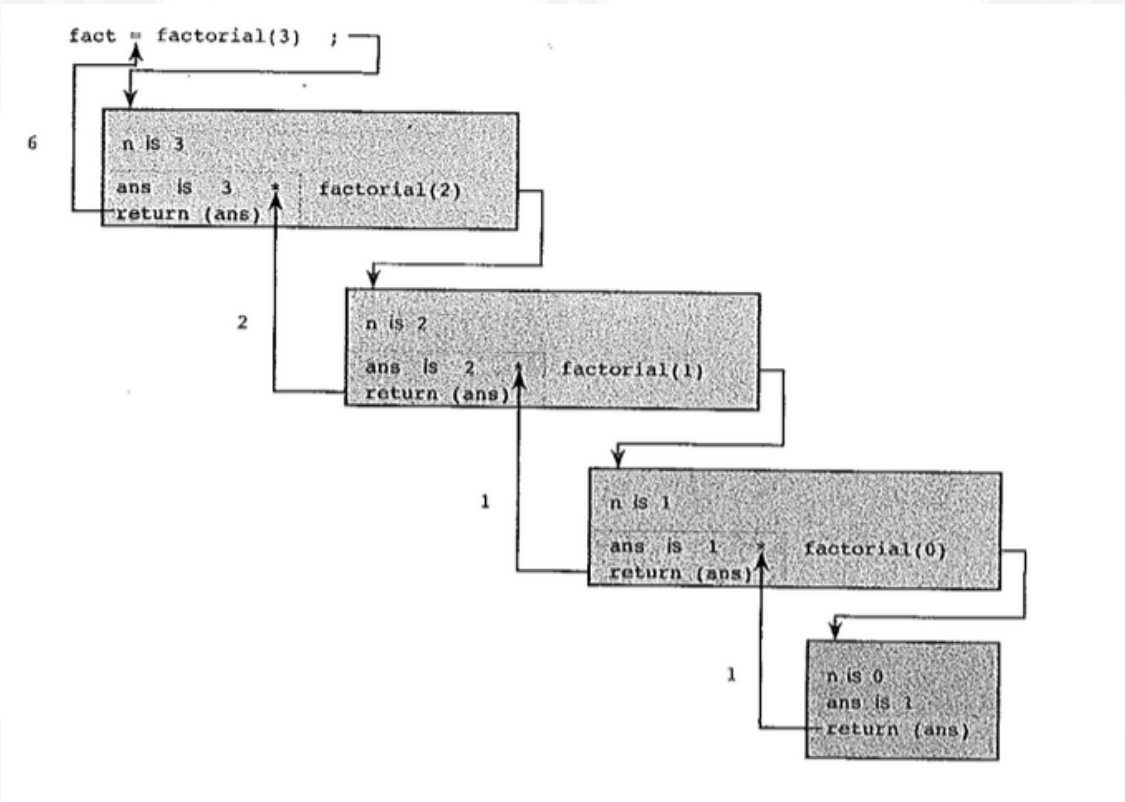


Figure: Extraído de: Problem Solving and Program Design in C, Elliot Koffman and Jeri R. Hanly

Ejemplos de algoritmos recursivos (cont) I

Calcular la suma de todos los números naturales comprendidos entre 0 y n usando recursión:

$$f(n) = 1 + 2 + 3 + ... + (n-2) + (n-1) + n \tag{1}$$

de forma genérica:

$$f(n) = \sum_{k=1}^{k=n} (k) \tag{2}$$

Ejemplos de algoritmos recursivos (cont) II

```
1  int sum(int n)
2  {
3      int temp;
4
5      if (n==0)//Caso base
6      {
7          return(n)
8      }
9      else //Caso recursivo
10     {
11         temp = n + sum (n-1);
12         return (temp);
13     }
14 }
```

Ejemplos de algoritmos recursivos (cont) III

- Serie de Fibonacci
  - ▸ Ver en Wikipedia
- El problema del Templo de Benarés
  - ▸ Ver en Wikipedia
- Algoritmo de resolución de laberintos
  - ▸ Ver en Wikipedia

Definición recursiva

Una definición recursiva se utiliza para definir los elementos de un conjunto en términos de otros elementos del conjunto.

Ejemplos:

- Un número entero es 1 o $n+1$ donde n también es un número natural
- Reglas BNF[1] son sintaxis utilizadas para describir reglas gramaticales en lenguajes de programación[2].

$$< ex >::=< number > |(< ex > * < ex >)|(< ex > + < ex >)$$

$$(5 * ((3 * 6) + 8))$$

[1] Por sus siglas en inglés Backus normal form

[2] Esto se profundizará en DHS

Ventajas y desventajas

- Ventajas
  - Algoritmos expresados de forma más clara y comprensible
  - La codificación de un algoritmo en forma recursiva en general es mas compacta que de forma iterativa
- Desventajas
  - Consumo de memoria
  - Ejecuciones mas lentas
  - Pueden resultas mas complejos de entender

Creando nuestra primera biblioteca I

- Desde los primeros programas que se han construido, se ha utilizado la directiva *#include*.
- Esto ha sido particularmente útil para no preocuparse por la codificación de las funciones *printf* o *scanf* entre otras.
- La motivación es crear nuestra propia biblioteca de funciones para, por ejemplo, poder trabajar con estructuras dinámicas re-utilizando el esfuerzo de codificación.

Para realizar esto, se trabajará utilizando el preprocesador de C [3]., junto a los calificadores estudiados anteriormente.

Creando nuestra primera biblioteca II

Algunas reglas a seguir:

1. Los archivos .h (headers) contienen sólo los prototipos de las funciones y variables globales si es que existen.
2. Los archivos .c/.cpp contiene la implementación de la función
3. Debemos garantizar que al momento de compilar no existen múltiples definiciones de variables y funciones en los diferentes archivos (utilizando extern/static)
4. Para evitar incluir múltiples veces a un mismo archivo .h, se propone el siguiente template

Creando nuestra primera biblioteca III

```
1 #ifndef FILENAME_H
2 #define FILENAME_H
3
4 //Codigo fuente
5
6 #endif
```

▸ Ver ejemplo completo en gitlab ▸ Ver ejemplo completo en gitlab

[3] https://en.wikipedia.org/wiki/C_preprocessor

¡Muchas gracias!

Consultas:

sperez@iua.edu.ar
drosso@iua.edu.ar