UNIVERSIDAD DE GUANAJUATO

Recursividad

Dr. Mario Alberto Ibarra Manzano
Departamento de Ingeniería Electrónica
División de Ingeniería del Campus Irapuato-Salamanca

Recursividad

- Definición
- Procedimientos de recursivos
- Ejemplos de casos recursivos

Definición de funciones

En la práctica es importante dividir un programa en unidades razonables e independientes, por ello, la mayoría de los lenguajes de programación, incluido C, proporcionan una forma de dividir un programa en segmentos, cada uno de los cuales puede escribirse de forma más o menos independiente de los demás.

En C, estos segmentos se denominan funciones. El código de programa en el cuerpo de una función está aislado del de otras funciones. Una función tendrá una interfaz específica con el mundo exterior en términos de cómo se le transfiere la información y cómo se transmiten los resultados generados por la función. Esta interfaz se especifica en la primera línea de la función, donde aparece el nombre de la función.

En C, el concepto de función se puede explicar de varias formas.

Por ejemplo:

- Una función es un subprograma. Te permite romper una gran tarea de computación en otras más pequeñas.
- Una función es un fragmento de código delimitado por llaves que realiza alguna tarea bien definida y también devuelve un valor.
- Una función es un componente básico de un programa. Es útil para hacer código existente reutilizable.
- Una función también se trata como un tipo derivado en C.
- Una función es una forma de ampliar el repertorio del lenguaje C

Categorías de funciones

Todas las funciones deben codificarse. Codificar una función significa escribir las instrucciones de programación para esa función.

Las funciones en C se pueden clasificar en tres categorías:

- La función *main ()*: esta función es necesaria en un programa en C. Habrá una (y solo una) función *main ()* en cada programa en C. Cuando el sistema operativo ejecuta un programa en C, en realidad ejecuta la función *main ()*. La función *main ()* llama a la biblioteca funciones y funciones definidas por el usuario, según los requisitos. Se requiere que el programador codifique la función *main ()*.
- Funciones de biblioteca: también se denominan definidas por el sistema o integradas. funciones. Por ejemplo, **printf ()** y **scanf ()** son bibliotecas funciones. Las funciones de la biblioteca son opcionales en un programa en C. Pero Es prácticamente imposible crear un programa útil que carezca de funciones de biblioteca. No es necesario que un programador codifique funciones de biblioteca. Los desarrolladores del compilador codifican las funciones de la biblioteca, compílelos y colóquelos en bibliotecas para su uso.
- Funciones definidas por el usuario: las funciones definidas por el usuario son opcionales en un programa C. Una función definida por el usuario llama a otra función definida por el usuario funciones o funciones de biblioteca, según los requisitos. La Se requiere que el programador codifique las funciones definidas por el usuario.

Elementos de una función

El último punto para tener en cuenta, antes de entrar en detalles sobre la creación de funciones, es que el cuerpo de cada función es un bloque (que puede contener otros bloques, por supuesto).

Como resultado, las variables automáticas que declara dentro de una función son local para la función y no existe en ningún otro lugar. Por lo tanto, las variables declaradas dentro de una función son bastante independientes de los declarados en otra función o en un bloque anidado.

No hay nada que le impida usar el mismo nombre para variables en diferentes funciones; permanecerán bastante separados. De hecho, esto es una ventaja. Eso sería muy difícil seguir creando nombres distintos para todas las variables en un programa grande. Es útil para poder usar el mismo nombre, como contador en diferentes funciones. Sigue siendo una buena idea evitar cualquier superposición engañosa de nombres de variables en sus diversas funciones y, por supuesto, debe intentar usar nombres que son importantes para que sus programas sean fáciles de seguir.

Declaración de una función

Cuando crea una función, especifica el encabezado de la función como la primera línea de la declaración de la función, seguida de las instrucciones que serán ejecutadas en la función encerrada entre llaves. El bloque de código entre llaves que sigue del encabezado de la función se denomina cuerpo de la función.

El encabezado de la función define el nombre de la función, los parámetros de la función (que especifican el número y los tipos de valores que se pasan a la función cuando se llama) y el tipo por el valor que devuelve la función.

El cuerpo de la función contiene las instrucciones que se ejecutan cuando se llama a la función, y estos tienen acceso a los valores que se pasan como argumentos a la función.

```
Tipo_de_dato_de_retorno identificador( Parámetros de entrada - separados por coma )
{
// Instrucciones ...
}
```

Declaración de una función

Las declaraciones en el cuerpo de la función pueden estar ausentes, pero las llaves deben estar presentes. Si no hay declaraciones en el cuerpo de una función, el tipo de retorno debe ser nulo y la función no tendrá ningún efecto. En este caso es utilizado el tipo void que significa "ausencia de cualquier tipo", y aquí significa que la función no devuelve un valor.

Una función que tiene declaraciones en el cuerpo de la función, pero no devuelve un valor también debe tener el tipo de retorno especificado como void.

Por el contrario, para una función que no tiene un tipo de retorno nulo, cada declaración de retorno en el cuerpo de la función debe devolver un valor del tipo de retorno especificado.

Definir una función con un cuerpo casi libre de contenido suele ser útil durante la fase de prueba de un programa complicado, por ejemplo, una función definida de modo que solo contiene una declaración de retorno, quizás devolviendo un valor predeterminado. Esto le permite ejecutar el programa con solo las funciones seleccionadas que realmente están haciendo algo; luego puede agregar el detalle de los cuerpos de función paso a paso, probando en cada etapa, hasta que todo se implemente y completamente probado.

El término parámetro se refiere a un marcador de posición en una definición de función que especifica el tipo de valor que debe pasarse a la función cuando se llama. El valor pasado a una función correspondiente a un parámetro se denomina argumento. Un parámetro de función consta del tipo seguido del nombre del parámetro que se utiliza dentro del cuerpo de la función para hacer referencia al valor de argumento correspondiente que se transfiere.

Declaración y prototipado de funciones

```c
// Ejemplo001.cpp: Forma de declaración por prototipado
//
```

Comentarios

```c
#include <stdio.h>
```

Librerías

```c
#define N 500000
```

Pre-procesamiento

```c
float tan(float x, int n);
float sin(float x, int n);
float cos(float x, int n);
```

Prototipos de funciones

```c
int main(int argc, char *arhv[])
{
    float x;
    int n;
    printf("Ingrese el valor de x: ");
    scanf("%f", &x);
    do {
        printf("Ingrese el numero de terminos: ");
        scanf("%d", &n);
    }while (n<1 || n>N);
    printf("tan(%f) = %f\n", x, tan(x, n));
    getchar();
    getchar();
    return 0;
}
```

Función main

```c
float tan(float x, int n)
{
    return sin(x, n) / cos(x, n);
}

float sin(float x, int n)
{
    float fct, sx;
    int i, s;
    for (i = 1, fct = x, s = 1, sx = 0; i < n; i++)
    {
        sx += (s*fct);
        s *= (-1);
        fct *= ((x/(2*i))*(x/(2*i+1)));
    }
    return sx;
}

float cos(float x, int n)
{
    static int s = 1, i = 1;
    static float fct = 1;
    s *= (-1);
    fct *= ((x/i)*(x/(i+1)));
    i += 2;
    if (n)
        return s*fct + cos(x, n - 1);
    else
        return 1;
}
```

Funciones

```c
// Ejemplo002.cpp: Forma de declaración por declaración
//
```

Comentarios

```c
#include <stdio.h>
```

Librerías

```c
#define N 500000
```

Pre-procesamiento

```c
float sin(float x, int n)
{
    float fct, sx;
    int i, s;
    for (i = 1, fct = x, s = 1, sx = 0; i < n; i++)
    {
        sx += (s*fct);
        s *= (-1);
        fct *= ((x/(2*i))*(x/(2*i+1)));
    }
    return sx;
}

float cos(float x, int n)
{
    static int s = 1, i = 1;
    static float fct = 1;
    s *= (-1);
    fct *= ((x/i)*(x/(i+1)));
    i += 2;
    if (n)
        return s*fct + cos(x, n - 1);
    else
        return 1;
}

float tan(float x, int n)
{
    return sin(x, n) / cos(x, n);
}
```

Funciones

```c
int main(int argc, char *arhv[])
{
    float x;
    int n;
    printf("Ingrese el valor de x: ");
    scanf("%f", &x);
    do {
        printf("Ingrese el numero de terminos: ");
        scanf("%d", &n);
    }while (n<1 || n>N);
    printf("tan(%f) = %f\n", x, tan(x, n));
    getchar();
    getchar();
    return 0;
}
```

Función main

Creación de librerías

Organizar una librería en lenguaje C implica estructurar el código de manera que sea modular, fácil de mantener y reutilizable.

- ***Definir la funcionalidad de la librería.***

Descripción clara: Define claramente qué problema resuelve la librería o qué funcionalidad proporciona.
Módulos: Si la librería ofrece múltiples funcionalidades, considera dividirla en módulos más pequeños.

- ***Estructura de archivos.***

Archivo de cabecera (.h): Contiene las declaraciones de funciones, macros, constantes, y definiciones de tipos de datos. También puede incluir las estructuras de datos utilizadas por la librería.
Archivo de implementación (.c): Contiene las definiciones de las funciones declaradas en el archivo de cabecera. Aquí es donde se implementa la lógica de la librería.

Ejemplo 1:

```
milibreria.h     // Declaraciones de funciones y macros.
milibreria.c     // Implementación de funciones.
```

Ejemplo 2:

```
include/
   milibreria.h   // Archivo de cabecera principal.
src/
   modulo1.c     // Implementación de un módulo.
   modulo2.c     // Implementación de otro módulo.
```

Creación de librerías

- ***Directivas de preprocesador.***

Guardas de inclusión: En el archivo .h, usa guardas de inclusión para evitar problemas de doble inclusión.

```
#ifndef MILIBRERIA_H
#define MILIBRERIA _H

// Declaraciones

#endif // MILIBRERIA _H
```

- ***Funciones y API.***

Archivo de implementación (.c): Contiene las definiciones de las funciones declaradas en el archivo de cabecera. Aquí es donde se implementa la lógica de la librería.
Nombres descriptivos: Usa nombres de funciones que describan claramente su propósito.
Abstracción: Proporciona una interfaz clara y oculta los detalles de implementación que no sean necesarios para el usuario.
Comentarios y documentación: Documenta cada función para que los usuarios sepan cómo utilizarla.

```
// milibreria.h
int suma(int a, int b);

// milibreria.c
int suma(int a, int b) {
    return a + b;
}
```

Creación de librerías

Las librerías estáticas y dinámicas son dos tipos de bibliotecas utilizadas en el desarrollo de software para compartir y reutilizar código. La principal diferencia entre ellas radica en cómo se integran con los programas que las utilizan y cuándo se realiza esta integración.

Librería estática

- **Extensión de archivo:** En sistemas Unix/Linux con extensión .a (archivo) y en Windows con extensión .lib.

- **Modo de enlace:** El código de la librería se copia y se integra directamente en el ejecutable en el momento de la compilación. Esto significa que el ejecutable resultante contiene una copia del código de la librería.

- **Tamaño del ejecutable:** Los ejecutables generados son generalmente más grandes, porque incluyen todo el código de las librerías que utilizan.

- **Distribución:** No se necesita distribuir la librería estática por separado junto con el ejecutable, ya que todo el código necesario está dentro del ejecutable.

- **Rendimiento:** Puede ser ligeramente más rápido porque no hay necesidad de cargar código externo en tiempo de ejecución.

- **Actualización:** Si se actualiza la librería estática, es necesario recompilar todos los ejecutables que la utilizan para incluir los cambios.

- **Uso:** Las librerías estáticas son útiles cuando se desea que el programa sea independiente y no dependa de archivos adicionales para ejecutarse.

Creación de librerías

Librería dinámica

- **Extensión de archivo:** En sistemas Unix/Linux con extensión .so (objeto compartido) y en Windows con extensión .dll.

- **Modo de enlace:** El código de la librería se carga y enlaza en tiempo de ejecución. El ejecutable resultante contiene referencias a la librería, pero el código de la librería no se copia en el ejecutable.

- **Tamaño del ejecutable:** Los ejecutables generados son generalmente más pequeños, ya que no incluyen el código de las librerías dinámicas. El código se carga desde la librería dinámica cuando el programa se ejecuta.

- **Distribución:** La librería dinámica debe estar disponible en el sistema donde se ejecuta el programa, ya que el ejecutable depende de ella en tiempo de ejecución.

- **Rendimiento:** Puede haber una ligera penalización en el rendimiento debido al proceso de carga de la librería en tiempo de ejecución. Sin embargo, la memoria del sistema se utiliza de manera más eficiente porque múltiples programas pueden compartir una única copia de la librería dinámica.

- **Actualización:** Las librerías dinámicas pueden actualizarse de manera independiente del programa principal. Si la librería se actualiza, los programas que la utilizan no necesitan ser recompilados, siempre y cuando la API de la librería no haya cambiado.

- **Uso:** Las librerías dinámicas son útiles en sistemas donde múltiples aplicaciones pueden compartir la misma librería, lo que ahorra memoria y facilita actualizaciones y mantenimiento.

Creación de librerías

Característica	Librería Estática	Librería Dinámica
Extensión de archivo	.a (Unix/Linux), .lib (Windows)	.so (Unix/Linux), .dll (Windows)
Enlace	En tiempo de compilación	En tiempo de ejecución
Tamaño del ejecutable	Más grande	Más pequeño
Distribución	No requiere la librería separada	Requiere la librería separada
Actualización	Requiere recompilación del ejecutable	No requiere recompilación del ejecutable
Rendimiento	Potencialmente más rápido	Ligeramente más lento, pero más eficiente en memoria
Uso	Programas independientes	Programas que comparten código en tiempo de ejecución

Creación de librerías

- ***Compilar una librería estática.***

Compilar el archivo fuente (.c) a un archivo objeto (.o)

gcc -c milibreria.c -o milibreria.o

Crear la librería estática

ar rcs libmilibreria.a milibreria.o

Compilar el programa usando la librería estática

gcc main.c -L. -lmilibreria -o miprograma

- ***Compilar una librería dinámica.***

Compilar el archivo fuente con la opción '-fPIC'

gcc -c milibreria.c -fPIC -o milibreria.o

Crear la librería dinámica (.so)

gcc -shared -o libmilibreria.so milibreria.o

Compilar el programa usando la librería dinámica

gcc main.c -L. -lmilibreria -o miprograma

Ejecutar el programa usando la librería dinámica

- **Usar 'LD_LIBRARY_PATH'**

export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
./miprograma

- **Instalar la librería en una ubicación estándar**

Copia libmilibreria.so a /usr/local/lib o a otro directorio estándar en tu sistema.

Creación de librerías

El objetivo de crear y utilizar un '*Makefile'* es automatizar el proceso de compilación y gestión de proyectos en lenguajes como C, C++, y otros lenguajes de programación. Un '*Makefile'* define un conjunto de reglas que indican cómo compilar y enlazar el código fuente para generar los ejecutables o librerías.

- ***Automatización de tareas repetitivas***

Compilación: Un '*Makefile'* automatiza el proceso de compilación de múltiples archivos fuente, eliminando la necesidad de escribir manualmente comandos de compilación para cada archivo.
Enlace: También automatiza el proceso de enlazar archivos objeto en un ejecutable o una librería.

- ***Eficiencia en la compilación***

Compilación condicional: *'Make'* recompila solo los archivos que han cambiado desde la última compilación, en lugar de recompilar todo el proyecto, lo que ahorra tiempo en proyectos grandes.
Detección de dependencias: Un *'Makefile'* puede gestionar las dependencias entre archivos. Por ejemplo, si un archivo de cabecera cambia, *'make'* recompila automáticamente los archivos fuente que dependen de él.

- ***Organización del proyecto***

Estructura: Ayuda a mantener una estructura de proyecto clara y organizada, definiendo cómo se debe construir el proyecto desde los diferentes archivos fuente y librerías.
Múltiples objetivos: Un *'Makefile'* puede definir múltiples objetivos, como compilar el proyecto, limpiar archivos temporales, ejecutar pruebas, generar documentación, etc.

Creación de librerías

- ***Portabilidad***

Consistencia en diferentes entornos: Al usar un *'Makefile'*, aseguras que el proyecto se compila de manera consistente en diferentes entornos y plataformas, siempre y cuando los requisitos sean los mismos.
Colaboración: Facilita la colaboración en equipos de desarrollo, ya que todos los desarrolladores pueden usar el mismo *'Makefile'* para compilar el proyecto de manera uniforme.

- ***Facilidad de uso***

Simplificación del proceso: Un *'Makefile'* permite compilar un proyecto completo con un solo comando (‘*make*’), en lugar de recordar y escribir manualmente complejos comandos de compilación.
Opciones personalizadas: Puedes configurar opciones específicas de compilación (como ‘*debug*’ o ‘*release*’), simplemente cambiando las variables en el *‘Makefile’*.

- ***Mantenimiento***

Modificaciones fáciles: Si cambian las dependencias, rutas de archivos o configuraciones de compilación, es más fácil actualizar un *‘Makefile’* que modificar manualmente los comandos de compilación.
Documentación implícita: Un *‘Makefile’* actúa como documentación del proceso de construcción del proyecto, indicando claramente cómo se espera que se compile y se ejecute el código.

Creación de librerías

- **Ejemplo de uso típico**

Un *'Makefile'* típico podría verse así:

```
# Variables
CC = gcc
CFLAGS = -Wall -g
TARGET = miprograma

# Regla principal (por defecto)
all: $(TARGET)

# Reglas para compilar
$(TARGET): main.o modulo1.o modulo2.o
	$(CC) $(CFLAGS) -o $(TARGET) main.o modulo1.o modulo2.o

main.o: main.c modulo1.h modulo2.h
	$(CC) $(CFLAGS) -c main.c

modulo1.o: modulo1.c modulo1.h
	$(CC) $(CFLAGS) -c modulo1.c

modulo2.o: modulo2.c modulo2.h
	$(CC) $(CFLAGS) -c modulo2.c

# Regla para limpiar los archivos generados
clean:
	rm -f *.o $(TARGET)
```

Creación de librerías

- ***Compilación y uso.***

Makefile: Si la librería es grande, usa un *'Makefile'* para automatizar la compilación. Define cómo se deben compilar y enlazar los archivos.

Ejemplo:

```
CC = gcc
CFLAGS = -Iinclude -Wall
OBJ = milibreria.o

all: milibreria.a

libmilibreria.a: $(OBJ)
    ar rcs liblibreria.a $(OBJ)

milibreria.o: src/milibreria.c
    $(CC) $(CFLAGS) -c src/milibreria.c -o milibreria.o

clean:
    rm -f *.o libmylibrary.a
```

Enlace: Los usuarios de la librería solo necesitarán incluir el archivo de cabecera (mylibrary.h) y enlazar la librería (libmylibrary.a o libmylibrary.so) durante la compilación.

Definición de recursividad

La recursividad es un concepto en programación y matemáticas donde una función o un algoritmo se llama a sí mismo para resolver un problema. En lugar de resolver el problema directamente, el algoritmo divide el problema en subproblemas más pequeños, cada uno de los cuales es una instancia del problema original. La recursión continúa hasta que se alcanza una condición base, que es un caso simple que puede resolverse sin necesidad de más recursión.

La recursividad es una técnica utilizada para diseñar algoritmos que procesan datos de manera jerárquica o repetitiva, aplicando una misma operación a subconjuntos de datos hasta alcanzar un nivel de granularidad que permite obtener un resultado final. Esta técnica es común en algoritmos de búsqueda, optimización y procesamiento de estructuras de datos complejas, como árboles de decisión, donde la operación recursiva descompone la tarea en subproblemas hasta que se alcanza una condición de finalización, conocida como condición de parada o base, que permite resolver el problema en su totalidad.

Conceptos clave

Caso base

- Es la condición que detiene la recursión. Sin un caso base, la función recursiva entraria en un bucle infinito, lo que eventualmente llevaría a un desbordamiento de la pila (stack overflow).

Caso recursivo

- Es la parte de la función en la que la función se llama a sí misma con un subproblema más pequeño, acercándose gradualmente al caso base.

Ventajas

- La recursividad puede simplificar la solución de problemas complejos al decomponerlos en subproblemas más manejables.

Deventajas

- Puede ser menos eficiente en términos de tiempo y espacio debido a la sobrecarga de múltiples llamadas de función y el uso de la pila. Además, si no se diseña cuidadosamente, puede llevar a errores como el desbordamiento de la pila.

Procedimientos de recursivos

El procedimiento general para implementar recursividad en un problema de programación es el siguiente:

1. **Definición del problema:** Primero, define claramente el problema que se va a resolver y determina si es adecuado para un enfoque recursivo, es decir, si se puede descomponer en subproblemas similares al original.

2. **Identificación de la condición base:** Determina una o más condiciones base que detendrán la recursión. Estas condiciones deben representar los casos más simples del problema, que pueden resolverse directamente sin más llamadas recursivas.

3. **División del problema:** Descompone el problema en subproblemas más pequeños que son instancias del mismo problema, pero con un tamaño o complejidad reducidos.

Procedimientos de recursivos

El procedimiento general para implementar recursividad en un problema de programación es el siguiente:

4. **Implementación de la función recursiva:**

   Escribe una función que:

- Verifique primero la condición base. Si se cumple, la función debe devolver una solución directa sin hacer más llamadas recursivas.

- Si la condición base no se cumple, la función debe realizar una llamada a sí misma con los subproblemas descompuestos. Los resultados de estas llamadas recursivas se utilizan para construir la solución del problema original.

Procedimientos de recursivos

El procedimiento general para implementar recursividad en un problema de programación es el siguiente:

5. **Combinación de resultados:** Si la recursión produce varios resultados parciales, combina estos resultados para formar la solución final al problema original.

6. **Prueba y validación:** Prueba la función recursiva con varios casos de entrada, para asegurarte de que la función se comporta como se espera y que la recursión siempre termine.

Ejemplos de casos recursivos

Factorial de un número n

- **Problema:**
  Calcular el factorial de un número n

  $0! = 1$
  $1! = 0! \times 1$
  $2! = 1! \times 2$
  $3! = 2! \times 3$
  $\vdots \qquad \vdots$
  $n! = (n-1)! \times n$

- **Condición base:**
  Sí $n = 0$, devolver 1 porque $0! = 1$

Ejemplos de casos recursivos

Factorial de un número n

- **División:**

  El factorial de n se expresa en función $n! = (n-1)! \times n$

  Podemos definir la función respecto de si misma de la siguiente forma:

$$factorial(n) = factorial(n-1) \times n$$

- **Implementación:**

  El pseudocódigo para la función factorial sería:

```
FUNCION Factorial(n)
        SI (n == 1) ENTONCES
                RETORNAR 1   // Condición base: factorial de 1 es 1
        FIN SI
        // Llamada recursiva: n * factorial de (n-1)
        RETORNAR n * Factorial(n - 1)
FIN FUNCION
```

Ejemplos de casos recursivos

Factorial de un número n

- **Implementación:**

  La implementación en lenguaje C podría ser:

```
int factorial(int n)
{
        if( n == 0) return 1;           // Condición base
        return n*factorial(n-1);        // Llamada recursiva
}
```

- **Combinación de resultados:**

  El algoritmo muestra solo un resultado por lo tanto no es necesario realizar una combinación

Ejemplos de casos recursivos

Factorial de un número n

- **Prueba:**

Considerando el código para $n = 6$

factorial(6)
return 6*factorial(5);
return 5*factorial(4);
return 4*factorial(3);
return 3*factorial(2);
return 2*factorial(1);
return 1*factorial(0)
return 1;
return 1*1=1;
return 2*1=2;
return 3*2=6;
return 4*6=24;
return 5*24=120;
return 6*120=720;
720

Procedimientos de recursivos

Diseñar un algoritmo basado en recursividad requiere un enfoque estructurado para garantizar que el algoritmo sea correcto y eficiente. Aquí se detalla el procedimiento general que puedes seguir:

1. **Entender el Problema.**
- Analiza el problema para determinar si puede dividirse en subproblemas más pequeños del mismo tipo.
- Identifica la recurrencia natural en el problema. Es decir, si una solución para el problema más grande se puede construir a partir de soluciones para problemas más pequeños.

2. **Definir el Caso Base.**
- Identifica el caso más simple del problema, aquel que se puede resolver sin necesidad de recurrir a la recursividad. Este caso base es crucial para detener la recursión.
- Asegúrate de que el caso base sea correcto y que realmente detenga la recursión. Un caso base incorrecto puede llevar a bucles infinitos.

Procedimientos de recursivos

3. **Definir el Caso Recursivo.**
- Desglosa el problema en subproblemas más pequeños. Pregúntate cómo puedes resolver el problema más grande si ya conoces la solución a un problema más pequeño.
- Expresa la solución del problema grande en términos de la solución de uno o más subproblemas.
- Asegúrate de que cada llamada recursiva se acerque al caso base. Esto es fundamental para garantizar que la recursión termine eventualmente.

4. **Diseñar la Función Recursiva.**
- Implementa la función que realiza la tarea recursiva. La función debe verificar primero si se ha alcanzado el caso base, y si no, realizar la llamada recursiva.
- Asegúrate de que los argumentos de la llamada recursiva cambien de manera que la recursión progrese hacia el caso base.

Procedimientos de recursivos

5. **Verificar Funcionamiento y Termino.**
- Verifica que la recursión esté correctamente definida para todos los posibles valores de entrada.
- Asegúrate de que cada rama de la recursión eventualmente alcanza un caso base. Si no, la función podría entrar en un bucle infinito..

6. **Optimización (si es necesario).**
- Considera optimizar el algoritmo si es posible o necesario. Algunas técnicas incluyen:
  - *Memorización:* Almacenar resultados de subproblemas ya resueltos para evitar cálculos redundantes.
  - *Transformar recursión en iteración:* A veces es posible reformular un algoritmo recursivo como uno iterativo, lo que puede ahorrar espacio en la pila.

Procedimientos de recursivos

7. **Probar el Algoritmo**
   - Realiza pruebas con varios casos, incluyendo casos extremos y el caso base, para asegurarte de que la función recursiva se comporta como se espera.
   - Analiza la complejidad en términos de tiempo y espacio. La recursividad puede implicar una sobrecarga de llamadas de función, lo que es importante considerar en problemas de mayor tamaño.

8. **Documentar y Refactorizar**
   - Documenta el algoritmo, explicando claramente el caso base, el caso recursivo y cómo se estructura la llamada recursiva.
   - Refactoriza si es necesario para mejorar la claridad, eficiencia o legibilidad del código.

Ejemplos de casos recursivos

Serie de Fibonacci

- **Entender el Problema:**
  Calcular la serie de Fibonacci $F(n)$ de un número n

  $F(0) = 0$
  $\mathrm{F}(1) = 1$
  $F(2) = F(1) + F(0) = 1$
  $F(3) = F(2) + F(1) = 2$
  $\vdots \qquad \vdots$
  $F(n) = F(n-1) + F(n-2)\ para\ n \geq 2$

- **Definir el Caso Base:**
  Sí $n = 0$, devolver 0 porque $\mathrm{F}(0) = 0$
  Sí $n = 1$, devolver 1 porque $\mathrm{F}(1) = 1$

Ejemplos de casos recursivos

- **Definir el Caso Recursivo:**
  Sí $n = 2$ devolver $F(2) = F(1) + F(0) = 1$
  Sí $n = 3$ devolver $F(3) = F(2) + F(1) = 2$
  Entonces el caso recursivo es $F(n) = F(n-1) + F(n-2)\ para\ n \geq 2$

- **Diseñar la Función Recursiva:**

```
int fb(int n)
        {
        if(n==0)
                return 0;
        else if(n==1)
                return 1;
        else
                return fb(n-1)+fb(n-2);
        }
```

Ejemplos de casos recursivos

- **Verificar Funcionamiento y Termino:**
  Sí $n = 0$ devolver $F(0) = 0$
  Sí $n = 1$ devolver $F(1) = 1$
  Sí $n \geq 2$ devolver $F(n) = F(n-1) + F(n-2)$
  Sí $n < 0$ devolver $F(n) = 0$

- **Optimización (si es necesario):**

```
int fb(int n)
        {
        return n>0?(n==1?1: fb(n-1)+fb(n-2)):0;
        }
```

Ejemplos de casos recursivos

- **Probar el Algoritmo:**

  Sí $n = 12$ *entonces evaluamos la función* $fb(n) = fb(12)$

```
int fb(int n)
        {
        return n>0?(n==1?1: fb(n-1)+fb(n-2)):0;
        }
```

fb(12) = fb(11)+fb(10) = 144
fb(11) = fb(10)+fb(9) = 89
fb(10) = fb(9)+fb(8) = 55
fb(9) = fb(8)+fb(7) = 34
fb(8) = fb(7)+fb(6) = 21
fb(7) = fb(6)+fb(5) = 13
fb(6) = fb(5)+fb(4) = 8
fb(5) = fb(4)+fb(3) = 5
fb(4) = fb(3)+fb(2) = 3
fb(3) = fb(2)+fb(1) = 2
fb(2) = fb(1)+fb(0) = 1

Ejemplos de casos recursivos

Usted tiene $15. Usted va a la tienda de chocolates y encuentra que el precio por chocolate es de $1. El encargado le comenta que usted puede cambiar tres envolturas por otro chocolate. ¿Cuál es el número máximo de chocolates que puede obtener?

Solución.

- Con $15 se compran 15 chocolates con un costo de $1 por chocolate.
- Se cambian las 15 envolturas por 5 chocolates y quedan 0 envolturas.
- Se cambian las 5 envolturas por 1 chocolate y queda 2 envolturas.
- Se cambian las 3 envolturas por 1 chocolate y quedan 0 envolturas.
- Debido a que las envolturas no son suficientes para cambiar chocolates se detiene le proceso.
- El total de chocolates sería 15+5+1+1=22

Caso general:

Entrada

- Monto inicial: $X.
- Precio por chocolate: $Y.
- Cambio de A envolturas por B chocolates.

Salida

- Z es el número total de chocolates.

Ejemplos de casos recursivos

Usted tiene $X. Usted va a la tienda de chocolates y encuentra que el precio por chocolate es de $Y. El encargado le comenta que usted puede cambiar A envolturas por B chocolates. ¿Cuál es el número máximo de chocolates que puede obtener?

Entrada

- Monto inicial: $X.
- Precio por chocolate: $Y.
- Cambio de A envolturas por B chocolates.

Salida

- Z es el número total de chocolates.

#	Monto inicial (X)	Precio por chocolate (Y)	Número de envolturas (A)	Número de chocolates (B)	Total de chocolates
1	15	1	3	1	22
2	20	2	3	2	26
3	30	5	2	1	11
4	120	3	5	1	49
5	300	10	5	2	48
6	250	30	5	3	14

Ejemplos de casos recursivos

Usted tiene $15. Usted va a la tienda de chocolates y encuentra que el precio por chocolate es de $1. El encargado le comenta que usted puede cambiar 3 envolturas por 1 chocolate. ¿Cuál es el número máximo de chocolates que puede obtener?

Solución.

- Con $\$15$ se compran $\lfloor {}^{15}/_{1} \rfloor = 15$ chocolates considerando un costo de $\$1$ por chocolate, estás son las primeras 15 envolturas.

- Se cambian las 15 envolturas por $\lfloor {}^{15}/_{3} \rfloor \times 1 = 5$ chocolates y quedan $15 - \lfloor {}^{5}/_{1} \rfloor \times 3 = 0$ envolturas, sumado los chocolates y envolturas se tendrán $0 + 5 = 5$ envolturas.

- Se cambian las 5 envolturas por $\lfloor {}^{5}/_{3} \rfloor \times 1 = 1$ chocolate y queda $5 - \lfloor {}^{1}/_{1} \rfloor \times 3 = 2$ envolturas, sumando los chocolates y envolturas se tendrán $2 + 1 = 3$ envolturas.

- Se cambian las 3 envolturas por $\lfloor {}^{3}/_{3} \rfloor \times 1 = 1$ chocolate y quedan $3 - \lfloor {}^{1}/_{1} \rfloor \times 3 = 0$ envolturas, sumando los chocolates y envolturas se tendrán $0 + 1 = 1$ envolturas.

- Debido a que las envolturas no son suficientes para cambiar chocolates se detiene le proceso.

- El total de chocolates sería $15 + 5 + 1 + 1 = 22$.

Ejemplos de casos recursivos

Usted tiene $20. Usted va a la tienda de chocolates y encuentra que el precio por chocolate es de $2. El encargado le comenta que usted puede cambiar 3 envolturas por 2 chocolate. ¿Cuál es el número máximo de chocolates que puede obtener?

Solución.

- Con $\$20$ se compran $\lfloor {}^{20}/_{2} \rfloor = 10$ chocolates considerando un costo de $\$2$ por chocolate, estás son las primeras 10 envolturas.
- Se cambian las 10 envolturas por $\lfloor {}^{10}/_{3} \rfloor \times 2 = 6$ chocolates y quedan $10 - \lfloor {}^{6}/_{2} \rfloor \times 3 = 1$ envoltura, sumado los chocolates y envolturas se tendrán $1 + 6 = 7$ envolturas.
- Se cambian las 7 envolturas por $\lfloor {}^{7}/_{3} \rfloor \times 2 = 4$ chocolate y queda $7 - \lfloor {}^{4}/_{2} \rfloor \times 3 = 1$ envolturas, sumando los chocolates y envolturas se tendrán $1 + 4 = 5$ envolturas.
- Se cambian las 5 envolturas por $\lfloor {}^{5}/_{3} \rfloor \times 2 = 2$ chocolate y quedan $5 - \lfloor {}^{2}/_{2} \rfloor \times 3 = 2$ envolturas, sumando los chocolates y envolturas se tendrán $2 + 2 = 4$ envolturas.
- Se cambian las 4 envolturas por $\lfloor {}^{4}/_{3} \rfloor \times 2 = 2$ chocolate y quedan $4 - \lfloor {}^{2}/_{2} \rfloor \times 3 = 1$ envoltura, sumando los chocolates y envolturas se tendrán $1 + 2 = 3$ envolturas.
- Se cambian las 3 envolturas por $\lfloor {}^{3}/_{3} \rfloor \times 2 = 2$ chocolate y quedan $3 - \lfloor {}^{2}/_{2} \rfloor \times 3 = 0$ envoltura, sumando los chocolates y envolturas se tendrán $0 + 2 = 2$ envolturas.
- Debido a que las envolturas no son suficientes para cambiar chocolates se detiene le proceso.
- El total de chocolates sería $10 + 6 + 4 + 2 + 2 + 2 = 26$.

Ejemplos de casos recursivos

Usted tiene $X. Usted va a la tienda de chocolates y encuentra que el precio por chocolate es de $Y. El encargado le comenta que usted puede cambiar A envolturas por B chocolate. ¿Cuál es el número máximo de chocolates que puede obtener?

Solución.

- Chocolates iniciales: $\lfloor {}^{X}/_{Y} \rfloor = Z_0$, se generan $Z_0 = E$ envolturas.

- Se cambien las E envolturas por $\lfloor {}^{E}/_{A} \rfloor \times B = Z_1$ chocolates, quedando $E - \left\lfloor {}^{Z_1}/_{B} \right\rfloor \times A$ envolturas y agregando las Z_1 envolturas generadas se tendrán $E - \left\lfloor {}^{Z_1}/_{B} \right\rfloor \times A + Z_1 = E$ envolturas.

- Se cambien las E envolturas por $\lfloor {}^{E}/_{A} \rfloor \times B = Z_2$ chocolates, quedando $E - \left\lfloor {}^{Z_2}/_{B} \right\rfloor \times A$ envolturas y agregando las Z_2 envolturas generadas se tendrán $E - \left\lfloor {}^{Z_2}/_{B} \right\rfloor \times A + Z_2 = E$ envolturas.

- De forma general se tiene: se cambien las E envolturas por $\lfloor {}^{E}/_{A} \rfloor \times B = Z_i$ chocolates, quedando $E - \left\lfloor {}^{Z_i}/_{B} \right\rfloor \times A$ envolturas y agregando las Z_i envolturas generadas se tendrán $E - \left\lfloor {}^{Z_i}/_{B} \right\rfloor \times A + Z_i = E$ envolturas.

- El proceso se detiene cuando el número de envolturas no es suficiente es decir si $Z_i = 0$.

- El total de chocolates sería $Z_0 + Z_1 + Z_2 + \cdots = \sum Z_i$.

Ejemplos de casos recursivos

Usted tiene $X. Usted va a la tienda de chocolates y encuentra que el precio por chocolate es de $Y. El encargado le comenta que usted puede cambiar A envolturas por B chocolate. ¿Cuál es el número máximo de chocolates que puede obtener?

Solución.

```
int cambio(int E, int A, int B)
{
        int Zi;
        Zi = (E/A)*B;
        return Zi?Zi+cambio(Zi+E-(Zi*A)/B,A,B):0;
}
int chocolate(int X, int Y, int A, int B)
{
        return X/Y+cambio(X/Y,A,B);
}
```

chocolate(15, 1, 3, 1)
15/1+cambio(15/1,3,1)=15+cambio(15,3,1)
5+cambio(5+15-(5*3)/1,3,1)=5+cambio(5,3,1) Zi=(15/3)*1=5
1+cambio(1+5-(1*3)/1,3,1)=1+cambio(3,3,1) Zi=(5/3)*1=1
1+cambio(1+3-(1*3)/1,3,1)=1+cambio(1,3,1) Zi=(3/3)*1=1
0 Zi=(1/3)*1=0
chocolate(15 1, 3, 1) = 15+5+1+1+0=22

Alcance de variables.

En C, el alcance (scope) de una variable se refiere al contexto dentro del cual la variable es visible y accesible. El alcance determina dónde una variable puede ser utilizada en el código. Hay varios tipos de alcance de variables en C:

- ***Alcance de Bloque (Local)***

Definición: Una variable declarada dentro de un bloque de código, como dentro de una función o un bloque de control (como un if, for, while, etc.), tiene un alcance local o de bloque.
Visibilidad: Solo es accesible dentro de ese bloque de código en el que fue declarada.
Duración: La memoria de la variable se asigna cuando se entra en el bloque y se libera al salir del bloque.

- ***Alcance de Función (Estático dentro de una Función)***

Definición: Una variable declarada dentro de una función con la palabra clave static tiene un alcance de función.
Visibilidad: Solo es accesible dentro de la función donde fue declarada, similar al alcance de bloque.
Duración: La variable mantiene su valor entre llamadas a la función. Su duración es la vida del programa.

Alcance de variables.

En C, el alcance (scope) de una variable se refiere al contexto dentro del cual la variable es visible y accesible. El alcance determina dónde una variable puede ser utilizada en el código. Hay varios tipos de alcance de variables en C:

- ***Alcance Global***

Definición: Una variable declarada fuera de todas las funciones tiene un alcance global.
Visibilidad: Es accesible desde cualquier función o bloque de código en el archivo donde fue declarada, después de su punto de declaración.
Duración: La variable tiene duración estática; existe durante toda la ejecución del programa.

- ***Alcance de Archivo (Estático Global)***

Definición: Una variable global declarada con la palabra clave static tiene un alcance de archivo.
Visibilidad: Solo es accesible dentro del archivo donde fue declarada, no es visible desde otros archivos incluso si están incluidos.
Duración: La variable tiene duración estática.

Alcance de variables.

En C, de forma predeterminada, las variables tienen un alcance local, lo que significa que su uso está limitado al bloque de código en el que fueron definidas, como dentro de una función o un bloque de control (if, for, while, etc.). Estas variables se denominan automáticas porque se crean automáticamente en el punto en el que se declaran y se destruyen automáticamente cuando la ejecución del programa sale del bloque en el que fueron declaradas. La memoria utilizada por estas variables se libera para que otras funciones la utilicen.

Este comportamiento es muy eficiente, ya que los datos almacenados en la memoria de una función solo se retienen mientras se ejecutan las declaraciones dentro de esa función. Sin embargo, hay situaciones en las que es útil retener información entre diferentes llamadas a una misma función. Por ejemplo, podrías querer llevar un recuento de cuántas veces se ha llamado a la función o cuántas líneas de salida se han escrito.

Con las variables automáticas, no es posible retener valores entre llamadas a la función porque su memoria se libera al final de cada ejecución. Para resolver esto, C proporciona variables de tipo *static*.

Una variable *static* tiene una duración de almacenamiento estática, lo que significa que conserva su valor entre diferentes llamadas a la función en la que fue declarada. Aunque su alcance es el mismo que el de una variable local (es decir, es visible solo dentro de la función o bloque donde fue declarada), su duración persiste durante toda la ejecución del programa. Esto permite que la función mantenga un estado entre llamadas sin necesidad de utilizar variables globales.

Alcance de variables.

Ejemplo de Variable Automática (Local)

```
#include <stdio.h>

void contar_llamadas_auto() {
    int count = 0;  // Variable automática (local)
    count++;
    printf("Llamada con variable automática: %d\n", count);
}

int main() {
    contar_llamadas_auto();  // Imprime 1
    contar_llamadas_auto();  // Imprime 1
    contar_llamadas_auto();  // Imprime 1
    return 0;
}
```

Alcance de variables.

Ejemplo de Variable Estática

```
#include <stdio.h>

void contar_llamadas_static() {
    static int count = 0;  // Variable estática
    count++;
    printf("Llamada con variable estática: %d\n", count);
}

int main() {
    contar_llamadas_static();  // Imprime 1
    contar_llamadas_static();  // Imprime 2
    contar_llamadas_static();  // Imprime 3
    return 0;
}
```

Alcance de variables.

Ejemplo de Variable Global vs. Estática Global

Variable Global:

```
#include <stdio.h>

int count = 0;  // Variable global

void incrementar_contador() {
    count++;
}

int main() {
    incrementar_contador();
    printf("Contador global: %d\n", count);  // Imprime 1
    incrementar_contador();
    printf("Contador global: %d\n", count);  // Imprime 2
    return 0;
}
```

Alcance de variables.

Ejemplo de Variable Global vs. Estática Global

Variable Estática Global:

```
#include <stdio.h>

static int count = 0;  // Variable estática global

void incrementar_contador() {
    count++;
}

int main() {
    incrementar_contador();
    printf("Contador estático global: %d\n", count);  // Imprime 1
    incrementar_contador();
    printf("Contador estático global: %d\n", count);  // Imprime 2
    return 0;
}
```

Alcance de variables.

Comparación entre Variable Automática y Estática en la Misma Función

```
#include <stdio.h>

void comparar_variables() {
    int local_count = 0;  // Variable automática
    static int static_count = 0;  // Variable estática

    local_count++;
    static_count++;

    printf("local_count = %d, static_count = %d\n", local_count, static_count);
}

int main() {
    comparar_variables();  // Imprime: local_count = 1, static_count = 1
    comparar_variables();  // Imprime: local_count = 1, static_count = 2
    comparar_variables();  // Imprime: local_count = 1, static_count = 3
    return 0;
}
```

Ejemplos de casos recursivos

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Solución.

```
float potencia(float x, int n)
{
        return n?x*potencia(x,n-1):1;
}

long int factorial(long int n)
{
        return n?n*factorial(n-1):1;
}

float exp1(float x, int n)
{
        return n?potencia(x,n)/factorial(n)+exp1(x,n-1):1;
}
```

Ejemplos de casos recursivos

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Solución.

```
float fct(float x, int n)
{
        return n?(x/n)*fct(x,n-1):1;
}
```

```
float exp3(float x, int n)
{
        return n?fct(x,n)+exp3(x,n-1):1;
}
```

Ejemplos de casos recursivos

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Solución.

```
float exp4(float x, int n)
{
        static float fct1=1;
        static int i=0;
        fct1*=(x/(i+1));
        if(n-i++)
                return fct1+exp4(x,n);
        else
        {
                fct1=1;
                i=0;
                return 1;
        }
}
```

Ejemplos de casos recursivos

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Solución.

```
#include<stdio.h>
#include<math.h>
#include<time.h>
#include"libarram.h"

int main(int argc, char *argv[])
{
        int i, n;
        float x, exr, ex1, ex2, ex3, err1, err2, err3;
        clock_t ini, fin;
        double t1, t2 ,t3;
        scanf("%f %d", &x, &n);
        exr = exp(x);
        for(i=1; i<n; i<<=1)
        {
                ini = clock();
                ex1 = exp1(x, i);
                fin = clock();
                err1 = fabs(exr-ex1);
                t1 = ((double) (fin - ini))/CLOCKS_PER_SEC;
                ini = clock();
                ex2 = exp3(x, i);
                fin = clock();
                err2 = fabs(exr-ex2);
                t2 = ((double) (fin - ini))/CLOCKS_PER_SEC;
                ini = clock();
                ex3 = exp4(x, i);
                fin = clock();
                t3 = ((double) (fin - ini))/CLOCKS_PER_SEC;
                err3 = fabs(exr-ex3);
                printf("%d\t%.4f (%.4lf)\t%.4f (%lf)\t%.4f (%lf)\n", i, err1, t1, err2, t2, err3, t3);
        }
        return 0;
}
```

Ejemplos de casos recursivos