Unidad 1: LENGUAJES PROCEDURALES

La comunicación necesita comprensión mutua de cierto conjunto de símbolos y reglas del lenguaje.

Lenguaje de Programación:
Un lenguaje de programación es un sistema notacional para describir computaciones en una forma legible tanto para la máquina como para el ser humano.
Su objetivo es construir programas, escritos por personas. Los programas se ejecutarán sobre una computadora que realizará las tareas descritas. Utilizar un lenguaje de programación requiere, comprensión mutua por parte de personas y máquinas.

Computar:
Se puede interpretar como "cualquier proceso que puede ser realizado por una computadora“. No solo cálculos matemáticos sino también manipulación de datos, procesamiento de textos, almacenamiento y recuperación de la información.

Cuestiones de Diseño:
El principal objetivo de la abstracción en el diseño de lenguajes de programación es el control de la complejidad. Se controla la complejidad elaborando abstracciones que ocultan los detalles cuando es apropiado.
El diseño de lenguajes exige:

- **legibilidad:** por parte del ser humano es un requisito complejo y sutil.
- **abstracción:** permitir concentrarse en un problema al mismo nivel de generalización, dejando de lado los detalles irrelevantes.

Tipos de Abstracciones:
• **Abstracciones de datos:** Técnica de inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, facilitar la escritura del programa. (Ejemplo: uso de struct en lenguaje C).
    • int x; abstracción de datos básica.
    • float notas[8]; abstracción de datos estructurada
• **Abstracciones de control:** Resume propiedades de la transferencia de control, es decir, la modificación de la trayectoria de ejecución de un programa en una determinada situación. (Ejemplo: for, if, while, etc, en C).
    • a=a+3; abstracción de control básica, resume el cómputo y almacenamiento de un valor en el lugar de almacenamiento de la variable “a”
    • sentencias estructuras for, if, while abstracciones de control estructurada
• **Abstracciones procedimentales:** Es una secuencia nombrada de instrucciones que tienen una función específica y limitada. (Ejemplo: uso de funciones en C).
    • Secuencia nombrada de instrucciones que tienen una función específica (conocidos como funciones o subprogramas).

Definición de un lenguaje de programación:
La definición de un lenguaje de programación necesita una descripción precisa y completa, además de un traductor que permita aceptar el programa en el lenguaje en cuestión y que, lo ejecute directamente o bien lo transforme en una forma adecuada para su ejecución.
Dos partes se pueden distinguir en la definición de un lenguaje: La sintaxis estudia las reglas de formación de frases. Las reglas de sintaxis nos dicen cómo se escriben los enunciados, declaraciones y otras construcciones del lenguaje. La semántica, sin embargo, hace referencia al significado de esas construcciones.

Computadoras Simuladas por Software y Traducción

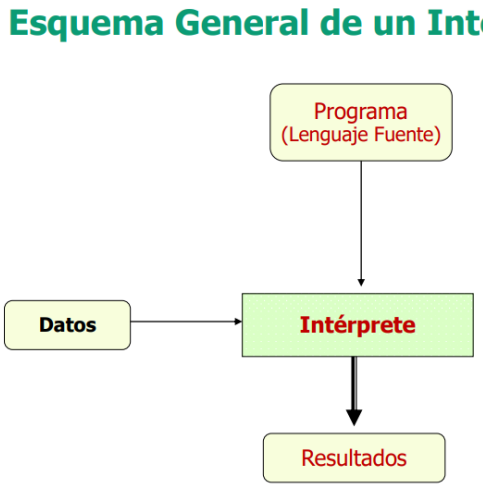
¿Cómo se pueden ejecutar programas en lenguajes de alto nivel en computadoras reales con lenguaje de bajo nivel?

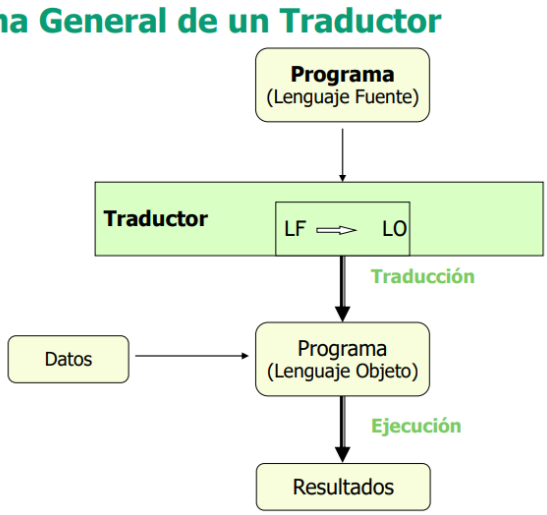
Simulación por software o Interpretación:

es un software que recibe un programa en lenguaje de alto nivel, lo analiza y lo ejecuta. Para analizar el programa completo, va traduciendo sentencias de código y ejecutándolas si están bien, así hasta completar el programa origen.

Traductor:

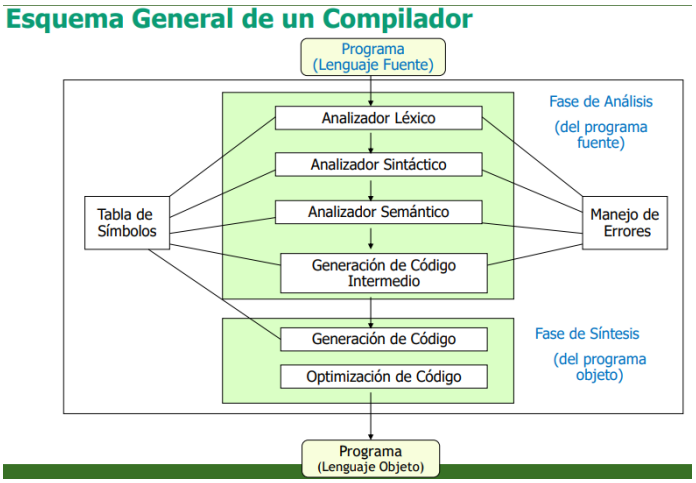
Es un "procesador de lenguajes" que acepta programas en cierto lenguaje fuente como entrada y produce programas funcionalmente equivalentes en otro lenguaje objeto.





Compilación:

Un compilador es un traductor que transforma un programa escrito en un lenguaje de alto nivel (lenguaje fuente) en otro programa escrito en un lenguaje de bajo nivel.



Implementación de un lenguaje de programación:

- Cuando se implementa un lenguaje de programación, las estructuras de datos y algoritmos que se utilizan en tiempo de ejecución de un programa escrito en ese lenguaje, definen un modelo de ejecución definida por la implementación del lenguaje.
- Múltiples decisiones debe tomar el implementador en función de los recursos de hardware y software disponibles en la computadora subyacente y los costos de su uso.

Existen dos principios importantes a tener en cuenta:
La eficiencia y la regularidad.
Eficiencia:

- **Eficiencia de código:** un diseño del lenguaje que permite que el traductor genere un código ejecutable eficiente. Ej. uso de declaraciones anticipadas de variables (estáticas).
- **Eficiencia de traducción:** un diseño del lenguaje que permite que el código fuente se traduzca con rapidez y con un traductor de tamaño razonable. Ejemplo: C standard permite un compilador de una sola pasada pues las variables se declaran antes de usarse. C++, el compilador debe realizar una segunda pasada para resolver referencias del identificador.
- **Eficiencia de implementación:** eficiencia con la que se puede escribir un traductor. Está relacionado con la eficiencia de la traducción y con la complejidad de la definición del lenguaje.
- **Eficiencia de la programación:** facilidad para escribir programas en el lenguaje. Entre otras, involucra capacidad de expresión del lenguaje y capacidad de mantenimiento al programa (localizar y corregir errores).

Regularidad:
La regularidad indica lo bien que se integran las características del lenguaje. Es una cualidad que implica que hayan pocas restricciones en el uso de sus constructores particulares y en la forma en la que se comportan las características del lenguaje.

Generalidad:
Un lenguaje alcanza generalidad evitando casos especiales en cuanto a disponibilidad o uso de constructores. La no generalidad está ligada a restricciones de los constructores dependiendo del contexto en que estén.
Ejemplos:

- En lenguaje C el operador de igualdad == carece de generalidad. Dos arreglos (o estructuras) no pueden compararse a través de este operador.
- En cuanto a la longitud de los arreglos, C tiene arreglos de longitud variable, mientras que esta generalidad no está presente en Pascal, que admite sólo arreglos de longitud fija.

Ortogonalidad: Se refiere al atributo de combinar varias características de un lenguaje de manera que la combinación tenga significado.
Ejemplos:

- Si un lenguaje provee de expresiones que pueden devolver un valor y también posee un enunciado condicional que compara valores; estas dos características son ortogonales si se puede usar dentro del enunciado condicional cualquier expresión.

if ((x+y -2) < (x*y))

Uniformidad:
Hace referencia a la consistencia de la apariencia y comportamiento de los constructores del lenguaje. Las no-uniformidades son de dos tipos: a) dos cosas similares que no parecen serlo, se comportan similarmente, o b) cosas no similares, se comportan similarmente cuando no debieran. Ejemplo En C de la situación b): los operadores & (and bit a bit) y && (and lógico) tienen una apariencia confusamente similar y producen resultados muy distintos.

Paradigma de Programación:
Un paradigma de programación consiste en un método para llevar a cabo cómputos y la forma en la que deben estructurarse y organizarse las tareas que debe realizar un programa.

Clasificación de los Paradigmas:
En general, la mayoría de paradigmas son variantes de los dos tipos principales de programación:

•**Imperativa:** En la programación imperativa se describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución. Es la más usada en general, se basa en dar instrucciones al ordenador de cómo hacer las cosas en forma de algoritmos, en lugar de describir el problema o la solución.

- **Programación orientada a objetos:** encapsula elementos denominados objetos que incluyen tanto variables como funciones. Ejemplo: C++, C#, Java, Python entre otros, Smalltalk que está completamente orientado a objetos.
- **Programación dinámica:** proceso de dividir problemas en partes pequeñas para analizarlos y resolverlos de forma cercana al óptimo. Este paradigma está basado en el modo de realizar los algoritmos.
- **Programación orientada a eventos:** es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.

•**Declarativa:** En la programación declarativa se programa diciendo lo que se quiere resolver a nivel de usuario, pero no las instrucciones necesarias para solucionarlo. Esto último se realizará mediante mecanismos internos de inferencia de información a partir de la descripción realizada. Está basada en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones. La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla. No existen asignaciones destructivas, y las variables son utilizadas con transparencia referencial. Uso de mecanismos matemáticos para optimizar el rendimiento de los programas. Ejemplos: Lisp y Prolog.

Lenguajes Imperativos:

Un lenguaje procedural o imperativo es un lenguaje controlado por mandatos o instrucciones. El proceso de ejecución de programas procedurales por la computadora, tiene lugar a través de una serie de estados, cada uno definido por el contenido de la memoria, los registros internos y los almacenes externos durante la ejecución.

- El almacenamiento inicial de estas áreas define el estado inicial de la computadora.
- Cada paso en la ejecución transforma este estado en otro nuevo, a través de la modificación de alguna de estas áreas, transición de estado.
- Cuando termina la ejecución del programa, estado final viene dado por el contenido final de las áreas antes mencionadas.

El lenguaje C, responde al paradigma imperativo

Lenguaje C

- Los primeros lenguajes de programación se denominaban lenguajes de bajo nivel o lenguajes de máquina (ceros y unos).
- Lenguajes de alto nivel, que distan mucho del lenguaje de máquina y se asemejan más al lenguaje que utilizan las personas para comunicarse (el inglés).
- El lenguaje C fue desarrollado en 1972 por Dennis Ritchie y Ken Thompson en los laboratorios Bell Telephone de AT&T. El lenguaje C comienza a desarrollarse como un lenguaje para escribir software y compiladores, específicamente para desarrollar el sistema operativo UNİX, pero su uso se ha generalizado y hoy en día muchos sistemas están escritos en C o en C++.
- El lenguaje C es un lenguaje de propósitos generales y, si bien es un lenguaje de alto nivel, provee sentencias de bajo nivel que permite facilidades similares a las que se encuentran en los lenguajes ensambladores.

Unidad 2: OBJETOS DE DATOS-TIPOS DE DATOS ELEMENTALES

Objetos de datos:
Es la representación de un elemento u objeto de la realidad de manera que pueda ser procesado por una computadora Recipiente para almacenamiento y recuperar valores de datos.

Tipos	**Objetos de datos definidos por el programador:** Son creados y manipulados por el programador a través de declaraciones y enunciados. Ejemplos: variables, constantes, arreglos, etc.
	Objetos de datos definidos por el sistema: Se construyen para mantenimiento durante la ejecución de un programa, se crean automáticamente sin intervención del programador: • Pilas de almacenamiento en tiempo de ejecución • Registros activación • Memoria intermedia de archivos
Atributos	**Valor:** se representa como un patrón de bits en el almacenamiento de la computadora subyacente.
	Tipo: conjunto de valores de datos que el objeto puede tomar.
	Identificador: nombre a través del cual se puede referenciar el objeto de datos en tiempo de ejecución. Este nombre se construye a partir de un conjunto de caracteres; dependiendo del lenguaje puede haber restricciones.
	Localización: Son lugares de memoria o direcciones de memoria donde se almacenan los valores
Clasificación	**Elemental:** si el objeto contiene un valor de datos que se manipula siempre como una unidad
	Estructurado: Si un objeto de datos es agregado de otros objetos de datos.

Variable:
Objeto de datos cuyo valor puede cambiar durante su tiempo de vida, pero no su nombre y su tipo. Antes de usar una variable hay que declararla, al hacerlo se debe dar su nombre (identificador) y su tipo. Para el identificador se sugiere un nombre significativo, de modo de lograr una mejor legibilidad del programa. Una variable se puede especificar a través de sus atributos, que incluyen nombre, localización, valor y otros atributos como tipo de datos y tamaño.

Constante:
Objeto de datos con nombre enlazado en forma permanente a un valor. Una constante es similar a una variable, excepto que no tiene un atributo de localización. Esto no significa que una constante no se almacene en la memoria en algunas ocasiones. En C las constantes se declaran con la directiva #define, esto significa que esa constante tendrá el mismo valor a lo largo de todo el programa. El identificador de una constante así definida será una cadena de caracteres que deberá cumplir los mismos requisitos que el de una variable (sin espacios en blanco, no empezar por un dígito numérico, etc).

Tiempo de Vida:
Durante la ejecución de un programa algunos objetos de datos existen desde el comienzo de la ejecución, otros pueden crearse dinámicamente durante la misma. Algunos objetos persisten durante toda la ejecución del programa, otros se destruyen durante la ejecución del mismo. Por lo tanto, definimos como tiempo de vida de un objeto, al tiempo durante el cual el objeto puede usarse para guardar valores de datos.

Enlace o Ligadura:
El enlace o ligadura se refiere a la conexión entre un elemento de programa (como una variable o una función) y una característica o propiedad específica en el código. Esto significa que estás relacionando un elemento con una entidad particular dentro de tu programa. Por ejemplo, en el caso de una variable, el enlace se establece cuando asignas un valor a esa variable. En esencia, el enlace determina a qué objeto o valor está asociado un elemento de programa en un momento dado. Puedes pensar en ello como una "ligadura" que une un elemento a su contenido o función específica.
Los **enlaces tempranos** se establecen antes de la ejecución para lograr eficiencia, mientras que **los enlaces tardíos** se determinan durante la ejecución para lograr flexibilidad. La Tabla de Símbolos es una herramienta esencial para gestionar estos enlaces y garantizar que los nombres se asocien correctamente con sus significados en todo momento.
Es importante mencionar que el traductor (el compilador o intérprete) debe manejar estos enlaces de manera adecuada durante la traducción y la ejecución del programa. Para hacer esto, se utiliza una estructura de datos llamada **"Tabla de Símbolos".** Esta tabla contiene una entrada por cada identificador diferente declarado en el programa fuente, y cada entrada incluye información adicional, como el tipo de variable, el nombre del subprograma, etc.
A medida que se avanza en la traducción del programa, la Tabla de Símbolos se actualiza para reflejar los cambios en los enlaces, como la eliminación o inserción de nuevos enlaces. Esto asegura que los nombres se manejen correctamente tanto en tiempo de traducción como en tiempo de ejecución.

Tipos de datos:
Un tipo de datos es una clase de objetos de datos ligados a un conjunto de operaciones para crearlos y manipularlos.
Los tipos de datos pueden ser:

- **Primitivos:** datos que están integrados al lenguaje.
- **Definidos por el programador:** el lenguaje provee recursos para definir nuevos tipos.

Especificación e implementación de un tipo de datos

Elementos básicos de una <u>especificación</u> de un tipo de datos:

- **Atributos**: distinguen a los objetos de datos de ese tipo. Por ejemplo, para un tipo de dato arreglo los atributos son: nombre, dimensiones, tipo de dato de las componentes, etc.
- **Valor:** especifica los valores de datos que pueden tomar los objetos de este tipo de dato.
- **Operaciones:** hacen referencias a las distintas formas de manipular los objetos de datos de ese tipo de datos.

Elementos básicos de la <u>implementación</u> de un tipo de datos:

- **Representación de almacenamiento:** hace referencia a la forma que se usa para representar los objetos de datos de un tipo de dato determinado de datos en memoria.
- **Algoritmos y procedimientos:** definen las operaciones del tipo, en términos de manipulaciones de su representación de almacenamiento.

Tipos de datos elementales

Los tipos de datos elementales son categorías de objetos de datos que almacenan un solo valor, mantienen atributos invariables, tienen un conjunto de valores ordenado y pueden tener operaciones asociadas, que pueden ser primitivas (definidas por el lenguaje) o definidas por el programador. Estos tipos de datos son fundamentales en la programación y se utilizan para representar datos simples como números, caracteres o fechas.

Representación de almacenamiento:

Ordinariamente, representación de almacenamiento hace referencia al tamaño de bloque de memoria que se requiere, a la disposición de los valores y atributos dentro de ese bloque y no a su ubicación dentro de la memoria.
La "representación de almacenamiento" se refiere a cómo se almacenan los datos en la memoria de una computadora, y esto puede influir en el rendimiento y la eficiencia de un programa.
Esta representación puede variar según el lenguaje de programación y la arquitectura de la computadora subyacente, y las decisiones sobre cómo se representan los datos pueden tener un impacto significativo en el rendimiento y la eficiencia de un programa.

Implementación de tipos de datos elementales:

Implica decidir cómo se almacenan los valores de ese tipo en la memoria, definir las operaciones y procedimientos asociados para manipular esos datos, y asegurarse de que la implementación sea eficiente y cumpla con los objetivos del lenguaje de programación. La elección de la implementación adecuada puede tener un impacto significativo en el rendimiento y la eficiencia de un programa.

Implementación de las operaciones:

La implementación de las operaciones en un lenguaje de programación puede variar desde operaciones de hardware eficientes hasta simulaciones por software a través de subprogramas o secuencias de código de línea. La elección del enfoque adecuado depende de la disponibilidad de hardware, los requisitos de eficiencia y la estructura general del código.

- **Operación de Hardware:** En este caso, la operación se realiza directamente a nivel de hardware. Esto significa que el hardware de la computadora tiene instrucciones específicas y circuitos diseñados para ejecutar esa operación de manera eficiente. Estas operaciones suelen ser muy eficientes en términos de velocidad de ejecución.
- **Simulada por Software a través de Subprogramas**: En algunos casos, las operaciones no tienen un soporte de hardware directo, pero se pueden implementar mediante subprogramas o funciones que el programador escribe en el código. Estos subprogramas realizan las operaciones necesarias utilizando las operaciones básicas que proporciona el lenguaje de programación. Por ejemplo, una operación matemática más compleja, como la potencia, puede implementarse como una función en el código.
- **Simulada por Software como una Secuencia de Código de Línea**: En situaciones en las que no es posible o práctico utilizar subprogramas, algunas operaciones se implementan directamente como una secuencia de código en línea. Este enfoque se utiliza a menudo para operaciones simples que no justifican la creación de un subprograma separado.

Declaraciones:

Las declaraciones en un programa de computadora son enunciados escritos por el programador, esenciales para comunicar información crítica al traductor y para establecer las ligaduras necesarias entre nombres y objetos de datos. Esto contribuye a la eficiencia en la traducción y la ejecución del programa, así como a la resolución adecuada de operadores homónimos cuando existen múltiples definiciones posibles.
El **propósito más importante de las declaraciones** es permitir la verificación estática de tipos.

Verificación de tipos:
La verificación de tipos es esencial para garantizar que un programa se ejecute sin errores relacionados con tipos de datos, este proceso lo realiza el intérprete o el compilador. Puede realizarse de forma dinámica en tiempo de ejecución o estática en tiempo de compilación, dependiendo del lenguaje de programación. La verificación estática se basa en la información proporcionada por el programador a través de declaraciones de tipo, mientras que la verificación dinámica utiliza marcas de tipo para verificar tipos en tiempo de ejecución.

- **Verificación Dinámica de Tipos:** La verificación dinámica de tipos implica que la información de tipos se verifica en tiempo de ejecución. Esto significa que los intérpretes realizan esta verificación durante la ejecución del programa.
- **Verificación Estática de Tipos:** En contraste, la verificación estática de tipos se realiza durante la traducción o compilación del programa. En este caso, el compilador verifica que todas las construcciones del programa sean coherentes en cuanto a los tipos declarados. Los programadores deben proporcionar información de tipos a través de declaraciones para que el compilador pueda realizar esta verificación.

¿Cómo se realiza la verificación de tipos?
La verificación de tipos es un proceso que se realiza durante la traducción del programa y que implica recopilar información de las declaraciones y verificar todas las operaciones en busca de errores de tipo. Los lenguajes fuertemente tipificados, como C++, aplican una verificación de tipos más estricta para prevenir errores en tiempo de ejecución. Es importante prestar atención a las advertencias que emite el compilador, ya que ignorarlas puede llevar a problemas potenciales en el programa.
La verificación de tipos se realiza recopilando información de las declaraciones en la tabla de símbolos, verificando la validez de los argumentos en cada operación del programa y determinando el tipo de datos del resultado. Esta verificación estática evita marcas de tipo en tiempo de ejecución y ahorra almacenamiento y tiempo de ejecución. Sin embargo, existe falta de consenso entre los diseñadores de lenguajes sobre cuánta información de tipos debe explicitarse antes de la ejecución del programa.

Conversión de tipos:
Cuando en la verificación de tipos hay una discordancia entre el tipo real de un argumento y el tipo esperado, el lenguaje puede:

1. marcar el error y ejecutar una acción de error apropiada
2. se puede realizar una conversión de tipos.

Conversión implícita o coerciones:
Son invocadas automáticamente en ciertos casos de discordancia. El principio básico que gobierna las coerciones es **no perder información.**
El compilador busca la operación de conversión para que de manera automática se realicen los cambios al tipo de dato adecuado. En este caso, estamos frente a una conversión implícita o coerción. En C, las coerciones son reglas:

a- Si los dos operandos son de tipo coma flotante con distinta precisión, el operando de menor precisión se transforma a la precisión del otro operando y el resultado se expresa en esta precisión (la más alta).
b- Si un operando es de tipo coma flotante y el otro es un char o un int, el char/int se convertirá al tipo coma flotante y el resultado se expresará en este tipo.
c- Si ninguno de los operandos es de tipo coma flotante, el compilador convierte los char y short int a int antes de evaluar, salvo que en la expresión aparezca un long int. En este caso los operandos y el resultado se convertirán a long int.

Conversión Explícita:
El programador llama a funciones integradas para realizar una conversión de tipos. En C, se puede forzar a una expresión a ser de un tipo específico, usando el operador unario cast.

Tipo de datos elementales:
Los tipos de datos elementales, también conocidos como tipos primitivos, son los tipos de datos originales proporcionados por un lenguaje de programación, que se utilizan para construir estructuras de datos y tipos de datos abstractos. Aquí se describen los principales tipos de datos elementales:

1. **Carácter:** Por lo general, se almacena en un solo byte y se utiliza para representar caracteres individuales y se basa en el código ASCII extendido para la representación de caracteres. Estos caracteres son fundamentales en la manipulación de texto y la construcción de cadenas de caracteres en la programación. Los caracteres están ordenados de 0 a 255, lo que permite la comparación de caracteres entre sí. Los caracteres incluyen letras minúsculas (a..z), letras mayúsculas (A..Z), dígitos decimales (0..9), caracteres de espacio en blanco, caracteres especiales (+, -, %, etc.), signos de puntuación (, ; :), entre otros. El orden de los caracteres es importante, ya que permite realizar operaciones como comparaciones alfabéticas y construcción de cadenas de caracteres ordenadas. En lenguajes como Pascal y C, no existen tipos de datos de cadena de caracteres como entidades separadas, sino que se construyen a partir de arreglos de caracteres. En C, el tipo carácter es un subtipo del tipo entero. Cada carácter ocupa un byte y puede ser manipulado como un entero.
2. **Entero:** Un objeto de datos de tipo entero tiene como atributo su tipo. Los valores que puede tomar un objeto de datos entero son un subconjunto ordenado de números enteros con límites inferior y superior. La implementación utiliza la representación de almacenamiento definida por el hardware. Las operaciones típicas que un lenguaje de programación define sobre objetos de datos enteros son las siguientes: Operaciones aritméticas, Operaciones relacionales, Operación de asignación.
   Dentro de las operaciones antes destacadas, resulta importante entender **la asignación**. La asignación es una operación de casi todos los tipos de datos elementales, y es la operación básica para cambiar el valor de una variable.
3. **Real:** Un objeto de datos de tipo real puede tomar cualquier valor real dentro de un rango limitado por un valor mínimo y máximo determinado por el hardware. La precisión decimal se puede especificar por el programador. La implementación utiliza representaciones de punto flotante, generalmente siguiendo estándares como IEEE 754, que definen la precisión y el rango de valores.
4. **Booleano o Lógico:** Este tipo de datos se utiliza para representar valores booleanos, es decir, verdadero (true) o falso (false). Se utiliza en evaluaciones lógicas y control de flujo en el programa. En algunos lenguajes, los enteros también se pueden utilizar para representar valores booleanos, donde 0 se interpreta como falso y cualquier otro valor como verdadero. Operaciones que soportan: Conjunción, Disyunción (inclusive) y Negación.
5. **Puntero o Apuntador:** Algunos lenguajes proporcionan el tipo de datos puntero, que se utiliza para almacenar direcciones de memoria. Los punteros son útiles para trabajar con estructuras de datos dinámicas y para acceder a ubicaciones de memoria específicas.

En cuanto a la implementación de estos tipos de datos, generalmente depende del hardware subyacente y de las especificaciones del lenguaje de programación. Los tipos de datos enteros y reales suelen utilizar la representación de almacenamiento y las operaciones relacionadas con hardware, mientras que los caracteres y booleanos se pueden representar en un solo byte.
Es importante tener en cuenta que algunos lenguajes pueden no proporcionar todos estos tipos de datos elementales o pueden tener variaciones en la forma en que se implementan. Además, la precisión y el rango de valores pueden variar según el estándar utilizado por el lenguaje y el hardware.

Declaración de Punteros:
Para declarar un puntero en C, se utiliza el operador '*' antes del nombre del puntero.
La sintaxis general para declarar un puntero es: <tipo_de_dato> *<nombre_del_puntero>;
Por ejemplo:

```
int *p, x; // Declaración de p como puntero a un entero y x como variable entera
```

Asignación de la Dirección de Memoria:
Para asignar al puntero la dirección de memoria de una variable, puedes utilizar el operador '&' que devuelve la dirección de esa variable.
La asignación se realiza de la siguiente manera:

```
p = &x; // Asignación de la dirección de x al puntero p
```

En este caso, &x devuelve la dirección de memoria de la variable entera x, y ese valor se almacena en el puntero p.
Después de esta asignación, el puntero p apunta a la dirección de memoria de la variable x, lo que significa que p contiene la dirección donde se encuentra almacenado el valor de x.
Al declarar un puntero en C, primero debes especificar el tipo de dato al que apuntará el puntero y luego asignarle la dirección de memoria de la variable que deseas que esté apuntada por ese puntero. Esto te permite manipular indirectamente el valor de la variable a través del puntero, lo que puede ser útil en diversas situaciones de programación.

Asignación de valores a punteros:

A través del Operador &:
El operador & se utiliza para obtener la dirección de memoria de una variable existente y asignarla a un puntero.
Ejemplo:

```
char c, *punt;
punt = &c; // Asignación de la dirección de memoria de c al puntero punt
```

En este ejemplo, se asigna al puntero punt la dirección de memoria de la variable c.

Por Medio de Otro Puntero:
Un puntero se puede asignar a otro puntero del mismo tipo, lo que hace que ambos apunten a la misma dirección de memoria.
Ejemplo:

```
int x = 10, y = 20, *p, *q;
q = &y;
p = &x;
q = p; // Ahora q apunta a la misma dirección de memoria que p
```

En este ejemplo, el puntero q se asigna con el valor del puntero p, lo que hace que ambos apunten a la misma dirección de memoria, en este caso, la de la variable x.
Con una Dirección de Memoria Constante:

También es posible asignar directamente una dirección de memoria constante a un puntero:
Ejemplo:

```
int *ptr;
int valor = 42;
ptr = (int *)0x1234; // Asignación de una dirección de memoria constante a ptr
```

En este ejemplo, el puntero ptr se asigna con la dirección de memoria constante 0x1234.

Inicialización de variables punteros:
La inicialización de variables punteros es una práctica importante en C, ya que los punteros no se inicializan automáticamente al declararse:

Inicialización a NULL:
Una forma segura de inicializar un puntero cuando no se tiene una dirección de memoria específica que asignar es usar NULL. NULL es una constante simbólica definida en el archivo de cabecera stdio.h. Inicializar un puntero a NULL significa que el puntero no apunta a ninguna dirección de memoria válida.
Ejemplo:

```
int *ptr = NULL; // Inicialización del puntero ptr a NULL
```

Inicialización con la Dirección de una Variable Existente:
También puedes inicializar un puntero con la dirección de una variable existente siempre que la variable esté previamente declarada.
Ejemplo:

```
int x = 42;
int *ptr = &x; //Inicialización del puntero ptr con la dirección de la variable x
```

Inicialización al Declarar el Puntero:
Puedes inicializar un puntero al momento de declararlo. Esto es especialmente útil cuando se conoce la dirección de memoria que se asignará al puntero.
Ejemplo:

```
int y = 10;
int *ptr =&y;//Declaración e inicialización del puntero ptr con la dirección de y
```

Inicialización sin Valor Inicial (Evitar):
Es importante evitar usar un puntero sin inicializar, ya que podría apuntar a una dirección de memoria desconocida donde pueden existir datos indeseados.
Ejemplo de puntero sin inicializar (no recomendado):

```
int *ptr; // Evitar declarar punteros sin inicializar
```

La inicialización adecuada de punteros ayuda a evitar comportamientos inesperados y errores en el programa. Siempre es una buena práctica inicializar los punteros antes de usarlos y, cuando no tengas una dirección de memoria específica que asignar, utilizar NULL para indicar que el puntero no apunta a ningún dato válido.

Comparación de punteros:
Los punteros, al igual que las demás variables, se pueden comparar. La comparación es posible siempre que los punteros apunten al mismo tipo de datos.
Los operadores usados en la comparación, son los operadores relacionales conocidos:
==, !=, <>, <=, >=.
La comparación de punteros es válida siempre y cuando respete una lógica. Existen tres formas para comparar punteros:

- Dos punteros entre sí.
- Un puntero con la dirección de memoria de una variable, expresada por medio del operador &.
- Un puntero con una dirección de memoria dada directamente.

Tipos de Datos Estructurados:
Un Objeto de datos estructurado o estructura de datos está constituido por un agregado de otros objetos de datos llamados componentes. Pueden ser:

- Definidos por el programador: Registros, Arreglos, etc.
- Definidos por el sistema: durante la ejecución del programa La pila que usa el sistema para trabajar con funciones.

Ejemplos de Tipos de Datos Estructurados:
Arreglos: Un arreglo es una estructura de datos que almacena un conjunto de elementos del mismo tipo. Puedes pensar en un arreglo como una lista ordenada de elementos, donde cada elemento tiene una posición única.
Registros: Un registro es una estructura de datos que contiene varios campos o componentes, y cada campo puede ser de un tipo de dato diferente. Los registros se utilizan para representar objetos complejos con múltiples atributos.
Cadenas de Caracteres: Las cadenas de caracteres son secuencias de caracteres almacenadas como un solo tipo de dato. Son útiles para manipular texto y representar palabras, frases o párrafos.

Atributos Principales de los Tipos de Datos Estructurados:

- **Número de Componentes**: Los tipos de datos estructurados pueden tener un número fijo o variable de componentes. Los arreglos y registros suelen tener un número fijo de componentes, mientras que otras estructuras, como las listas enlazadas, pueden tener un número variable de elementos.
- **Tipo de Componente:** Las estructuras pueden ser homogéneas o heterogéneas en términos de los tipos de datos que contienen. Los arreglos son homogéneos, ya que todos los elementos son del mismo tipo, mientras que los registros son heterogéneos, ya que pueden contener componentes de diferentes tipos.
- **Nombres de Selección:** Para acceder a los componentes individuales de una estructura, se utilizan nombres de selección. Por ejemplo, en un registro, cada componente tiene un nombre único para identificarlo.
- **Organización de Componentes:** Los componentes de una estructura pueden estar organizados de manera unidimensional o multidimensional. Los arreglos y las cadenas de caracteres son ejemplos de organización unidimensional, mientras que los registros que contienen registros anidados representan una organización multidimensional.

Operaciones sobre Estructuras de Datos:
Las operaciones que se pueden realizar sobre estructuras de datos incluyen:

- **Selección Directa:** Permite el acceso arbitrario a un componente específico de una estructura.
- **Selección Secuencial:** El acceso a los componentes se realiza en un orden predeterminado.

Operaciones sobre una estructura de datos completa:

- **Asignación de Estructuras de Datos**: En lenguaje C, puedes asignar una estructura de datos completa a otra utilizando el operador de asignación =. Esto copiará todos los componentes de una estructura a otra.
- **Inserción/Eliminación de Componentes:** Estas operaciones implican agregar o quitar componentes de una estructura de datos. Por ejemplo, en una lista enlazada, la inserción implica agregar un nuevo elemento a la lista, mientras que la eliminación implica quitar un elemento existente.
- **Creación/Destrucción de Estructuras de Datos**: La creación de una estructura de datos implica reservar memoria para ella y configurar sus componentes según sea necesario. La destrucción implica liberar la memoria utilizada por la estructura de datos cuando ya no es necesaria.

Representación de tipos de datos estructurados:
Representación Secuencial: En esta representación, la estructura utiliza un solo bloque de memoria contigua que almacena tanto las componentes como el descriptor (si es necesario). Se utiliza para estructuras de tamaño fijo o para estructuras homogéneas de tamaño variable. Por ejemplo, registros y arreglos pueden utilizar representaciones secuenciales.
Representación Vinculada: En esta representación, la estructura utiliza varios bloques de memoria no contiguos que están vinculados entre sí a través de punteros. Se utiliza comúnmente para estructuras de tamaño variable, como listas enlazadas. Cada bloque de memoria puede contener una parte de la estructura de datos y un puntero al siguiente bloque.

Es importante tener en cuenta que la representación de estructuras de datos y las operaciones relacionadas generalmente se implementan mediante software en lugar de depender del hardware subyacente. Esto significa que el lenguaje de programación y su biblioteca estándar proporcionarán las herramientas y las funciones necesarias para trabajar con estructuras de datos de manera eficiente.

Arreglos Unidimensionales:
Un arreglo unidimensional (también llamados vectores) es una estructura de datos formada por un número fijo de componentes del mismo tipo de datos, organizadas como una serie lineal simple.
Sus atributos son: Número de componentes, Tipo de dato de las componentes, Subíndice que se puede usar para seleccionar una componente.

Operaciones sobre arreglos unidimensionales:

- **Selección de una componente (subindización):** Operación implica acceder a un elemento específico del arreglo utilizando un índice o subíndice.
- **Operación de Referenciamiento:** Similar a la selección de una componente, pero se refiere al proceso de obtener la dirección de memoria de un elemento del arreglo en lugar de su valor.
- **Operación de Selección:** Implica seleccionar un conjunto de elementos del arreglo que cumplan ciertos criterios o condiciones. Esto se hace generalmente utilizando bucles y estructuras de control.

Arreglos Bidimensionales (Matriz):
Son una estructura de datos en la que los elementos se organizan en filas y columnas, formando una tabla rectangular. Estas estructuras se utilizan para representar datos en forma de cuadrícula o cuadro, donde cada elemento se puede acceder mediante dos índices: uno que indica la fila y otro que indica la columna.
Arreglo Bidimensional es un arreglo de arreglos: esta representación implica que el arreglo se divide en subarreglos para cada elemento del intervalo del primer subíndice. Cada uno de estos subarreglos es un arreglo.

Cadena de caracteres:
Las cadenas de caracteres son objetos de datos que representan secuencias de caracteres en un lenguaje de programación. Están diseñadas para almacenar texto y permiten trabajar con palabras, frases o cualquier tipo de contenido textual.

Declaración y sintaxis:
En C, las cadenas de caracteres generalmente se implementan como arreglos de caracteres (tipo char) con una serie de caracteres consecutivos que representan el contenido de la cadena.

```
char nombre[] = "FIN";
```

Librería <string.h>:
C no proporciona un tipo de datos cadena incorporado como algunos otros lenguajes. En su lugar, las funciones de manipulación de cadenas se encuentran en la biblioteca estándar <string.h>. Estas funciones, como strcpy, strcat, strlen, strcmp, entre otras, se utilizan para realizar operaciones comunes en cadenas de caracteres.

Carácter nulo ('\0'):
En C, cada cadena de caracteres termina con un carácter nulo ('\0'), que indica el final de la cadena. Este carácter se agrega automáticamente al final de una cadena cuando se inicializa o se asigna. La presencia de este carácter permite a las funciones de manipulación de cadenas identificar el final de una cadena.

```
printf("\n Ingrese cadena minúsculas( máximo 10 car.: " );
gets(cad);
// Qué pasa si uso scanf(“%s”, cad) ?
```

scanf("%s", cad), esta es una opción válida para leer una cadena de caracteres, pero puede tener problemas si la entrada contiene espacios en blanco. scanf("%s", cad) solo leerá una secuencia de caracteres hasta el primer espacio en blanco, lo que significa que no capturará una cadena completa si contiene espacios.

Registros:
Los registros, también conocidos como estructuras en lenguaje C y otros lenguajes de programación, son objetos de datos compuestos que pueden contener un número fijo de componentes, campos o miembros, cada uno de los cuales puede ser de igual o distinto tipo de datos y tiene una longitud fija. Estos registros son utilizados para agrupar diferentes tipos de datos relacionados bajo una sola entidad.

Atributos de un Registro:

- **Número de Componentes:** Un registro puede tener un número fijo de componentes, y este número se define durante la declaración del registro. Cada componente representa una parte específica de los datos que se almacenan en el registro.
- **Tipo de Datos de Cada Componente**: Cada componente en un registro puede tener un tipo de datos diferente. Por ejemplo, un registro podría contener un componente de tipo entero, otro componente de tipo flotante y otro componente de tipo cadena de caracteres.
- **Identificador para Nombrar Cada Componente:** Cada componente dentro de un registro se identifica mediante un nombre o identificador único. Estos nombres se utilizan para acceder a cada componente individualmente.

Diferencias entre un arreglo(vector) y un registro:
Los componentes de un registro pueden ser heterogéneos u homogéneos Los componentes se designan con nombres simbólicos en lugar de indizarse con subíndices.

Operación básica:
La operación básica es la operación de selección.

Atributos de un registro:

- Número de componentes
- Tipo de datos de cada
- Selector que se usa para nombrar cada componente

Implementación:
En C y en otros lenguajes de programación, la implementación de un registro se basa generalmente en una representación secuencial de almacenamiento. Las componentes individuales del registro se almacenan en memoria de manera consecutiva, de modo que la selección de una componente se hace de manera eficiente utilizando el conocimiento de su posición en relación con otras componentes.

Acceso a los miembros de una estructura:
Los miembros de una estructura se procesan generalmente en forma individual. El acceso a cada miembro de una estructura se realiza utilizando el operador de miembro de estructura (.) también conocido como operador punto, de la siguiente manera: variable . miembro

Operaciones sobre estructuras:
- Las estructuras son objetos de datos compuestos por un número fijo de componentes de igual o distinto tipo, con longitud fija.
- Las operaciones básicas en estructuras implican la lectura y escritura de sus componentes de manera individual.
- Para asignar valores a un miembro de una estructura, se accede a él a través del operador de miembro de estructura (.).
- La asignación entre estructuras es válida si ambas son del mismo tipo y no contienen miembros del tipo puntero.
- Los punteros a estructuras permiten acceder a las variables estructuradas usando el operador flecha (->).

Registros Variantes:
- Los registros variantes son estructuras donde algunos componentes pueden estar presentes o ausentes dependiendo del valor de una "marca".
- En la implementación, se asigna almacenamiento para la variante más grande posible durante la traducción.
- La selección de componentes se realiza de manera similar a los registros ordinarios, pero puede haber problemas si se selecciona una componente inexistente.
- Las soluciones incluyen verificación dinámica o no verificar, según el lenguaje de programación y la implementación.

Union vs. Struct en C:
- Las uniones y estructuras son estructuras de datos en C, pero difieren en su almacenamiento.
- En una estructura, los campos se almacenan en posiciones contiguas de memoria, y se usan cuando se necesita acceder a múltiples variables al mismo tiempo.
- En una unión, los miembros comparten memoria en la misma posición y se usan cuando no es necesario acceder a todas las variables simultáneamente.

Unidad 3: Funciones

En lenguajes de programación, se proporcionan tipos de datos primitivos y operaciones para ocultar los detalles de la representación de datos. El objetivo actual es permitir a los programadores crear nuevos tipos de datos y operaciones de alto nivel. Se exploran los mecanismos en lenguajes procedurales como C para definir nuevos tipos de datos.

Definición de tipos:
Una definición de tipos consiste en un nombre de tipo y una descripción de la estructura de una clase de objetos de datos. Estos nombres de tipo simplifican la manipulación de objetos de datos específicos sin necesidad de repetir la estructura completa.

Constructores de tipos:
Los lenguajes permiten construir tipos complejos a partir de tipos básicos. Ejemplos de constructores de tipo en C son struct, union y arreglos.

Lenguaje C y typedef:
C ofrece la palabra clave "typedef" para crear alias o nombres alternativos para tipos de datos, lo que facilita la declaración de variables y parámetros.

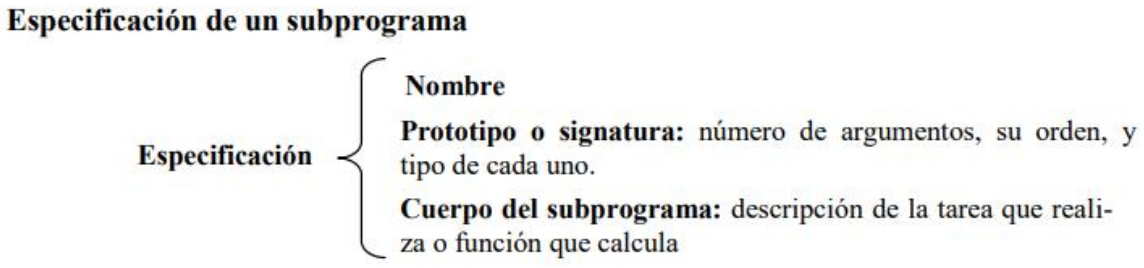
Sistema de tipos:
El sistema de tipos incluye métodos de construcción de tipos, algoritmo de equivalencia de tipos y reglas de inferencia y corrección de tipos. Se distingue entre lenguajes fuertemente tipificados, como Pascal, que detectan errores de tipo estática y tempranamente, y lenguajes débilmente tipificados, como C, que tienen menos restricciones.

Ventajas de la definición de tipos:
La definición de tipos aporta ventajas como simplificar la estructura del programa, permitir cambios eficientes en la estructura definida y facilitar el paso de argumentos en el uso de subprogramas, evitando la repetición de descripciones de tipos de datos.

Subprogramas:
La modularización es una técnica de programación que busca simplificar y optimizar el código al dividir un problema complejo en partes más simples, llamadas subproblemas, para resolver cada uno por separado. En C, los subprogramas o funciones son fragmentos de código utilizados para dividir un programa en tareas específicas que abordan partes individuales del problema. Los subprogramas representan operaciones abstractas definidas por el programador y se recomienda su uso en varias situaciones:

1. Cuando un programa necesita realizar repetidamente una tarea particular, se puede definir un subprograma que realice esa tarea y se puede acceder a él tantas veces como sea necesario. Cada invocación del subprograma puede trabajar con datos diferentes.
2. Para mejorar la claridad y la legibilidad del programa principal, es beneficioso definir subprogramas que realicen tareas específicas, incluso si estas tareas no se invocan de manera repetida.

Los subprogramas son una herramienta fundamental en la modularización y organización de programas, permitiendo que el código sea más estructurado y mantenible al dividirlo en componentes más pequeños con propósitos definidos.



Implementación de un subprograma:
Un subprograma es un conjunto de sentencias que realiza una tarea específica y utiliza estructuras de datos y operaciones del lenguaje. Es una operación definida por el programador y su implementación se

encuentra en el cuerpo del subprograma. Algunos lenguajes permiten que el cuerpo contenga definiciones de subprogramas encapsulados, aunque esto no es posible en C. La verificación de tipos es estática en lenguajes como C, ya que se proporciona información sobre los tipos de datos de los argumentos y el resultado. En C, los subprogramas se denominan funciones.

Definición, invocación y activación de subprogramas:
La definición de un subprograma es una propiedad estática del lenguaje y se refiere al tipo de argumentos y variables locales. Un subprograma es un proceso que recibe parámetros y devuelve un valor. Puede invocarse desde cualquier parte del programa y se activa en tiempo de ejecución cuando se llama. La activación se destruye al finalizar la ejecución. La definición actúa como una plantilla para crear activaciones en tiempo de ejecución. Esta plantilla se divide en una parte estática (código ejecutable) y una parte dinámica (parámetros, resultados, datos locales, etc.). Los registros de activación se crean y destruyen en cada invocación de un subprograma. El tamaño y la estructura del registro de activación se determinan en tiempo de traducción.

Prólogo - Epílogo:
El prólogo es un conjunto de código introducido al comienzo del segmento de código ejecutable para realizar tareas como la creación del registro de activación y la transmisión de parámetros. El epílogo es un conjunto de instrucciones al final del bloque de código ejecutable que se usa para devolver resultados y liberar el almacenamiento destinado al registro de activación. Ambos ayudan en la gestión de las activaciones de subprogramas.

Funciones en Lenguaje C
En el lenguaje C, las funciones permiten a los programadores definir y construir unidades de código que realizan tareas específicas. Estas son algunas de las características clave relacionadas con las funciones en C. Las funciones en C son componentes clave para modularizar programas, dividir tareas y mejorar la organización del código. Cada función tiene su propia cabecera que define sus parámetros formales, y su cuerpo contiene las instrucciones que se ejecutan cuando se llama la función. La sentencia return se utiliza para devolver resultados al punto de llamada, y su uso es fundamental para el funcionamiento correcto de las funciones.

Modularización:
El uso de funciones permite dividir un programa en componentes más pequeños, cada uno con un propósito determinado, lo que facilita la codificación y la depuración.

Definición de funciones:
Una función es un conjunto de sentencias que realiza una tarea específica y puede devolver cero o un valor como resultado. Por ejemplo, la función getch() devuelve un carácter, mientras que clrscr() no devuelve ningún valor. Un programa en C está formado por una o más funciones, siendo main() la función principal desde donde comienza la ejecución.

Cabecera y cuerpo de la función:
La definición de una función consta de dos partes: la cabecera y el cuerpo. La cabecera incluye el tipo de valor devuelto, el nombre de la función y los argumentos o parámetros entre paréntesis. Los argumentos son los datos que la función recibe cuando se llama. El cuerpo de la función contiene el conjunto de sentencias que se ejecutan cuando la función es invocada.

Parámetros formales y reales:
Los argumentos se comunican a la función a través de parámetros formales, que se definen en la cabecera de la función. Los parámetros formales son locales a la función y se corresponden con los argumentos

reales que se pasan cuando se llama a la función. Los nombres de los parámetros formales y reales no necesariamente deben coincidir.

Uso de la sentencia return:
Una función debe incluir al menos una sentencia return para devolver un valor al punto de invocación. La sentencia return se utiliza para especificar el valor devuelto por la función. Si la función no retorna un valor, se puede declarar con el tipo de valor devuelto void.

Control de flujo:
Una vez que se invoca una función, se ejecutan las sentencias que componen su cuerpo, y el control vuelve al punto desde donde se llamó la función. Si una función no contiene una sentencia return, el control vuelve automáticamente al punto de llamada.

Coherencia en parámetros:
Los parámetros reales deben coincidir en tipo, orden y cantidad con los parámetros formales en la cabecera de la función.

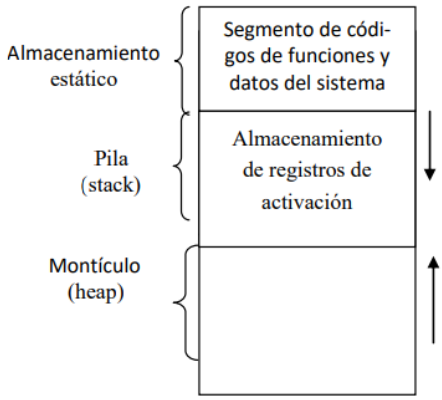
Invocación o llamada a una función:
La invocación o llamada a una función en C provoca la ejecución de las sentencias contenidas en su cuerpo hasta encontrar una sentencia return o la llave de cierre }.
Para invocar una función, se utiliza su nombre seguido de una lista de argumentos reales (parámetros reales) encerrados entre paréntesis y separados por comas.
La llamada puede formar parte de una expresión si la función retorna un valor.

Declaración de una función o prototipo de función:
En C, se puede definir una función antes o después de la función main(). Sin embargo, si se invoca una función antes de su definición, es necesario proporcionar un prototipo de función para que el compilador sepa que la función será definida posteriormente en el programa.
Un prototipo de función es similar a la primera línea de la definición de una función e informa al compilador sobre el tipo de valor devuelto y los tipos de parámetros que espera la función. Se termina con un punto y coma.
Los parámetros formales en el prototipo de función pueden omitir los nombres, ya que lo esencial es especificar los tipos de datos de los parámetros.

Organización de la memoria en C:
La memoria en C se divide en tres áreas: almacenamiento estático, pila (stack) y montículo (heap).

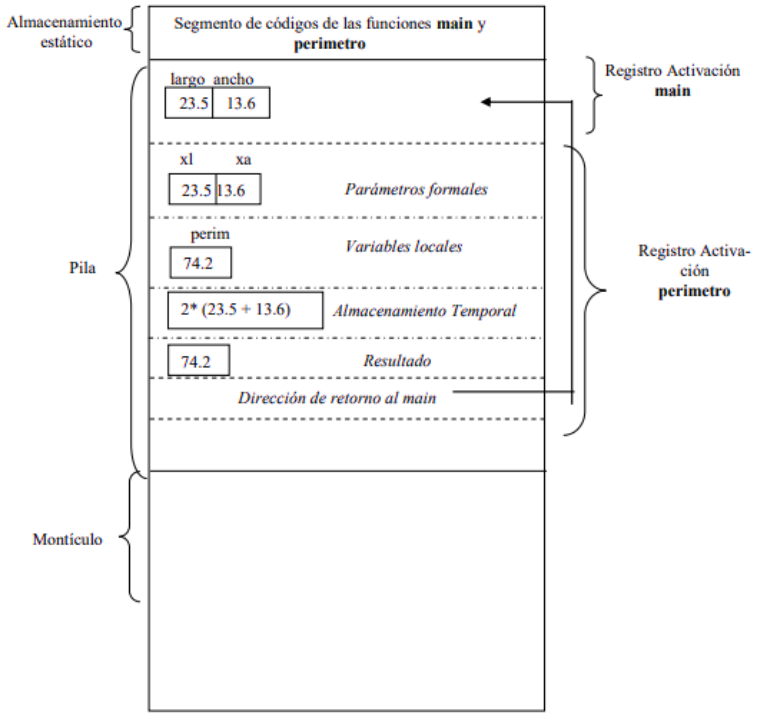
- El almacenamiento estático se asigna en tiempo de traducción y contiene el código de las funciones y otros datos del sistema.
- La pila se utiliza para gestionar las llamadas a funciones y almacena registros de activación de funciones. La asignación y liberación de memoria en la pila se realizan secuencialmente, generalmente creciendo hacia direcciones de memoria más altas.
- El montículo es un área de almacenamiento dinámico donde se asignan y liberan segmentos de memoria. Se utiliza para trabajar con estructuras de datos dinámicas.

La pila y el montículo se ubican en extremos opuestos de la memoria para permitir su crecimiento adecuado.



Ejecución de un programa en C:
Cuando se inicia la ejecución de un programa en C, se generan en tiempo de compilación los segmentos de código correspondientes a las distintas funciones que el programa contiene. Durante la ejecución de un programa en C, se gestiona la memoria en la pila para almacenar registros de activación de funciones. Cada función tiene su propio registro de activación, que se crea al invocar la función y se destruye al finalizar su ejecución. La gestión de la pila permite que las funciones se llamen y retornen de manera ordenada y eficiente.

- En la pila (stack), el primer registro de activación que se almacena corresponde a la función main(), ya que la ejecución del programa comienza por esta función.
- Cada vez que se invoca una nueva función, se "apila" su registro de activación correspondiente, que incluye información como los parámetros formales, variables locales, almacenamiento temporal, resultado de la función y dirección de retorno.
- Una vez que la ejecución de una función finaliza, su registro de activación se "desapila", y el control vuelve a la función llamante.
- Al iniciar la ejecución del programa, se genera el registro de activación del main(). Este registro no reserva espacio para los parámetros ni para el resultado de la función main().
- Cuando se encuentra una invocación a una función, se crea un nuevo registro de activación para esa función, donde se asigna espacio para los parámetros formales, variables locales, almacenamiento temporal, el resultado de la función y la dirección de retorno.



Tipos de Almacenamiento en C:

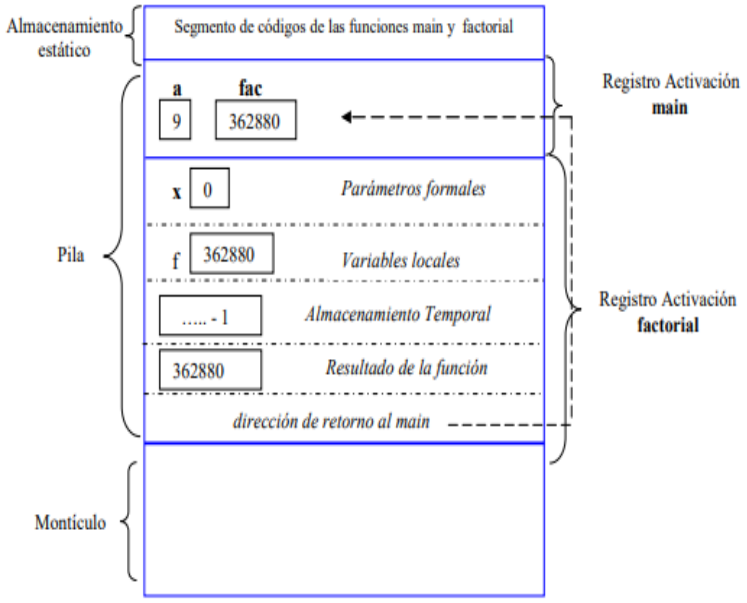
- Variables Automáticas: Son locales a una función, se crean al iniciar la función y se destruyen al salir de ella, almacenadas en la pila.
- Variables Externas: Tienen un alcance global, se definen una vez fuera de funciones y deben declararse en cada función que las use, almacenadas en el área de almacenamiento estático.
- Variables Estáticas: Son locales a la función donde se declaran, mantienen sus valores entre llamadas sucesivas a la misma función, almacenadas en el área de almacenamiento estático.

Alcance de Variables en C:
El alcance de una variable en C se basa en el alcance léxico, lo que significa que una variable es visible solo en el bloque donde se declara y en bloques anidados dentro de ese bloque.

Pasaje de parámetros a una función:
Un programa se comunica con sus funciones a través de los parámetros. Existen distintas formas de pasar parámetros a una función.

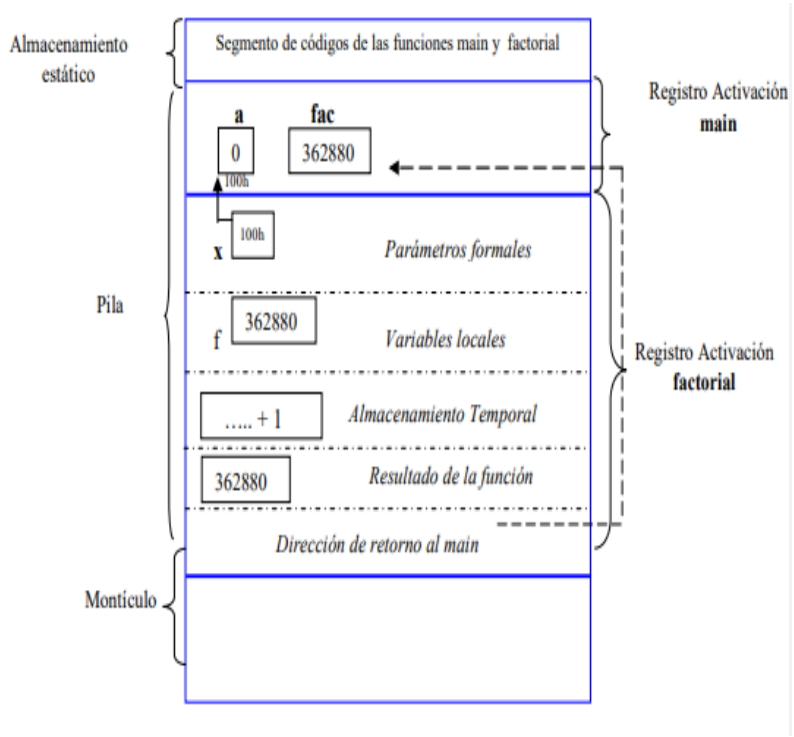
Pasaje por valor:
El pasaje por valor, comúnmente utilizado en lenguajes como C, C++, Pascal y Java, implica que los valores de los parámetros reales se copian en los parámetros formales de una función al llamarla. Esto permite que los parámetros formales actúen como variables locales con valores iniciales iguales a los parámetros reales. Los cambios en los parámetros formales no afectan a los valores originales de los parámetros reales. Sin embargo, este enfoque puede llevar a la duplicación de memoria al copiar valores, lo que puede ser una desventaja en términos de eficiencia.

```
Ejemplo int factorial ( int x ) {
        int f=1;
        while (x) {
                f*=x; x--;
        }
       return f;
 }

int main (void ) {
int a;
int fac;
printf (" \n Iingrese un valor para a: ");
scanf("%d", &a); fac=factorial(a);
printf (" \n El factorial de %d es %ld", a,fac);
getch();
return 0;
}
```



Pasaje de direcciones:
El pasaje de parámetros por dirección en C implica que, si el parámetro es un puntero (una dirección de memoria), se puede acceder y modificar la memoria apuntada desde fuera de la función. Esto permite cambiar valores fuera de la función y también puede utilizarse para devolver más de un resultado desde la función. En C, para pasar un parámetro por dirección, se utiliza el operador de dirección & al invocar la función. Luego, dentro de la función, se utiliza el operador de indirección * para acceder al valor almacenado en esa dirección.

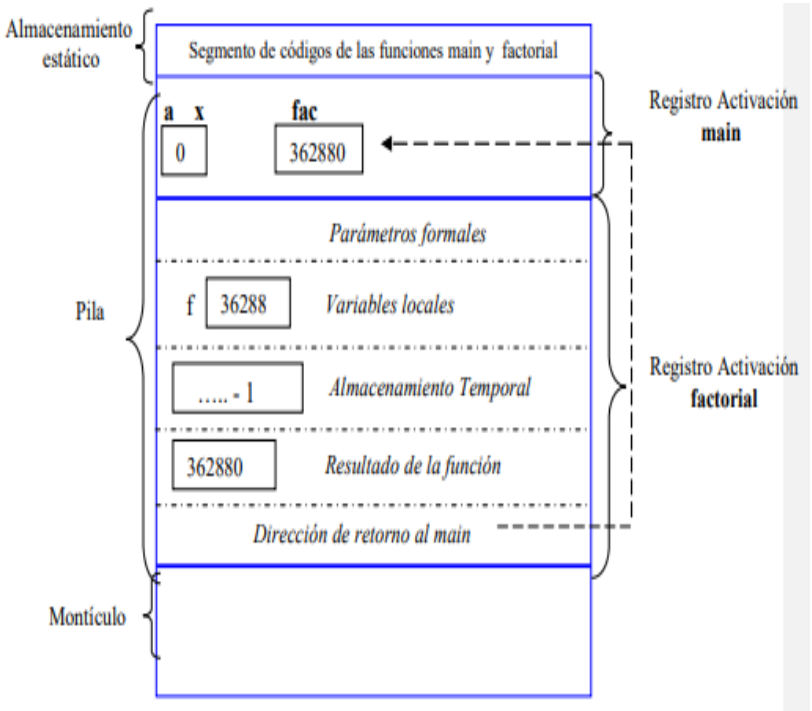
```
int factorial ( int * x ) {
int f=1;
while (*x) {
            f*=*x; --(*x);
 }
return f;
}

int main (void ) {
int a;
long int fac;
printf (" \n Ingrese un valor para a: ");
scanf("%d", &a);
printf("Valor de a antes de invocar a la función
factorial %d\n \n ", a);
fac=factorial(&a);
printf("Valor de a después de invocar a la
función factorial %d\n \n ", a);
printf (" \n El factorial es %ld", fac);
getch();
}
```



Pasaje por referencia:

El pasaje por referencia implica que el argumento que se pasa a la función debe ser una variable con una dirección asignada. A diferencia del pasaje por valor, que permite pasar variables, expresiones o constantes, el pasaje por referencia se basa en pasar la dirección de memoria de la variable. Cuando se utiliza el pasaje por referencia, el parámetro formal de la función se convierte en un alias o un segundo nombre para el parámetro actual. Esto significa que cualquier cambio realizado en el parámetro formal se reflejará directamente en el parámetro actual. Es como si ambas variables apuntaran a la misma área de memoria. Lenguajes como C++ y Pascal permiten el uso del pasaje por referencia. En C++, se indica el pasaje por referencia colocando el operador & después del tipo de dato del parámetro formal. Este enfoque permite modificar directamente la variable original desde dentro de la función sin la necesidad de trabajar con punteros.

```
int factorial ( int * x ) {
int f=1;
while (*x) {
            f*=*x; --(*x);
 }
return f;
}
int main (void ) {
int a;
long int fac;
printf (" \n Ingrese un valor para a: ");
scanf("%d", &a);
printf("Valor de a antes de invocar a la
función factorial %d\n \n ", a);
fac=factorial(&a);
printf("Valor de a después de invocar a la
función factorial %d\n \n ", a);
printf (" \n El factorial es %ld", fac);
getch();
}
```



Funciones que retornan más de un resultado:

Hay situaciones en las cuales es necesario que la función devuelva más de un resultado. La forma de lograr resolver este problema, es a través del pasaje de parámetros por direcciones o referencias.

```
int acumula_pares_impares(int &si,  int xnum)
{  int sp=0;
   while (xnum)
   {
       if (xnum%2==0)
            sp+=xnum;
       else
            si+=xnum;
       xnum--;
    }
    return sp;
}

void main(void)
{int sumai=0, num;
printf("ingrese un número \n"); scanf("%d", &num);
printf(" suma de pares  = %d \n", acumula_pares_impares(sumai,num));
printf(" suma de impares menores que %d = %d \n",num,sumai);
getch();
}
```

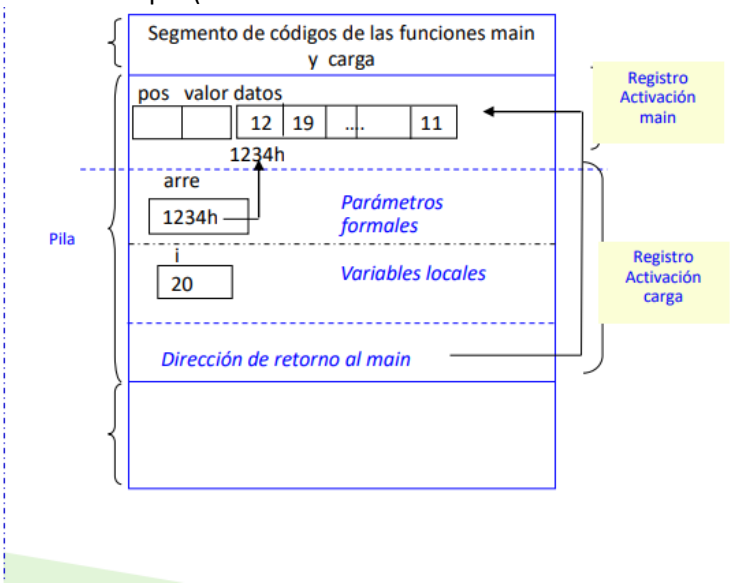
A tener en cuenta para la definición de funciones:

Es importante tener en cuenta que una función define una operación por lo cual debe ser definida de manera sencilla (fácil de entender) y la comunicación con ella debe ser lo más simple posible. Por lo tanto, no es eficiente comunicarnos con una función a través de más parámetros de los necesarios.

Arreglos como parámetros de funciones:

El nombre de un arreglo es una dirección constante que apunta a la primer componente del mismo, por lo cual el pasaje de un arreglo es un pasaje por valor de un parámetro simple (una dirección de memoria:

```
void carga ( float arre[], n )
{  int i;
   for (i=0; i < n ; i++ )
        scanf(" %f", &arre [i]);
   return ;
}

void main (void )
{
   int pos,valor;
   float datos [20];
   carga(datos, 20);
    ----------
    ----------
   getch();
}
```



Variables Automaticas:

Son las declaradas en la lista de parámetros o en el cuerpo de una función. Se almacenan en el registro de activación de la función y su alcance es restringido a la función en la que se declaran.

¿Qué sucede cuando dos variables correspondientes a distintas funciones tienen el mismo nombre?

Cuando dos variables correspondientes a distintas funciones tienen el mismo nombre, no hay ningún conflicto en términos de alcance o ambigüedad. Cada función trata sus variables locales (variables automáticas) como si fueran

únicas dentro de su propio contexto. Esto significa que una variable con el mismo nombre en diferentes funciones no se "mezcla" ni interfiere entre sí. Cada función trabaja con su propia copia local de la variable.

¿Qué ocurre cuando una variable automática está inicializada?
Cuando una variable automática está inicializada, significa que se le asigna un valor específico en el momento de su declaración. Esta inicialización puede ocurrir en la declaración de la variable o más adelante en el cuerpo de la función. Cuando una variable automática está inicializada, su valor inicial será el que se le haya asignado.

Variables Externas:
Se definen una sola vez, fuera de cualquier función y pueden ser inicializadas. Su alcance no se restringe a una función, todas las funciones puede tener acceso a ella a través de su nombre. Se almacenan en el área de almacenamiento estático. En algunos casos, para el uso de una variable externa en una función es necesario el uso del especificador extern.

¿Qué sucede si una función altera el valor de una variable externa?
Cuando una función altera el valor de una variable externa, esos cambios son reflejados en la variable externa en todo el programa, ya que todas las funciones dentro del programa pueden acceder a la misma variable externa a través de su nombre. Esto se debe a que las variables externas tienen un alcance global en todo el programa.

Variables Estáticas:
Son locales a una función, se declaran con el especificador static, pueden ser inicializadas. Su alcance se restringe a la función en la que se declaran y se almacenan en el área de almacenamiento estático, por ello mantienen la información a lo largo de la ejecución.

Bloques en lenguaje C:
Un bloque es una secuencia de declaraciones, seguidas por una secuencia de enunciados, rodeados por marcadores sintácticos (llaves de inicio-terminación).
Las declaraciones brindan información muy importante. A partir de las declaraciones, tiempo de traducción, se construye la Tabla de Símbolos, para realizar verificación estática de tipos.

Alcance de un vínculo en C:
Las declaraciones vinculan atributos a un nombre. Alcance de este vínculo es la región del programa donde este vínculo se mantiene. En C, lenguaje estructurado en bloques, el alcance de un vínculo queda limitado al bloque donde aparece la declaración asociada, y a los bloques contenidos en el interior. Desde el punto siguiente a la declaración, hasta el final del bloque en el cual ésta se encuentra (alcance léxico).

Apertura en el Alcance: Visibilidad:
La visibilidad incluye únicamente aquellas regiones de un programa donde las ligaduras de una declaración son aplicables.

Tabla de Símbolos:
La tabla de símbolos en un lenguaje de programación es una estructura de datos que se crea durante el proceso de traducción del programa fuente. Su función principal es mantener un registro de los identificadores (nombres de variables, funciones, etc.) utilizados en el programa y sus atributos asociados. Aquí hay un resumen de los conceptos clave relacionados con la tabla de símbolos en el lenguaje C:

Tabla de Símbolos en Lenguaje C:
La tabla de símbolos es una estructura de datos utilizada para mantener información sobre los identificadores (nombres de variables, funciones, etc.) utilizados en un programa.

Atributos en la Tabla de Símbolos:
Cada entrada en la tabla de símbolos se llama "registro de la tabla de símbolos" (symbol-table record) y contiene información sobre un identificador. Los atributos almacenados en un registro de la tabla de símbolos pueden incluir:

- Nombre del identificador.
- Dirección en tiempo de ejecución para variables.
- Tipo del identificador, que puede ser el tipo de datos o el tipo de retorno de una función.
- Número de dimensiones para arreglos.
- Tamaño máximo o rango de cada dimensión para arreglos de dimensión estática.
- Información sobre miembros de estructuras, uniones o clases.
- Tipo y forma de acceso para miembros de estructuras, uniones o clases.
- Tipo de cada parámetro de funciones, entre otros.

Operaciones en la Tabla de Símbolos:

- Las dos operaciones principales realizadas en la tabla de símbolos son la inserción y la búsqueda.
- La inserción ocurre cuando se procesa una declaración que introduce un nuevo identificador en el programa.
- La búsqueda se utiliza para encontrar información sobre un identificador existente en la tabla de símbolos.

Set y Reset:
Además de inserción y búsqueda, se utilizan operaciones SET (activar un bloque) y RESET (desactivar un bloque) para gestionar el alcance de los identificadores en bloques de código anidados.

- SET se usa al comenzar un bloque y agrega registros de identificadores al alcance actual.
- RESET se utiliza al salir de un bloque y elimina registros de identificadores que ya no están en alcance.

Organización en Forma de Pila:
La tabla de símbolos se organiza comúnmente como una pila (LIFO - Last In, First Out) para manejar bloques anidados.
Cada bloque nuevo puede agregar registros de identificadores a la parte superior de la pila, y cuando se sale de un bloque, se eliminan los registros correspondientes.
Los registros en la parte superior de la pila representan los identificadores actualmente en alcance.

Función SET y RESET:
La función SET se utiliza para guardar el índice de bloque (BLOCK INDEX) en una pila auxiliar cuando comienza un bloque. El índice representa dónde se encuentra almacenado el primer registro del bloque.
RESET se utiliza cuando se detecta el final de un bloque y se eliminan registros de identificadores que ya no están en alcance.

Este manejo de la tabla de símbolos permite gestionar el alcance de los identificadores en un programa C, asegurando que las variables tengan la visibilidad y duración adecuadas según su contexto dentro del código fuente.

Unidad 4: Recursividad

La recursividad es una alternativa a la iteración, es un proceso mediante el cual se puede definir una función en términos de sí misma.

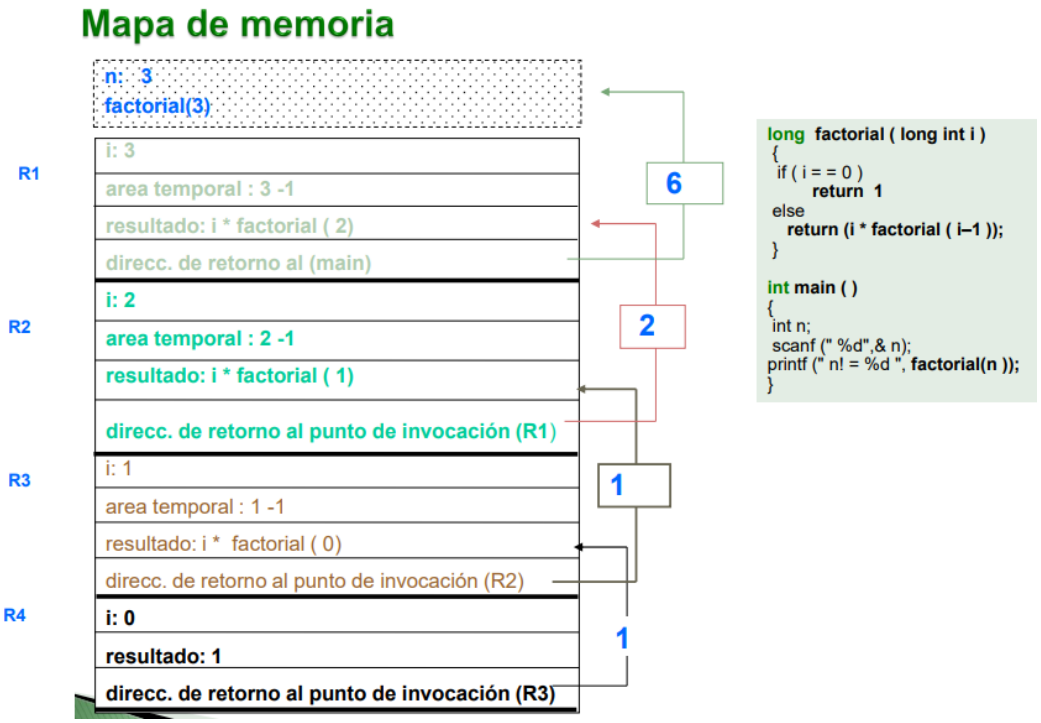
Características de la Recursividad:
La recursividad suele ser menos eficiente en términos de tiempo de ejecución y uso de memoria en comparación con la iteración, pero puede ser más simple y natural para algunos problemas.
Es importante definir al menos un caso base o condición de terminación que indique cuándo debe detenerse la recursión para evitar un bucle infinito. También es importante definir uno o más casos generales, que permiten que la función se invoque a sí misma con valores de parámetros que cambian en cada llamada; acercándose cada vez más al caso base.

Ventajas de la Recursividad:
La recursividad puede hacer que el código sea más claro y conciso al expresar soluciones de manera elegante para problemas complejos.
En algunos casos, la recursividad es la única forma natural de abordar un problema, como en problemas relacionados con estructuras de datos recursivas, árboles, etc.

Desventajas de la Recursividad:
La recursión puede tener un mayor uso de memoria debido a la pila de llamadas que se utiliza para almacenar las llamadas recursivas pendientes.
En problemas grandes, la recursión puede resultar en un mayor tiempo de ejecución debido al costo de administrar múltiples llamadas de función.

Construcción y ejecución de funciones recursivas:
En la ejecución, las sentencias que aparecen después de cada invocación no se resuelven inmediatamente, éstas quedan pendientes. Cada invocación recursiva, genera un registro de activación y se apila en el stack o pila. Esos registros de activación se van desapilando en el orden inverso a como fueron apilados.

Mapas de memoria:



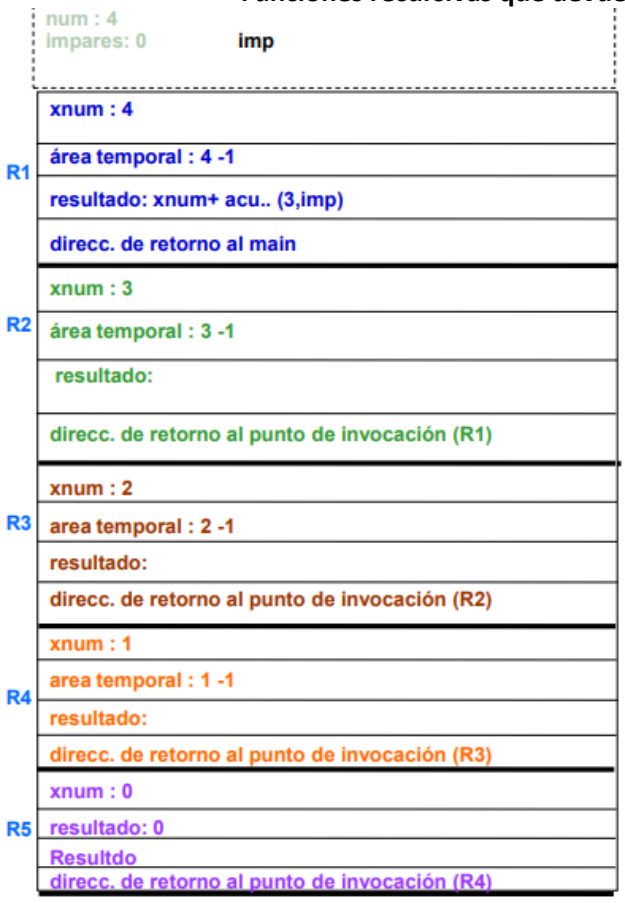
Funcionamiento de la Recursión:
Cuando se invoca una función recursiva, se apilan sucesivamente registros de activación, correspondientes a cada invocación recursiva. Estos registros de activación almacenan información sobre el estado de cada llamada recursiva.

Al llegar al caso base, comienza el proceso de desapilado de los registros de activación, resolviendo tareas pendientes si las hay.
La recursión puede ocupar una cantidad considerable de memoria en la pila debido a las múltiples llamadas recursivas en espera de resolución.

Variables Locales en la Recursión:
Si una función recursiva tiene variables locales, se creará un conjunto diferente de estas variables para cada llamada recursiva.
Aunque los nombres de las variables locales son los mismos en todas las llamadas, representan conjuntos de valores diferentes en cada invocación.

Recursión vs. Iteración:
La recursión y la iteración son lógicamente equivalentes, lo que significa que cualquier algoritmo recursivo puede expresarse de manera iterativa y viceversa.
La recursión tiende a ser más simple y sintética al escribirse, lo que puede hacer que el código sea más legible y claro.
La desventaja de la recursión suele radicar en el mayor uso de memoria y, en algunos casos, en un mayor tiempo de ejecución debido a la gestión de la pila de llamadas.

Funciones recursivas que devuelven más de un resultado:

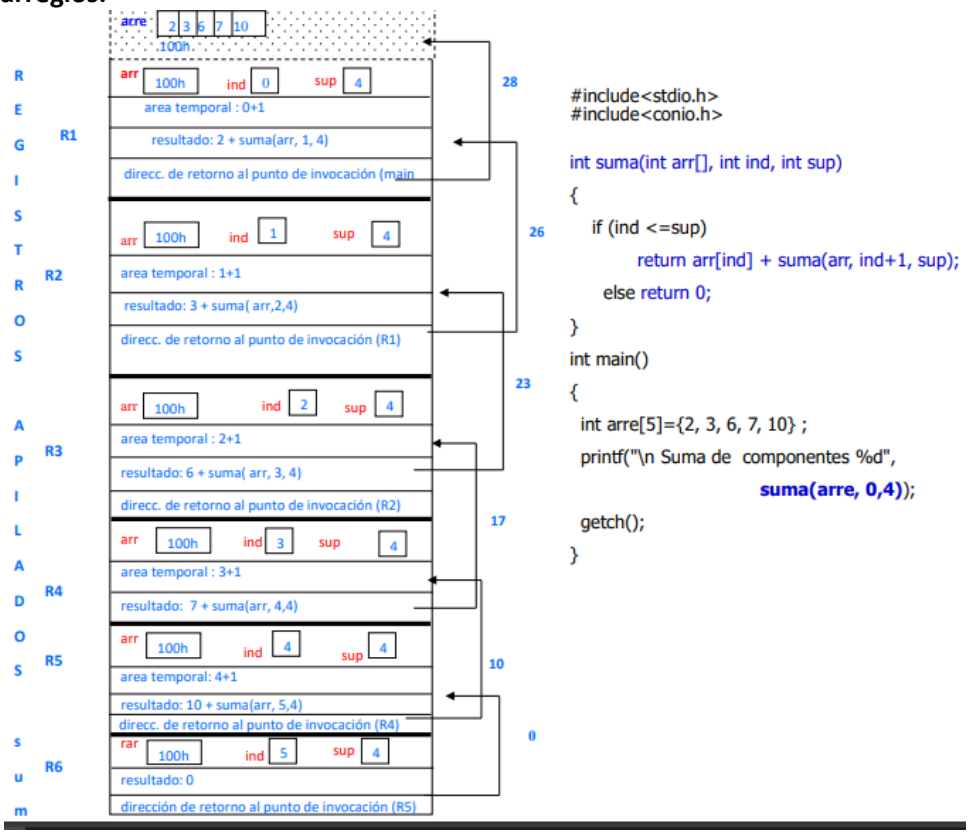



```
int acumula_pares_impares ( int xnum, int &imp)
{
 if  (xnum)
   if ((xnum % 2)== 0)
   return (xnum + acumula_pares_impares ( xnum - 1,imp));
   else { imp+= xnum;
          return(acumula_pares_impares ( xnum - 1,imp));
         }
 else return (0);
}
```

```
 int main(void)
  {
   int  num;
   int impares = 0;
   printf("\n ingrese un numero");
   scanf("%d", &num);
   printf("\n  la suma de los números
                                  pares <=  %d  ", num);
   printf("es : %d:",
            acumula_pares_impares (num,impares));
    printf("\n impares es : %d:", impares);
    getch();
   }
```

Programación Procedural: Resumen Parcial 1

Recursividad con arreglos:



Ordenación Rápida (Quicksort) con recursividad:
El Quicksort es uno de los métodos más rápidos de ordenamiento. El método consiste en:

1. Elegir un elemento del arreglo denominado pivote
2. Dividir el arreglo en dos subarreglos, de modo que todos los elementos menores que el pivote estén en un subarreglo y los elementos mayores que el pivote estén en el otro.
3. Repetir el proceso para cada subarreglo hasta llegar a subarreglos con un único elemento.

```
#include <stdio.h>
#define long 8

void carga(int arr[long], int i) {
   if (i < long) {
      printf("\nIngrese %dº numero: ", i + 1);
      scanf("%d", &arr[i]);
      carga(arr, i + 1);
   }
}
```

```
void quicksort(int arr[], int inf, int sup) {
   int t[long];
   int i, piv, m, n;
   if (inf < sup) {
      i = (inf + sup) / 2;
      piv = arr[i];
      for (i = inf; i <= sup; i++)
         t[i] = arr[i];
      m = inf;
      n = sup;
      for (i = inf; i <= sup; i++)
         if (t[i] < piv) {
            arr[m] = t[i];
            m = m + 1;
         } else if (t[i] > piv) {
            arr[n] = t[i];
            n = n - 1;
         }
      for (i = m; i <= n; i++)
         arr[i] = piv;
      quicksort(arr, inf, m - 1);
      quicksort(arr, n + 1, sup);
   }
   return;
}

int main() {
   int i;
   int a[long];
   carga(a, 0);
   quicksort(a, 0, long - 1);
   for (i = 0; i < long; i++)
      printf("a[%d] = %d ", i, a[i]);
   return 0;
}
```