

# PROGRAMACIÓN PROCEDURAL

## Tipos de datos y Funciones

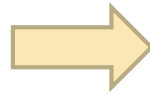
### Unidad 3



# Definición de Tipos

La definición de tipos permite crear nuevos tipos de datos y operaciones sobre ese tipo, de modo que pueda ser usado como un tipo de dato provisto por el lenguaje.

**Tipos básicos**  
(int, char, float, double)



**Tipos complejos**

*constructores de tipo*

Una definición de tipos proporciona un **nombre** de tipo junto con una declaración que describe la estructura de una clase de objetos de datos. El nombre del tipo se convierte en el nombre de esa clase de objetos de datos, y cuando se necesita trabajar con un objeto de datos particular basta con proporcionar el nombre del tipo en lugar de repetir la descripción completa de la estructura de datos.

*¿Qué ocurre si deben declararse dos arreglos de 10 componentes enteras ?*

```
int a[20], b[20]
```

*Las variables a y b no tienen un nombre de tipo asociado*

*¿Cómo dar en nombre de tipo ?*

```
typedef int arre[20];  
arre a,b
```

## Definición de Tipos

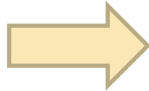
Formato general:

```
typedef <definición de tipo > <nombre del tipo>
```

# Definición de Tipos

## Lenguaje C:

```
struct partido_politico  
{  
    int Numero;  
    int Votos[19];  
};
```



**struct partido\_politico** define un nuevo tipo de datos. Es el único tipo de datos que provee lenguaje C.

```
struct partido_politico P;
```

El lenguaje C provee la construcción **typedef** para asignar un nombre a este tipo de datos, de manera que “parezca” una extensión del lenguaje y sea sintáctica y semánticamente similar a los tipos primitivos de datos.

```
typedef struct  
{  
    float real;  
    float imaginario;  
} complejo;
```

```
complejo c1, c2;
```

# Definición de Tipos

**typedef** no crea un nuevo tipo de datos, el efecto producido es el de una por macro

**Formato general:**

```
typedef <definición de tipo > <nombre del tipo>
```

Dada la declaración **typedef** char **cadena**[30];

Declarar 2 cadenas de ese tipo

```
typedef char cadena[30];
int main(void)
{
    cadena ciudad[10];
    int i;
    for(i=0; i<=10;i++)
    { printf("\n Ingrese el nombre de la ciudad: ");
      gets(ciudad[i]);
    }
    printf("\n Ciudades ingresadas\n");
    for(i=0; i<=10;i++)
        puts(ciudad[i]);
    getch();
    return 0;
}
```

*Código que lee y escribe los  
nombres de 10 ciudades*

# Sistema de tipos

Un **sistema de tipos** incluye los métodos utilizados para la construcción de tipos, el algoritmo de equivalencia de tipos, y las reglas de inferencia y corrección de tipos.

Si un lenguaje presenta un sistema de tipos completo que pueda aplicarse estáticamente y que garantice que todos los errores de corrupción se detecten lo antes posible, entonces se dice que el lenguaje es **fuertemente tipificado**. Ej Pascal

Lenguaje C es un lenguaje que incluso tiene más fallas por eso a veces se lo conoce como un lenguaje **débilmente tipificado**.

## Ventajas de la definición de tipos:

- ✓ Simplificación la estructura del programa
- ✓ Modificación más eficiente.
- ✓ En el uso de subprogramas, facilita el pasaje de argumentos, pues evita repetir la descripción del tipo de datos.

# Subprogramas

Representan una operación abstracta definida por el programador

```
typedef int vector[10]; // Vector se usa como nombre de un Tipo de datos
```

```
#define N 10;
```

```
int escalar (vector a, vector b, int N) // Operación producto escalar
```

```
{ int e=0,i;  
  for(i=0; i < N; i++)  
    e+=a[i]*b[i];  
  return e;  
}
```

```
void carga(vector x,int N) // Operación de carga del vector
```

```
{ int i;  
  printf(" \n ingrese las componentes del vector");  
  for(i=0; i<N; i++)  
    scanf("%d",&x[i]);  
  return;  
}
```

```
void main(void)
```

```
{ vector v1, v2;
```

```
  carga(v1,N);
```

```
  carga(v2,N);
```

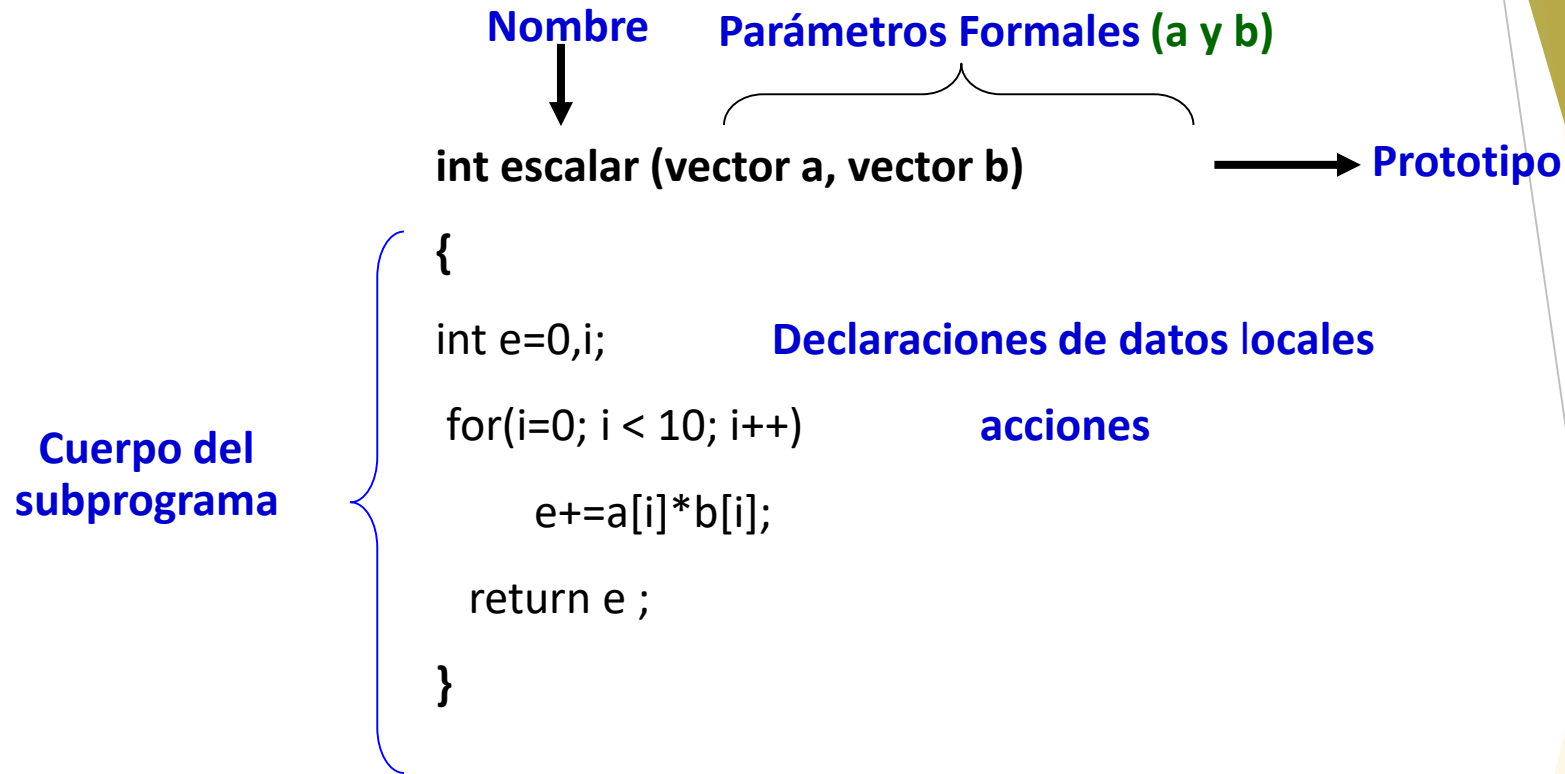
```
  printf(" el producto escalar es %d ", escalar(v1,v2,N));
```

```
}
```

## Ventajas de usar subprogramas ??

- Reusabilidad de tareas.
- Mayor claridad y legibilidad al programa principal, aún cuando los subprogramas no se invoquen de manera repetida.

## Especificación e Implementación de un subprograma



1. En Pascal, el cuerpo puede incluir otras definiciones de subprogramas , en lenguaje C no es posible.
2. En ambos lenguajes, la verificación de tipos es estática, se conoce el tipo de datos de argumentos y del resultado.



# Definición, invocación y activación de subprogramas

**Definición de un subprograma:** es una propiedad estática de lenguaje; **en tiempo de traducción** es la única información disponible. Ejemplo: tipo de variables de argumentos, de variables locales, etc.

*La traducción de la definición de un subprograma es una plantilla que permite generar activaciones en tiempo de ejecución.*

La plantilla se divide en partes con el fin de ahorrar memoria:  
**segmento de código y registro de activación**

El **segmento de código** es la parte estática compuesta por las constantes y el código ejecutable generado a partir de **enunciados del cuerpo de la función.** (*invariable durante la ejecución*)

# Definición, invocación y activación de subprogramas

**Activación de un subprograma:** se genera **en tiempo de ejecución** cuando se lo llama o invoca. Al terminar la ejecución, la activación se destruye.

El **registro de activación** es la parte dinámica, compuesta por:

- Parámetros
- Resultados de la función
- Datos locales
- Punto de retorno
- Áreas temporales de almacenamiento
- Vinculaciones para referencias de variables no locales

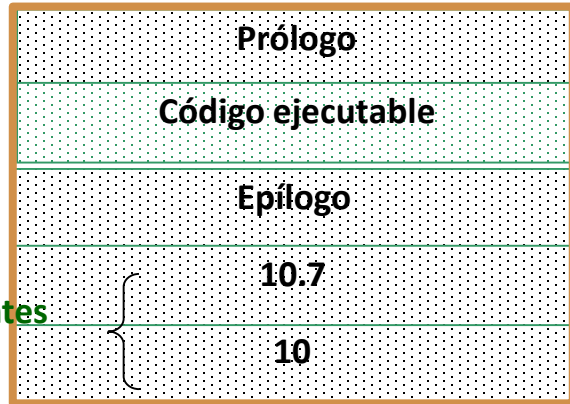
Los registros de activación se **crean cada vez que se invoca un subprograma** y **destruyen cada vez que el mismo concluye con un retorno.**

# Definición, invocación y activación de subprogramas

Segmento de código

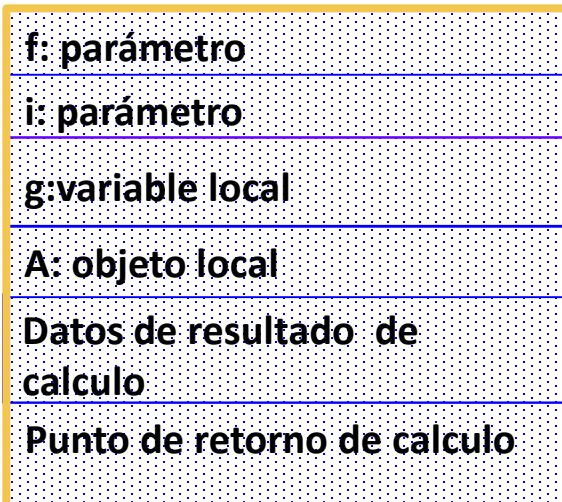


Constantes



**Prólogo:** bloque de código que el traductor introduce al comienzo del segmento de código. Permite *tareas de creación del registro de activación*, transmisión de parámetros, creación de vínculos y actividades similares de mantenimiento.

**Epílogo:** conjunto de instrucciones que el traductor inserta al final del bloque de código ejecutable, para realizar acciones que permitan *devolver resultados y liberar almacenamiento destinado al registro de activación*.



Registro de activación

```
float calculo (float f, int i)
{
    const por=10.7
    float g;
    int A[10];
    .....}
main()
{.....
    g=calculo( m, n);
}
```

# Funciones en C

Una función es un conjunto de sentencias que realiza una determinada tarea, que retorna como resultado cero o un valor.

Un programa en C está formado por una o más funciones, siendo *main* la función principal por donde se inicia la ejecución del programa.

La función `getch()`, es una *función predefinida* que devuelve un único valor, un carácter.

```
char car;  
car=getch();
```

El uso de *funciones definidas por el programador* permite dividir un programa en un cierto número de componentes más pequeñas, cada una de éstas con un propósito específico y determinado, logrando así programas más fáciles de codificar y depurar.

# Funciones en C

**<tipo de dato> <identificador>(< tipo1 arg1, . . ., tipon argn>)**

```
float perimetro ( float xl, float xa )
```

```
{  
float perim;
```

```
perim = 2 * ( xl + xa ) ;
```

```
return perim;
```

```
}
```

**primera línea:** incluye declaración de argumentos

**cuerpo de la función**

## ¿ Cómo se utiliza?

Al **invocarla** desde algún punto del programa, se ejecutan las sentencias que forman parte de ella.

Al **finalizar su ejecución**, se devuelve el control al punto desde donde fue invocada: al **main** (programa principal) o a la función que la llamó.

```
void main(void)
```

```
{
```

```
:
```

```
leer();
```

```
.
```

```
.
```

```
.
```

```
p=perimetro();
```

```
void leer(...);
```

```
{ int p;
```

```
.
```

```
return};
```

```
float perimetro (...)
```

```
{ :
```

```
return (r)} };
```

## Funciones en C - Parámetros

```
float perimetro ( float xl, float xa )    // xl y xa son parámetros formales
{ float perim; // perim es variable local
  perim = 2 * ( xl + xa ) ;
  return perim ;
}
```

```
int main(void)
{ float largo, ancho;
  printf("ingrese el largo y el ancho"); scanf("%f", &largo);
  scanf("%f", &ancho); // largo y ancho son parámetros actuales
  printf ("perímetro:%f ",perimetro( largo,ancho));
}
```

- ❑ *Los parámetros actuales deben coincidir en tipo, orden y cantidad con los parámetros formales.*
- ❑ *El cuerpo de la función debe incluir al menos una sentencia **return** para devolver cero o un valor*
- ❑ *La sentencia **return** permite que se devuelva el control al punto de llamada o invocación.*

## Funciones en C – Ejemplos

```
int factorial (int a)
{
    int i, f=1;
    for(i=1; i <=a; i++)
        f*=i;
    return f;
}
```

```
void sumatoria ( int a, int b )
{
    int i, s=0;
    for (i=a; i<=b; i++)
        s+=2 * i + 1;
    printf(" la suma es %d ", s);
}
```

```
char signo ( int num)
{
    if ( num > 0)
        return ' p ';
    else
        if ( num < 0 )
            return ' n ';
        else
            return ' c ';
}
```

```
void cabecera(void)
{
    printf("      EMPRESA UNION S.A  \n");
    printf("      Prado 123 (S) - Tel: 2644378990 \n");
    printf("      San Juan  \n");
    return;
}
```

## Funciones en C – Declaración y Definición

```
#include <stdio.h>

int factorial (int a)
{   int i, f=1;
    for(i=1; i <=a; i++)
        f*=i;
    return f;
}

void main(void)
{   int c, m,n;
    printf(" ingrese m y n \n");
    scanf("%d",&m);
    scanf("%d", &n);
    c= factorial(m)/(factorial(n) * factorial(m-n)) ;
    printf("Combinaciones %d", c) ;
}
```

```
#include <stdio.h>

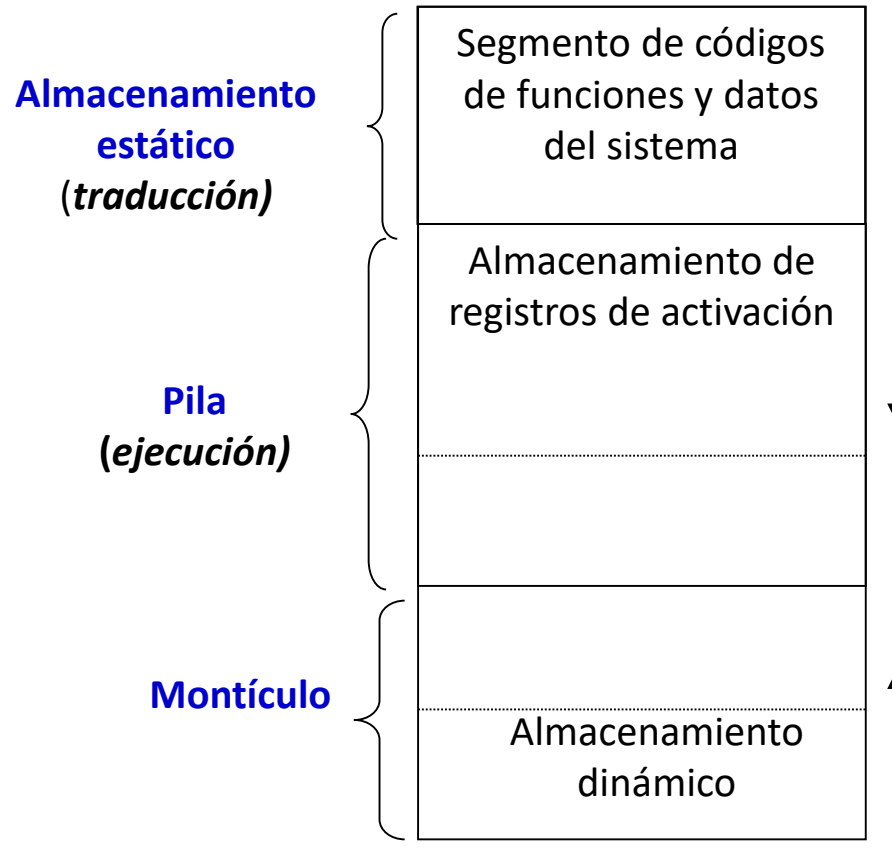
int factorial (int x);

void main(void)
{   int c, m,n;
    printf(" ingrese m y n \n");
    scanf("%d",&m);
    scanf("%d", &n);
    c= factorial(m)/(factorial(n) * factorial(m-n)) ;
    printf("Combinaciones %d", c) ;
}

int factorial (int a)
{   int i, f=1;
    for(i=1; i <=a; i++)
        f*=i;
    return f;
}
```



## Organización de la memoria en C

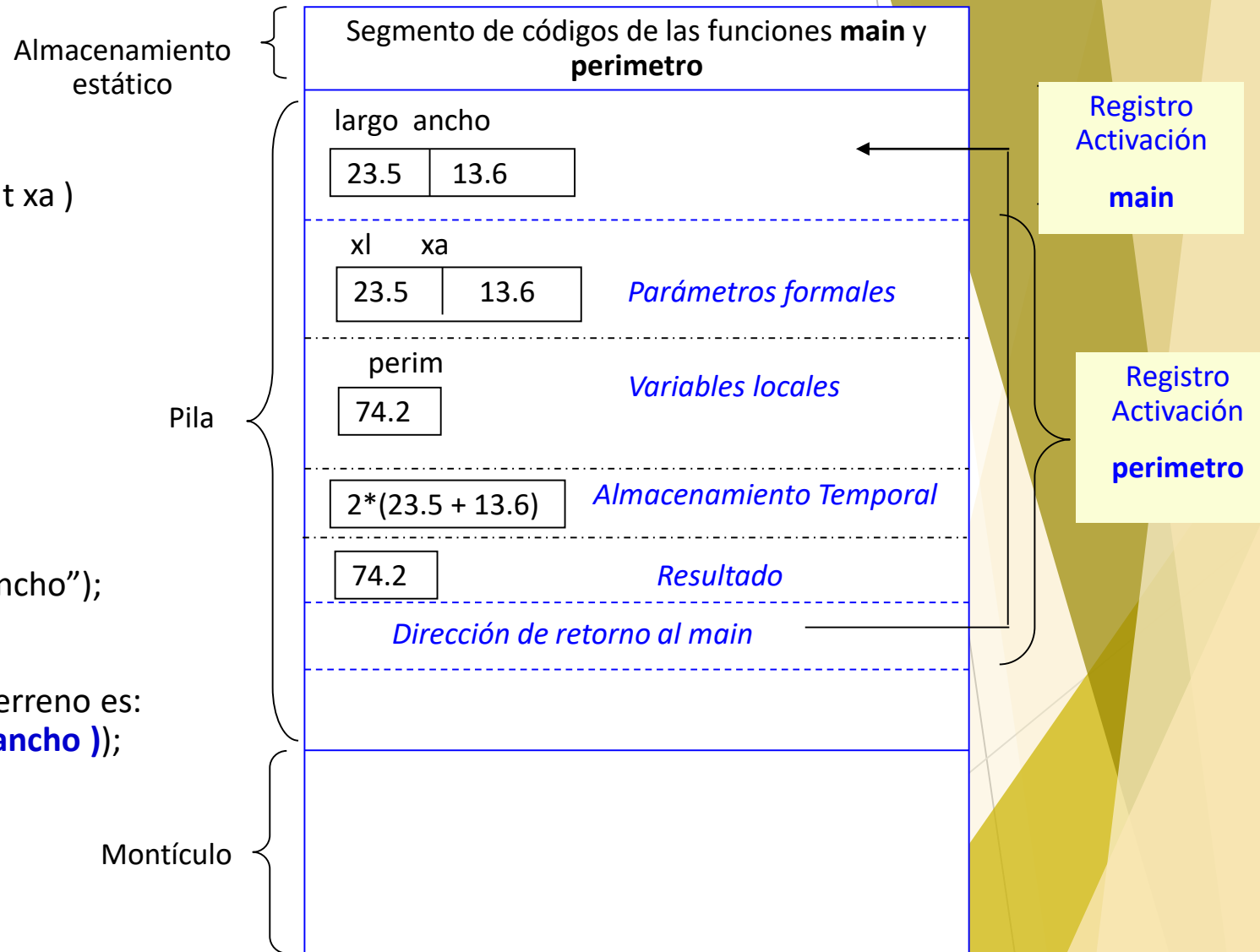


Organización de la memoria durante la ejecución de un programa

# Ejecución de un programa en C

```
# include <stdio.h>
# include <conio.h>
float perimetro ( float xl, float xa )
{
    float perim;
    perim = 2 * ( xl + xa );
    return perim;
}

void main(void)
{
    float largo, ancho;
    printf("ingrese el largo y el ancho");
    scanf("%f", &largo);
    scanf("%f", &ancho);
    printf("\n el perímetro del terreno es:
    %f ", perimetro( largo,ancho ));
}
```



**Fig 2.** Organización de la memoria durante la ejecución de la función `perimetro`

# Pasaje de Parámetros

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

## Pasaje por valor

```
void Func (int b)
```

```
{--
```

```
---
```

```
}
```

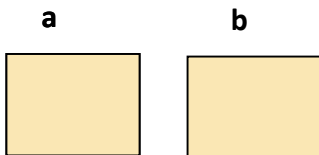
```
main()
```

```
{int a;
```

```
.....
```

```
Func(a)
```

```
}
```



## Pasaje por dirección

```
void Func (int * b)
```

```
{--
```

```
---
```

```
}
```

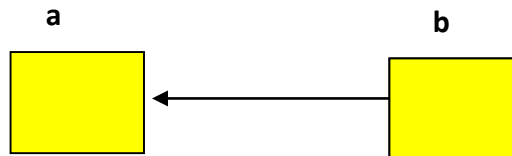
```
main()
```

```
{int a;
```

```
.....
```

```
Func(&a)
```

```
}
```



## Pasaje por referencia

```
void Func (int & b)
```

```
{--
```

```
---
```

```
}
```

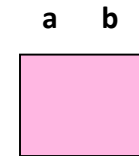
```
main()
```

```
{int a;
```

```
.....
```

```
Func(a)
```

```
}
```



# Pasaje por valor

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

```
# include < conio.h >
```

```
void modificar ( int x )
```

```
{
```

```
    x += 3;
```

```
    printf ( " \n x = %d (desde la función, modificando el valor) ", x);
```

```
    return;
```

```
}
```

```
void main (void )
```

```
{
```

```
    int a = 2;
```

```
    printf ( " \n a = %d (desde main, antes de invocar a la función modificar) ", a);
```

```
    modificar (a);
```

```
    printf ( " \n a = %d (desde main, después de invocar a la función modificar) ", a);
```

```
    getch();
```

```
}
```

# Pasaje por valor

```
long int factorial ( int x )
```

```
{ long int f=1;
```

```
  while (x)
```

```
  {
```

```
    f*=x;
```

```
    x--;
```

```
  }
```

```
  return f;
```

```
}
```

```
int main (void )
```

```
{ int a;
```

```
  long int fac;
```

```
  printf (" \n lingrese un valor para a: ");
```

```
  scanf("%d", &a);
```

```
  fac=factorial( a );
```

```
  printf (" \n El factorial de %d es %ld", a,fac);
```

```
}
```

**Salida:**

**Ingrese un valor para a: 9**

**El factorial de 9 es 362880**

Almacenamiento  
estático

Segmento de códigos de las funciones  
main y factorial

**a**      **fac**

**9**

362880

Registro  
Activación  
main

**x**

**0**

*Parámetros formales*

**f**

362880

*Variables locales*

..... 0

*Almacenamiento temporal*

362880

*Resultado de la función*

*Dirección de retorno*

Pila

Registro  
Activación  
factorial

Montículo

# Pasaje por valor

- Los parámetros reales se evalúan al momento de la llamada a la función y se transforman en los valores que toman los parámetros formales durante la ejecución de la función.
- El paso por valor es el mecanismo por omisión **de lenguajes como C++ y Pascal, siendo el único mecanismo de pasaje de parámetros de C y Java.**
- En todos estos lenguajes **los parámetros formales se consideran como variables locales** que toman como valor inicial el valor de los parámetros reales.
- Los parámetros formales pueden cambiar sus valores a través de asignaciones sin que estos cambios afecten los valores de los parámetros reales.

**Desventaja** : se produce duplicación del área de memoria.

# Pasaje por dirección

```
long int factorial ( int * x )
```

```
{ long 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 función %d\n\n", a);
fac=factorial(&a);
printf("Valor de a despues de invocar función %ld ", a);
printf (" \n El factorial es %ld", fac);
}
```

## **SALIDA:**

Ingrese un valor para a: 9

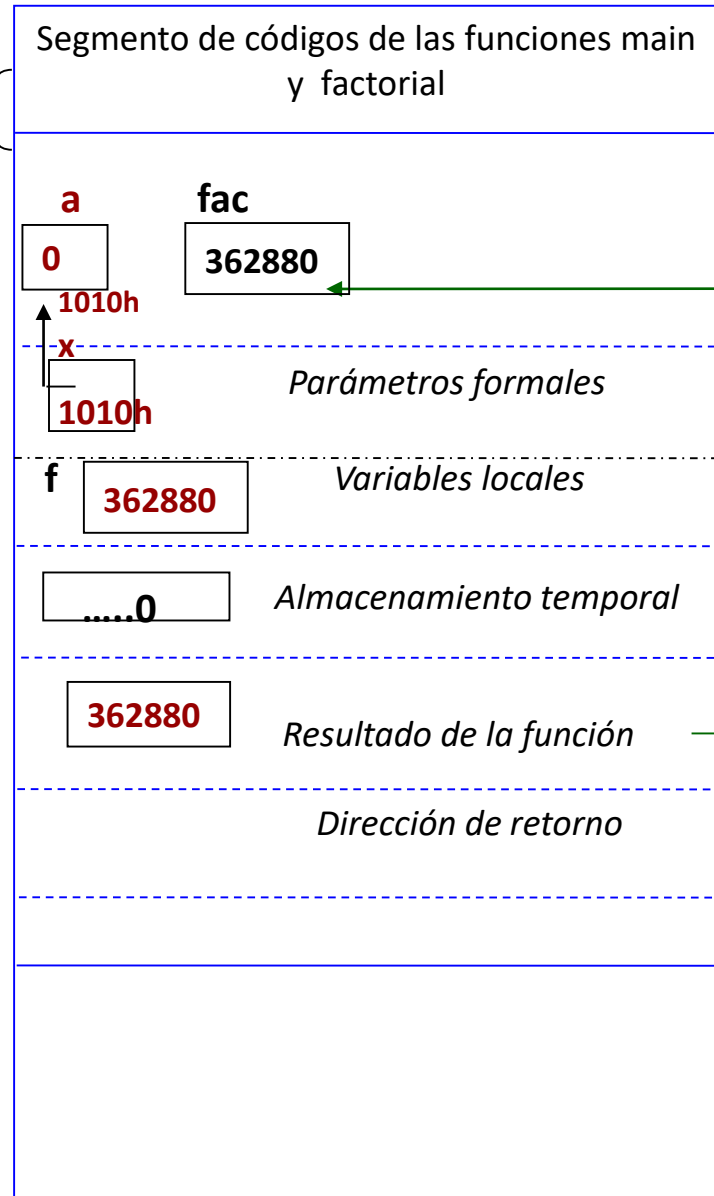
Valor de a antes de invocar a la función factorial 9

Valor de a después de invocar a la función factorial 0

El factorial es 362880

Almacenamiento  
o estático

Pila



# Pasaje por dirección

- Consiste en el paso por valor de una dirección, por ello puede usarse para cambiar el contenido de la memoria apuntada por ese puntero.
- El pasaje de parámetros por dirección permite retornar más de un resultado desde la función.
- En C, para pasar un parámetro por dirección, se utiliza el operador de **dirección &**, al momento de invocar a la función. Luego, el operador de **indirección \*** debe utilizarse en la función para acceder al valor almacenado en esa dirección.



# Pasaje por referencia

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

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

## SALIDA:

Ingrese un valor para a: **9**

Valor de a antes de invocar a la función factorial **9**

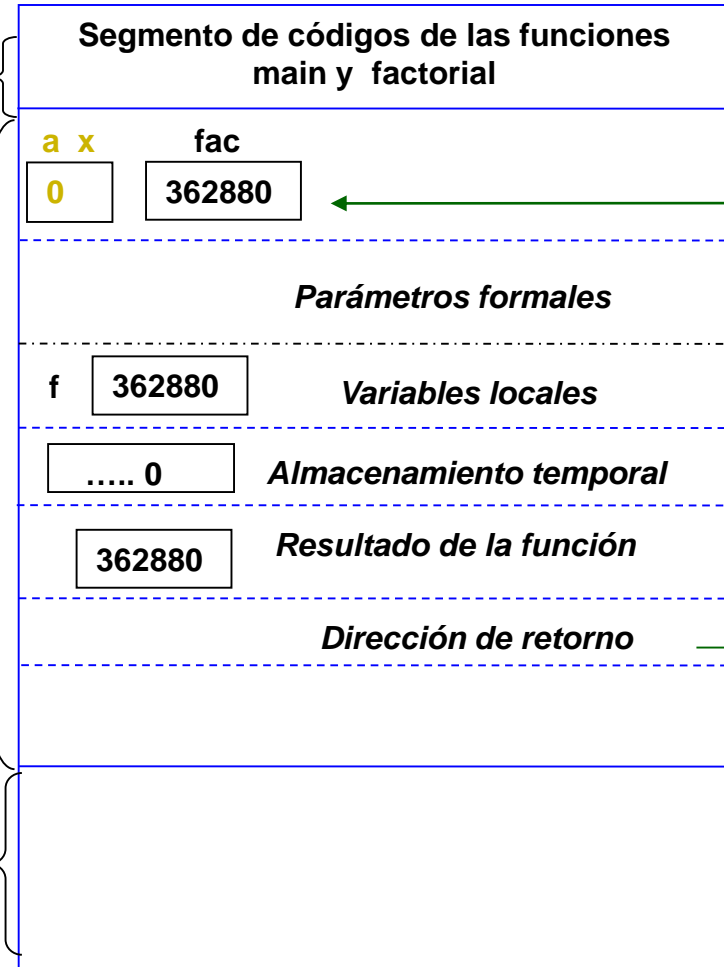
Valor de a después de invocar a la función factorial **0**

El factorial es **362880**

Almacenamiento  
estático

Pila

Montículo



Registro  
Activación  
main

Registro  
Activación  
factorial

# Pasaje por referencia

- Para realizar un pasaje por referencia el **parámetro actual a pasar debe ser una variable con una dirección asignada** .
- El pasaje por referencia pasa la ubicación de la variable, por lo que el parámetro formal se transforma en un **alias del parámetro actual** de modo que cualquier cambio que se realiza en el parámetro formal se siente en el parámetro actual.
- Esto puede interpretarse como que a un mismo área de memoria se asignan dos nombres distintos. Lenguajes como C++ y Pascal permiten el uso de pasaje por referencias.
- *En lenguaje C todas las llamadas son por valor, las llamadas por referencia se las puede simular pasando un puntero a la variable, y accediendo al contenido desreferenciando el puntero (pasaje por dirección).*

# Funciones que retornan más de un resultado

```
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();
}
```

## Algunas cuestiones a tener en cuenta para la definición de funciones

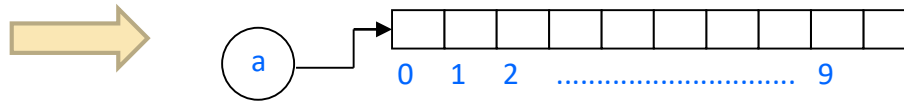
Es importante tener en cuenta que una función define una operación por lo cual debe ser definida de manera sencilla (fácil de entender) y la comunicación con ella debe ser lo más simple posible.

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)

**int a[10]**

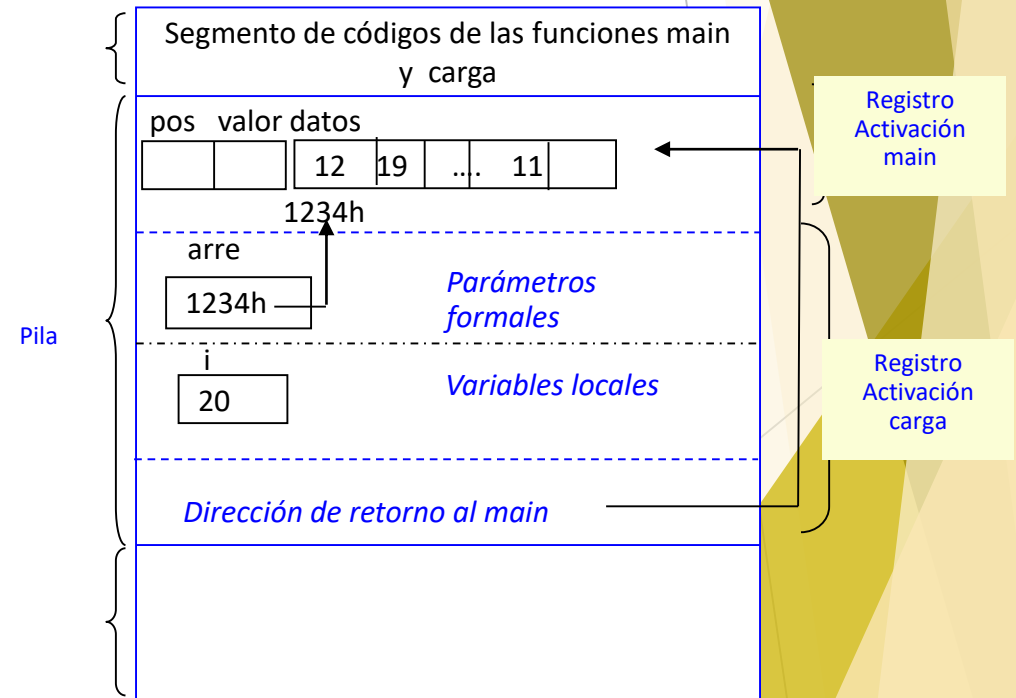


```
void carga ( float arre[], n )
```

```
{ 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();  
}
```



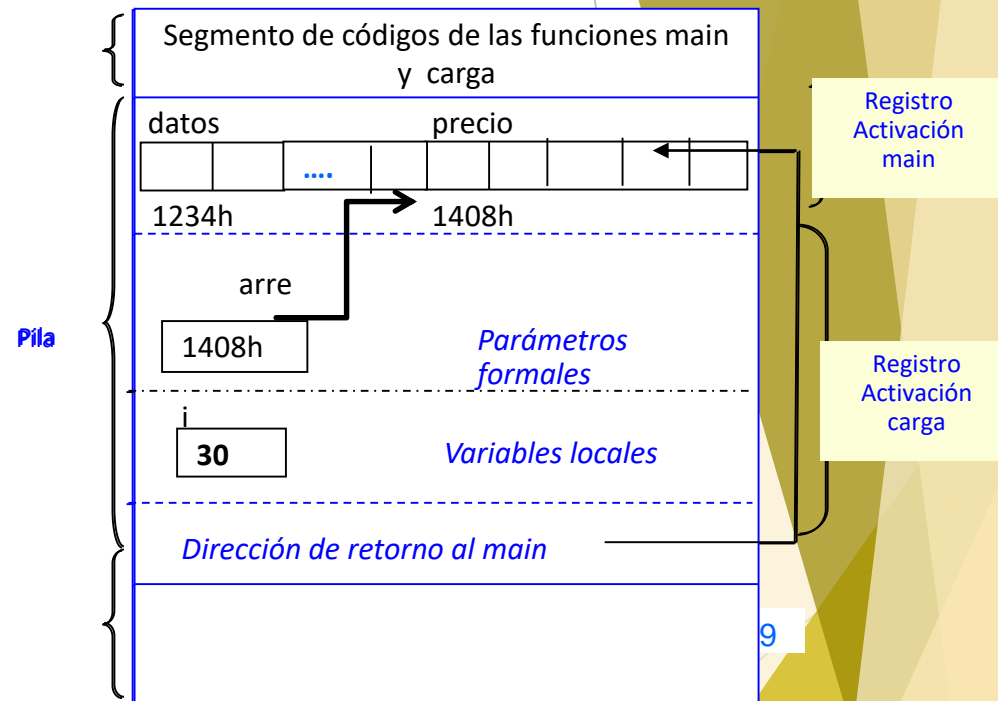
# Arreglos como parámetros de funciones

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

```
void mostrar( float arre[], int lim )
{ int i;
  for (i=0; i < lim ; i++)
    printf("\n %f", arre[i]);
}
```

```
int main (void )
{
  float datos[5], precios[30];
  carga(datos,5);
  mostrar(datos, 5);
  carga(precios, 30);
  mostrar(precios ,30);
  .....
  getch();
}
```

`void carga( float * arre, int n ) /*carga del arreglo*/`



# Arreglos como parámetros de funciones

```
int bus_secuencial (float *arre, int lim, float elem) /* busqueda secuencial*/
{
    int posi=0;
    while ( (posi!=lim) && (elem != arre[posi]))
        posi++;
    if (posi==lim)
        return -1;
    else return posi;
}
```

```
int main (void )
{
    float datos[5],valor;
    int pos;

    carga(datos,5);
    printf("\n DATOS DEL ARREGLO\n "); mostrar(datos, 5);
    printf("\nEl promedio de valores es %f ", media(datos,5));
    printf("\n INGRESE UN VALOR A BUSCAR\n"); scanf("%f",&valor);
    pos = bus_secuencial (datos,5, valor);
    if (pos != -1)
        printf("\n elemento encontrado en posición: %d", pos +1 );
    else printf("\n el elemento no se encontró");
}
```

# Arreglos como parámetros de funciones / Buenas prácticas

```
void carga( float * arre, int n )    /*carga del arreglo*/
{
    int i;
    printf("\n INGRESE las %d componentes del arreglo\n", n );
    for (i=0; i < n ; i++ )
    {
        printf("ingrese valor \n");
        scanf(" %f", &arre [i]);
    }
    return;
}
```

```
void mostrar( float const * arre, int lim )    /* muestra el arreglo*/
{
    int i;
    for (i=0; i < lim ; i++ )
        printf("\n %f", arre[i]);
}
```

```
float media ( float arre[ ], int lim )    /* función media */
{
    int i;
    float acum=0;
    for (i=0; i < lim ; i++ )
        acum=acum + arre [i];
    return acum/lim;
}
```

```
int main (void )
{
    float datos[5],valor;
    int pos;
    carga(datos,5);
    printf("\n DATOS DEL ARREGLO\n ");
    mostrar(datos, 5);
    printf("\nEl promedio es %f", media(datos,5));
    printf("\n INGRESE UN VALOR A BUSCAR\n");
    scanf("%f",&valor);
    pos = bus_secuencial (datos,5, valor);
    if (pos != -1)
        printf("\n elemento encontrado en posición: %d", pos +1 );
    else printf("\n el elemento no se encontró");
}
```



# Pasaje constante en arreglos

```
void intenta_modificar (const int b [ ] )
{
    b [0] = 2;    /*error*/
    b [1] = 2;    /*error*/
    b [2] = 2;    /*error*/
}

void main (void )
{
    int a [ ] = { 10 , 20 , 30 };
    intenta_modificar ( a );
    printf (" %d %d %d \n", a [0], a [1], a [2 ] );
}
```

# Tipos de almacenamiento

Clasificación  
de variables

- Por su tipo (int, float, char, etc.)
- Por su almacenamiento ( tiempo de vida y alcance)

De acuerdo al  
almacenamiento

- Almacenamiento Estático: globales y statics
- Automáticas
- Dinámicas

# Variables Automáticas

Son las declaradas en la lista de parámetros o en el cuerpo de una función

```
void p (float r)
{
    doble d;
    ...int x
}
```

**Alcance:** restringido a la función en la que se declaran

```
void q (void)
{   int x;
    ...
}
```

**Almacenamiento:** registro de activación de la función

```
void main(void)
{   char z;
    ...
}
```

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

- ¿Qué ocurre cuando una variable automática está inicializada?

# 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 pueden 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**.

```
int x=5;
void p (void)
{  doble d;
  ...
}

void q (void)
{  int y;
  ...
}

void main(void)
{  char z;
  ...
}
```

¿ Qué sucede si una función altera el valor de una variable externa?

# 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
- ▶ Se almacenan en el área de **almacenamiento estático**, por ello mantienen la información a lo largo de la ejecución.

```
void stat()
{
    int a=0;
    static int s=0;
    printf("auto= %d, static= %d \n", a,s);
    a++;
    s++ ;
}
main()
{
    int i;
    for(i=0;i<5;++i)
        stat();
    getchar();
}
```

funcione.cpp | varefere.cpp | actividad8 FILMINAS.cpp | variab statics.cpp

```
#include<stdio.h>
void stat()
{
    int a=0;
    static int s=0;
    printf("auto= %d,      static= %d \n", a,s);
    printf("\n");
    printf("\n");
    a++;
    s++ ;
}
int main()
{
    int i;
    for(i=0;i<5;++i)
        stat();
    getchar();
}
```

C:\ F:\PROCEDURAL 2014\UNIDAD 3\variab statics.exe

```
auto= 0,      static= 0
auto= 0,      static= 1
auto= 0,      static= 2
auto= 0,      static= 3
auto= 0,      static= 4
```

# 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)

```
int x;  
float p (void)  
{ doble d;  
  ...  
}  
  
void q (void)  
{ int y; //inicio de un bloque  
  ....  
  ...  
  while .. // bloque anidado  
  { int z;  
    ...  
  }  
} // fin de bloque  
  
void main(void)  
{ char z;  
  ...  
}
```

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 lenguaje 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 bloque contenidos en el interior**. Desde el punto siguiente a la declaración, hasta el final del bloque en el cual ésta se encuentra. (alcance léxico).

```
int x;           —————> global
float p (void)  —————> global
{
  doble d;      —————> local
  ...
}
void q (void)   —————> global
{
  int y;        —————> local
  ...
}

int main(void) —————> global
{
  char z;       —————> local
  ...
}
```



# Apertura en el Alcance: Visibilidad

La visibilidad incluye únicamente aquellas regiones de un programa donde las ligaduras de una declaración son aplicables.

```
int x;  
void p (void)  
{  doble d;  
    float x  
    {  
        ..  
    }  
}  
void q (void)  
{  int y;  
    x= ??  
    d= ??...  
}  
void main(void)  
{  char z;  
    x=.... ??  
    ...  
}
```

# Apertura en el Alcance: Visibilidad

```
int i,j;
```

```
float cambia(void )
```

```
{
```

```
    float i=17.5;
```

```
    char j;
```

```
    j='a';
```

```
    i=i/2;
```

```
    printf("\n variable j es ahora el caracter %c",j );
```

```
    return i;
```

```
}
```

```
int main(void)
```

```
{
```

```
    printf("ingrese un entero ");    scanf("%d",&i);
```

```
    printf("\n ingrese un entero "); scanf("%d",&j);
```

```
    printf("\n j es un entero en el main: %d",j);
```

```
    printf("\n Cambio en subprograma: 5.2f",cambia());
```

```
    printf("\n Valor de i en principal : %d",i);
```

```
}
```

```

#include <stdio.h>
int i,j;
float cambia(void )
{ float i=17.5;
  char j;
  j='a';
  i =i/2;
  printf("\n  variable j es ahora el caracter %c",j  );
  return i;
}

int main(void)
{printf("ingrese un entero ");scanf("%d",&i);
  printf("\n ingrese un entero ");scanf("%d",&j);
  printf("\n  j es un entero en el main: %d",j);
  printf("\n Cambio  en subprograma:%5.2f",cambia());
  printf("\n  Valor de i en principal : %d",i);
  getchar();
  getchar();
}

```

D:\IVO\PROCEDURAL 2015\Unidad 3\Ejercicios teoría\va

ingrese un entero 10

ingrese un entero 15

j es un entero en el main: 15  
 variable j es ahora el caracter a  
 Cambio en subprograma: 8.75  
 Valor de i en principal : 10\_

```

# include <stdio.h>
# include <conio.h>
int b=10;
void mostrar(int x, int y)
{
    x=b+y;
    while (x>9)
    {
        int b=2;
        printf("\n valor de b en bloque interior = %d", b);
        printf("\n valor de x= %d", x);
        x-=b;
    }
    printf("\n valor de b fuera del bloque= %d", b);
}
int main(void)
{
    int a=2, b=3;
    mostrar(a,b);
    printf("\n valor de b en principal = %d", b);
    getch();
}

```

F:\IVO\PROCEDURAL 2015\Unidad 3\activ5 filmina.exe

```

valor de b en bloque interior = 2
valor de x= 13
valor de b en bloque interior = 2
valor de x= 11
valor de b fuera del bloque= 10
valor de b en principal = 3

```

# Tabla de Símbolos


La tabla de símbolos es una estructura de datos, se crea en tiempo de traducción, permite dar apoyo a la inserción, búsqueda y cancelación de identificadores con sus atributos asociados,

## Estructura de datos que soporta una Tabla de Símbolos

puede ser una tabla, una pila, un árbol, etc.

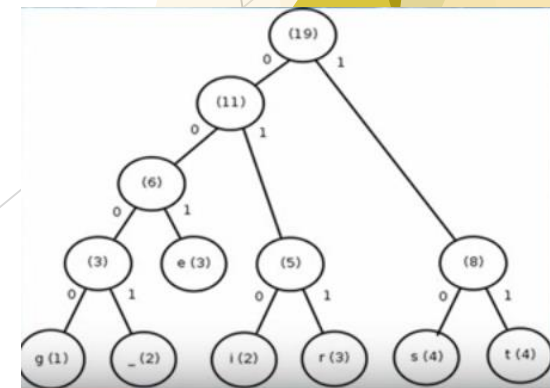
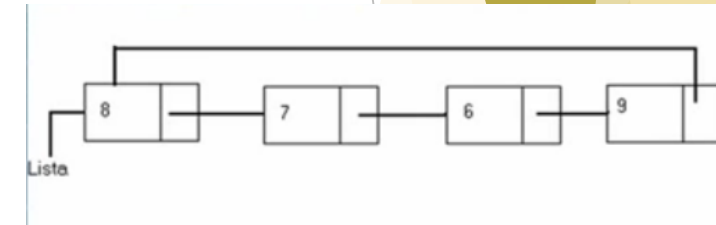
Los símbolos se guardan en la tabla con su nombre y una serie de atributos opcionales que dependen del lenguaje y objetivos del procesador. Entre ellos se encuentra:

- **Nombre de identificador.**
- **Dirección a partir de la cual se almacenará la variable en tiempo de ejecución , dirección a partir de la cual se colocará el código en caso de funciones.**
- **Tipo del identificador. si es una función, el tipo del resultado.**
- **Tipo de los parámetros de las funciones o procedimientos.**

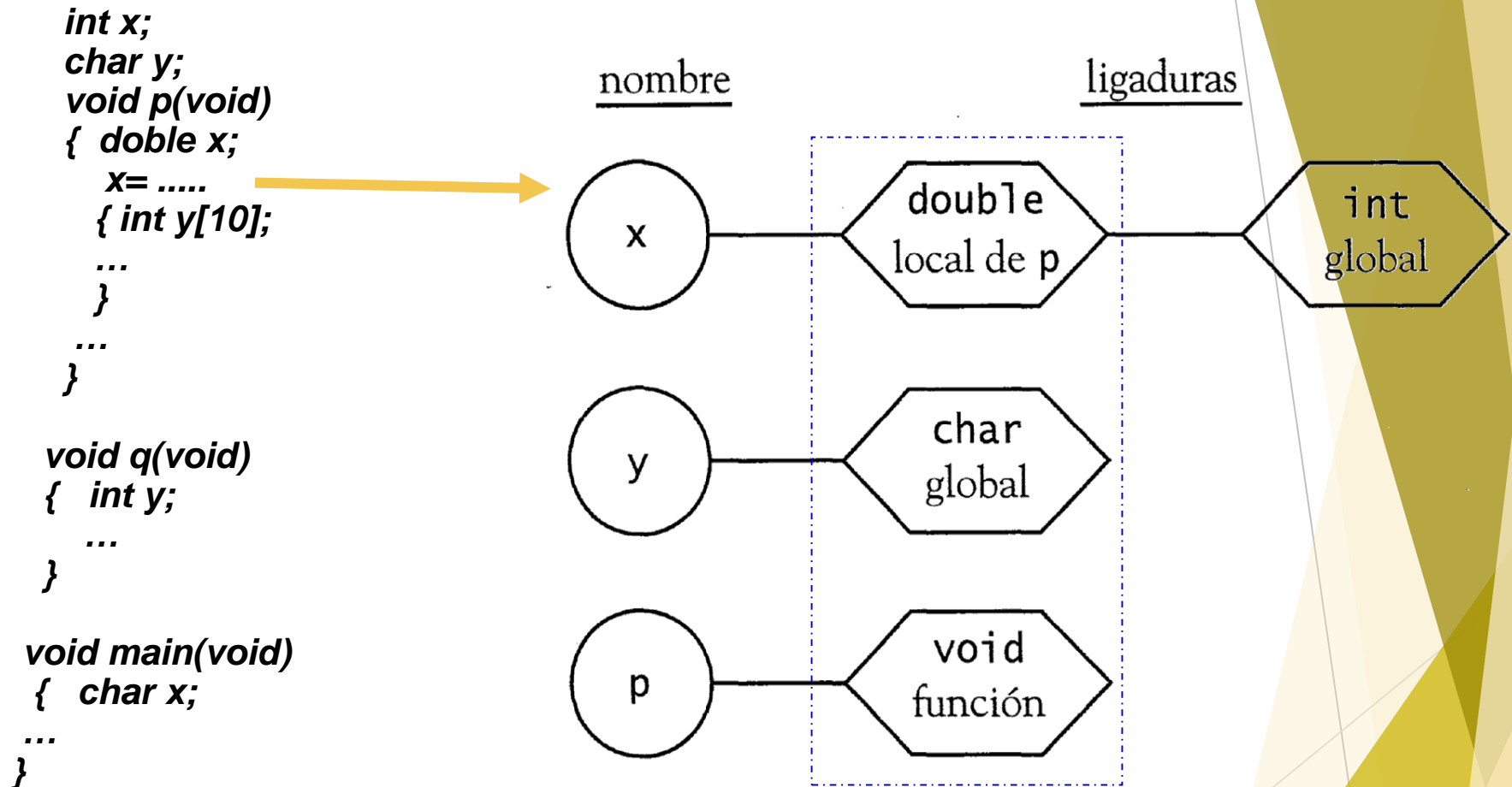


```
int a;  
int b = 5;  
int c = 8;  
  
a = b + c;
```

Nombre	Ambito	Tipo	Dato
a	Global	Var	Int
b	Global	Var	Int
c	Global	Var	Int



## Tabla de Símbolos (2)



La TS es un conjunto de nombres, cada uno de los cuales **tiene una pila** de declaraciones asociados. La declaración en la parte superior de la pila es aquella cuyo alcance está actualmente activo

# Operaciones con la Tabla de Símbolos : Inserción, búsqueda

La TS Se utiliza para **verificación estática de tipos**.

## Operaciones con la tabla de Símbolos:

### Inserción ( cuando se declaran variables)

- 1) En operaciones de inserción se detectan identificadores previamente declarados **en un mismo bloque**, emitiéndose mensaje de error.  
*multiple declaration for 's' si s ya estaba en el bloque*

### Búsqueda ( cuando se hacen referencias a variables)

- 2) En operaciones de búsqueda se detectan identificadores no declarados previamente emitiéndose el mensaje de error

Ejemplo: *Undefined símbolo 'x,' si x es una variable que desea usarse pero no se declaró.*

## Operaciones con la Tabla de Símbolos: Set, Reset

- A la entrada de un bloque o función todas las declaraciones se procesan y se agregan las vinculaciones correspondientes a las TS ( **Operación Set** ) .
- A la salida de un bloque o función se eliminan todas las ligaduras proporcionadas por las declaraciones, restaurando cualquier vínculo anterior que pudiera haber existido. ( **Operación Reset** )



# Ejemplo de funcionamiento de la TS en lenguaje C (1)

La tabla de símbolos cambia conforme se avanza en la traducción, reflejando las inserciones o eliminaciones de ligaduras dentro del programa que se está traduciendo.

En ella se conserva información del **Alcance de Ligaduras**

```
1)  int x;  
2)  char y;  
3)  float func1(int n)  
4)  { float x=10.50;  
5)    x*=2;  
6)    {  
7)      .....  
8)    int y[10];  
9)    }  
10)  ...  
11)  }  
12)  void func2(void)  
13)  { int y;  
14)  ...  
15)  }  
16)  void main(void)  
17)  { char x;  
18)  ...  
19)  }
```

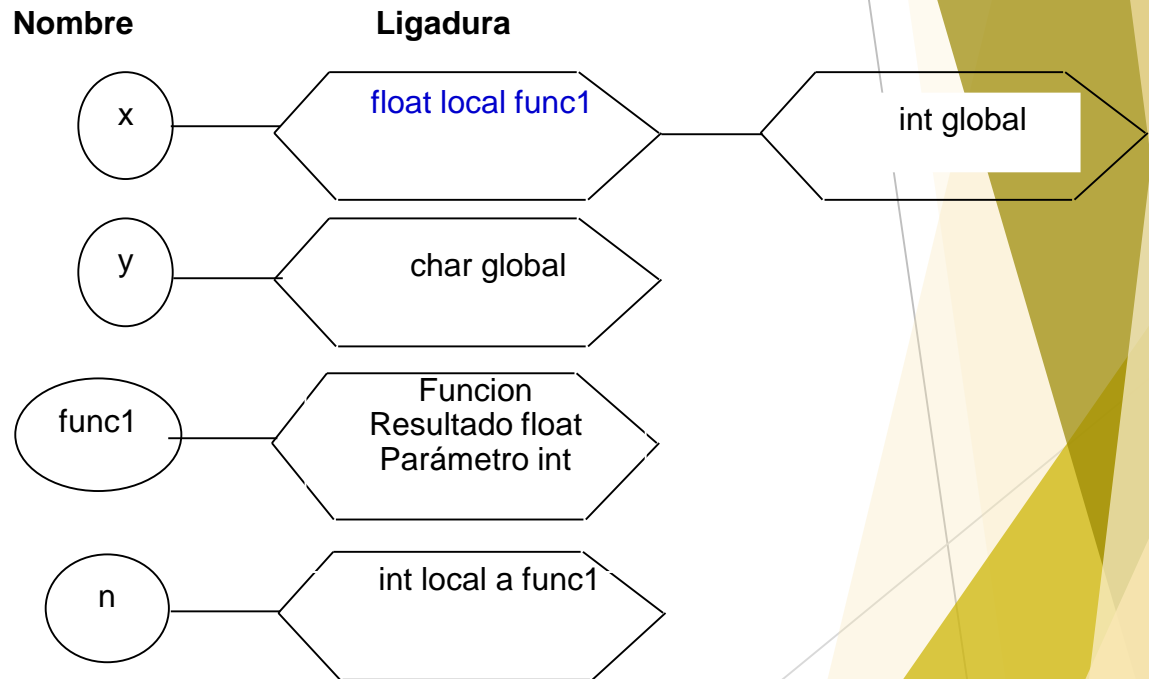


Tabla de Símbolos en la linea 4

## Ejemplo de funcionamiento de la TS en lenguaje C (2)

```
1)  int x;  
2)  char y;  
3)  float func1(int n)  
4)  { float x=10.50;  
5)    x*=2;  
6)    {  
7)      .....  
8)    int y[10]; ←  
9)    }  
10)  ...  
11)  }  
12) void func2(void)  
13) { int y;  
14)  ...  
15) }  
16) void main(void)  
17) { char x;  
18)  ...  
19) }
```

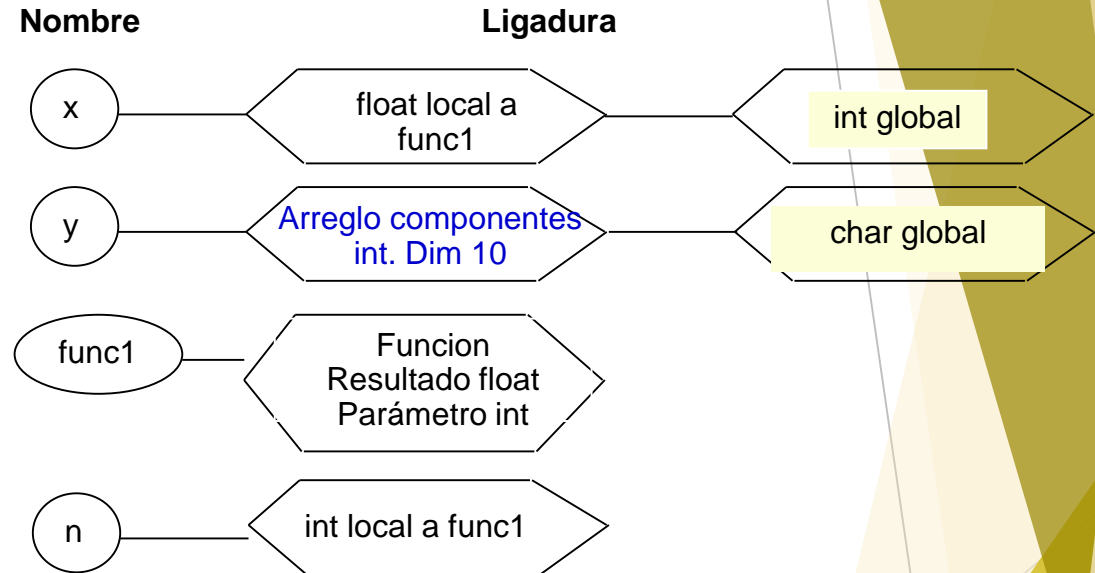


Tabla de símbolos en la línea 8

# Ejemplo de funcionamiento de la TS en lenguaje C (3)

```
1)  int x;  
2)  char y;  
3)  float func1(int n)  
4)  { float x=10.50;  
5)    x*=2;  
6)    {  
7)      .....  
8)    int y[10];  
9)    }  
10)  ...  
11)  }  
12)  void func2(void)  
13)  { int y;  
14)    ...  
15)  }  
16)  void main(void)  
17)  { char x;  
18)    ...  
19)  }
```

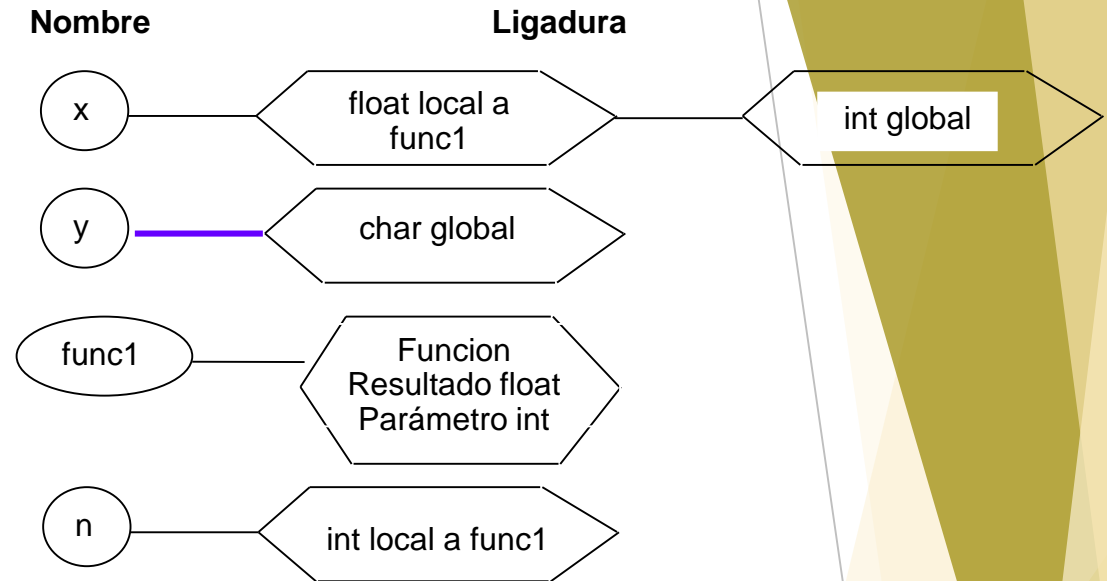


Tabla de símbolos en la línea 10

# Ejemplo de funcionamiento de la TS en lenguaje C (4)

```
1) int x;  
2) char y;  
3) float func1(int n)  
4) { float x=10.50;  
5)   x*=2;  
6)   {  
7)   .....  
8)   int y[10];  
9)   }  
10)  ...  
11) } ←  
12) void func2(void)  
13) { int y;  
14)  ...  
15) }  
16) void main(void)  
17) { char x;  
18)  ...  
19) }
```

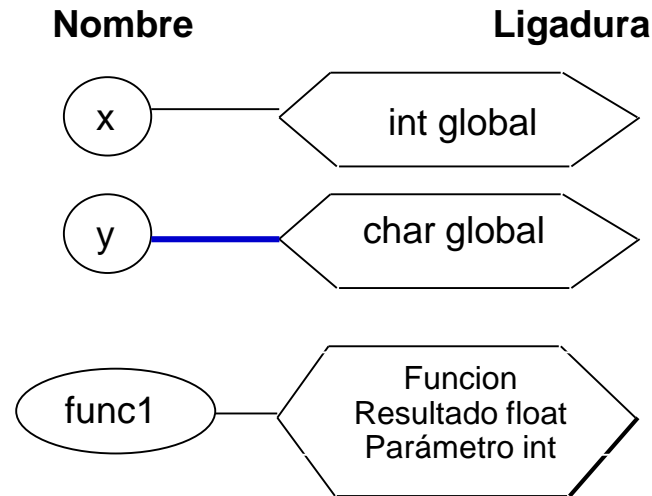


Tabla de símbolos en la línea 11

# Ejemplo de funcionamiento de la TS en lenguaje C (5)

```
1) int x;  
2) char y;  
3) float func1(int n)  
4) { float x=10.50;  
5)   x*=2;  
6)   {  
7)     .....  
8)   int y[10];  
9)   }  
10)  ...  
11) }  
12) void func2(void)  
13) { int y; ←  
14)  ...  
15) }  
16) void main(void)  
17) { char x;  
18)  ...  
19) }
```

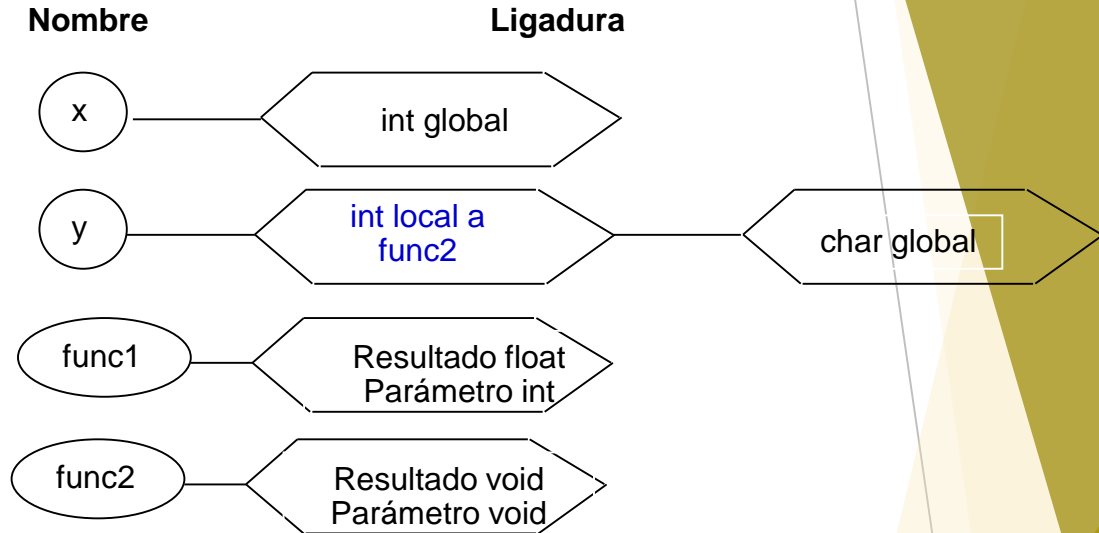


Tabla de símbolos en la línea 13

# Ejemplo de funcionamiento de la TS en lenguaje C (6)

```
1)  int x;  
2)  char y;  
3)  float func1(int n)  
4)  { float x=10.50;  
5)    x*=2;  
6)    {  
7)      .....  
8)    int y[10];  
9)      }  
10)   ...  
11)   }  
12)  void func2(void)  
13)  { int y;  
14)    ...  
15)  } ←  
16)  void main(void)  
17)  { char x;  
18)    ...  
19)  }
```

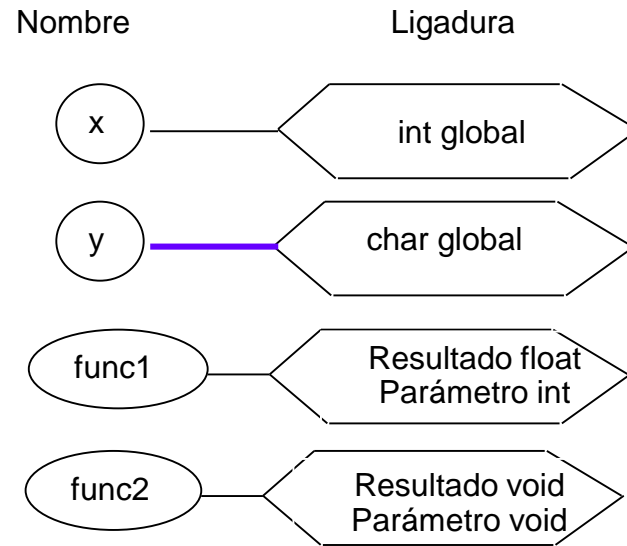


Tabla de símbolos cuando termina la línea 15

# Ejemplo de funcionamiento de la TS en lenguaje C (7)

```
1) int x;  
2) char y;  
3) float func1(int n)  
4) { float x=10.50;  
5)   x*=2;  
6)   {  
7)     .....  
8)   int y[10];  
9)   }  
10)  ...  
11)  }  
12) void func2(void)  
13) { int y;  
14)  ...  
15) }  
16) void main(void)  
17) { char x; ←  
18)  ...  
19) }
```

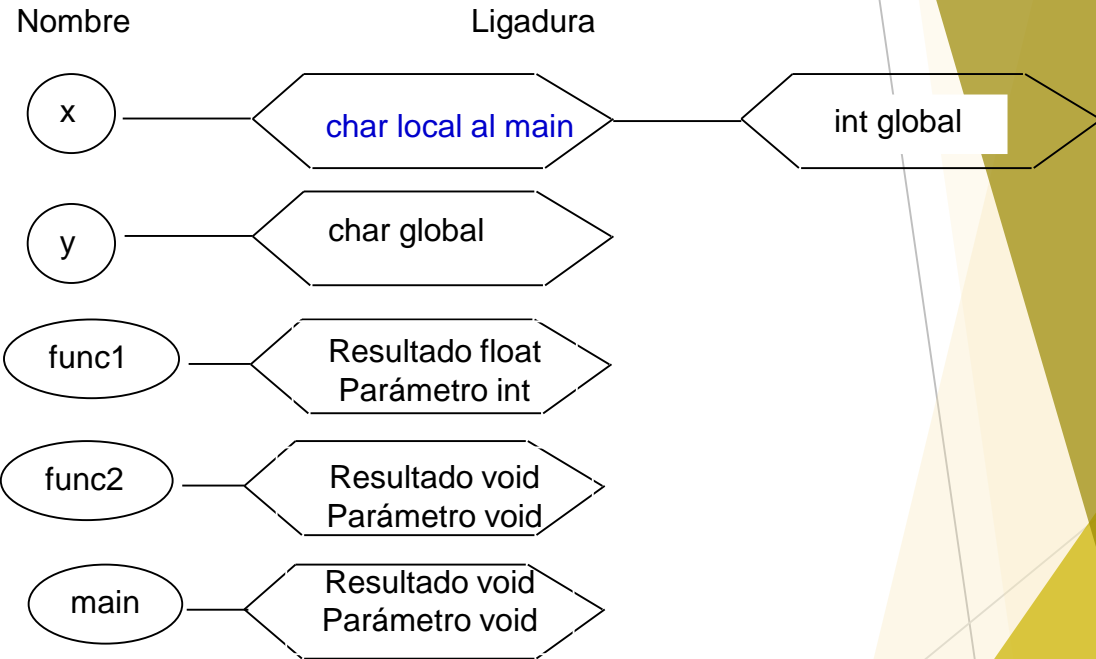


Tabla de símbolos en la línea 17

# Manejo de la TS en PILA

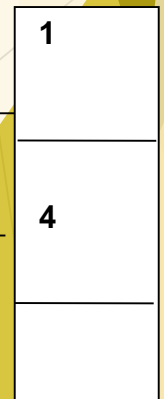
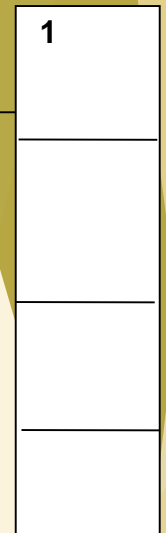
```
int x; /1****/  
char y;  
float func1(int n) /2****/  
{ float x=10.50;  
  x*=2;  
  { .....  
    int y[10];  
    char m }  
    .....  
  }  
void func2(void)  
{ int y;  
  ...  
}  
void main(void)  
{ char x;  
  ...  
}
```

Tabla de Símbolos

Posición	Nombre	Atributos
1	x	int

Posición	Nombre	Atributos
1	x	int
2	y	char
3	func1	float
4	n	int

Pila Auxiliar





# Manejo de la TS en PILA

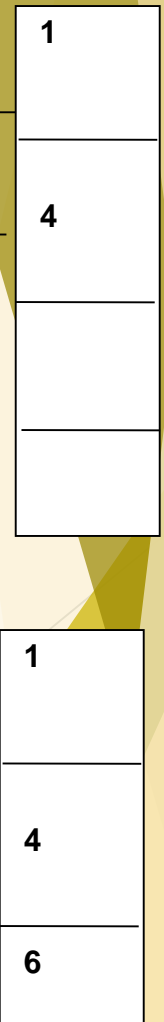
```
int x;  
char y;  
float func1(int n)  
{ float x=10.50; /1****/  
  x*=2;  
  { .....  
    int y[10]; /2****/  
    char m  
  }  
  .....  
}  
void func2(void)  
{ int y;  
  ...  
}  
void main(void)  
{ char x;  
  ...  
}
```

Tabla de Símbolos

Posición	Nombre	Atributos
1	x	int
2	y	char
3	func1	float
4	n	int
5	x	float

Posición	Nombre	Atributos
1	x	int
2	y	char
3	func1	float
4	n	int
5	x	float
6	y	arreglo

Pila Auxiliar



# Manejo de la TS en PILA

```
int x;  
char y;  
float func1(int n)  
{ float x=10.50;  
  x*=2;  
  { .....  
    int y[10];  
    char m  
    } /1*****/  
    .....  
  }  
void func2(void)  
{ int y;  
  ...  
}  
void main(void)  
{ char x; /2****/  
  ...  
}
```

Tabla de Símbolos

Posición	Nombre	Atributos
1	x	int
2	y	char
3	func1	float
4	n	int
5	x	float

Posición	Nombre	Atributos
1	x	int
2	y	char
3	func1	float
4	func2	void
5	Main	void
6	x	char

Pila Auxiliar

