Funciones

- El tipo que retorna puede ser `void` o bien cualquier tipo completo, excepto arreglo o función, pero si puede devolver un puntero a cualquiera de estos.
- Si al declararla la lista de parámetros contiene solo `void` implica que no lleva ningún parámetro
- Si al declararla la lista de parámetros está vacía, significa que no se conoce cuales son (y no se controlan).
  - Esto fue eliminado en C23, pasa a ser igual que en C++, es decir, sin parámetros es equivalente a poner `void`.
- Si se llega al final de la función, que no devuelve `void`, sin un `return` y la función que invocó usa el resultado, el comportamiento es no definido.

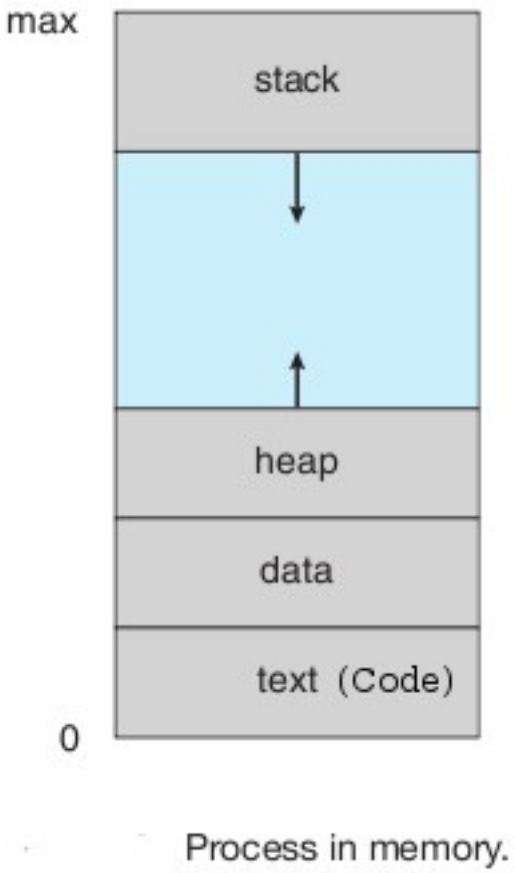
Funciones

- Similitudes con arreglos, en cuanto a los punteros:
  - El identificador de una función se toma como un puntero a ella, por eso no uso & antes del nombre de una función para tomar su dirección.
    - 6.3.2.1 Lvalues, arrays, and function designators- 4. A function designator is an expression that has function type ... "function returning type" is converted to an expression that has type "pointer to function returning type".
  - De modo similar puedo declarar un parámetro función sin poner un puntero:
    - 6.7.7.4 Function declarators - 7. A declaration of a parameter as "function returning type" shall be adjusted to "pointer to function returning type".
  - Los paréntesis son el operador necesario para invocar a la función.
  - No está definido el orden de evaluación de los parámetros.

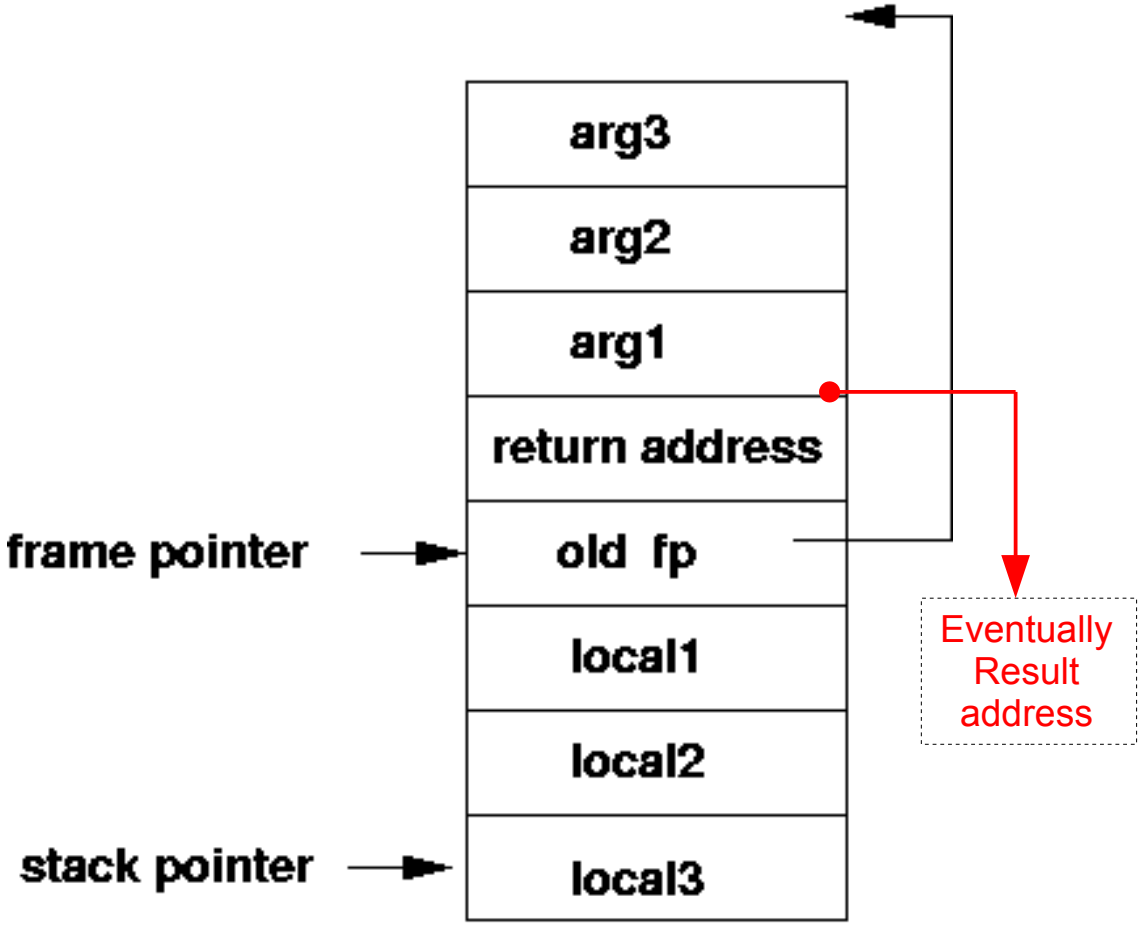
Funciones - ejemplos

```
void quicksort (float vector[], int dim);
void shellsort (float vector[], int dim);
void (*ordenar) (float vector[], int dim);
// en lo siguiente no uso &quicksort
ordenar = quicksort; // o sino ordenar = shellsort;
//invoco como si fuese función.
ordenar (mivector, 100);

//Lo explícito es
void ordenar
(float vector[], int dim, int (*criterio)(float, float));

//Pero puedo escribirlo como
void ordenar
(float vector[], int dim, int criterio(float, float));
```

Manejo de la memoria (Calling convention)



Process in memory.



Estructuras

- Una estructura, conocida en otros lenguajes como registro, permite agrupar en una unidad un conjunto, posiblemente heterogéneo, de variables relacionadas.
- Para poder definir una variable de tipo estructura primero debemos declarar el tipo completo, con sus miembros. Luego podremos definir la variable de este “tipo estructura”

```
struct empleado {//declaro el tipo
        int legajo;
        char nombre[30];
};

struct empleado e1, e2; //defino variables
```

Declaración y definición

- Puedo declarar la estructura y definir las variables en un solo paso

```
struct punto {
        double x;
        double y;
} pto1, pto2;
```

- Puedo definir una variable de una estructura anónima

```
struct { //anónima (falta nombre del struct)
        int i, j; //posible pero
desaconsejado
} anon1;
```

Inicialización

- Puedo incializar valores al declarar la variable

```
struct punto pto3 = { 1.0, 2.0};
struct empleado e3 = { 525, "Pablo"};
```

- Puedo hacer todo junto (declarar e inicializar)

```
struct ejemplo {
        int a;
        int b;
} vejemplo = {1, 5};
```

- A partir de C99 se puede incializar en cualquier orden si “designo” el campo al que asigno.

```
struct punto pto = {.y = 2.0, .x = 1.0};
/* idéntico a pto = {1.0, 2.0} */
```

Uso

- Para acceder a un miembro de una estructura uso el operador . (punto) si lo que tengo es una variable

```
e1.legajo = 123;
strcpy(e1.nombre, "Juan");
```

- Puedo asignar para copiar TODOS los campos

```
e2 = e1;
```

- Puedo pasar una estructura como parámetro o devolverla como resultado de una función

Punteros

- Si declaro un puntero a una estructura y quiero acceder a un campo, por tema de precedencias debo usar paréntesis

```
struct punto *p;
p = &pto;
(*p).x = 5.0;
```

- Como esto es engorroso se definió el operador -> que es lo que se acostumbra utilizar.

```
p->x = 5.0; //equivalente a (*p).x = 5.0;
```

typedef y autoreferencias

- Se puede declarar una estructura dentro de otra y es posible tener un miembro de tipo puntero a la estructura que lo contiene

```
struct enlazada {
    int clave;
    struct {
        double medida;
        char *descrip;
    } datos;
    struct enlazada *siguiente;
};
typedef struct enlazada *ptr2en;
```

```
struct enlazada enl01, enl02;
enl01.clave = 1;
enl01.datos.medida = 25.6;
enl01.siguiente = &enl02;

ptr2en primero = &enl01;
primero->siguiente->clave = 2;
/*equivale a: enl02.clave = 2 */

struct enlazada enl03 = {1, {3.0}};
/*descrip y siguiente puestos en
NULL */
```

Modo de almacenamiento

- Así como un vector calcula un desplazamiento para cada elemento, en una estructura hay un desplazamiento para cada miembro, pero este no se puede calcular (el compilador arma una tabla)
- El sizeof de una estructura es mayor o igual a la suma de los sizeof de sus miembros. Esto se debe a las conveniencias de alineación, pudiendo dejar espacios (en inglés gap o filler) entre campo y campo o al final de la estructura (para que en un arreglo la siguiente estructura esté bien alineada).
- En stddef.h se define la macro offsetof(type, member-designator) que permite conocer el desplazamiento de un miembro en particular.

Ejemplo de como almacena

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

struct id {
    int i;
    double d;
};
struct di {
    double d;
    int i;
};

int main(void)
{
    struct id id1, id2 = {3};
    struct di di1;

    printf ("id1.i = %d # id1.d = %g\n", id1.i, id1.d);
    printf ("id2.i = %d # id2.d = %g\n\n", id2.i, id2.d);

    printf ("sizeof(id) = %ld # sizeof(id.i) = %ld # sizeof(id.d) = %ld\n"
           , sizeof(struct id), sizeof(id1.i), sizeof(id1.d));
    printf ("offsetof(id.i) = %ld # offsetof(id.d) = %ld\n\n"
           , offsetof(struct id, i), offsetof(struct id, d));

    printf ("sizeof(di) = %ld # sizeof(di.d) = %ld # sizeof(di.i) = %ld\n"
            , sizeof(struct di), sizeof(di1.d), sizeof(di1.i));
    printf ("offsetof(di.d) = %ld # offsetof(di.i) = %ld\n"
            , offsetof(struct di, d), offsetof(struct di, i));

    return EXIT_SUCCESS;
}
```

Salida

```
id1.i = -202348087 # id1.d = 4.94066e-322
id2.i = 3 # id2.d = 0

sizeof(id) = 16 # sizeof(id.i) = 4 # sizeof(id.d) = 8
offsetof(id.i) = 0 # offsetof(id.d) = 8

sizeof(di) = 16 # sizeof(di.d) = 8 # sizeof(di.i) = 4
offsetof(di.d) = 0 # offsetof(di.i) = 8
```

Uniones

- Al declarar una estructura se asigna memoria para todos sus miembros, uno detrás de otro, en el orden en que fueron declarados, con posibles espacios intermedios por cuestiones de alineación
- Una unión es similar pero asigna espacio para el miembro más grande ya que todos se alinean al principio. Es decir, en la unión solo uno de los miembros puede estar almacenado en un momento dado.
- Salvo por el modo de almacenaje, en todos los demás aspectos se comporta igual que una estructura

Uniones

```
enum  modos_venta
{XUNIDAD, XPESO, OTROS};

struct pedido {
    int tipo_pedido;
    union {
        int unidades;
        double peso;
        char descrip[20];
    } dato;
} vped;
```

```
/* Supongamos que ya leímos y
cargamos vped.tipo_pedido */

switch (vped.tipo_pedido) {
case XUNIDAD:
    scanf("%d"
          , &vped.dato.unidades);
    break;
case XPESO:
    scanf("%lf"
          , &vped.dato.peso);
    break;
case OTROS:
    fgets(vped.dato.descrip, 20
          , stdin);
}
```

Licencia