REPASO CLASE 3

Cada uno de los atributos de una entidad van a tener valores. Al momento de usar esa entidad los atributos tienen que tener un valor determinado.

El momento en el que a un atributo de una entidad se le asigna un valor se llama momento de ligadura o momento de binding.Esa ligadura puede darse:

- En momentos no es en ejecución → ligadura estática
- En ejecución → ligadura dinámica

El concepto de ligadura también está asociado al concepto de estabilidad, porque puede ser que un atributo tenga un valor pero que no sea fijo hasta el final de la ejecución de programa.

Los distintos momentos de binding son:

- Definición del lenguaje → Estática
- Implementación → Estática
- Compilación → Estática
- Ejecución → Dinámica

Todas las ligaduras que se definan antes de ejecución son consideradas como estáticas.

La entidad Variable es una abstracción de una celda de memoria y tiene los siguientes atributos asociados:

- Nombre: identificador. Los lenguajes tiene reglas para los identificadores que varian entre si (si puede empezar con _ , con número, etc)
- Alcance: Conjunto de sentencias donde esa variable es conocida (donde se puede usar). Cuando una unidad usa una variable que no esta definida localmente en esta, los lenguajes aplican dos cadenas para resolver qué variable se está referenciando
    - Estático: La que utilizan la mayoría de lenguajes. Con solo ver la estructura del programa puedo definir el alcance (se van fijando que unidades contienen a otras)
    - Dinámico: Tengo que simular una ejecución ya que para determinar que variable usar se fija quien llamó a la unidad que quiere usar la variable. No me importa donde está contenida, sino quien la llamó

  El alcance está asociado al nombre
    -
- Tipo: Puede asignarse de forma dinámica (python, ruby, php) o de forma estática (Java, Pascal, C)
- L-valor: Cuando aparece del lado izq. Representa la dirección donde una variable es almacenada en la memoria. Está asociada con el concepto de tiempo de vida: Cuando y cuanto tiempo esa variable se aloca en la memoria y está en la memoria

- R-valor: Cuando aparece del lado derecho. La asignación generalmente es dinámica excepto en el caso de las constantes. Tiene que ver con las inicializaciones, si no tiene inicialización por defecto entonces no puedo comenzar a usar la variable sin asignarle nada manualmente porque sino me toma basura

RESUMEN:
- Clasificación de acuerdo al alcance: global, no local, local
- Clasificación de acuerdo al tiempo de vida: Automáticas, semiautomáticas, dinámicas, estáticas.

CLASE 4 - UNIDADES DE PROGRAMA

Las **unidades** representan la abstracción de una acción, puede estar representando una sentencia o una operación.

Procedimientos y funciones

Hay distintos tipos de unidades. A las unidades que están basadas en un esquema con return son aquellas que se denominan rutinas y en general nos permiten definir definir **procedimientos** o **funciones**. Se dice que estan basadas en un esquema con return porque esas unidades, para que puedan ejecutarse este código tienen que ser llamadas y una vez que terminan devuelven el control.
Tanto los procedimientos como las funciones van a contener un conjunto de sentencias que se van a ejecutar en el momento que se llaman, además ambas pueden compartir información a través de sus parámetros. La diferencia es que la función devuelve un valor, por eso se dice que la función es una abstracción de un operador, porque es como si estuviéramos definiendo un nuevo operador.
No todos los lenguajes me permiten definir procedimientos, algunos solo tienen funciones y simulan los procedimientos con funciones que devuelven void como python, c, c++

Atributos de las rutinas:

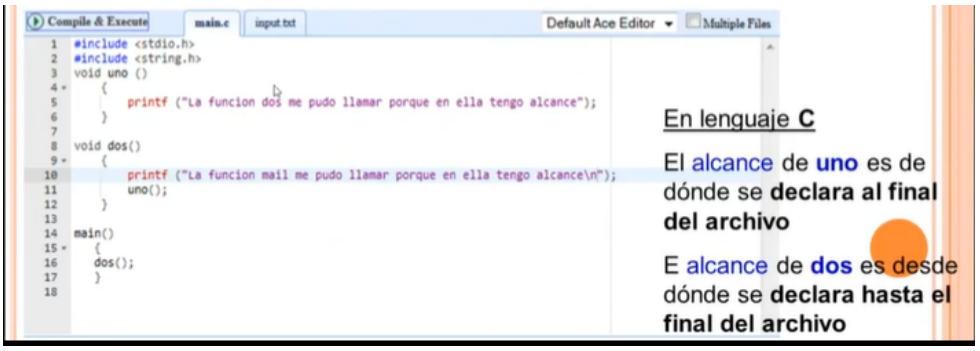
- Nombres: String de caracteres que se usa para invocar a la rutina (identificador)
    - El nombre de la rutina se introduce en su declaración
    - El nombre de la rutina es lo que se usa para invocarlas. La llamada debe estar dentro del alcance del nombre de la rutina

- Los lenguajes tienen reglas para definir los identificadores de rutinas que en general coinciden con las reglas de definición de identificadores de variables

```c
#include <stdio.h>

int suma (int a, int b)
{

    return a + b;
}

int main(void)
{
    /* Se llama a la función*/
    printf("%i ", suma( 7, 3));

    return 0;

}
```

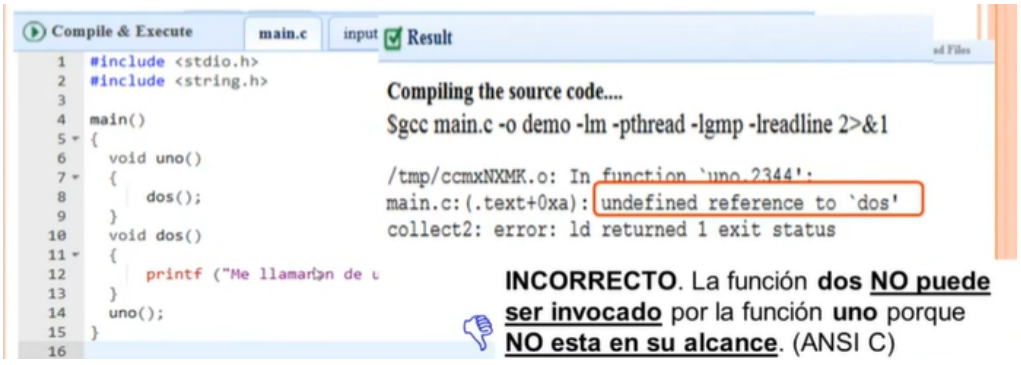
primero defino suma y después la invoco

- Alcance: Rango de instrucciones donde se conoce su nombre
  - Se extiende desde el punto de su declaración hasta algún constructor de cierre
  - Según el lenguaje puede ser estático o dinámico
  - Puede ser estático o dinámico al igual que en las variables, pero en general es desde donde se define la rutina hasta abajo



En el caso de C, su programa puede estar contenido en varios archivos, si yo declaro una función, esa función va a tener alcance desde que se declara hasta abajo (A MENOS QUE LA ENMASCARE)

ACTIVACIÓN: La llamada puede estar solo dentro del alcance de la rutina



Puede ser que algunas veces no de error y solo de un warning (Ver [Composición de una rutina]())

- Tipo: El encabezado de la rutina define el tipo de los parámetros y el tipo del valor de retorno (si lo hay)
    - Generalmente está dado por la signatura: permite especificar el tipo de una rutina. Una rutina `fun` que tiene como entrada parámetros de tipo T1, T2, Tn y devuelve un valor de tipo R, puede especificarse con la siguiente signatura:

      `fun: T1 x T2 x … Tn → R`
    - Un llamada a una rutina es correcto si está de acuerdo al tipo de la rutina
    - La conformidad requiere la correspondencia de tipos entre parámetros formales y reales

  Ejemplo:

```
/* sum es una funcion que suma los n primeros naturales,
1 + 2 + + n; suponemos que el parametro n es positivo */
int sum(int n)
{
        int i, s;
        s = 0;
        for (i = 1; i <= n; ++i)
        s+= i;
        return s;
}
```

El tipo de la función sería:

sum: enteros ⟶ *enteros*

***sum** es una rutina con un parámetro entero que devuelve un entero*

- L-value: Es el lugar de memoria en el que se almacena el cuerpo de la rutina. Donde se coloca la rutina en la memoria
- R-value: La llamada a la rutina causa la ejecución de su código, eso constituye su r-value. Esos identificadores que aparecen en el programa que se refieren a distintas invocaciones.
    - Puede ser estático: SI tengo una función que es llamada varias veces, cuando se compile la referencia a esa función se pone como una referencia a donde va a estar el código de la función. Generalmente es estático
    - Puede ser dinámico: A veces hay variables que representan nombres de funciones, esa variable a lo largo del programa puede tomar destinos nombres de funciones. Ejemplo:

main.c

```c
/*****************************************************************************
                            Online C Compiler.
                Code, Compile, Run and Debug C program online.
Write your code in this editor and press "Run" button to compile and execute it.
*****************************************************************************/

#include <stdio.h>
void uno( int valor)
  { if (valor == 0) printf ("Me invocaron con el identificador uno\n");
        else printf ("Me invocaron a través de un puntero a función\n");
  }

int main()
{
    int y;
    void (*punteroAFuncion)();

    printf("Probando R-VALUE funciones\n");

   /* Pureba de llamada a función R-VALUE estático*/
    y= 0;
    uno(y);

    /* Pureba de llamada a función R-VALUE dinámico*/
    y= 1;
    punteroAFuncion = &uno;
    punteroAFuncion(y);

    return 0;
}

```

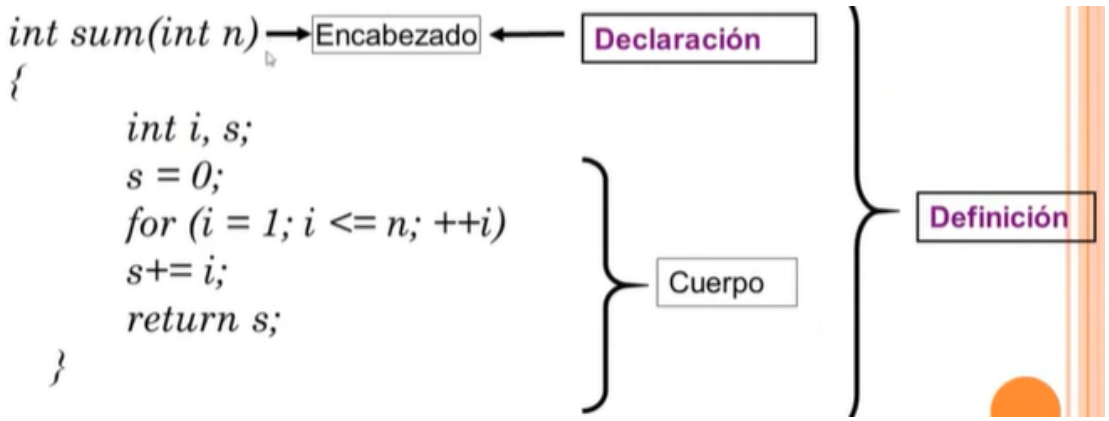
Ejemplo en C de variables rutinas (binding dinámico). Puedo definir variables que son punteros a funciones. En este caso, la variable `punteroAFuncion` es una variable que apuntará a una función que no recibe parámetros y que va a devolver un `void`. Cuando hago `punteroAFuncion` = &uno; le estoy asignando la dirección de la función `uno` y despues la invoco a través de `punteroAFuncion`. Se dice que el r-value es dinámico porque a la variable de tipo puntero se le está asignando una dirección a una dirección de memoria pero se le está asignando de forma dinámica

Registro de activación:

Cada vez que se ejecuta una rutina se ejecuta una rutina se arma un registro de activación que va a contener toda la información que pertenece a la rutina.

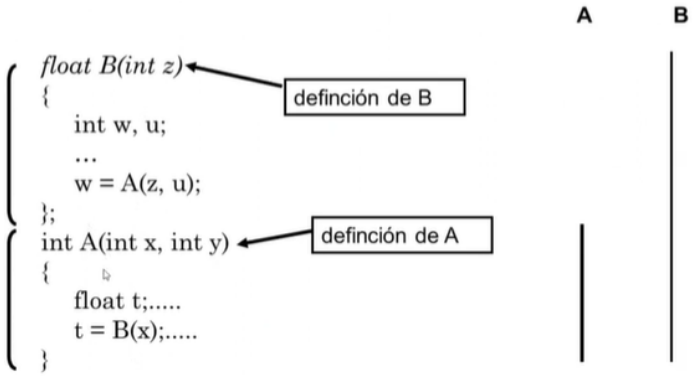
Composición de una rutina:

- Encabezado/Definición: También se lo llama declaración. Corresponde a la linea en donde se le da el nombre a la rutina y se establecen los datos que vana compartir a través de los parámetros y en el caso que sea de tipo función se establec e cual es el valor de retorno
- Cuerpo: El conjunto de sentencias



Encabezado + Cuerpo = Definición de la rutina

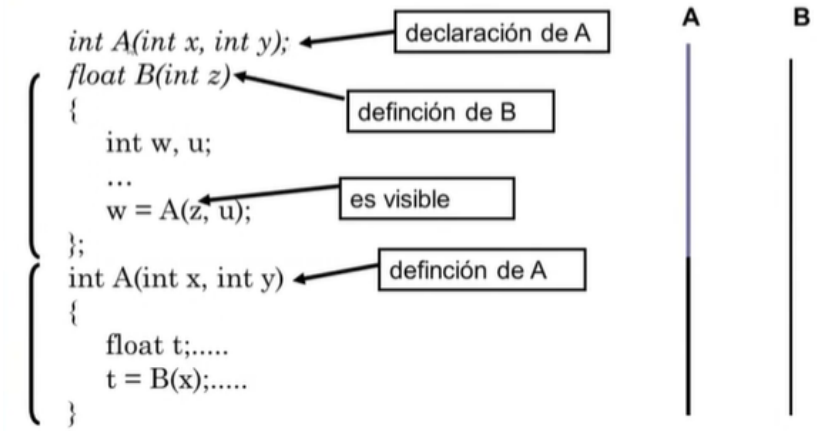
Posible problema:



Aca voy a tener conflicto porque B llama a A en una zona en la cual A no tiene alcance. SI intento dar vuelta las rutinas, poniendo a A primero, el problema persistirá porque ahora sería A quien estaría llamando a B fuera de su alcance.

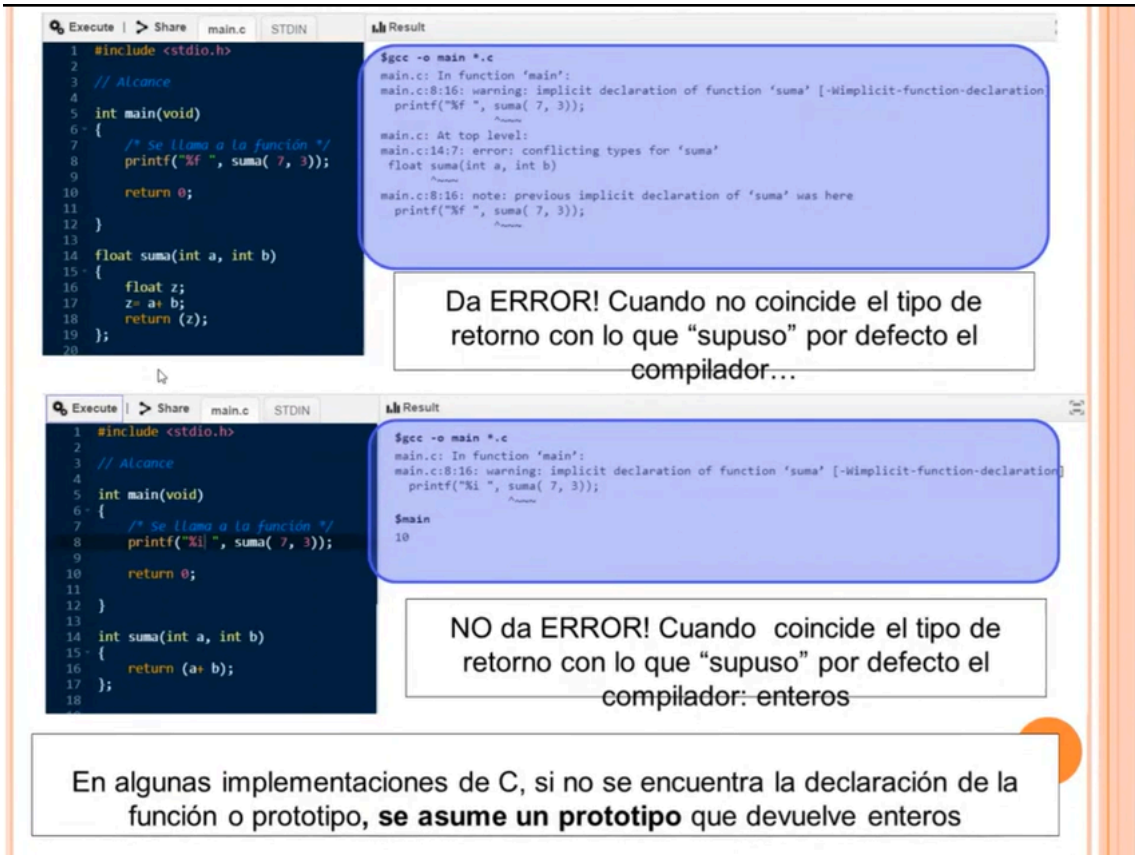
Posible solución:

Muchos lenguajes permiten poner solo la declaración de una de las rutinas para poder extenderle el alcance. Esto solo se puede en lenguajes que distinguen entre la declaración y la definición de una rutina

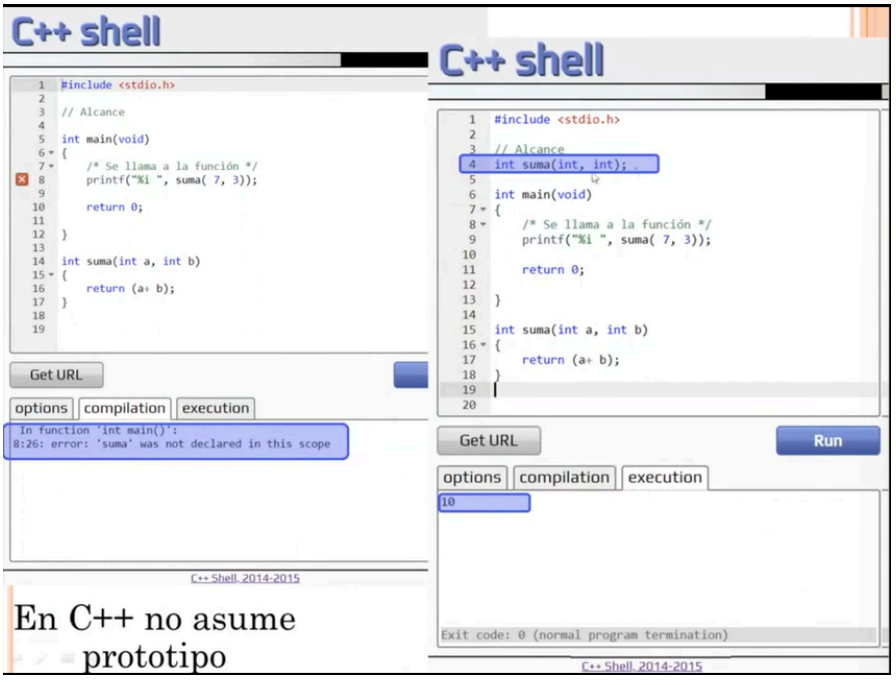


Ejemplo en C

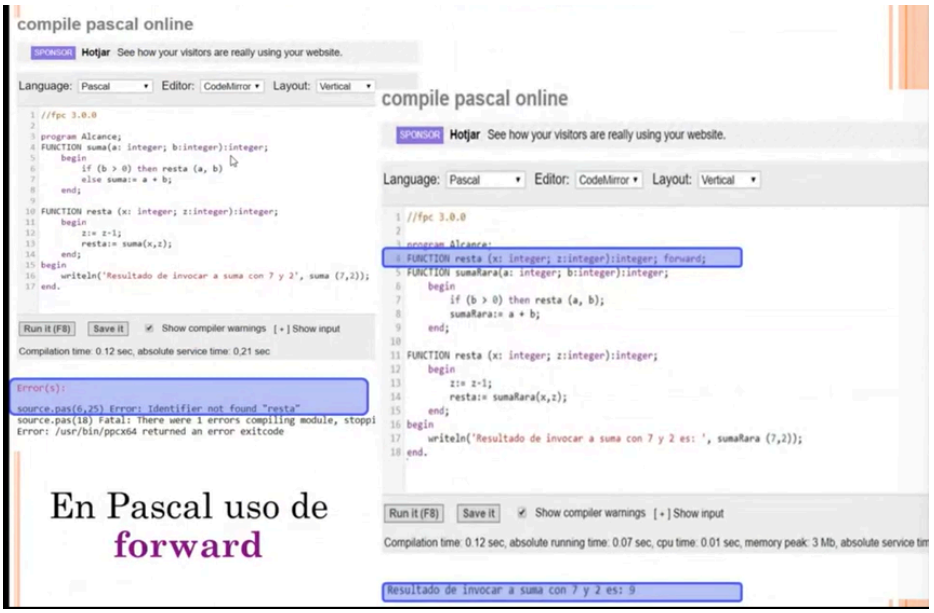
Importante, hay versiones de C, en donde, si no puse la definición arriba para extender el alcance me lo pone automaticamente pero como una función que devuelve un dato de tipo entero



En la primera imagen da error porque 'suma' ya fue declarada como flotante, entonces no puede darle ese valor por defecto.



Ejemplo en C++, donde no se asume ningún prototipo



Ejemplo en Pascal donde extiendo el alcance utilizando la palabra forward en el encabezado.

Comunicación entre rutinas:

Van a trabajar con datos que pueden ser locales o compartidos. hay dos maneras de compartir esos datos:

- Ambiente no local: Si referencia a una variable que no fue definida localmente ni fue recibida por parámetro. Para saber qué variable es tengo que seguir la cadena estática/dinámica. No es muy recomendable ya que para poder saber a quién me estoy refiriendo tengo que hacer una búsqueda mayor y corro riesgos de manipular información que no debía
- Parámetros: Es la mejor opción, ya que determino bien, a través de la lista de parámetros que datos voy a compartir, de esta forma no corro riesgo de modificar información que no debía. Además a la hora de hacer modificaciones en mi código, tan solo con mirar la rutina puedo darme cuenta que datos estan siendo utilizados en el procedimiento/función (mayor legibilidad y modificabilidad).
  Hay dos tipos de parámetros:
  - Parámetros formales: los que aparecen en la definición (encabezado) de la rutina
  - Parámetros reales: los que aparecen en la invocación de la rutina (dato o rutina)

<u>Ligadura entre parámetros formales y reales:</u>

- Método posicional: se lugan uno a uno, relacionando el primer real con el primer formal, el segundo real con el segundo formal, etc

routine S (F1,F2,……,Fn) **Definición**
call S (A1, A2,….. An) **Llamado**
Fi se liga a Ai para i de 1 a n.
deben **conocerse** las **posiciones**

Variante: combinación con valores por defecto
C++: int *distancia* (**int a = 0**, **int b = 0**)
distancia() ⟶ *distancia (0,0)*
distancia*(10)* ⟶ *distancia (10,0)*

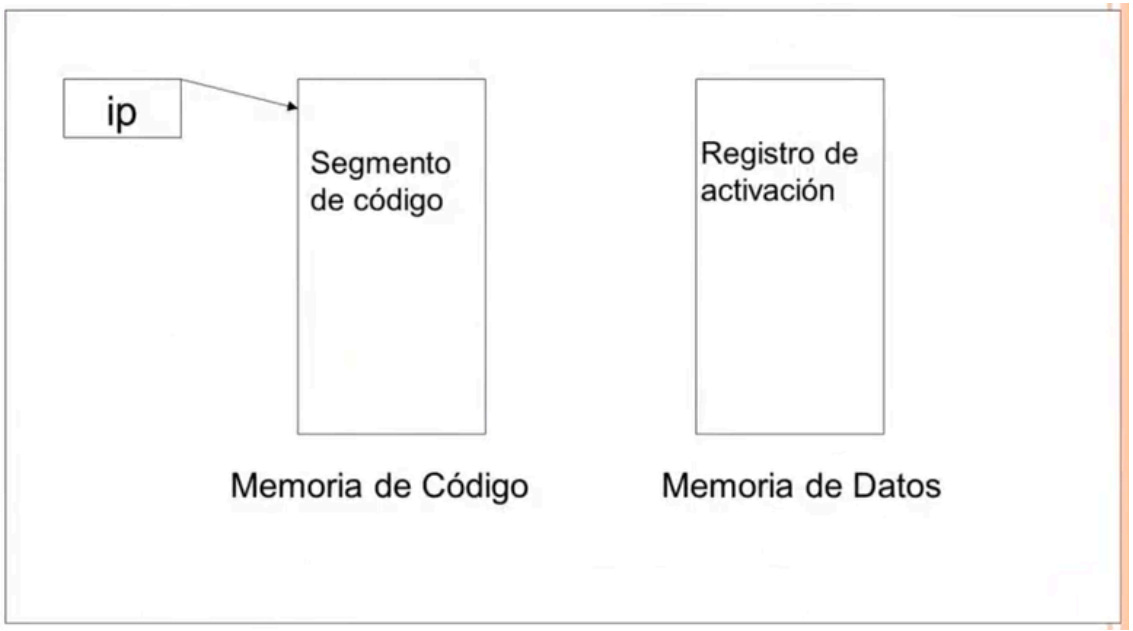
Representación en ejecución:

Qué es lo que pasa en la memoria cada vez que nosotros mandamos a ejecutar un programa.

- La definición de la rutina especifica un proceso de cómputo
- Cuando se invoca una rutina se ejecuta una instancia del proceso con los particulares valores de los parámetros
- **instancia de unidad**: representación de la rutina en ejecución. Cada instancia de unidad va a formar su propio registro de activación (Ver registro de activación)

Zonas de la memoria:

- Segmento de código: Instrucciones de la unidad se almacenan en la memoria de instrucción C. Contenido fijo
- Registro de activación: Datos locales de la unidad se almacenan en la memoria de datos D. Contenido cambiante



ip: puntero a la sentencia que estoy ejecutando

Simplesem: Lenguaje/ procesador abstracto que se utiliza por la cátedra para poder ver que pasa con las sentencias de alto nivel que nosotros utilizo en los programas y cómo se traducen a las de más bajo nivel

Utilidades de Simplesem:

- El procesador nos servirá para comprender que efecto causan las instrucciones del lenguaje al ser ejecutadas.
- Semántica intuitiva.
- Se describe la semántica del lenguaje de programación través de reglas de cada constructor del lenguaje traduciéndolo en una secuencia de instrucciones equivalentes del procesador abstracto

Memoria de Código: C(y) valor almacenado en la yésimacelda de la memoria de código. Comienza en cero
Memoria de Datos: D(y) valor almacenado en la yésimacelda de la memoria de datos. Comienza en cero y representa el l-valor, D(y) o C(y) su r-valor
Ip: puntero puntero a la instrucción instrucción que se está ejecutando.

- Se inicializa en cero. En cada ejecución se actualiza cuando se ejecuta cada instrucción.
- Direcciones de C

Ejecución:

- Obtener la instrucción actual para ser ejecutada (C[ip]): Cuando hable de “C de ip” significa que estoy trabajando y ejecutando la sentencia que esta en la parte de código, apuntada por ip
- Incrementar ip
- Ejecutar la instrucción actual

Instrucciones:

SET: setea valores en la memoria de datos

set target,source

Copia el valor representado por source en la dirección representada por target

`set 10,D[20]`

copia el valor almacenado en la posición 20 en la posición 10.

E/S: read y write permiten la comunicación con el exterior.

`set 15,read`

el valor leído se almacenara en la dirección

`15set write,D[50]`

se transfiere el valor almacenado en la posición 50 .

combinación de expresiones

`set 99, D[15]+D[33]*D[4]` expresión para modificar el valor

JUMP: bifurcación incondicional a direcciones de C

jump 47

la próxima instrucción a ejecutarse será la que estealmacenada en la dirección 47 de C

JUMPT: bifurcación condicional, bifurca si la expresión seevalúa como verdadera

jumpt 47,D[13]>D[8]

bifurca si el valor almacenado en la celda 13 es mayor que el almacenado en la celda 8

direccionamiento indirecto:

set D[10],D[20] //anda a la posicion 20 (que tiene un 5) y colocalo en lo qur esta la dirección que esta guardada en la posición 10

jump D[30]

Ip= 5 posición 5 en C

Elementos de ejecución

- **Punto de retorno**

Es una pieza cambiante de información que debe ser salvada en el registro de activación de la unidad llamada. Cuando llego al punto de invocación de la función el ip va a cambiar y va a estar apuntando a la primera sentencia de la función. Si ese ip se cambia, cuando termina de ejecutar la función, a donde retorna? → tengo que salvarlo antes

- **Ambiente de referencia**

- Ambiente local: variables locales, ligadas a los objetos almacenados en su registro de activación
- Ambiente no local: variables no locales, ligadas a objetos almacenados en los registros de activación de otras unidades

Estructuras de ejecución de los lenguajes de programación:

- Estático: Espacio fijo
    - El espacio necesario para la ejecución se deduce del código

    - Todo los requerimientos de memoria necesarios necesarios se conocen conocen antes de la ejecución
    - Cargan todo en la memoria cuando se está cargando el programa (Y QUEDA AHÍ HASTA EL FINAL DE LA EJECUCIÓN, se use o no, por ejemplo si las rutinas tienen cosas locales, esos tambien se cargan ni vien cargo en la memoria el programa)--> antes de ejecutar ya se cargó todo
    - La alocación de todas las variables explicitadas(se hace antes de ejecución) puede hacerse estáticamente
    - No puede haber recursión ya que no pueden mantener diferentes instancias.(ya carga todo y lo deja fijo)
    - Permite el uso de un solo tipo de variables que son las variables estáticas
    - Lenguajes estáticos como las versiones originales FORTRAN y COBOL

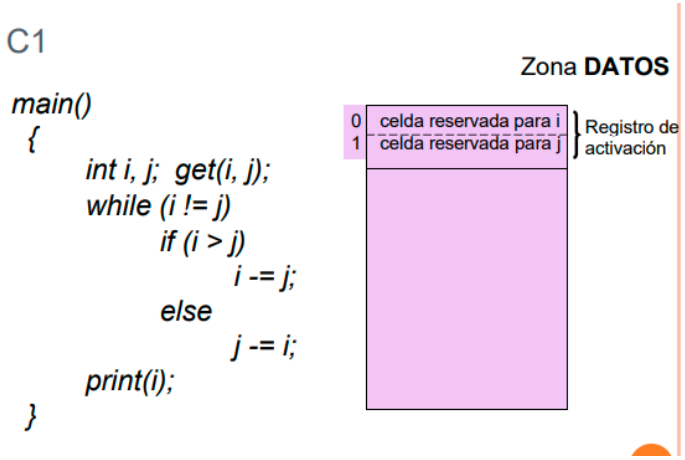
- Basado en pila:Espacio predecible
    - El espacio se deduce del código. Algol-60
    - Programas más potentes cuyos requerimientos de memoria no puede calcularse en traducción.
    - A diferencia del esquema estático, si ese programa tiene muchos procedimientos o funciones internas no va a cargar los registros de activación de esas rutinas, solo va a comenzar la ejecución con el registro de activación del programa principal.Luego, a medida que van invocando las rutinas se vana a ir cargando en la zona de datos, aquellas rutinas que se van llamando, y se van desalocando del registro de activación aquellas rutinas que terminan de ser ejecutadas (ej, si llamo 4 veces a una rutina se va a alocar 4 veces y desalocar 4 veces)
    - La memoria memoria a utilizarse es predecible predecible y sigue una disciplina last-in-first-out.
    - Las variables se alocan automáticamente y se desalocan cuando el alcance se termina
    - Se permiten variables automáticas, estáticas, semiautomáticas, dinámicas.
    - Se utiliza una estructura de pila para modelizarlo. (el último que entró es el primero en salir)
    - EJ: C

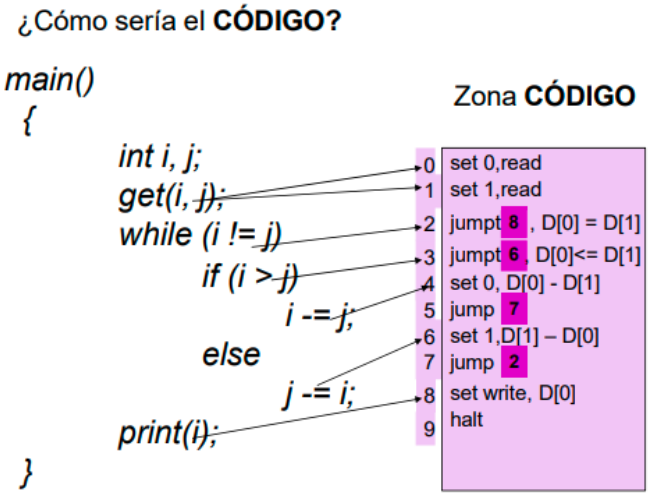
- Dinámico: Espacio impredecible
    - Lenguajes con impredecible uso de memoria.
    - Los datos son alocados dinámicamente sólo cuando se los necesita durante la ejecución.
    - No pueden modelizarse con una pila, el programador puede crear objetos de datos en cualquier punto arbitrario durante la ejecución del programa.
    - Los datos se alocan en la zona de memoria heap
    - Ruby, Php, Python

Estructura de ejecución estática:

CASO 1:

- Sentencias simples
- Tipos simples
- Sin funciones
- Datos estáticos de tamaño fijo
- un programa = una rutina main()
    - Declaraciones
    - Sentencias
- E/S: get/print



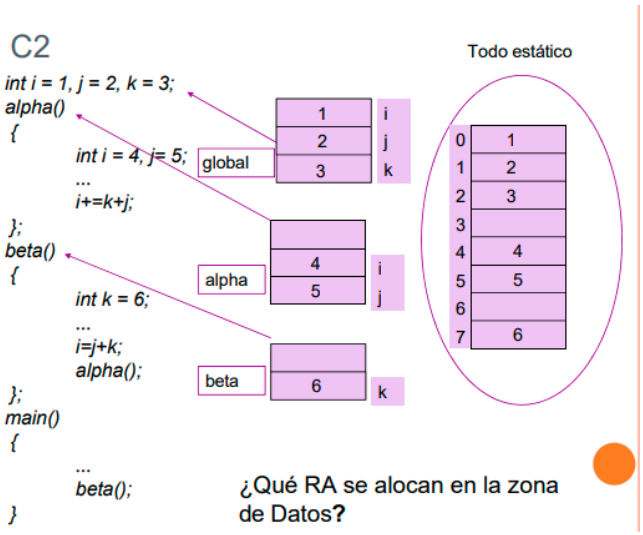


CASO 2: C1 + RUTINAS INTERNAS

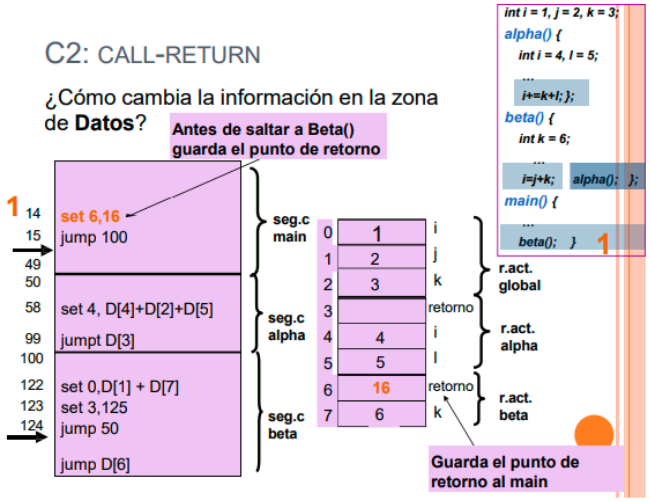
Definición de rutinas internas al main (no anidamiento)

- Programa puede tener:
    - Datos globales
    - Declaraciones de rutinas
    - Rutina principal
        - Datos locales
- Se invoca automáticamente en ejecución
- En zona de datos: SOLO datos locales y punto de retorno
- NECESITO EL PUNTO DE RETORNO
- El tamaño de cada R.A de cada unidad puede determinarse en compilación.

- Todos los R.A pueden alocarse antes de la ejecución(alocación estática), se invoquen o no.
- Cada variable puede ser ligada a direcciones de memoria D antes de ejecución.
- Rutinas internas
    - Disjuntas: no pueden estar anidadas
    - No son recursivas
- Ambiente de las rutinas internas
    - Datos locales
    - Datos globales



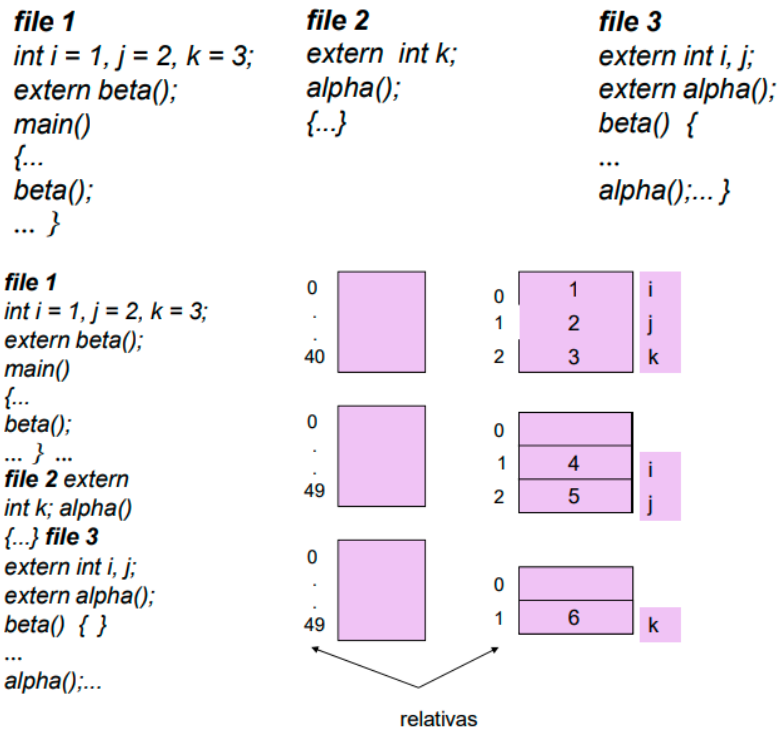
En el R.A de alpha() dejo un espacio para guardar el punto de retorno. Lo mismo para beta()



En el R.A de beta() se guarda la dirección de la siguiente instrucción que tendrías que ejecutar una vez que termine de ejecutarse beta() (valor 16) y luego si, bifurcá.

CASO 2' C2 PERO CON RUTINAS COMPILADAS POR SEPARADO

- En este caso, las unidades del programa se encuentran en archivos separados.
- Cada archivo es compilado por separado y en orden arbitrario.



En este caso, como cada unidad es compilada en distinto orden y por separado, entonces el compilador ya no puede:

- Ligar variables variables locales a locales a direcciones absolutas direcciones absolutas.
- Tampoco variables globales pueden ligarse con sus desplazamientos en el R.A. global.
- Las invocaciones a las rutinas no pueden ligarsecon la dirección de inicio de los correspondientes segmentos de código

Surge el Linkeditor:

- encargado de combinar los módulos
- ligar la información faltante

- El Linkeditor se encarga de asignar los varios segmentos de código a almacenamiento en C
- Se encarga de asignar los varios registros de activación dentro de D
- Y completa toda información faltante que el compilador no podía evaluar.
- C2 y C2’ no difieren semánticamente, una vez que el linkeditor reuna todas las unidades compiladas separadamente.