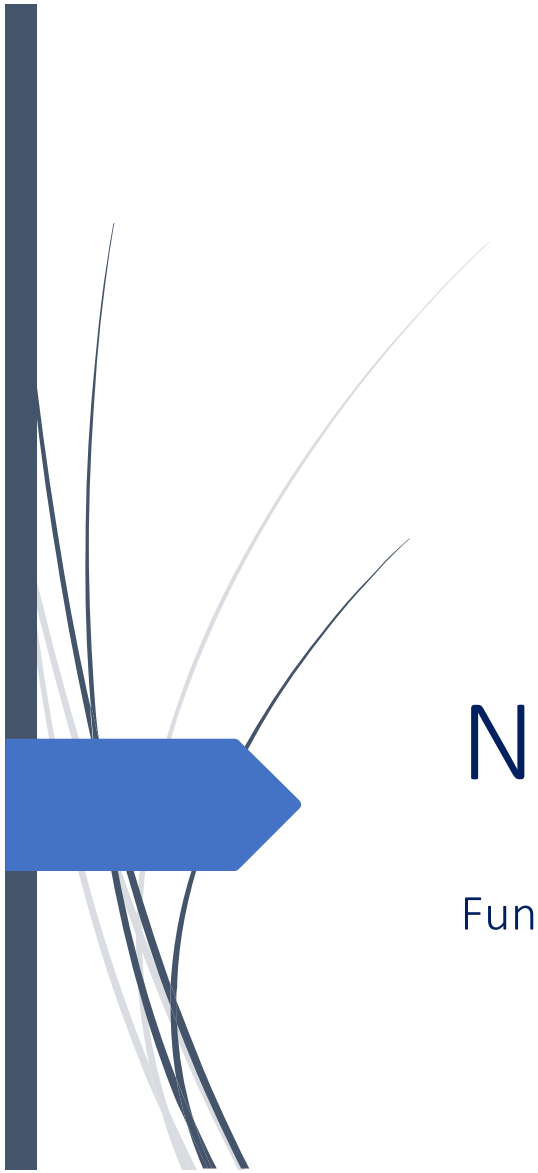


Nombres y vínculos

Fundamentos de Lenguajes de Programación – IIC2560

Fortran, Algol y Lisp fueron llamados "de alto nivel":

- su sintaxis y semántica eran mucho más *abstractas* que las de los lenguajes *assembly*

Programa en lenguaje de alto nivel (C)

```
swap(size_t v[], size_t k):
    size_t temp
    temp = v[k]
    v[k] = v[k+1]
    v[k+1] = temp
```

compilador

```
swap:
    slli x6,x11,3
    add x6,x10,x6
    lw  x5,0(x6)
    lw  x7,4(x6)
    sw  x7,0(x6)
    sw  x5,4(x6)
    jalr x0,0(x1)
```

Programa en **lenguaje assembly RISC-V**

¡¡ Los programadores usaron el computador para que los ayudara a programar el propio computador !!

assembler

Programa en **lenguaje de máquina** (binario) **RISC-V**

```
0000 … 010011
0000 … 110011
0000 … 000011
0000 … 000011
0000 … 100011
0000 … 100011
0000 … 100111
```

Cada línea (instrucción) tiene 32 bits de largo

Fortran, Algol y Lisp fueron llamados "de alto nivel":
- su sintaxis y semántica eran mucho más *abstractas* que las de los lenguajes *assembly*

Más distantes del hardware

Permitió escribir programas que corrían en una amplia diversidad de computadores

Los programas eran más fáciles de entender para las personas

La facilidad de programación sigue siendo el motor detrás del diseño de los lenguajes modernos

Los lenguajes imperativos son abstracciones de la *arquitectura von Neumann*:

- memoria —almacena instrucciones y datos
- procesador —operaciones para modificar el contenido de la memoria

En un lenguaje de programación ⟶ variables:

- desde una variable integer
- hasta un *array* tridimensional

Un *nombre* es un string de caracteres mnemónico usado para representar algo

Un *nombre* es un string de caracteres mnemónico usado para representar algo

En la mayoría de los lenguajes son identificadores (tokens alfanuméricos)

Un *nombre* es un string de caracteres mnemónico usado para representar algo

Nos permiten referirnos a

… variables, constantes, operaciones, tipos de datos, etc.,

… usando identificadores simbólicos en vez de, p.ej., direcciones de memoria

… y son esenciales en el contexto de un segundo significado de *abstracción*

Abstracción es un proceso:

El/la programador/a asocia un nombre con un fragmento de programa

… el cual puede ser considerado a partir de ese momento en términos de su propósito o función, en lugar de cómo se lleva a cabo esta función

Abstracción es un proceso:

El/la programador/a asocia un nombre con un fragmento de programa

… el cual puede ser considerado a partir de ese momento en términos de su propósito o función, en lugar de cómo se lleva a cabo esta función

Al esconder detalles irrelevantes, se reduce la complejidad conceptual

… y el/la programador/a se puede enfocar en un subconjunto manejable del texto del programa

Abstracción es un proceso:

El/la programador/a asocia un nombre con un fragmento de programa

… el cual puede ser considerado a partir de ese momento en términos de su propósito o función, en lugar de cómo se lleva a cabo esta función

Las **subrutinas** son abstracciones de control:

• esconden código arbitrariamente complejo detrás de una interfaz simple

Las **clases** son abstracciones de datos:

• esconden los detalles de la representación de datos detrás de un conjunto (comparativamente) simple de operaciones

En este curso, una **vinculación** es una asociación entre dos cosas:

- p.ej., un nombre y la cosa que éste nombra

El *momento de la vinculación* es el momento en el tiempo en el cual

… se crea la vinculación

… o, más en general, se toma una decisión de implementación

Al diseñar el lenguaje

P.ej.,
… los constructos de control de flujo
… el conjunto de tipos de datos fundamentales
… los mecanismos disponibles para crear tipos de datos más complejos

Al implementar el lenguaje
(a criterio del implementador)

El conjunto de valores posibles de `count` y la representación interna del literal 5

P.ej.,
… el número de bits (precisión) de los tipos de datos fundamentales
… el acoplamiento del I/O a la noción de archivos del sistema operativo
… la organización y los tamaños máximos del *stack* y del *heap*

Al escribir el programa
(los programadores)

P.ej.,
… algoritmos
… estructuras de datos
… nombres

P.ej., la siguiente asignación en C++:

```
count = count + 5;
```

Al compilar

El tipo de datos de `count` y el significado del símbolo `+`

P.ej.,

… correspondencia entre constructos de alto nivel y código de máquina

… disposición en la memoria de los datos estáticos

Al hacer el *linking*

P.ej.,

… direcciones virtuales

… disposición global de los módulos, de unos con respecto a otros

… vinculación entre un nombre en un módulo y el objeto en otro módulo al cual se refiere

Al hacer el *loading*

P.ej.,

… direcciones físicas

Al ejecutar el programa

El valor de `count`

P.ej.,

… los valores de las variables

… momento en que parte la ejecución, se entra a un módulo, se "ve" una declaración por primera vez, se llama a una subrutina, se entra a un bloque, se evalúa una expresión

Al compilar

P.ej.,
… correspondencia entre constructos de alto nivel y código de máquina
… disposición en la memoria de los datos estáticos

Al hacer el *linking*

P.ej.,
… direcciones virtuales
… disposición global de los módulos, de unos con respecto a otros
… vinculación entre un nombre en un módulo y el objeto en otro módulo al cual se refiere

Al hacer el *loading*

P.ej.,
… direcciones físicas

Vinculación dinámica

Al ejecutar el programa

P.ej.,
… los valores de las variables
… momento en que parte la ejecución, se entra a un módulo, se "ve" una declaración por primera vez, se llama a una subrutina, se entra a un bloque, se evalúa una expresión

Hay que distinguir entre los nombres y los objetos a los que ellos se refieren

… y hay que identificar eventos clave:

- creación y destrucción de objetos
- creación y destrucción de vinculaciones
- desactivación y reactivación de vinculaciones que pueden dejar usarse temporalmente
- referencias a variables, subrutinas, tipos de datos, etc., todas las cuales usan vinculaciones

Hay que distinguir entre los nombres y los objetos a los que ellos se refieren

… y hay que identificar eventos clave:

- creación y destrucción de objetos
- creación y destrucción de vinculaciones
- desactivación y reactivación de vinculaciones que pueden dejar usarse temporalmente
- referencias a variables, subrutinas, tipos de datos, etc., todas las cuales usan vinculaciones

El período de tiempo entre uno y otro es el *tiempo de vida* del objeto

El período de tiempo entre uno y otro es el *tiempo de vida* de la vinculación

El tiempo de vida de un objeto

… y el tiempo de vida de la vinculación de un nombre con ese objeto

… no tienen porqué coincidir:

- un objeto puede retener su valor y la posibilidad de ser accesado aún cuando un nombre dado no pueda ser más usado para accesarlo

El tiempo de vida de un objeto

… y el tiempo de vida de la vinculación de un nombre con ese objeto

… no tienen porqué coincidir:

• un objeto puede retener su valor y la posibilidad de ser accesado aún cuando un nombre dado no pueda ser más usado para accesarlo

P.ej., cuando una variable es pasada por referencia a una subrutina

… la vinculación entre el nombre del parámetro y la variable tiene un tiempo de vida más corto que la variable

El tiempo de vida de un objeto

… y el tiempo de vida de la vinculación de un nombre con ese objeto

… no tienen porqué coincidir:

• un objeto puede retener su valor y la posibilidad de ser accesado aún cuando un nombre dado no pueda ser más usado para accesarlo

P.ej., si un objeto creado mediante un operador **new** es pasado como parámetro por referencia y luego eliminado antes que la subrutina retorne

… entonces la vinculación entre el nombre del parámetro y el objeto tiene un tiempo de vida más largo que el objeto

El tiempo de vida de un objeto

… y el tiempo de vida de la vinculación de un nombre con ese objeto

… no tienen porqué coincidir:

• un objeto puede retener su valor y la posibilidad de ser accesado aún cuando un nombre dado no pueda ser más usado para accesarlo

P.ej., si un objeto creado mediante un operador **new** es pasado como parámetro por referencia y luego eliminado antes que la subrutina retorne

… entonces la vinculación entre el nombre del parámetro y el objeto tiene un tiempo de vida más largo que el objeto

Una referencia *dangling* —una vinculación a un objeto no vigente— que normalmente es señal de un error en el programa

Los tiempos de vida corresponden a uno de tres mecanismos de asignación de memoria:

- los objetos *estáticos* tienen una dirección (de memoria) absoluta que es mantenida a lo largo de toda la ejecución del programa
- a los objetos del *stack* se les asigna y se les quita espacio en memoria en orden *last-in, first-out*
- a los objetos del *heap* se les asigna y se les quita espacio en memoria en momentos arbitrarios

Los tiempos de vida corresponden a uno de tres mecanismos de asignación de memoria:

- los objetos *estáticos* tienen una dirección (de memoria) absoluta que es mantenida a lo largo de toda la ejecución del programa
- a los objetos del *stack* se les asigna y se les quita espacio en memoria en orden *last-in, first-out*
- a los objetos del *heap* se les asigna y se les quita espacio en memoria en momentos arbitrarios

Usualmente, en conjunción con llamadas y retornos de subrutinas

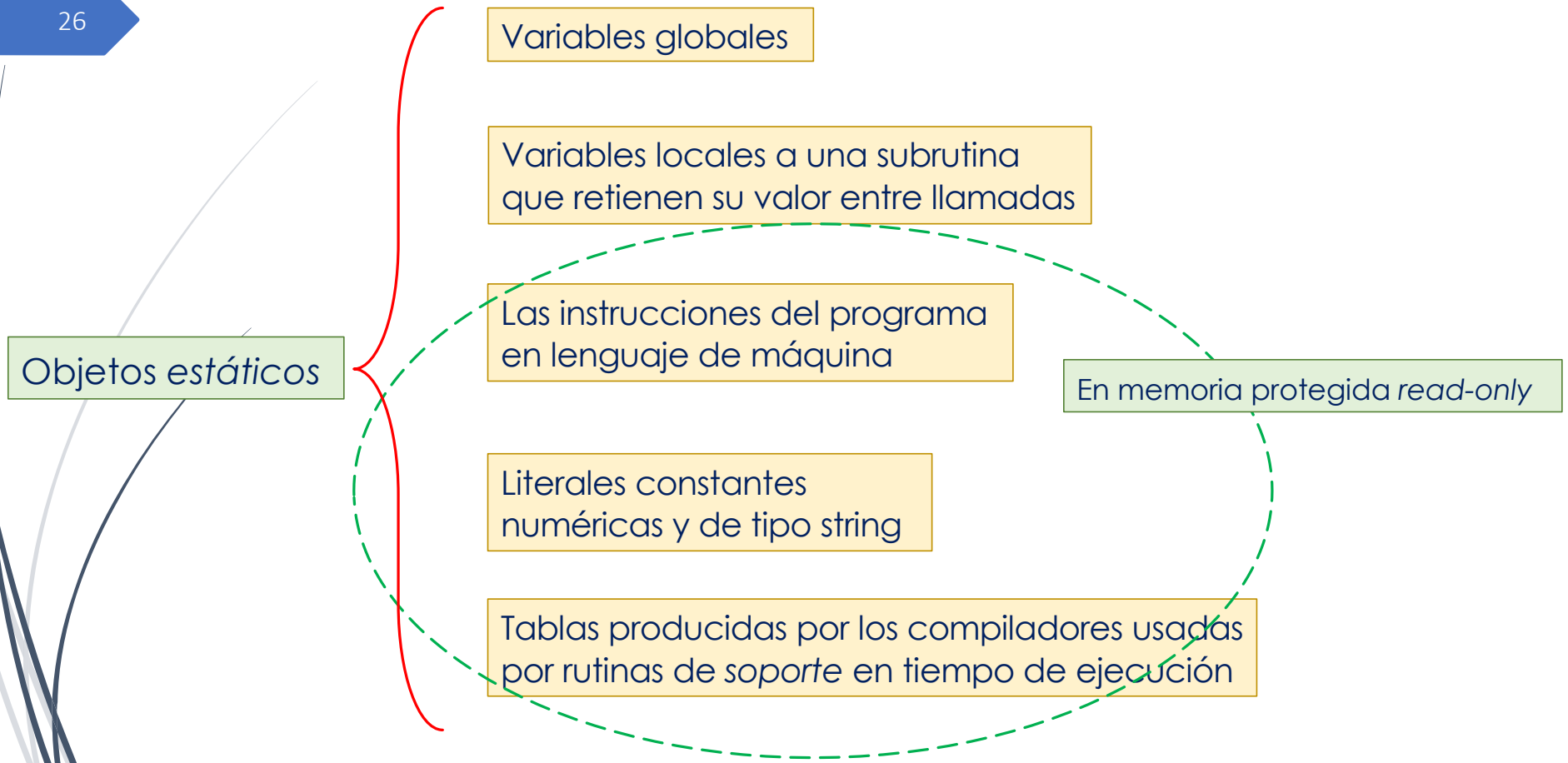
Requieren de un algoritmo general (y caro) de manejo de almacenamiento

Objetos *estáticos*

- Variables globales
- Variables locales a una subrutina que retienen su valor entre llamadas
- Las instrucciones del programa en lenguaje de máquina
- Literales constantes numéricas y de tipo string
- Tablas producidas por los compiladores usadas por rutinas de *soporte* en tiempo de ejecución

Objetos *estáticos*

- Variables globales
- Variables locales a una subrutina que retienen su valor entre llamadas
- Las instrucciones del programa en lenguaje de máquina
- Literales constantes numéricas y de tipo string
  - Las más pequeñas se almacenan dentro de la misma instrucción
- Tablas producidas por los compiladores usadas por rutinas de *soporte* en tiempo de ejecución
  - Debugging, chequeo dinámico de tipos, recolección de basura, manejo de excepciones, etc.


Objetos *estáticos*
Variables globales
Variables locales a una subrutina que retienen su valor entre llamadas
Las instrucciones del programa en lenguaje de máquina
Literales constantes numéricas y de tipo string
Tablas producidas por los compiladores usadas por rutinas de *soporte* en tiempo de ejecución
En memoria protegida *read-only*

Objetos del *stack*

(*asignación de memo-ria basada en* stack)

Objetos del *stack* (*asignación de memoria basada en* stack)

Si el lenguaje permite recursión

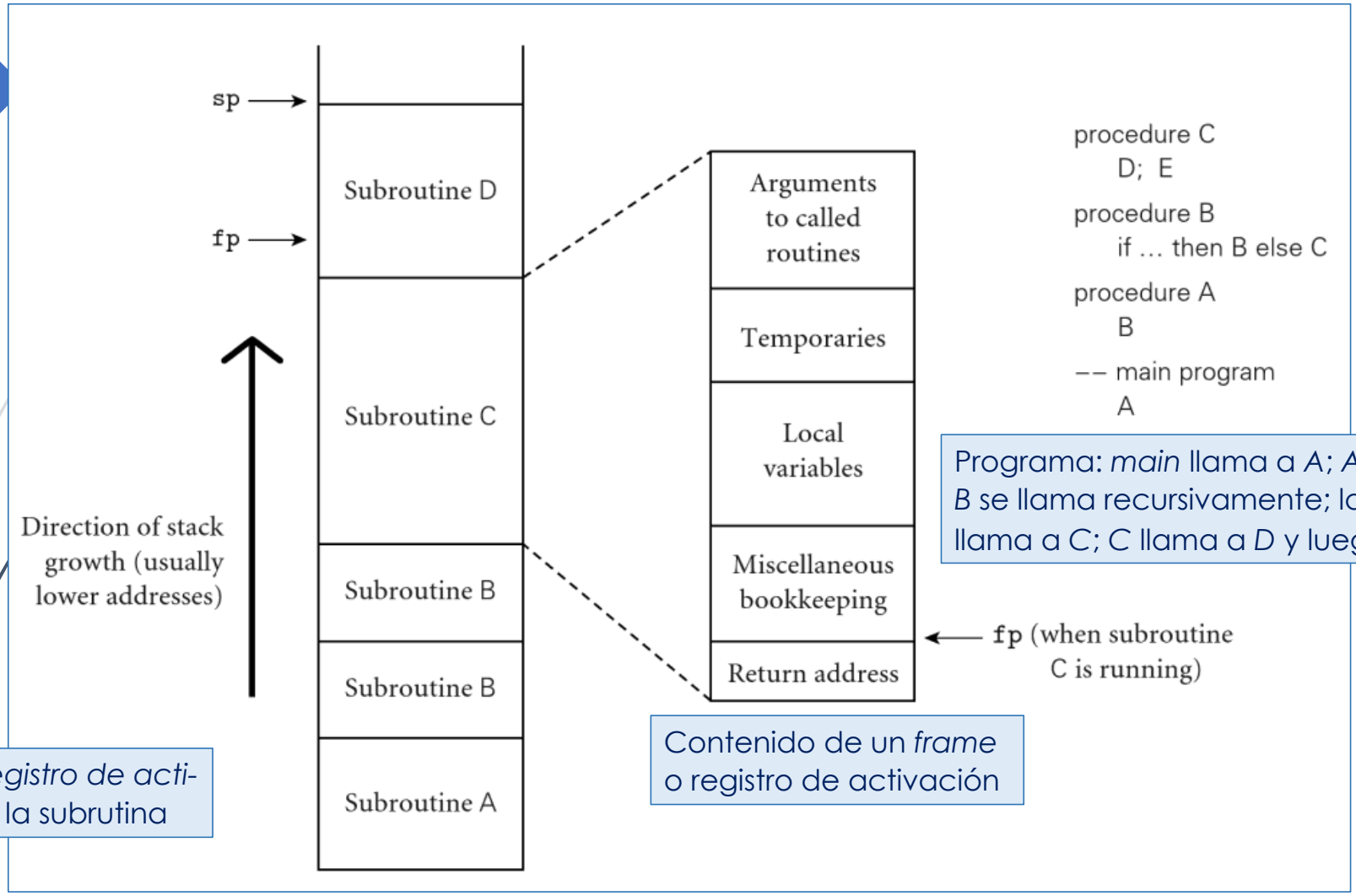
El número de instancias de una variable que pueden tener que existir al mismo tiempo es conceptualmente ilimitado

Objetos del *stack* (*asignación de memoria basada en* stack)

Si el lenguaje permite recursión

El número de instancias de una variable que pueden tener que existir al mismo tiempo es conceptualmente ilimitado

La anidación natural de las llamadas a subrutinas facilita asignar espacio en un stack



Programa: *main* llama a *A*; *A* llama a *B*; *B* se llama recursivamente; la recursión llama a *C*; *C* llama a *D* y luego a *E*

Contenido de un *frame* o registro de activación

Frame o *registro de activación* de la subrutina

Objetos del *stack* (*asignación de memoria basada en* stack)

Si el lenguaje permite recursión

El número de instancias de una variable que pueden tener que existir al mismo tiempo es conceptualmente ilimitado

La anidación natural de las llamadas a subrutinas facilita asignar espacio en un stack

El mantenimiento del *stack* es responsabilidad de la *secuencia de llamadas* a subrutinas

… y del *prólogo* y del *epílogo* de la propia subrutina

El código ejecutado por el que hace la llamada inmediatamente antes e inmediatamente después de la llamada

El código ejecutado al comienzo y el código ejecutado al final

Objetos del *stack* (*asignación de memoria basada en* stack)

Si el lenguaje permite recursión

El número de instancias de una variable que pueden tener que existir al mismo tiempo es conceptualmente ilimitado

La anidación natural de las llamadas a subrutinas facilita asignar espacio en un *stack*

El mantenimiento del *stack* es responsabilidad de la *secuencia de llamadas* a subrutinas

… y del *prólogo* y del *epílogo* de la propia subrutina

El código ejecutado por el que hace la llamada inmediatamente antes e inmediatamente después de la llamada

El código ejecutado al comienzo y el código ejecutado al final

El registro *frame pointer* apunta siempre a una ubicación conocida dentro del *frame* vigente

La ubicación de un *frame* no puede predecirse durante la compilación

… pero sí pueden determinarse estáticamente los *offsets* de los objetos dentro del *frame*

Objetos del *heap*

(*asignación de memo-ria basada en* heap)

Objetos del *heap* (*asignación de memoria basada en* heap)

Región de almacenamiento en que bloques de memoria se asignan y se quitan en momentos arbitrarios

Para estructuras de datos ligadas, strings, listas, conjuntos de tamaño variable

Objetos del *heap* (*asignación de memoria basada en* heap)

Región de almacenamiento en que bloques de memoria se asignan y se quitan en momentos arbitrarios

Para estructuras de datos ligadas, strings, listas, conjuntos de tamaño variable

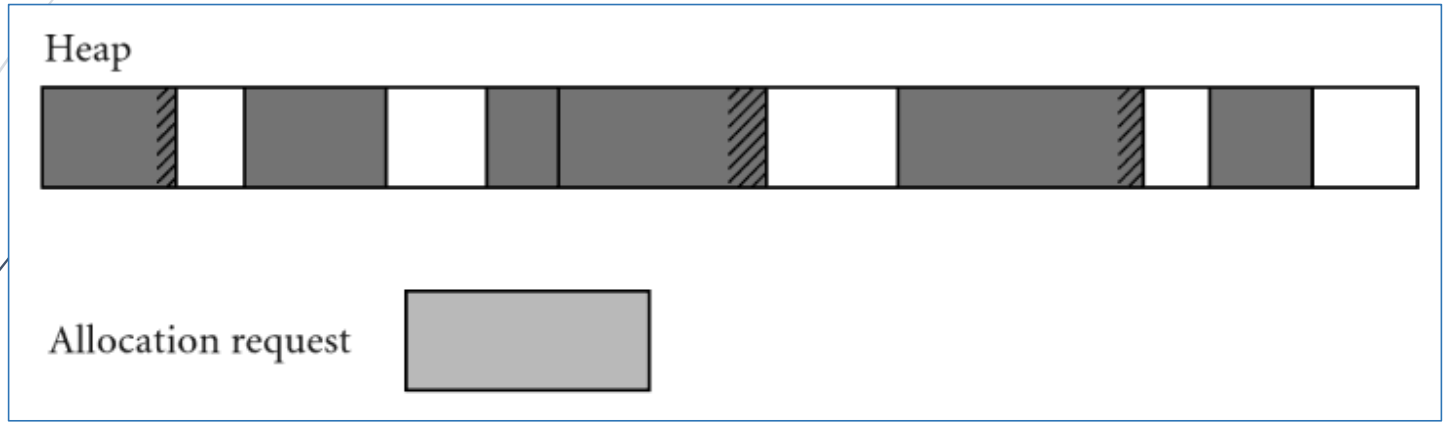
Hay varias estrategias posibles para manejar el espacio de un heap

Las preocupaciones principales son velocidad y espacio

… y hay compromisos entre ellos

C++

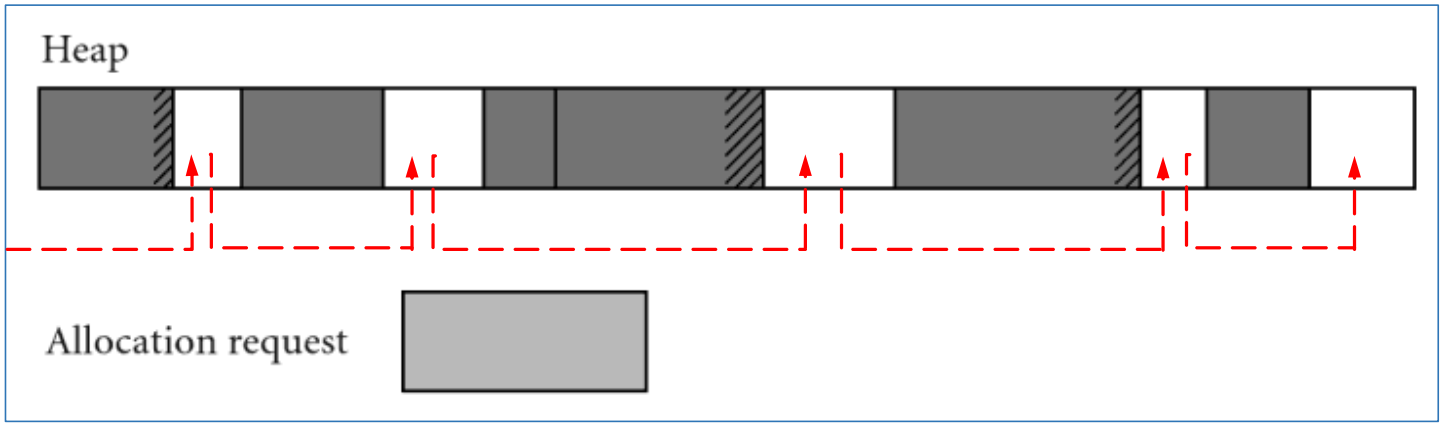
```
int  *intnode;       // Create a pointer
intnode =  new int; // Create the heap-dynamic variable
. . .
delete  intnode;    // Deallocate the heap-dynamic variable
                         // to which intnode points
```



JavaScript

```
highs = [74, 84, 86, 90, 71];
```

Algoritmos de manejo de almacenamiento



Mantienen una lista ligada —la *free list*—de bloques del heap desocupados

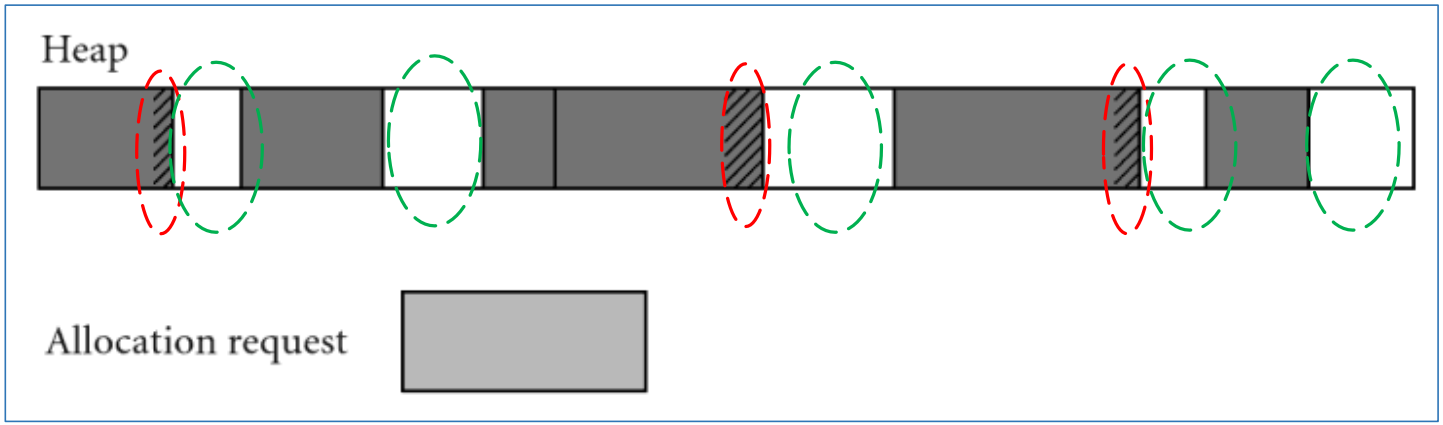
Inicialmente la lista contiene un único bloque que es todo el heap

Ante cada solicitud de asignación, se busca en la lista un bloque del tamaño apropiado:

- *first fit*, *best fit*

Si el bloque encontrado es mucho más grande que lo necesario, entonces se divide en dos y se devuelve la parte no usada a la *free list* como un bloque más pequeño

Al devolver un bloque a la *free list* se revisa si los bloques adyacentes están libres, en cuyo caso se unen



Fragmentación interna

El algoritmo asigna un bloque más grande que lo necesario y el espacio extra queda sin usar:

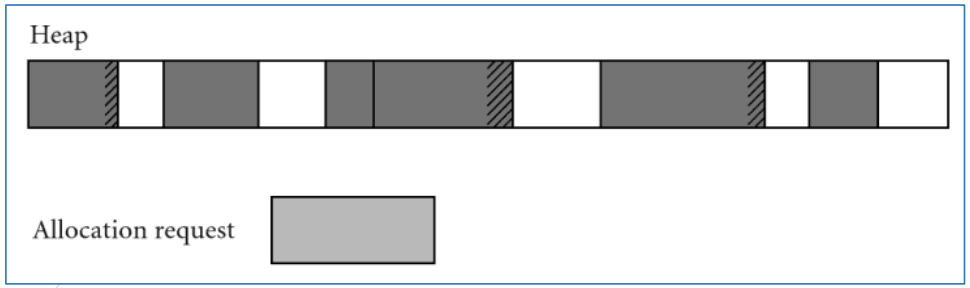
Fragmentación externa

Los bloques asignados están repartidos a través del heap, de manera que el espacio no usado consiste en múltiples bloques, pero ninguno suficientemente grande para una próxima solicitud:

Best fit

Hace un mejor trabajo de reservar bloques grandes para solicitudes grandes

… pero debe recorrer toda la lista —mayor costo de asignación— y tiende a producir un número grande de bloques desocupados muy pequeños

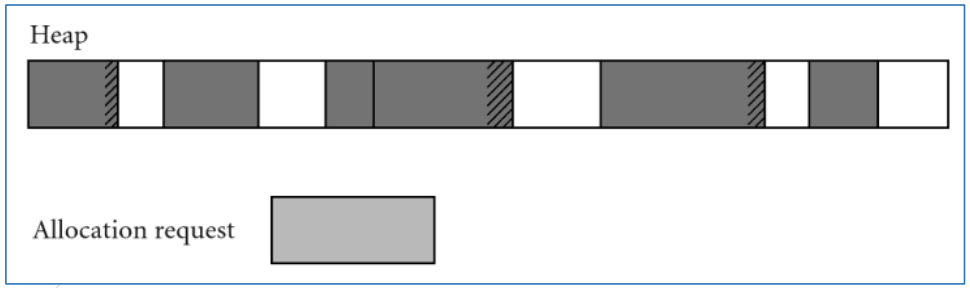


Asignación de memoria a objetos en el heap es *explícita*

⟶ gatillada siempre por alguna operación en el programa

P.ej.,

- instanciación de un objeto
- agregar al final de una lista
- agregar un valor largo a un string previamente corto

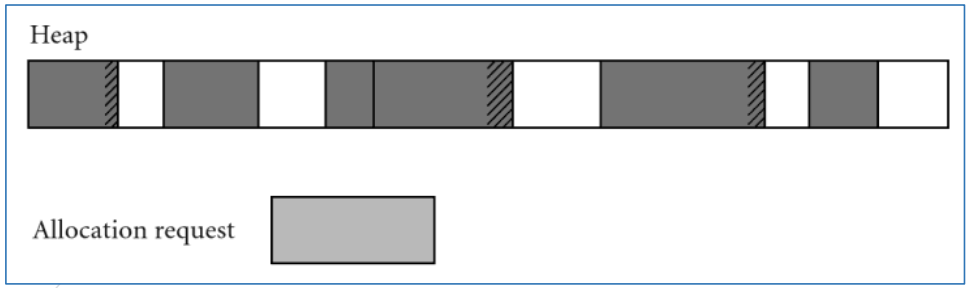


"Desasignación" —recolección— es explícita en algunos lenguajes C, C++, Rust

… pero muchos lenguajes especifican que debe ser *implícita*

⟶ cuando ya no es posible alcanzar el objeto desde ninguna variable del programa

Funcionales, de *scripting*
Java, C#, Python



"Desasignación" —recolección— es explícita en algunos lenguajes C, C++, Rust

… pero muchos lenguajes especifican que debe ser *implícita*

⟶ cuando ya no es posible alcanzar el objeto desde ninguna variable del programa

Funcionales, de *scripting*
Java, C#, Python

La librería de *run time* del lenguaje debe proveer un mecanismo de *garbage collection*

⟶ identificar tales objetos

… y devolver la memoria que ocupan a la *free list*

Recolección manual (explícita):

- simplicidad de implementación del lenguaje
- velocidad de ejecución

Recolección automática agrega mucha complejidad a la implementación de un lenguaje que tenga un sistema sofisticado de tipos de datos

… y su ejecución puede tomar mucho tiempo

Recolección manual (explícita):

- simplicidad de implementación del lenguaje
- velocidad de ejecución

Recolección automática agrega mucha complejidad a la implementación de un lenguaje que tenga un sistema sofisticado de tipos de datos

… y su ejecución puede tomar mucho tiempo

Recolección automática (implícita):

- errores debidos a recolección explícita son los más comunes y costosos (difíciles de identificar y corregir)

Si un objeto es recolectado demasiado pronto, el programa puede seguir a una *dangling reference*, accesando memoria usada ahora por otro objeto

Si el objeto no es recolectado al final de su tiempo de vida, el programa puede quedarse sin espacio en el heap

Hoy, la recolección automática es una propiedad de muchos lenguajes

Los algoritmos han mejorado, reduciendo el *overhead* en *run time*

La complejidad de implementación se ha vuelto marginal frente a la complejidad general de las implementaciones de lenguajes

Sus beneficios son más atractivos porque las aplicaciones se han vuelto más grandes y complejas

The scope of a binding

El **alcance** de una vinculación es la región del texto del programa en la cual la vinculación está activa

En la mayoría de los lenguajes modernos, es determinado estáticamente ⟶ al momento de compilar

The scope of a binding

El **alcance** de una vinculación es la región del texto del programa en la cual la vinculación está activa

En la mayoría de los lenguajes modernos, es determinado estáticamente ⟶ al momento de compilar

P.ej., en C:

Nuevos alcances al entrar a una subrutina:

- se crean vinculaciones para objetos locales
- se desactivan vinculaciones para objetos globales que son ocultados por objetos locales con el mismo nombre

Al sali de la subrutinar:

- se destruyen las vinculaciones para las variables locales
- se reactivan las vinculaciones para todos los objetos globales que estaban ocultos

The scope of a binding

El **alcance** de una vinculación es la región del texto del programa en la cual la vinculación está activa

En la mayoría de los lenguajes modernos, es determinado estáticamente ⟶ al momento de compilar

C es un lenguaje con **alcance estático**:

- Las partes del programa en que una vinculación está activa están totalmente determinadas al momento de la compilación
- Al mirar un programa en C sabemos qué nombres se refieren a qué objetos en qué partes del programa en base a reglas sólo del texto del programa

P.ej., en C:

Nuevos alcances al entrar a una subrutina:

- se crean vinculaciones para objetos locales
- se desactivan vinculaciones para objetos globales que son ocultados por objetos locales con el mismo nombre

Al sali de la subrutinar:

- se destruyen las vinculaciones para las variables locales
- se reactivan las vinculaciones para todos los objetos globales que estaban ocultos

A scope

Un **ámbito** es una región de programa de tamaño máximo en la cual ninguna vinculación cambia:

- el cuerpo de un módulo, clase, subrutina, o sentencia de flujo de control estructurado, a veces llamado un *bloque*
  - P.ej., en C es delimitado por { y }

En un lenguaje con alcance estático, la vinculación "vigente" para un nombre dado se encuentra en la declaración correspondiente cuyo bloque rodea más estrechamente un punto dado en el programa

Muchos lenguajes permiten la anidación de unas subrutinas dentro de otras: anidación estilo Algol

- P.ej., Python, JavaScript, Common Lisp, Ada, F#, …

Se crean ámbitos estáticos anidados

Muchos lenguajes permiten la anidación de unas subrutinas dentro de otras: anidación estilo Algol

- P.ej., Python, JavaScript, Common Lisp, Ada, F#, …

Se crean ámbitos estáticos anidados

En lenguajes que no permiten la anidación de subrutinas, los ámbitos estáticos anidados son creados sólo por bloques y definiciones de clases anidados

- P.ej., lenguajes basados en C

Regla del **ámbito anidado más cercano** para vincular nombres con objetos:

Un nombre que es introducido en una declaración es conocido en el ámbito en el cual está declarado

… y en cada ámbito anidado internamente

… a menos que sea ocultado por otra declaración del mismo nombre en uno o más ámbitos anidados

Regla del **ámbito anidado más cercano** para vincular nombres con objetos:

Un nombre que es introducido en una declaración es conocido en el ámbito en el cual está declarado

… y en cada ámbito anidado internamente

… a menos que sea ocultado por otra declaración del mismo nombre en uno o más ámbitos anidados

Para encontrar el objeto correspondiente a un determinado uso de un nombre, buscamos una declaración con ese nombre en el ámbito vigente más interno

… de lo contrario, buscamos una declaración en el ámbito inmediatamente circundante

… continuamos hacia afuera, examinando ámbitos circundantes sucesivos

… hasta alcanzar el nivel de anidación exterior del programa

Si no encontramos la declaración en ningún nivel, entonces hay un error en el programa

P2 es llamado sólo por P1

… y no necesita ser visible fuera de P1

⟶ es declarado dentro de P1, limitando su ámbito (su región de visibilidad) a la porción del programa que se muestra

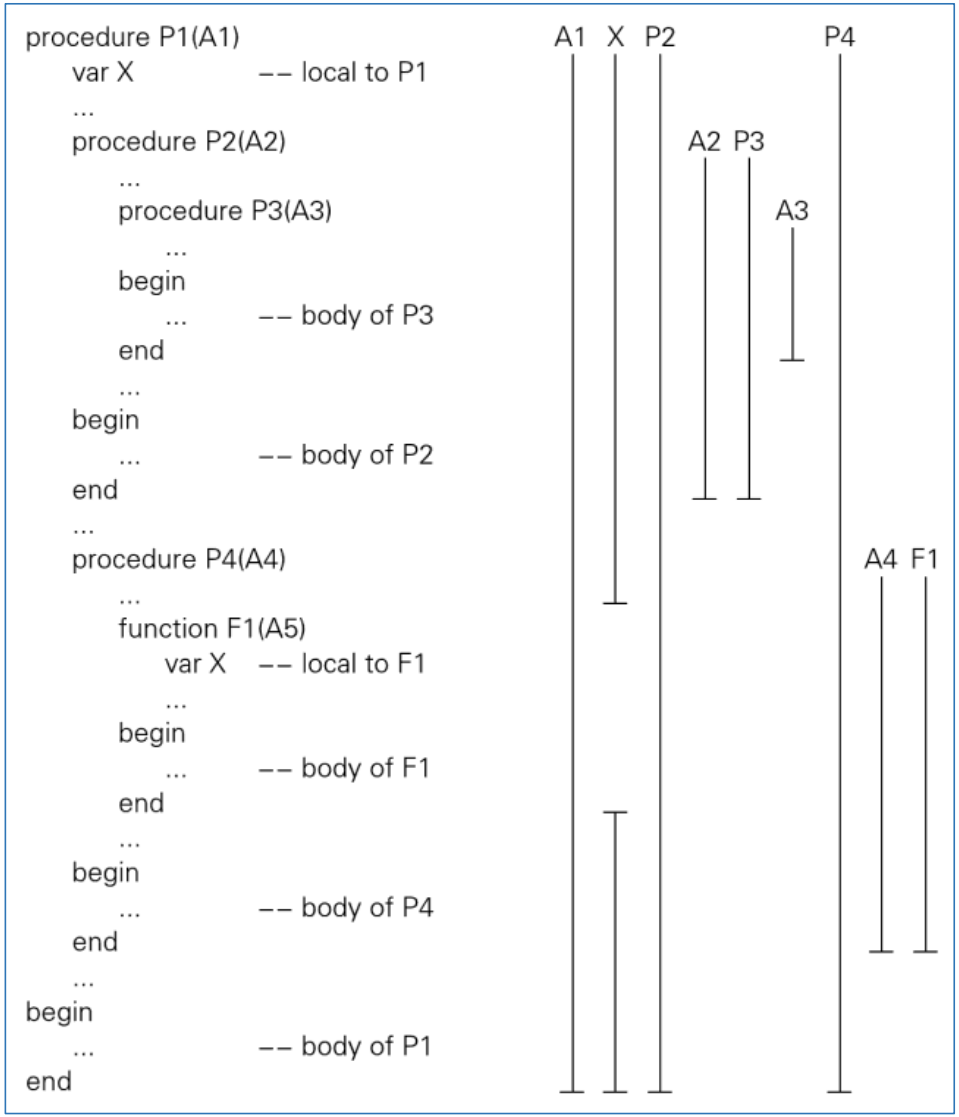
P4 es visible sólo dentro de P1

P3 es visible sólo dentro de P2

F1 es visible sólo dentro de P4

Según regla de ámbitos anidados:

- F1 podría llamar a P2
- P4 podría llamar a F1
- P2 no podría llamar a F1



También, las subrutinas anidadas pueden tener acceso a parámetros y variables locales (y otros objetos locales) de los ámbitos circundantes:

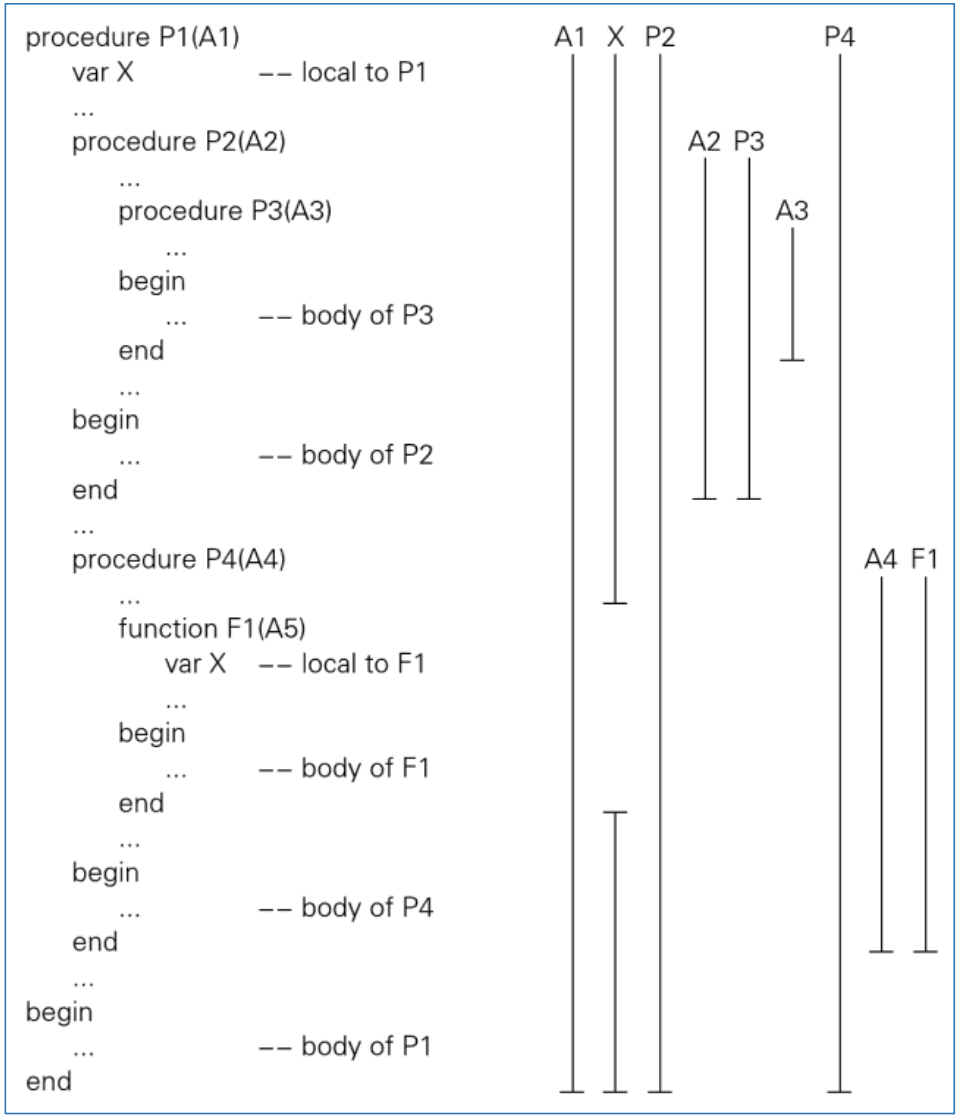
• P3 puede nombrar (y modificar) A1, X y A2, además de A3

Tanto P1 como F1 declaran variables locales que se llaman X

⟶ la declaración más interior oculta a la más exterior dentro de una parte de su ámbito:

• el uso de X en F1 se refiere al X de más adentro

• el uso de X en otras regiones del código se refiere al X de más afuera



¿Cómo se tiene acceso a objetos no locales?

El compilador se asegura que el registro **fp** apunte en *run time* al *frame* de la subrutina actualmente en ejecución

fp se usa como base para *direccionamiento por desplazamiento* (registro más offset), para tener acceso a los objetos dentro de la subrutina actual

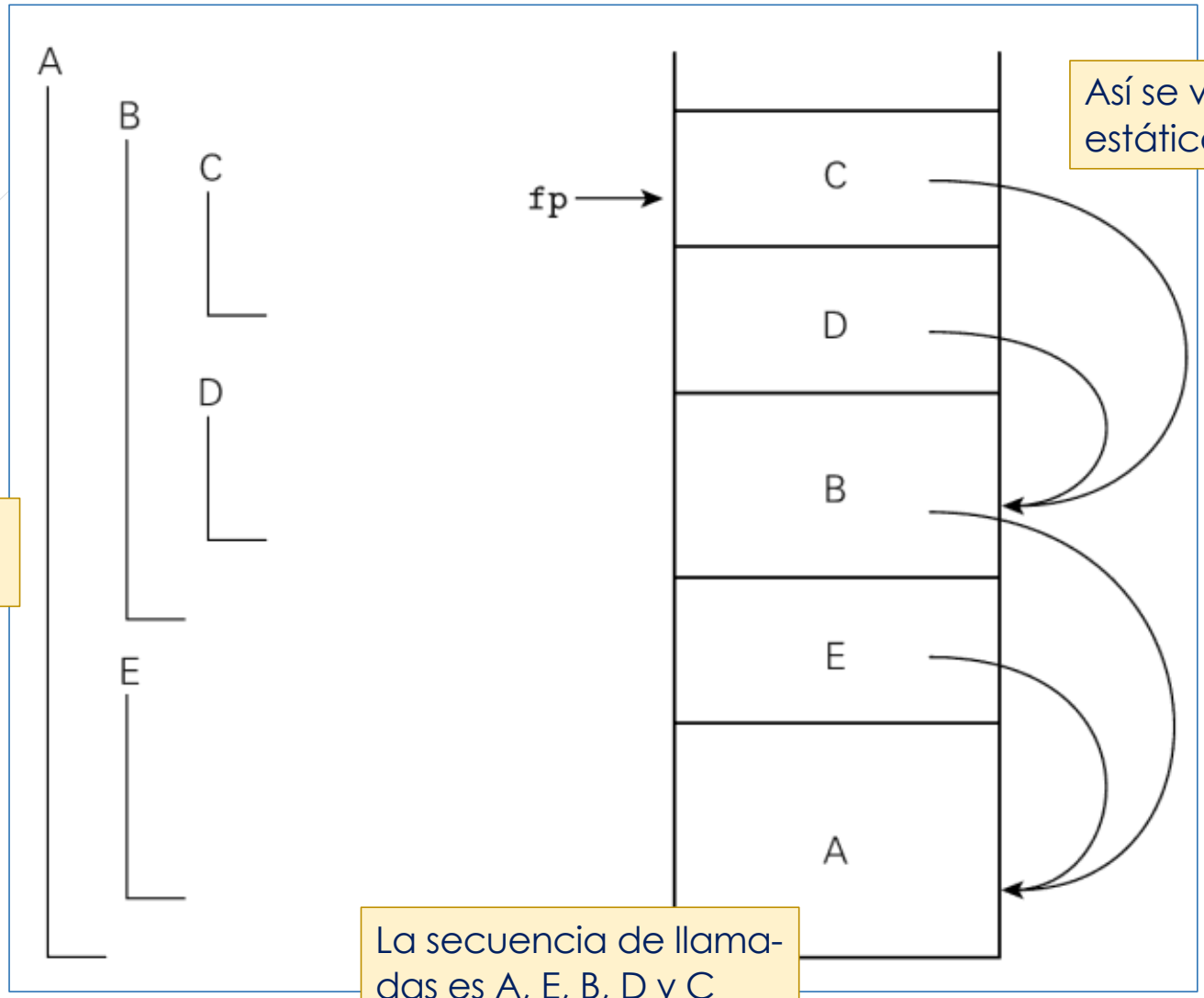
Para los objetos en subrutinas léxicamente circundantes necesitamos encontrar en *run time* los frames correspondientes a esos ámbitos

Una subrutina anidada puede llamar a otra en su ámbito exterior

⟶ el orden de los frames en el stack no corresponde necesariamente al orden de la anidación léxica

⟶ mantenemos un *link estático* en cada frame que apnta al frame "padre:" el frame de la llamada más reciente de la subrutina léxicamente circundante

Anidación de las sub-rutinas A, B, C, D y E



Así se ven los links estáticos en el stack

La secuencia de llama-das es A, E, B, D y C

The scope of a binding

El **alcance** de una vinculación es la región del texto del programa en la cual la vinculación está activa

En la mayoría de los lenguajes modernos, es determinado estáticamente → al momento de compilar

The scope of a binding

El **alcance** de una vinculación es la región del texto del programa en la cual la vinculación está activa

En la mayoría de los lenguajes modernos, es determinado estáticamente ⟶ al momento de compilar

P.ej., en C:

Nuevas vinculaciones al entrar a una subrutina:

- se crean vinculaciones para objetos locales
- se desactivan vinculaciones para objetos globales que son ocultados por objetos locales con el mismo nombre

Al salir de la subrutina:

- se destruyen las vinculaciones para las variables locales
- se reactivan las vinculaciones para todos los objetos globales que estaban ocultos

The scope of a binding

El **alcance** de una vinculación es la región del texto del programa en la cual la vinculación está activa

En la mayoría de los lenguajes modernos, es determinado estáticamente ⟶ al momento de compilar

P.ej., C es un lenguaje con **alcance estático**:

- Las partes del programa en que una vinculación está activa están totalmente determinadas al momento de la compilación
- Al mirar un programa en C sabemos qué nombres se refieren a qué objetos en qué partes del programa en base a reglas sólo del texto del programa

P.ej., en C:

Nuevas vinculaciones al entrar a una subrutina:

- se crean vinculaciones para objetos locales
- se desactivan vinculaciones para objetos globales que son ocultados por objetos locales con el mismo nombre

Al salir de la subrutina:

- se destruyen las vinculaciones para las variables locales
- se reactivan las vinculaciones para todos los objetos globales que estaban ocultos

A scope

Un **ámbito** es una región de programa de tamaño máximo en la cual ninguna vinculación cambia:

- el cuerpo de un módulo, clase, subrutina, o sentencia de flujo de control estructurado, a veces llamado un *bloque*
  - P.ej., en C es delimitado por { y }

En un lenguaje con alcance estático, la vinculación "vigente" para un nombre dado se encuentra en la declaración correspondiente cuyo bloque rodea más estrechamente un punto dado en el programa

Muchos lenguajes permiten la anidación de unas subrutinas dentro de otras: *anidación estilo Algol*

- P.ej., Python, JavaScript, Common Lisp, Ada, F#, …

Se crean *ámbitos estáticos anidados*

Muchos lenguajes permiten la anidación de unas subrutinas dentro de otras: *anidación estilo Algol*

- P.ej., Python, JavaScript, Common Lisp, Ada, F#, …

Se crean *ámbitos estáticos anidados*

En lenguajes que no permiten anidación de subrutinas, los ámbitos estáticos anidados son creados por bloques anidados y/o definiciones de clases anidados

- P.ej., lenguajes basados en C

```
function  big() {
  function  sub1() {
    var  x = 7;
    sub2();
  }
  function  sub2() {
    var  y = x;
  }
  var  x = 3;
  sub1();
}
```

¿ Cuál **x** es esta **x** ?

```
function  big() {
  function  sub1() {
    var  x = 7;
    sub2();
  }
  function  sub2() {
    var  y = x;
  }
  var  x = 3;
  sub1();
}
```

Esta es una posibilidad

¿ Cuál **x** es esta **x** ?

Esta es otra posibilidad

En lenguajes con alcance estático, cuando encontramos una referencia a una variable, los atributos de la variable deben ser determinados encontrando **la sentencia en la cual la variable está declarada**:

- buscamos primero en las declaraciones del subprograma en que aparece la referencia a la variable

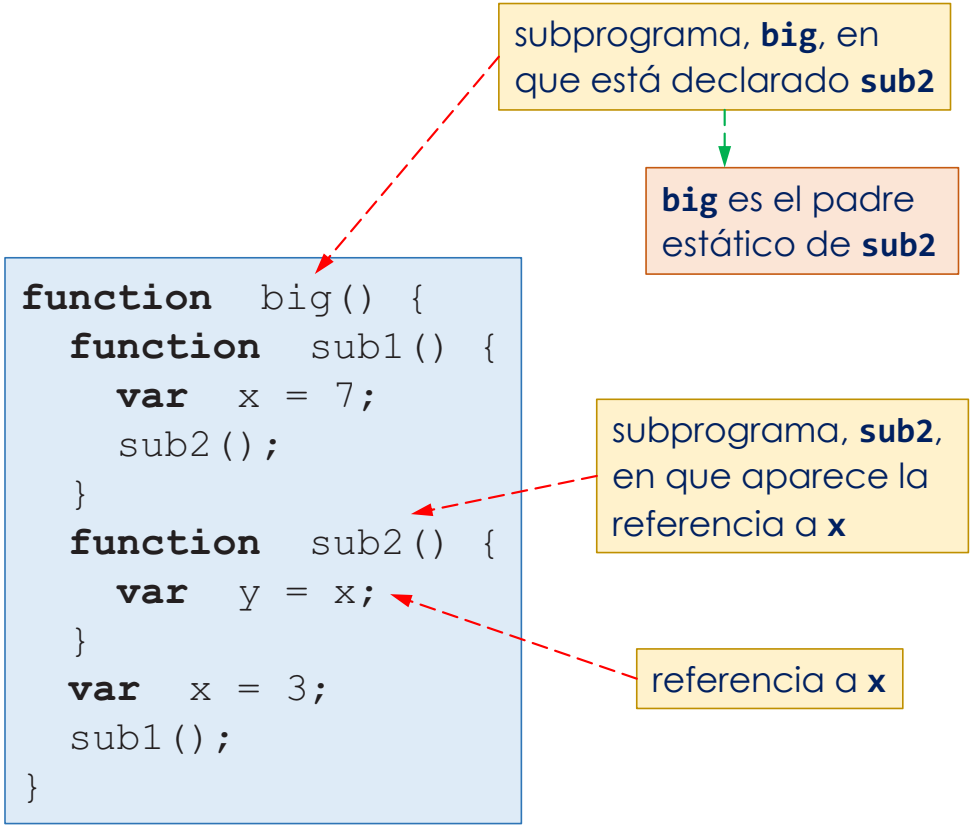
```
function  big() {
  function  sub1() {
    var  x = 7;
    sub2();
  }
  function  sub2() {
    var  y = x;
  }
  var  x = 3;
  sub1();
}
```

subprograma, **sub2**, en que aparece la referencia a **x**

referencia a **x**

En lenguajes con alcance estático, cuando encontramos una referencia a una variable, los atributos de la variable deben ser determinados encontrando **la sentencia en la cual la variable está declarada**:

- buscamos primero en las declaraciones del subprograma en que aparece la referencia a la variable
- (si allí no aparece la declaración de la variable, entonces …) buscamos en las declaraciones del subprograma en que está declarado el subprograma anterior ⟶ el **padre estático** del subprograma anterior
- (si allí no aparece la declaración de la variable, entonces …) buscamos en las declaraciones del subprograma en que está declarado el padre estático del subprograma en que aparece la referencia a la variable
- … y así sucesivamente




```
function  big() {
  function  sub1() {
    var  x = 7;
    sub2();
  }
  function  sub2() {
    var  y = x;
  }
  var  x = 3;
  sub1();
}
```

Regla del **ámbito anidado más cercano** para vincular nombres con objetos:

Un nombre que es introducido en una declaración es conocido en el ámbito en el cual está declarado

… y en cada ámbito anidado internamente

… a menos que sea ocultado por otra declaración del mismo nombre en uno o más ámbitos anidados

Regla del **ámbito anidado más cercano** para vincular nombres con objetos:

Un nombre que es introducido en una declaración es conocido en el ámbito en el cual está declarado

… y en cada ámbito anidado internamente

… a menos que sea ocultado por otra declaración del mismo nombre en uno o más ámbitos anidados

Para encontrar el objeto correspondiente a un determinado uso de un nombre, buscamos una declaración con ese nombre en el ámbito vigente más interno

… de lo contrario, buscamos una declaración en el ámbito inmediatamente circundante

… continuamos hacia afuera, examinando ámbitos circundantes sucesivos

… hasta alcanzar el nivel de anidación exterior del programa

Si no encontramos la declaración en ningún nivel, entonces hay un error en el programa

P2 es llamado sólo por P1

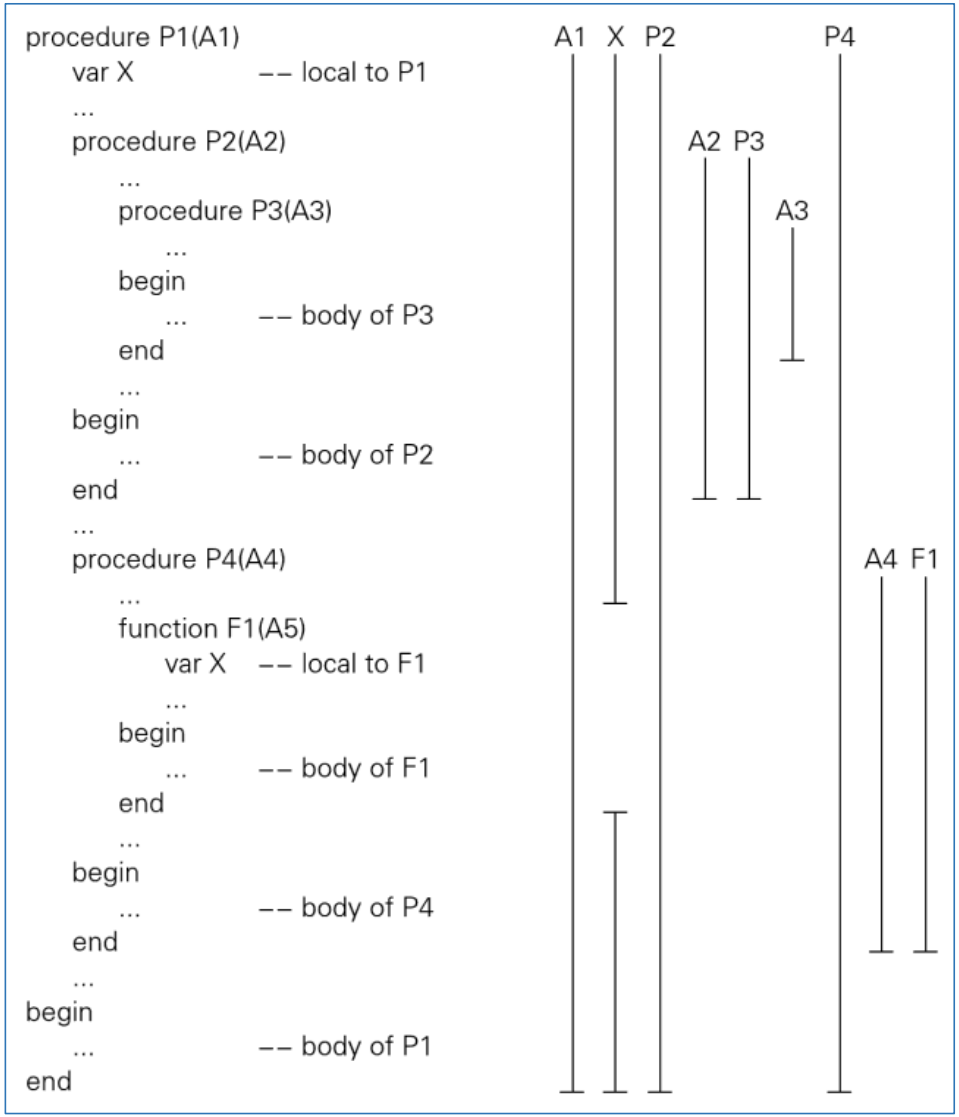
… y no necesita ser visible fuera de P1

→ es declarado dentro de P1, limitando su ámbito (su región de visibilidad) a la porción del programa que se muestra

P4 es visible sólo dentro de P1

P3 es visible sólo dentro de P2

F1 es visible sólo dentro de P4



P2 es llamado sólo por P1

… y no necesita ser visible fuera de P1

⟶ es declarado dentro de P1, limitando su ámbito (su región de visibilidad) a la porción del programa que se muestra

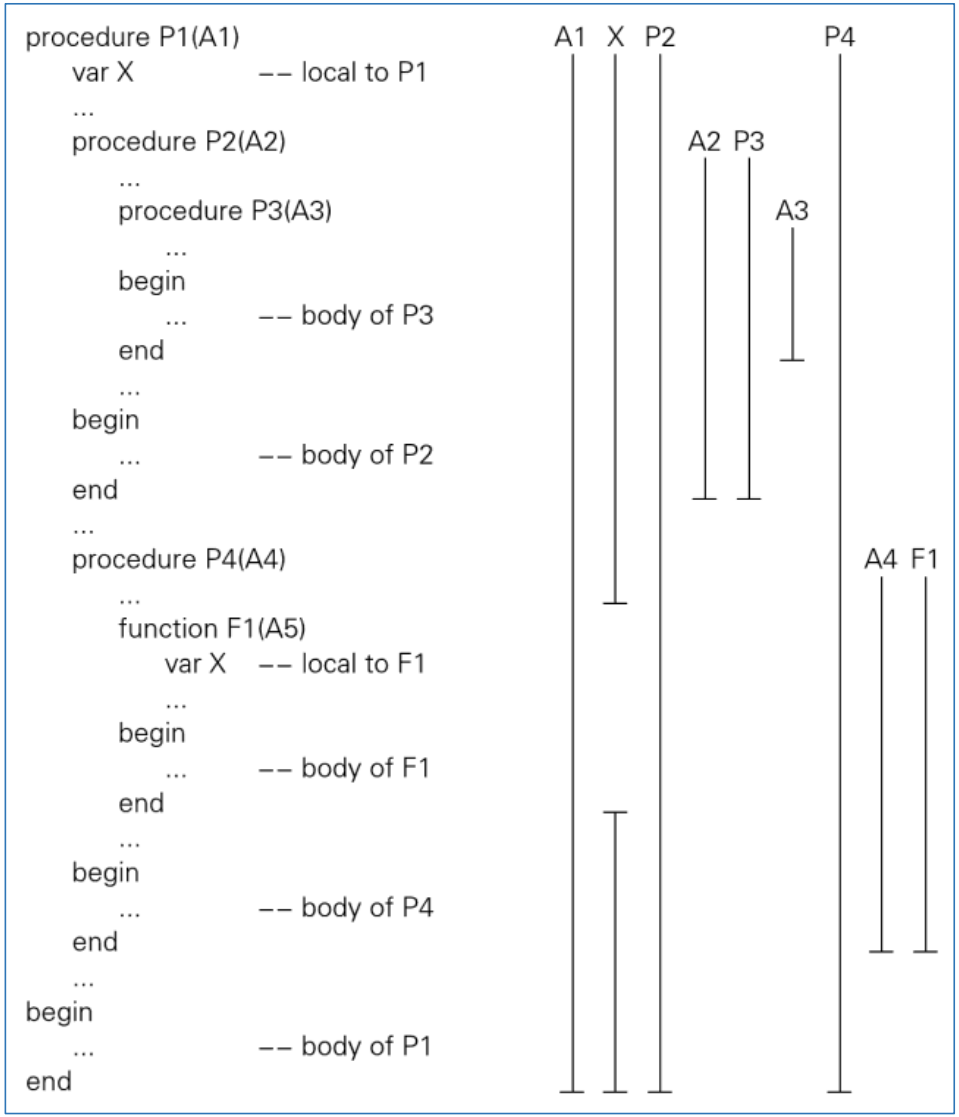
P4 es visible sólo dentro de P1

P3 es visible sólo dentro de P2

F1 es visible sólo dentro de P4

Según regla de ámbitos anidados:

- F1 podría llamar a P2
- P4 podría llamar a F1
- P2 no podría llamar a F1



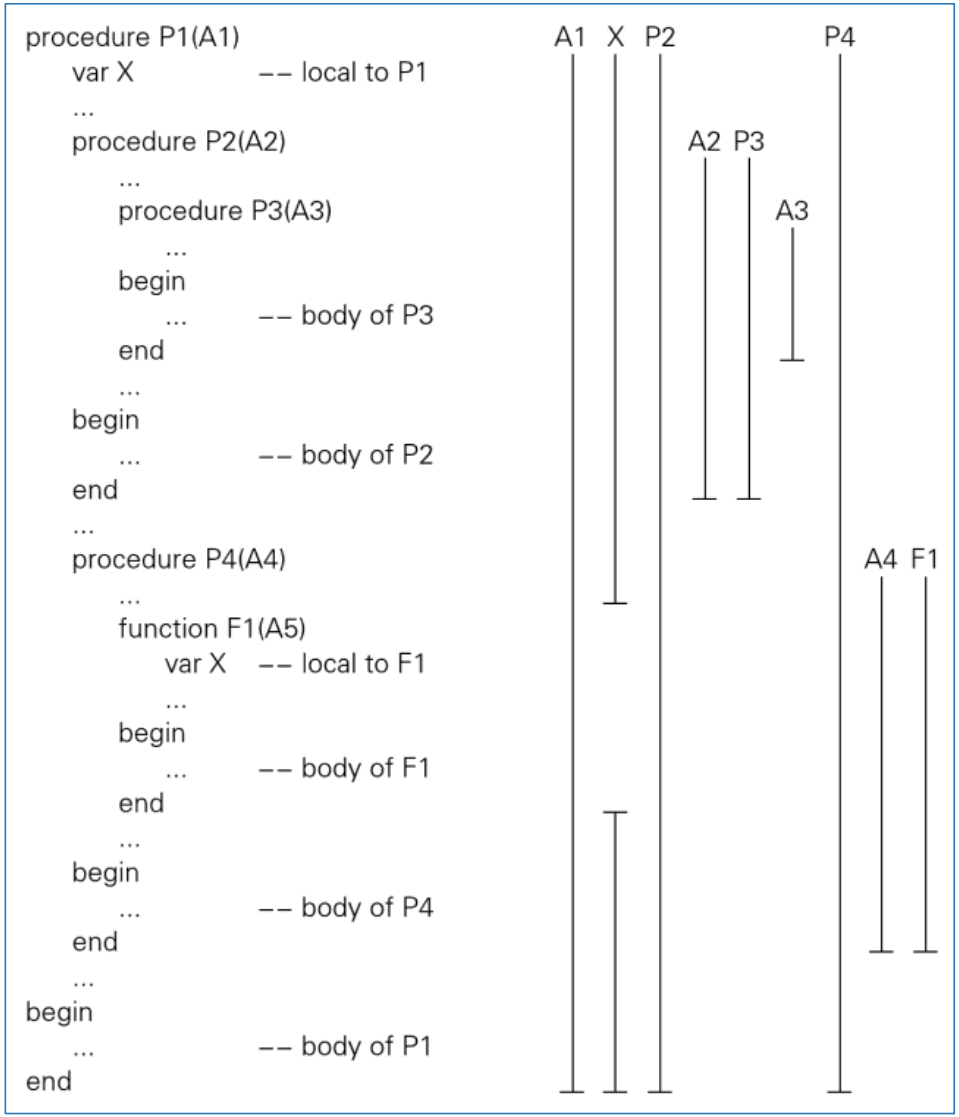
También, las subrutinas anidadas pueden tener acceso a parámetros y variables locales (y otros objetos locales) de los ámbitos circundantes:

- P3 puede nombrar (y modificar) A1, X y A2, además de A3

Tanto P1 como F1 declaran variables locales que se llaman X

⟶ la declaración más interior oculta a la más exterior dentro de una parte de su ámbito:

- el uso de X en F1 se refiere al X de más adentro
- el uso de X en otras regiones del código se refiere al X de más afuera



¿Cómo se tiene acceso a objetos locales?

El compilador se asegura de que el registro **fp** apunte en *run time* al *frame* de la subrutina actualmente en ejecución

fp se usa como base para *direcciona-miento por desplazamiento* (registro más *offset*), para tener acceso a los objetos dentro de la subrutina actual

¿Cómo se tiene acceso a objetos no locales?

Para los objetos en subrutinas léxicamente circundantes, necesitamos encontrar en *run time* los *frames* correspondientes a esos ámbitos

Una subrutina anidada puede llamar a otra en su ámbito exterior:

⟶ el orden de los *frames* en el stack no corresponde necesariamente al orden de la anidación léxica

⟶ mantenemos un **link estático** en cada frame que apunta al *frame* "padre": el *frame* de la llamada más reciente de la **subrutina léxicamente circundante**

La subrutina *léxicamente circundante* a P3 es P2

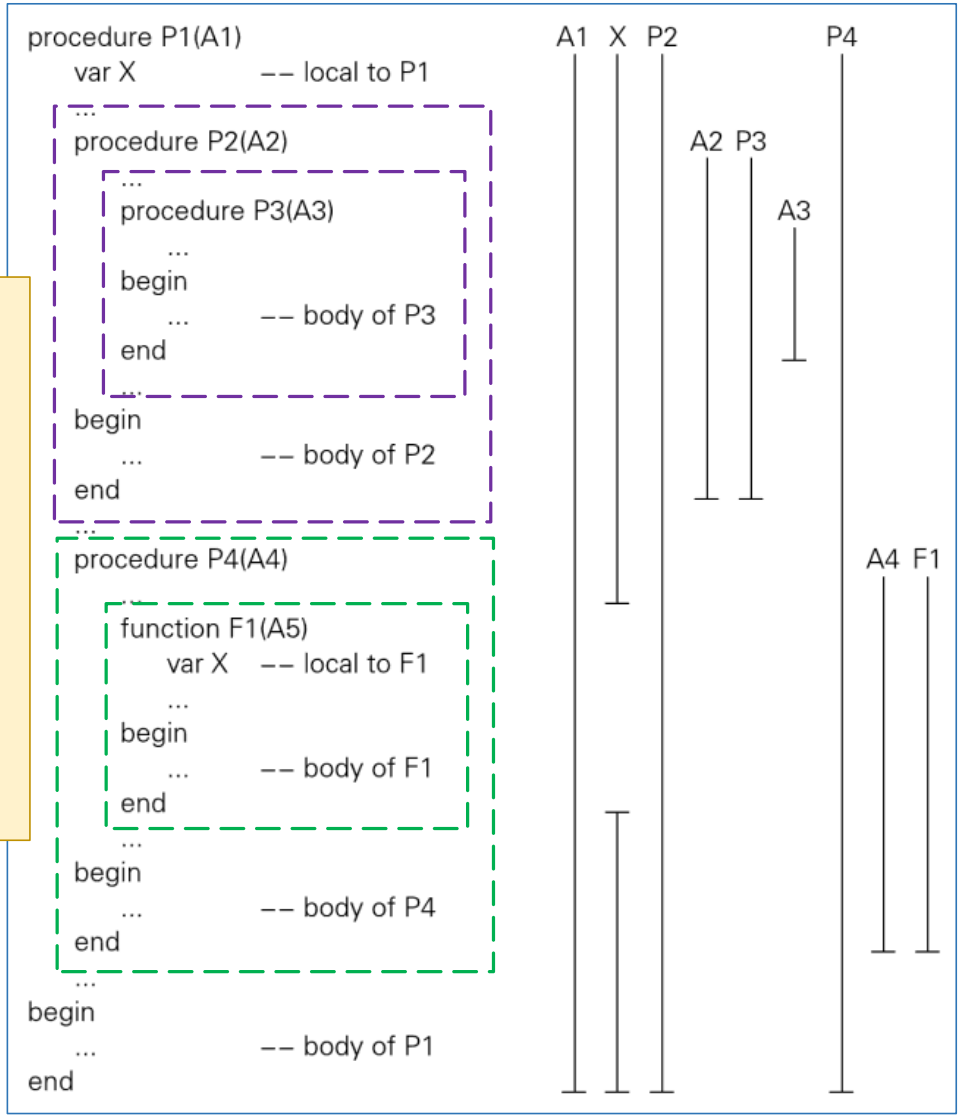
… y la subrutina léxicamente circundante a P2 es P1

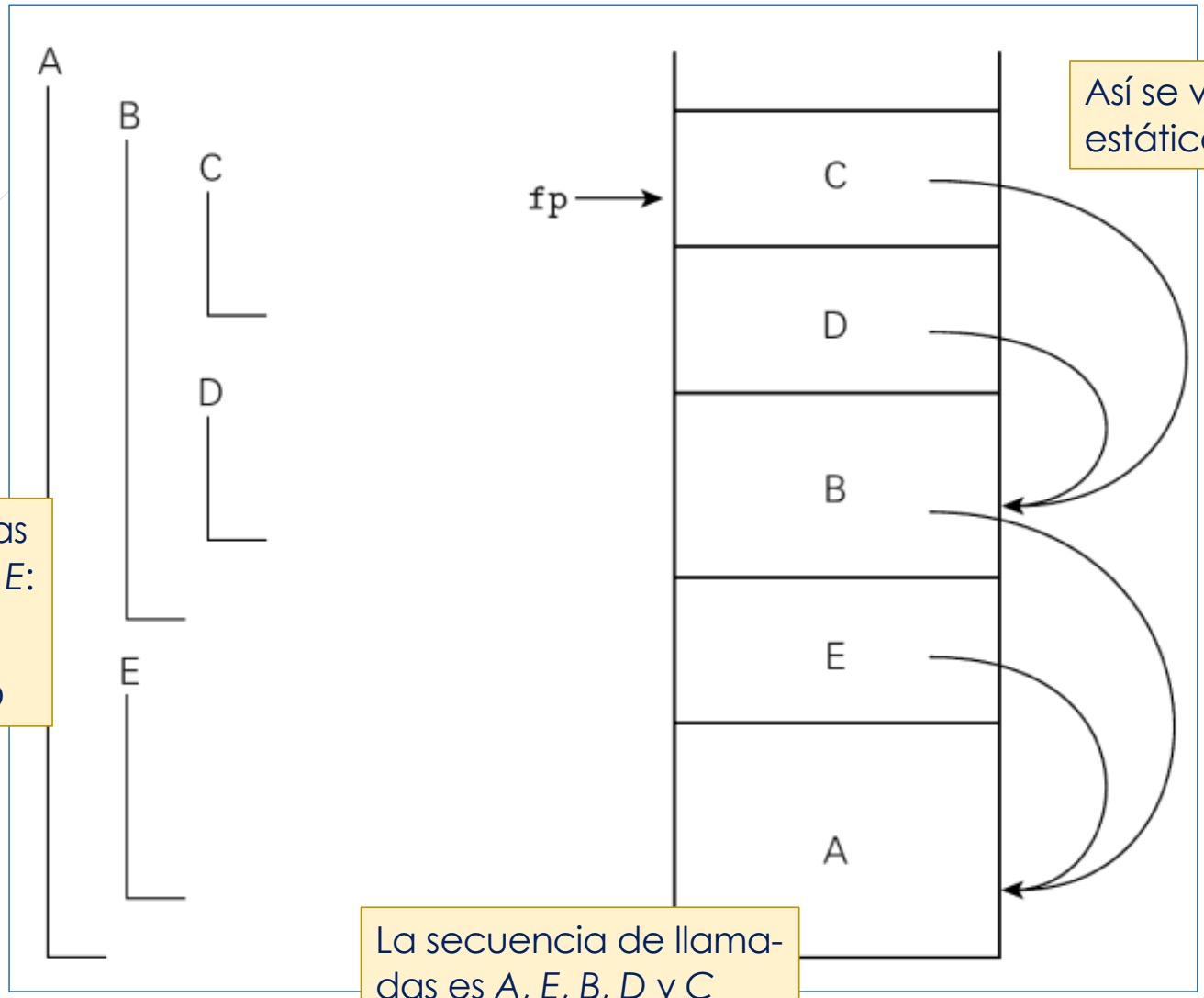
La subrutina léxicamente circundante a F1 es P4

… y la subrutina léxicamente circundante a P4 es P1

Aunque P2 puede llamar a P4 (y vice versa),

… P2 **no es** léxicamente circundante a P4 (y vice versa)





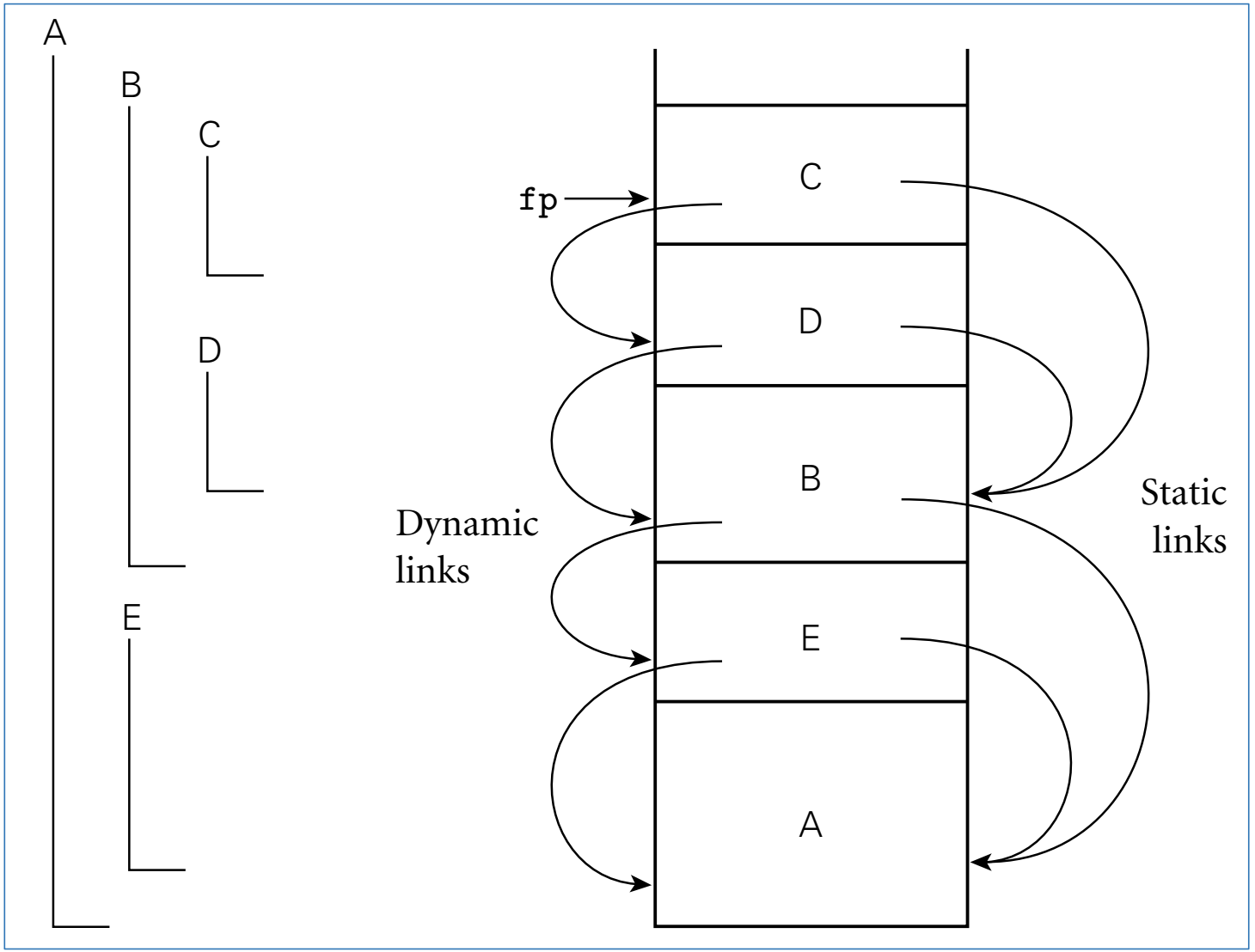
Así se ven los links estáticos en el stack

Anidación léxica de las subrutinas *A*, *B*, *C*, *D* y *E*:

A es léx.circ. a *B* y a *E*

B es léx.circ. a *C* y a *D*

La secuencia de llamadas es *A*, *E*, *B*, *D* y *C*


A
B
C
D
E
fp
C
D
B
E
A
Dynamic
links
Static
links

¿Cómo se mantienen los *static links*?

La subrutina que llama determina el *static link* de la subrutina llamada y lo pasa como un parámetro extra

Hay dos casos:

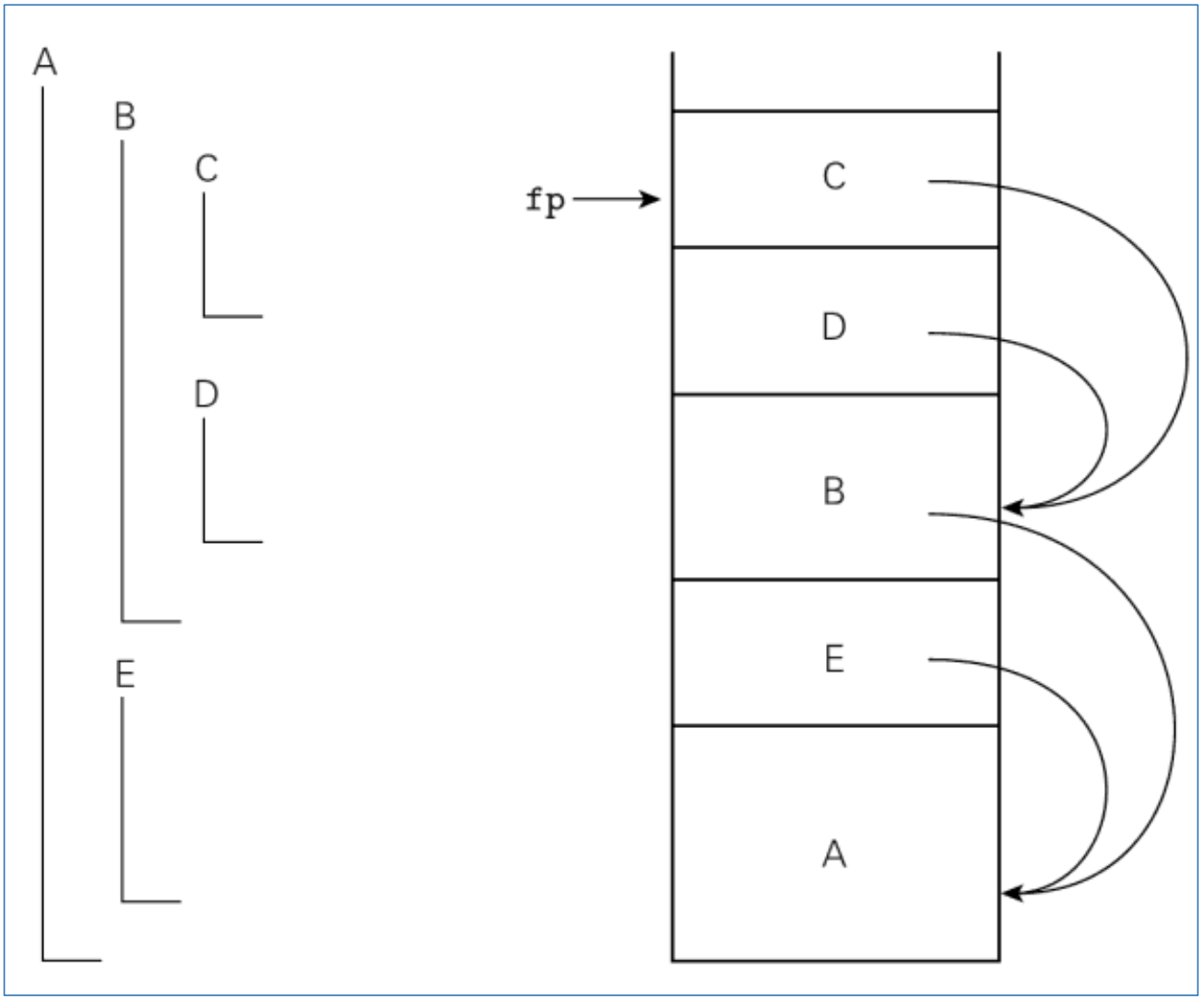
- La subrutina llamada está anidada directamente dentro de la subrutina que llama (la que hace la llamada)
- La subrutina llamada está $k \geq 0$ ámbitos "hacia afuera"

La subrutina llamada está anidada directamente dentro de la subrutina que llama:

- p.ej., *A* llama a *E*; *B* llama a *D*

→ el *static link* de la subrutina llamada debe hacer referencia al *frame* de la subrutina que llama

→ la subrutina que llama pasa su propio **fp** como *static link* de la subrutina llamada



La subrutina llamada está $k \geq 0$ ámbitos "hacia afuera"

- p.ej., *C* llama a *D* ($k = 0$); *C* llama a *B* ($k = 1$)

→ todos los ámbitos que circundan a la subrutina llamada también ciurcundan a la subrutina que llama

→ la subrutina que llama "desreferencia" su propio *static link* k veces y pasa el resultado com *static link* de la subrutina llamada

