4º Ingeniería Informática

II26 Procesadores de lenguaje

Organización y gestión de la memoria

Esquema del tema

1. Introducción
2. Organización de la memoria en tiempo de ejecución
3. Memoria estática
4. Memoria de pila
5. Memoria con reserva dinámica
6. Otros aspectos de las subrutinas
7. Resumen

1. Introducción

Una vez terminadas las fases de análisis, comienza la generación de código. Como paso previo a la generación, en este tema veremos cómo se organiza la memoria de los programas. Nos centraremos en dos aspectos fundamentales: la gestión de las variables globales y de la memoria asociada a las subrutinas. Además, comentaremos algunos aspectos de la gestión de la memoria dinámica.

2. Organización de la memoria en tiempo de ejecución

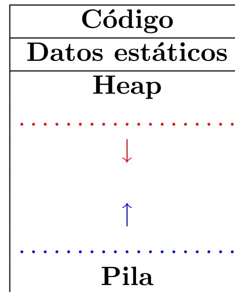
Podemos distinguir tres tipos de memoria. En primer lugar, está la ocupada por el código del programa. Por otro lado, está la destinada a datos estáticos, que se gestiona en tiempo de compilación. Finalmente, tenemos la memoria que debe gestionarse en tiempo de ejecución.

Hay poco que decir sobre la memoria ocupada por el programa. Su tamaño es conocido en tiempo de compilación y, en la mayoría de los lenguajes modernos, se puede decir que es "invisible": el código no puede referirse a sí mismo ni modificarse.

La memoria destinada a datos estáticos se utiliza para las variables globales y algunas otras como las variables estáticas de las subrutinas. La gestión de esta memoria es sencilla y se puede realizar en tiempo de compilación.

Al hablar de la memoria gestionada en tiempo de ejecución, podemos distinguir entre la memoria que se utiliza para albergar los objetos que se crean y destruyen con la ejecución de las subrutinas (parámetros, variables locales y algunos datos generados por el compilador) y la que se suele conocer como memoria dinámica, que se reserva explícitamente por el programador o que se necesita para almacenar objetos con tiempos de vida o tamaños desconocidos en tiempo de compilación. La memoria asociada a las subrutinas será gestionada mediante una pila. La memoria dinámica se localizará en un *heap*.

Normalmente, la memoria se organiza de una manera similar a:



Es habitual que la pila crezca "hacia abajo"[1] debido a que muchos procesadores tienen instrucciones que facilitan la creación de pilas "descendentes". Dado que la pila puede colisionar con el *heap*, el compilador deberá generar código para que en caso de colisión el programa pueda pedir memoria adicional al sistema o terminar adecuadamente. Normalmente, el sistema operativo ofrece ayuda tanto para la detección de la colisión (suele hacerlo con ayuda del hardware) como para facilitar el aumento de tamaño del bloque. Obviamente, la organización dependerá en última instancia del sistema operativo que es el que determina el modelo de proceso que se utiliza.

Sin embargo, en los restantes ejemplos supondremos que la pila crece hacia arriba[2] ya que esta es la forma más cómoda de implementarla en la máquina virtual que utilizamos en prácticas.

3. Memoria estática

La gestión de la memoria estática es la más sencilla. De hecho, algunos lenguajes de programación antiguos, como las primeras versiones de FORTRAN, únicamente tienen reserva estática de memoria. Las principales ventajas son la sencillez de implementación y que los requerimientos de memoria del programa son conocidos una vez compilado el mismo.

Sin embargo, existen bastantes inconvenientes:

- El tamaño de los objetos debe ser conocido en tiempo de compilación: no se puede trabajar con objetos de longitud variable.
- Es difícil para el programador definir el tamaño de las estructuras que va a usar: si son demasiado grandes, desperdiciará memoria; si son pequeñas, no podrá utilizar el programa en todos los casos.
- Sólo puede haber una instancia de cada objeto: no se pueden implementar procedimientos recursivos.

Por estas razones, casi todos los lenguajes de programación tienen además la posibilidad de gestionar parte de la memoria de manera dinámica.

La gestión de esta memoria se puede hacer íntegramente en tiempo de compilación. Para acceder a un objeto, el programa únicamente necesita saber en qué dirección se encuentra. Estas direcciones se pueden asignar secuencialmente. Basta con que el compilador tenga anotada la primera dirección libre del bloque de memoria estática y la vaya actualizando sumando la talla de los objetos que va reservando.

Supongamos que A es la primera dirección libre para datos globales. Las acciones que tiene

[1] También es habitual que "hacia abajo" signifique "hacia arriba" en el dibujo: la dirección 0 ocupa la parte superior.

[2] Y haremos los dibujos con arriba en la parte superior `:-)`.

que realizar el compilador para asignar las direcciones son:

Algoritmo Asignación estática
  dlibre := A;
  para toda declaración global d **hacer**
    $d.dir$:= *dlibre*;
    dlibre := *dlibre* + $d.talla$;
  fin para toda
Fin Asignación estática

Respecto a este algoritmo, hay que aclarar una serie de cosas. En primer lugar, el valor de A dependerá de la arquitectura de la máquina que vayamos a utilizar. Si el espacio de direcciones se dispone como hemos comentado al comienzo del tema, el valor inicial deberá ser la dirección inmediatamente posterior al código. Dado que la longitud de éste no se conoce hasta el fin de la compilación, puede ser necesario generar las direcciones relativas a una etiqueta y dejar que el ensamblador o el enlazador las resuelva adecuadamente. Por otro lado, si la máquina permite que los datos estén en su propio segmento o se va a utilizar un enlazador que los pueda mover, lo habitual es comenzar por la dirección cero.

En cuanto al momento de ejecutar el algoritmo, no hace falta que lo hagamos en una fase separada. Si tenemos *dlibre* como una variable global (o accesible en algún módulo del compilador) podemos realizar la actualización al crear el nodo correspondiente a la declaración. Es más, puede suceder que de esta manera no necesitemos crear nodos en el AST para las declaraciones, bastando con los objetos que se creen para albergar la información acerca de los identificadores.

4. Memoria de pila

En esta sección supondremos que un programa se compone de una serie de subrutinas, entre las cuales hay una "principal". Si la subrutina devuelve un valor, la llamamos *función* y, si no, *procedimiento*. Una *definición de subrutina* es una declaración que asocia un identificador (*nombre* de la subrutina) a un *perfil* (tipo de los parámetros y valor de retorno) y una serie de sentencias (*cuerpo* de la subrutina). Si el nombre de una subrutina (con los parámetros que necesite) aparece en una sentencia ejecutable, decimos que se *llama* a la subrutina, y eso implica la ejecución de su cuerpo. Quien llama es el *llamador* (*caller*) y el otro es el *llamado* (*callée*).

Ciertos identificadores declarados en una subrutina son especiales: los *parámetros formales*, que recibirán valores concretos durante la ejecución mediante los *parámetros de hecho* (*actual parameters*) o *argumentos*. Denominamos *activación* de una subrutina a cada una de sus ejecuciones.

4.1. Activación y tiempo de vida de las subrutinas

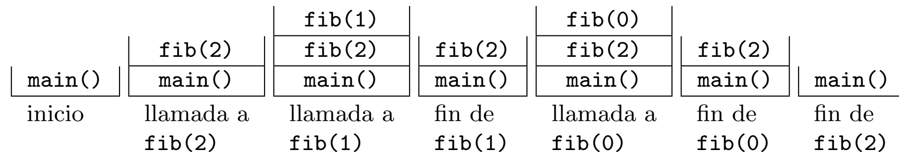
Es habitual suponer que las subrutinas se ejecutan de modo que su ejecución comienza por el inicio de su cuerpo y que, al finalizar la ejecución, el control regresa al punto desde el que la subrutina fue llamada. Esto implica que, en un momento dado de la ejecución del programa, tendremos una serie de subrutinas activas de modo que cada una ha llamado a la siguiente y la última es la que está en ejecución. Cuando esta termine, devolverá el control a la anterior y así sucesivamente. Esta forma de actuar sugiere que podemos representar las activaciones de las subrutinas mediante una pila.

Por ejemplo, sea el siguiente programa:

```
int fib(int n) {
  if (n <= 1) return 1;
  else return fib(n-1)+fib(n-2);
}
main() {
  fib(2);
}
```

© Universitat Jaume I 2006-2007

Concentrándonos en la llamada a `fib(2)`, podemos obtener una representación similar a la siguiente:



La memoria asociada a las subrutinas reflejará esta estructura. Para eso, el código generado gestionará una pila que mantendrá la información relativa a todas las subrutinas activas en un momento dado de la ejecución.

4.2. Organización de la pila

La pila está dividida en una serie de celdas que contienen información relativa a la activación: dirección de retorno, variables locales, argumentos... Llamaremos *registros o tramas de activación* a estas celdas. Cada registro se apila (se crea) cuando se activa una subrutina, y se desapila (se destruye) cuando finaliza la activación. En el momento de la llamada, se debe:

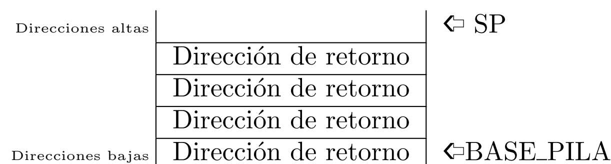
- reservar espacio para los *parámetros* (y valor de retorno, si es pertinente),
- guardar *información de estado* (dirección de retorno, valores de registros, etc.) que permita reanudar la ejecución cuando finalice la activación,
- reservar espacio para las *variables locales*, cuyo tiempo de vida está incluido en el tiempo de vida de la activación.

La forma exacta de cada los registros de activación dependerá de muchos factores: la presencia de instrucciones especializadas en el procesador, el sistema operativo, la necesidad de enlazar con bibliotecas o programas escritos en otros lenguajes, etc.

Empezaremos por preocuparnos únicamente de cómo "saltar" a una subrutina y cómo retornar al punto desde el que la habíamos llamado una vez finaliza su activación. Complicaremos progresivamente los registros de activación añadiéndoles información. Si la pila va de las direcciones bajas de memoria a las altas, reservar espacio de pila es sumar una cantidad positiva a la cima de la pila y liberar espacio de pila es restar una cantidad positiva.

La pila empieza en una dirección fija (típicamente donde finaliza la zona de memoria para variables estáticas) que denominaremos BASE_PILA. Usaremos un registro[3], el `SP` (por *stack pointer*), para apuntar *en tiempo de ejecución* a la cima de la pila. El compilador debe generar código que gestione la pila. (Casi todos los ordenadores tienen instrucciones que facilitan la gestión eficiente de la pila.)

Primero vamos a preocuparnos del caso en que no hay parámetros ni variables locales. En esta situación, el registro de activación contiene sólo una *dirección de retorno*, es decir, la dirección a la que debe regresar el flujo de control cuando finalice la activación de la subrutina.



Utilizaremos un pseudo-código fácilmente traducible a código de máquina para describir las acciones que debemos ejecutar cuando se activa o se desactiva una subrutina. Con `PC` denotamos el contador de programa. El código correspondiente a la activación (llamada) de la subrutina sería:

[3] Si no tenemos registros, utilizaremos una posición fija de memoria.

```
0[SP] := PC + 3     # Apilamos la dirección a la que regresar
SP := SP + 1
SALTO subrutina     # Saltamos a la subrutina
```

Mientras que el de desactivación sería:

```
SP := SP - 1        # Restauramos la pila
SALTO 0[SP]         # Volvemos
```

El código de activación se denomina *secuencia de llamada*, y el de desactivación, *secuencia de retorno*. ¿Dónde deben escribirse las secuencias de llamada y retorno? ¿En el llamador o en el llamado? Obviamente, la secuencia de retorno se escribe en la propia subrutina (llamado) y se denomina *epílogo*.

En principio, parece que la secuencia de llamada debe estar en el llamador, pero interesa que la mayor parte esté en la propia subrutina (de la cual sólo hay una copia en el código objeto), es decir, en el llamado. En otro caso, cada llamada a una subrutina (y puede haber muchas) obliga a escribir una copia de la secuencia de llamada. Aun así, es inevitable que parte de la secuencia de llamada esté en el llamador. La parte que escribimos en el llamado se denomina *prólogo*.

Dependiendo de las instrucciones disponibles en nuestra máquina, podremos tener distintas divisiones de la activación entre el llamador y el llamado. Por ejemplo, si tenemos las instrucciones `JAL`, que guarda el valor del `PC` en el registro `RA`, y `JR`, que salta a la dirección almacenada en un registro, la activación quedaría:

- Acciones del *llamador* (instrucción `JAL subrutina`):

  ```
  RA := PC+2
  SALTO subrutina
  ```

- Prólogo (en el *llamado*):

  ```
  0[SP] := RA
  SP := SP + 1
  ```

Y el retorno:

- En el *llamado* (instrucción `JR RA`):

  ```
  SP := SP - 1
  RA:= 0[SP]
  SALTO RA   # JR RA
  ```

Por ejemplo, dado el programa C:

```
int a;
void f(void)    { a=1;}
void main(void) { f(); a=2;}
```

El resultado de la compilación sería:

```
Inicio:   SP := BASE_PILA    # Inicialización
          JAL rut_main       # Llamada a main
          FINAL

rut_f:    0[SP] := RA        # Prólogo
          SP := SP + 1       # Prólogo
          a := 1
          SP := SP - 1       # Epílogo
          RA:= 0[SP]         # Epílogo
          JR RA              # Epílogo

rut_main: 0[SP] := RA        # Prólogo
          SP := SP + 1       # Prólogo
          JAL rut_f          # Llamada a f
          a := 2
          SP := SP - 1       # Epílogo
          RA:= 0[SP]         # Epílogo
          JR RA              # Epílogo
```

© Universitat Jaume I 2006-2007

4.3. Paso de parámetros

Para se realmente útiles, las subrutinas deben poder recibir información mediante el uso de parámetros. Los dos tipos más comunes de paso de parámetros son:

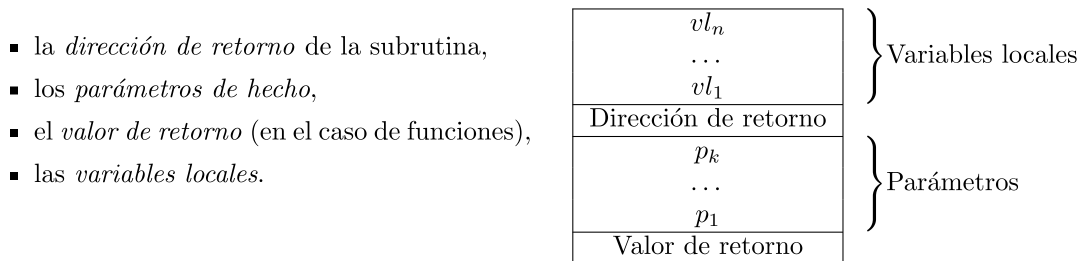
Paso por valor: es el método más sencillo. Los parámetros de hecho se evalúan y los valores resultantes se pasan a la subrutina. Las modificaciones que se hagan a los parámetros formales en la subrutina no afectan a los parámetros de hecho.

Paso por referencia (o por dirección, o por lugar): Se pasa la dirección de memoria donde se encuentra la información que pasamos.

En el caso de las funciones, podemos considerar el valor de retorno un parámetro más.

Por otra parte, muchos lenguajes permiten declarar variables con tiempo de vida limitado por el tiempo de activación de una subrutina: las *variables locales*. Dado que el tiempo de vida de las variables locales está incluido en el tiempo de activación, conviene ubicar la memoria que ocupan en el registro de activación.

Una posible organización es hacer que la trama de activación contenga, al menos:

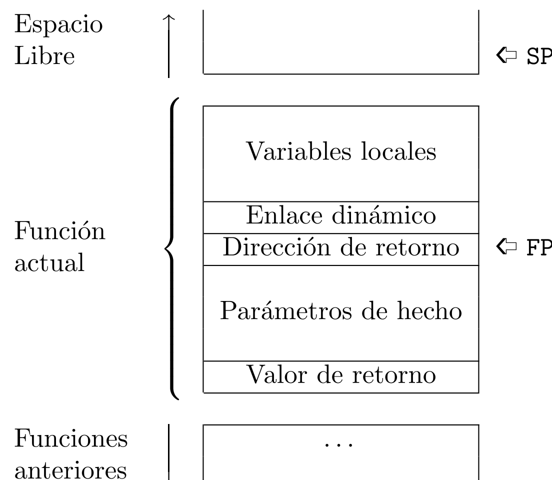


- la *dirección de retorno* de la subrutina,
- los *parámetros de hecho*,
- el *valor de retorno* (en el caso de funciones),
- las *variables locales*.

El valor de retorno puede no estar en la trama de activación. Un convenio habitual es utilizar un registro para devolverlo si su tamaño es "pequeño" (cabe en un registro). También es posible pasar algunos parámetros mediante registros. Para simplificar, en lo que sigue asumiremos que todo pasa a través de la trama de activación.

Para poder acceder a las variables locales y a los parámetros de la subrutina, utilizamos un registro especial, el `FP` (de *Frame Pointer*, puntero de trama). La idea es acceder uniformemente a variables locales y parámetros calculando su dirección a partir del desplazamiento respecto de la dirección apuntada por `FP`. Así, la tabla de símbolos trata de modo similar los parámetros y las variables locales: registra su posición en memoria con un *desplazamiento* sobre el registro `FP`, y no con una dirección absoluta.

La pila tiene un aspecto similar a:



El enlace dinámico es el valor del `FP` al entrar en la subrutina y se utiliza para restaurarlo al salir de ella.

Un posible convenio es hacer que se acceda a las variables locales con desplazamientos positivos sobre `FP`, y a los parámetros con desplazamientos negativos sobre `FP`.

Cuando desactivamos un registro, `FP` pasa a apuntar donde diga el *enlace dinámico*, que contiene el valor de `FP` que teníamos antes de activar la subrutina que ahora desactivamos. Este orden se

aplica si la pila crece “hacia arriba”. Observa que con este orden, el llamador no necesita preocuparse de las variables locales de la subrutina para colocar los parámetros, le basta con irlos apilando. El llamado puede después preparar la pila a su gusto. Lógicamente, si la pila crece “hacia abajo”, el orden se invierte.

Por ejemplo, el registro de activación para

```
int f(int a, int b) {
  int c, d;
  ...
}
```

será de la forma

d
c
Enlace dinámico
Dirección de retorno
a
b
Valor de retorno

4.4. Generación de código

La tabla de símbolos debe registrar cierta información sobre cada función definida en un programa y utilizarla cada vez que se llama a la función, tanto para hacer comprobación de tipo (de argumentos y valor de retorno) como para generar código que lleve a cabo la gestión de la pila de activaciones.

Cuando detectamos una función `f` anotamos en la tabla de símbolos:

- el tipo del valor de retorno y su tamaño,
- el tipo de cada parámetro formal, su tamaño y su “distancia” a `FP`,
- finalmente, anotamos también el tamaño total del registro de activación que corresponderá a `f`; este valor es la suma de los tamaños anteriores, del tamaño de otros campos del registro (la dirección de retorno, el enlace dinámico, etc.) y del tamaño total de la variables locales definidas en la función.

Aparte, para cada parámetro y variable local anotamos una entrada con su nombre y la información que necesitamos de ella (tipo, dirección, etc.).

Un aspecto que no hemos tratado es qué hacer con los registros que estén en uso en el momento de la llamada. Por ejemplo, al generar código para la expresión `f(a,b)*f(c,d)`, se guardará el resultado de `f(a,b)` en un registro. Dado que este registro puede ser utilizado por la llamada `f(c,d)`, habrá que asegurarse de que tenga el valor correcto a la hora de hacer la multiplicación. Tenemos varias posibilidades. El convenio *caller-saves* estipula que el llamador guarda cualquier registro que le interese conservar. Otro convenio es el *callee-saves*, con el que el llamado guarda aquellos registros que va a modificar. También se pueden mezclar ambos convenios, de esta manera algunos registros son responsabilidad del llamado y otros del llamador. Por simplicidad, supondremos que el llamador guarda en la pila los registros que quiera conservar.

El código correspondiente a una llamada a la función `f` hace lo siguiente:

- Guardar los registros activos.
- Calcular el valor o dirección de cada parámetro de hecho y apilarlo en el registro de activación.
- Apilar la dirección de retorno y saltar a la subrutina.
- Recuperar los registros activos.

El código de la función incluirá un prólogo que debe realizar las siguientes acciones:

- Rellenar la información del enlace dinámico en el registro de activación.

© Universitat Jaume I 2006-2007

- Actualizar `SP` (sumándole el tamaño total del registro de activación) y `FP`.

El código generado para una llamada a la función `f` de la sección anterior sería:

```
...              # Guardar registros
...              # Cálculo de a
0[SP] := a
SP := SP + 1
...              # Cálculo de b
0[SP] := b
SP := SP + 1
JAL rut_f        # Llamada a f
...              # Recuperar registros
```

Puedes pensar que podemos ahorrar alguna instrucción si agrupamos los incrementos del `SP` al final o, mejor aún, si el incremento se realiza en el llamado. Es cierto, pero de hacerlo así se nos complica la generación de código por dos aspectos. El más sencillo de solucionar es que el almacenamiento en la pila ya no se hace con un desplazamiento 0 sino que hay que ir cambiándolo en cada parámetro. El problema más complejo es que puede suceder que tengamos llamadas anidadas (por ejemplo `f(f(1,2),3)`), lo que, si no se tiene mucho cuidado con el valor de `SP`, puede hacer que el registro de activación de la llamada interna *machaque* el que estamos construyendo.

El prólogo de `f` sería:

```
rut_f:  0[SP] := RA      # Guardamos RA
        1[SP] := FP      # Guardamos FP
        FP := SP         # Nuevo FP
        SP := SP + 4     # Incrementamos SP
```

Y el epílogo:

```
SP := SP - 7     # Restauramos SP
RA := 0[FP]      # Recuperamos RA
FP := 1[FP]      # Restauramos FP
JR RA            # Volvemos
```

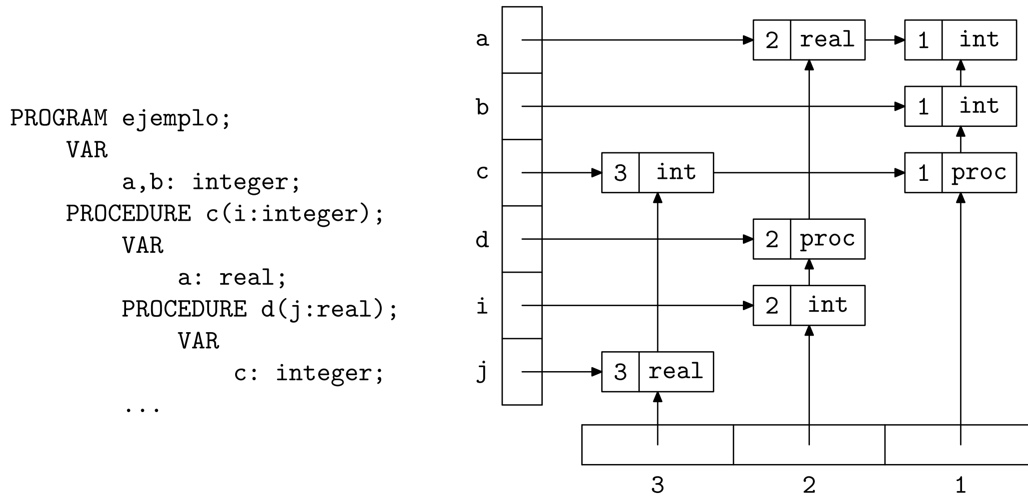
Es interesante darse cuenta de la asimetría entre lo que sumamos a `SP` en el prólogo y lo que restamos en el epílogo. Esto es debido a que podemos aprovechar el epílogo para *limpiar* los parámetros de hecho. También puedes darte cuenta de que esta asimetría es la que hace que usemos `FP` en lugar de `SP` para restaurar la dirección de retorno y el propio `FP`.

4.5. Reflejo de los ámbitos en la tabla de símbolos

Para poder almacenar la información que hemos comentado en la sección anterior, es necesario que la tabla de símbolos sea capaz de reflejar los ámbitos del programa. Lo habitual es que las reglas de ámbito sigan la estructura en bloques del programa: las declaraciones que se encuentran en un bloque son válidas únicamente en ese bloque y ocultan otras declaraciones para los mismos identificadores que estén fuera. Esto sugiere añadir a las operaciones de la tabla dos más: **entrabloque** y **salbloque**. La primera la utilizaremos cuando entremos en un bloque para señalar que las nuevas declaraciones estarán dentro de él. La segunda es la que se encargará de eliminar las declaraciones contenidas en el bloque y que dejan de tener efecto.

La forma de funcionamiento de la tabla sugiere una organización de la información de los identificadores similar a una pila. Tendremos una tabla *hash* de manera que la información de cada identificador será un puntero a una pila. Al encontrar una declaración, la información adecuada se almacenará en el tope de la pila. Al salir del bloque correspondiente, se eliminarán todas las declaraciones creadas dentro del bloque. La entrada en el bloque consistirá en crear una lista que enlazará las entradas que se creen en el bloque para facilitar su borrado a la salida.

Aquí podemos ver la situación de la tabla en un momento del análisis de un programa:



4.6. Bloques

Otra manera de crear ámbitos es el empleo de variables locales a *bloques*. Un *bloque* es una sentencia que contiene declaraciones locales propias. Los bloques pueden anidarse. Por ejemplo:

```
main() {                      /* Inicio bloque 0 */
  int i;
  for(i=0; i<10; i++) {       /* Inicio bloque 1 */
    int j;
    for(j=0; j<10; j++) {     /* Inicio bloque 2 */
      printf("%d %d\n",i,j);
    }                         /* Final bloque 2 */
    for(j=0; j<3; j++) {      /* Inicio bloque 3 */
      printf("%d %d\n",i,j);
    }                         /* Final bloque 3 */
  }                           /* Final bloque 1 */
}                             /* Final bloque 0 */
```

Muchos lenguajes siguen la *regla del bloque anidado más próximo*: el ámbito de una declaración hecha en un bloque alcanza a todo el bloque y si un nombre se referencia en un bloque B y no está declarado en él, se aplica la declaración del bloque más próximo que contiene a B.

La estructura de bloques puede implementarse mediante reserva de *memoria de pila*: cada bloque puede considerarse una subrutina sin parámetros que sólo se llama una vez. Sin embargo, es más eficiente reservar memoria para todas las variables de la función simultáneamente. Para reducir la ocupación en memoria, se puede asignar la misma dirección a variables que estén en bloques distintos (y no anidados uno dentro del otro).

5. Memoria con reserva dinámica

La memoria que ocupan los objetos con tiempo de vida no predecible a partir del código fuente se gestiona dinámicamente: cuando se crea un objeto *en tiempo de ejecución*, se pide memoria para él, y cuando se elimina, la memoria se libera.

En algunos lenguajes encontramos características que necesitan una gestión de memoria con reserva dinámica:

- Los operadores `new` y `dispose` del PASCAL,

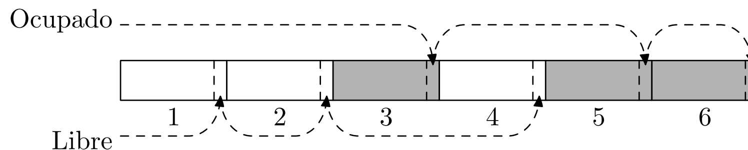
© Universitat Jaume I 2006-2007

- la creación de elementos de listas en LISP y su liberación automática (*garbage collection*) por ausencia de referencias a la memoria,
- las llamadas a funciones de bibiloteca `malloc` y `free` del C,
- la creación de objetos en Java y su eliminación cuando ninguna variable mantiene referencias a dichos objetos,
- los operadores `new` y `delete` o las llamadas implícitas que comportan la creación y destrucción de objetos en C++,
- etc.

La zona de memoria destinada a atender las demandas de memoria dinámica se denomina *heap*. Vamos a estudiar muy por encima el problema de la gestión del *heap*.

5.1. Gestión con bloques de tamaño fijo

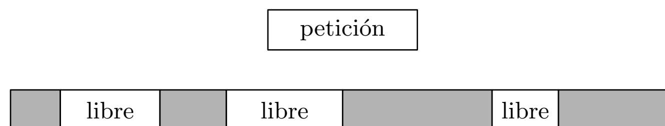
Empezamos por un *supuesto* sencillo: *la memoria se solicita siempre en fragmentos de tamaño fijo*. El *heap* puede estructurarse en dos listas: una que encadena los bloques ocupados y otra que encadena los bloques libres.



Al solicitar un bloque de memoria se nos da el primer bloque libre, que pasa a añadirse a la lista de bloques ocupados. Liberar un bloque lo elimina de la lista de bloques ocupados y lo añade a la lista de bloques libres. Es habitual que la lista de bloques ocupados sea "virtual" si no se necesita hacer ningún recorrido sobre ella.

5.2. Caso general

Es muy restrictivo exigir que los bloques sean siempre del mismo tamaño. Cuando podemos pedir bloques de tamaños diferentes, aparece el problema de la *fragmentación*. Si hay fragmentación puede suceder que no se atiendan peticiones de memoria aún cuando haya suficiente memoria libre debido a que ésta está repartida en bloques excesivamente pequeños:



Para evitar una excesiva fragmentación cuando se libera memoria, hay que mirar si los bloques adyacentes están libres y si es así, unirlos en un bloque libre único.

Un método para reservar bloques de memoria dinámica es el de "*primero donde quepa*" (*first fit*): si se piden b bytes, buscamos el primer trozo de memoria contigua con tamaño B superior a b, la fragmentamos en dos partes, una de b bytes y la otra de $B - b$ bytes, y marcamos la primera como ocupada. Este método es muy costoso: puede obligar a recorrer todo el *heap* con cada petición de memoria.

Hay técnicas muy sofisticadas para reservar/liberar memoria eficientemente, pero estudiarlas con detalle queda fuera de los objetivos del curso. En la bibliografía encontrarás más información.

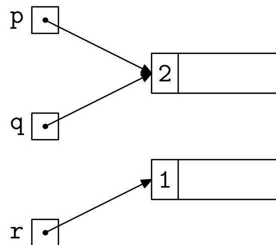
5.3. Liberación automática

Ciertos lenguajes (Java, Python, LISP, etc.) liberan automáticamente la memoria que no se sigue utilizando. *Grosso modo* hay dos técnicas básicas de liberación automática de memoria: uso de contadores de referencia y técnicas de marcado. Aunque ambos métodos tienen sus problemas, esta es un área de investigación muy activa actualmente y existen métodos que hacen que la liberación automática de memoria sea una alternativa muy atractiva con un impacto muy reducido en el tiempo de ejecución.

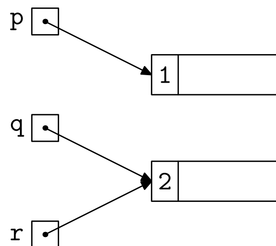
5.3.1. Contadores de referencia

Cada bloque lleva un contador de los punteros que le apuntan. Cada vez que se hace una asignación a un puntero, se decrementa el contador del bloque al que apuntaba y se incrementa el contador del nuevo bloque. Cuando el contador vale cero, el bloque se libera.

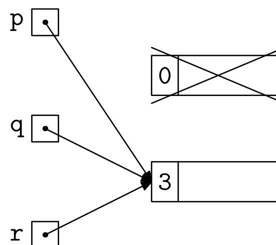
Supongamos que tenemos dos bloques de memoria, el primero está apuntado por `p` y `q` y el segundo por `r`. Los contadores de referencia estarían así:



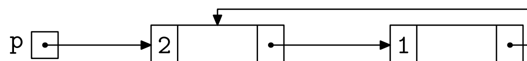
Si se produce la asignación `q:=r`, la situación cambia a:



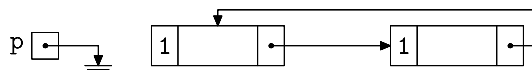
Tras la asignación `p:= q`, el contador del primer bloque se anula y podemos liberarlo:



El problema de los contadores referencia es que no pueden liberar correctamente estructuras circulares. Imaginemos que tenemos una lista circular:



Si hacemos que `p` apunte a `NIL`, nos encontramos con que las cuentas de los bloques no se anulan, aunque sean inaccesibles:



© Universitat Jaume I 2006-2007

5.3.2. Técnicas de marcado

Con las técnicas de marcado, cada cierto tiempo se detiene la ejecución del programa de usuario y se ejecuta una rutina que marca todos los bloques como "libres" para, a continuación, recorrer todos los punteros del programa para ir marcando como "utilizados" los bloques apuntados; finalmente, los bloques que han quedado marcados como "libres" se liberan.

El principal problema de estas técnicas es que si no se implementan con suficiente eficiencia, la detención periódica del programa puede resultar muy molesta para el usuario. Se puede aliviar el problema en parte mediante el empleo de *threads* paralelos que se encarguen del marcado y liberación. De todas formas, la implementación eficiente es muy compleja.

6. Otros aspectos de las subrutinas

Ciertas características de lenguajes de programación modernos pueden complicar bastante la gestión de la memoria asociada a las subrutinas. Comentaremos brevemente, sin entrar en demasiados detalles, algunas de estas características.

6.1. Ámbito dinámico

Hasta el momento hemos considerado la organización de la memoria en lenguajes de programación con *ámbito léxico*, es decir, lenguajes en los que podemos resolver la referencia a parámetros o variables analizando únicamente el programa fuente.

Pero hay lenguajes, como APL o algunos dialectos de LISP, con *ámbito dinámico*, es decir, lenguajes en los que las referencias sólo pueden resolverse en tiempo de ejecución. Por ejemplo, observa la diferencia de resultados en el siguiente programa:

```
program dinamico;
var i: integer;
  procedure escribe;
    writeln(i);
  end;
  procedure mini;
    var i: integer;
    i := 1;
    escribe;
  end;
begin
  i := 2;
  escribe; mini;
end.
```

Con ámbito léxico	Con ámbito dinámico
2	2
2	1

Utilizando el ámbito léxico, la llamada a `writeln` siempre hace referencia a la variable global `i`. Sin embargo, con ámbito dinámico, la llamada a `escribe` dentro de `mini` hace que la `i` del `writeln` sea la local de `mini`.

Implementar correctamente el ámbito dinámico supone mantener en la pila información con los nombres de las variables y recorrerla o bien emplear una estructura separada.

6.2. Subrutinas con parámetros o variables locales de tamaño variable

La técnicas estudiadas reservan en la pila una cantidad de memoria fija cuando se llama a una subrutina. El tamaño de este trozo de memoria puede determinarse en tiempo de compilación, y es el resultado de sumar lo que ocupan los parámetros, lo que ocupan las variables locales y lo que ocupan los campos propios del registro de activación (dirección de retorno, enlaces, etc.).

Si nuestro lenguaje admite parámetros o variables locales de tamaño desconocido en tiempo de compilación (por ejemplo, vectores con rango de índices acotado por variables), no podemos aplicar directamente la estrategia propuesta.

Una solución es representar los objetos de tamaño desconocido en tiempo de compilación mediante punteros a memoria gestionada dinámicamente: los punteros sí tienen tamaño conocido en tiempo de compilación.

6.3. Subrutinas anidadas

El poder anidar subrutinas introduce numerosas complicaciones, especialmente si se asocia con la capacidad de pasar subrutinas como parámetro o devolverlas como resultado.

Casi todos los problemas vienen del hecho de que la subrutina anidada puede acceder a variables locales de la subrutina en la que está. Por ejemplo, sean las funciones (en pseudo-C):

```
int fuera(int m)
  {
    int a;
    int dentro(int n)
      {
        if (n> 1)
          {
            a++;
            dentro(n-1);
          }
        return a;
      }
    dentro(m);
  }
```

El problema aquí es cómo conseguir acceder desde `dentro` al registro de activación de `fuera`. Dado que `dentro` es recursiva, no podemos saber a qué "distancia" está `a`. Hay varias maneras de resolver esto. Una es añadir un campo más al registro de activación de modo que cada subrutina sepa dónde está su padre. Otra posibilidad es mantener una estructura separada (llamada *display*). Finalmente, podemos añadir un nuevo parámetro a `dentro` que contenga la dirección de `a`.

Como hemos comentado, la cosa se complica un poco más si además podemos pasar funciones como parámetros. Entonces es prácticamente obligatorio añadir el parámetro ficticio. Si además las funciones se pueden devolver como resultado o se pueden almacenar en variables, el problema es mucho más complejo: podemos intentar acceder a la variable `a` mucho después de que `fuera` haya dejado de existir. En estos casos, hay que recurrir a reservar dinámicamente memoria para almacenar los objetos que son accesibles desde las funciones devueltas.

6.4. Parametrización parcial

La parametrización parcial (también llamada *currying*) permite definir nuevas funciones especificando el valor de alguno de los argumentos de otras funciones ya existentes. Por ejemplo, en pseudo-C podríamos tener la siguiente definición de la función `suma`:

```
int suma(int a, int b) { return a+b; }
```

Y después podríamos definir la función `incrementa` de forma similar a:

```
incrementa= suma(,1);
```

La manera de tratar esto es reservando con cada función parcialmente parametrizada una lista con los parámetros ya instanciados.

© Universitat Jaume I 2006-2007

7. Resumen

- Tres tipos de memoria:
  - Código.
  - Memoria gestionada en tiempo de compilación.
  - Memoria gestionada en tiempo de ejecución: pila y *heap*.
- Memoria estática:
  - Se gestiona en tiempo de compilación.
  - Sencilla.
  - Limitada.
- Memoria de pila:
  - Para la información de las subrutinas.
  - Registros de activación: parámetros, variables locales, otros.
- Memoria dinámica:
  - Gestión más compleja.
  - Técnicas automáticas: contadores de referencia, técnicas de marcado.
- Aspectos interesantes de las subrutinas:
  - Ámbito dinámico.
  - Parámetros y variables de tamaño variable.
  - Anidamiento.
  - Parametrización parcial.