Lenguajes de Programación.
Capítulo 3.
Variables.

Carlos Ureña Almagro

Curso 2011-12

Contents

1 Introducción

Invención del concepto de variable

Históricamente, el concepto de variable surge en los primeros lenguajes de programación (p.ej.: Plankalkul, ensambladores, Fortran). Se intenta resolver varios problemas del código máquina:

- direccionamiento usando direcciones numéricas
- ausencia de información sobre la interpretación válida de las secuencias de bits en memoria
- ausencia de información semántica.

Direccionamiento usando direcciones numéricas:

- En código máquina, cada zona de memoria donde se almacenan secuencias de bits se identifica a su vez por una secuencia de bits (representa un valor numérico).
- A ese valor numérico se le llama localización de la secuencia de bits almacenada.

Direccionamiento usando direcciones numéricas:

- La dirección de un valor en memoria depende de la distribución (layout) de todos los otros valores en memoria.

Ausencia de información sobre la interpretación válida

- Es posible realizar cualquier interpretación de dichas secuencias. Es fácil cometer errores.

Ausencia de información semántica

- No hay información en el programa sobre la semántica (el significado) de una secuencia de bits almacenada en una posición de memoria

El concepto de Variable

Una variable es una abstracción sobre un valor almacenado en una localización de memoria, usando la representación asociada a un tipo de datos.

El concepto de Variable

- Una variable puede tener un nombre
- El valor almacenado puede cambiar durante la ejecución de un programa.
- La localización puede cambiar durante la ejecución del programa

Las variables como abstracciones

Abstracción aquí significa que podemos olvidarnos de:

- El valor concreto (solo hay que conocer su tipo)
- La localización y tamaño de la zona de memoria donde se almacena la representación del valor
- La representación concreta asociada al tipo de datos

Beneficios de las abstracción

Olvidarnos de todo esto nos libera de la necesidad de ocuparnos de estos aspectos:

- Esto impide que cometamos los errores que se pueden cometer al ocuparnos de ellos (mayor fiabilidad)
- El uso de variables con nombres y tipos facilita la escritura de programas, y aumenta su legibilidad.

Atributos

En un pograma, una variables una entidad que está compuesta de atributos, que son:

- Identidad (un valor de un tipo dependiente de la implementación)
- Nombre (una secuencia de dígitos y letras)
- Localización de memoria (un valor de tipo puntero)
- Valor (una secuencia de bits)
- Tipo (entidad formal, determina la representación)
- Tiempo de vida (intervalo o intervalos de tiempo)

Nombre de las variables

- Es un identificador (una secuencia de dígitos y letras).
- Representa a la variable en el programa fuente
- Permite al programador aportar información sobre el significado del valor almacenado
- Pueden existir variables sin nombre (se denominan anónimas)
- Pueden existir variables con varios nombres (a esos nombres se le llaman alias)

Localización de memoria

- Se puede calcular a partir de la identidad
- Identifica donde está alojado el valor
- Puede cambiar durante el tiempo de vida de la variable.
- Los lenguajes deben permitir al programador abstraerse de este valor

Identidad de las variables

- Es un valor de un tipo dependiente del lenguaje y/o la implementación
- Cada variable del programa tiene asociada una identidad única
- No puede cambiar durante el tiempo de vida de la variable.
- Existe una función que asocia a cada identidad una localización de memoria
- La identidad puede implementarse mediante el tipo puntero (C/C++), mediante un índice en una tabla de variables (podría ser en Java o C#), mediante el nombre de la variable (Python,Perl,Php)

Tipo de las variables

- Determina el conjunto de valores que puede tomar la variable, y las operaciones que pueden hacerse sobre ella.
- Algunos lenguajes fuerzan a asociar un tipo único a cada variable
- Algunos lenguajes permiten variables sin tipo (la secuencia de bits se maneja tal cual)
- Algunos lenguajes permiten que el tipo de una variable cambie durante su tiempo de vida.

Tiempo de vida

La ejecución de un programa ocupa un intervalo de tiempo:

- El tiempo de vida de una variable es el intervalo o intervalos de tiempo durante los cuales la variable existe
- Los intervalos están normalmente incluidos en el intervalo de tiempo de ejecución del programa. A cada uno de ellos se les denomina intervalo de existencia
- Una variable existe durante la ejecución cuando se puede acceder a su valor, en el sentido de que se puede garantizar que dicho valor está almacenado en su localización usando la representación asociada a su tipo.

Ámbito de las variables

- Es la porción o porciones del programa fuente en los cuales la aparición del nombre de la variable es correcta y hace referencia a dicha variable, a sus atributos o a su valor.
- Las variables sin nombre no tienen ámbito.

Otros atributos de las variables

Las variables pueden tener otros atributos, p.ej.:

- Una variable puede ser modificable o no modificable, las segundas no pueden cambiar de valor durante un intervalo de existencia (se les sueles llamar variables de solo lectura.

2 Referencias

Referencias

Sea T un tipo cualquiera:

- En un momento de la ejecución de un programa, podemos considerar el conjunto de las identidades de todas las variables de tipo T que efectivamente existan en ese momento.
- Este conjunto es un tipo, que llamaremos el tipo de las referencias a T, (lo notamos como $ref(T)$)
- En algunos lenguajes, podemos considerar el tipo de todas las referencias, sin tener en cuenta el tipo de la variable referenciada.

La referencia nula

Existe un valor de tipo referencia, especial, llamado la referencia nula, que no es la identidad de ninguna variable, y que llamamos *null*

- Normalmente, no consideraremos un valor nulo entre las referencias

$$null \notin ref(T)$$

  ya que no sirve para acceder a ninguna variable.

- El conjunto $ref^\star$ es el conjunto de todas las referencias a variables de tipo T, incluyendo un valor nulo:

$$ref^\star(T) = ref(T) \cup \{null\}$$

Punteros en C/C++

El lenguaje C/C++ contempla las referencias a un tipo T, es decir, el tipo $ref^\star(T)$, se describen con el siguiente descriptor de tipo `T *` (donde `T` es el descriptor de tipo del tipo T)

- En la bibliografía sobre el lenguaje C/C++, a este tipo de referencias se les llama punteros (pointers)
- Se representan en memoria usando el formato para localizaciones de memoria dictado por el hardware subyacente.
- Se contempla el valor nulo (el cero). Se puede usar el literal entero `0` (se convierte al entero nulo), aunque es más legible usar `NULL` que está definido como `0`

Referencias en C/C++

Los punteros en C/C++ presentan algunos problemas:

- El tipo `T *` incluye todas las referencias válidas, más algunas que no lo son.
- Se incluye todos los valores asociados a localizaciones donde alguna vez se ha almacenado una variable de tipo `T`
- Lo anterior da lugar a frecuentes errores.

Referencias (access) en Ada

- En Ada 83, el tipo de las referencias a variables de tipo `T` se declara como `access T`
- En Ada 95, además, es válido usar `access all T`

Referencias en otros lenguajes

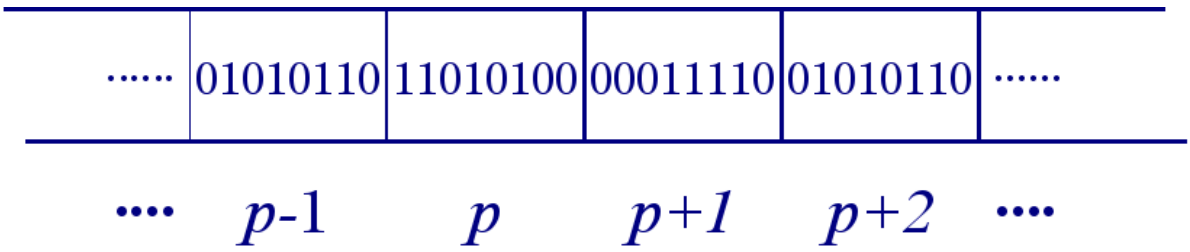
En los lenguajes Java, C#, Python y otros no hay una sintaxis específica para las referencias

- En estos lenguajes, el esquema de almacenamiento de variables (se verá más adelante) dicta que variables son referencias.
- Lo anterior depende esencialmente del tipo de la variable.
- Se hace implicitamente, no hay construcciones explícitas para declarar variables de tipo referencia.

3 Almacenamiento de valores

Las celdas de memoria

- Una celda de memoria es una secuencia contigua de bits de la memoria, del tamaño mínimo que puede ser asociado con una localización (o dirección) única, y transferido entre la CPU y la memoria.
- El conjunto de localizaciones o direcciones de celdas en un conjunto finito ordenado y numerable de valores, cada uno de los cuales esta asociado a una celda de memoria disponible para el programa.



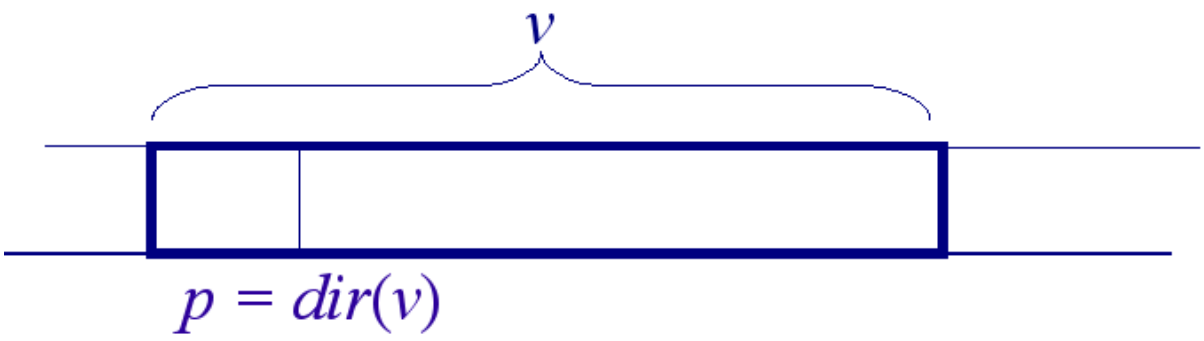
Las celdas de memoria

- El tamaño en bits de las celdas es fijo para cada arquitectura hardware, sistema operativo y compilador o intérprete.
- Este tamaño suele ser múltiplo de 8 (un número entero de bytes).
- El diseño de los lenguajes de programación es independiente del tamaño de las celdas

La dirección de una variable

- Cada variable v de un tipo T ocupará una secuencia de celdas consecutivas en memoria (un bloque)

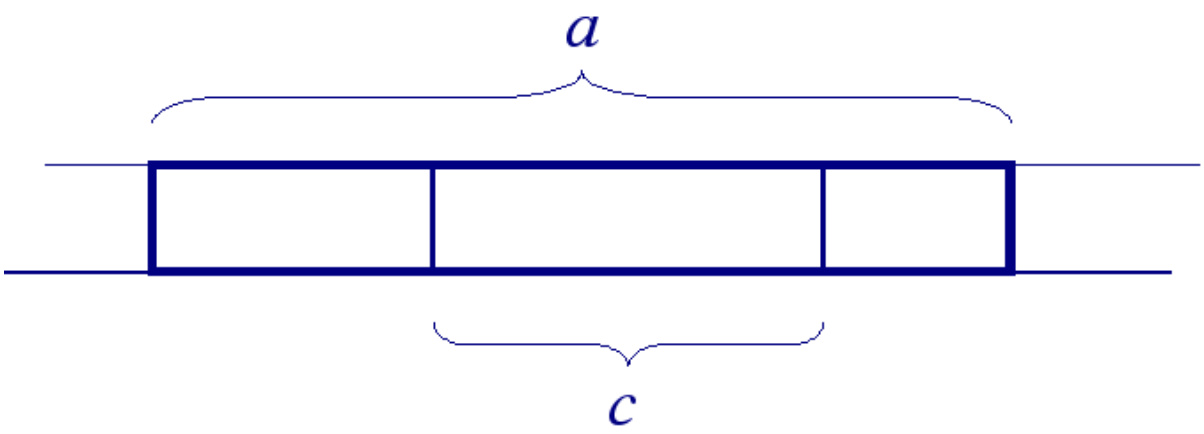
- La localización de la variable (dir(v)) es la localización de la primera celda que ocupa (p en la figura)



3.1 Almacenamiento de tipos compuestos

Almacenamiento de tipos compuestos

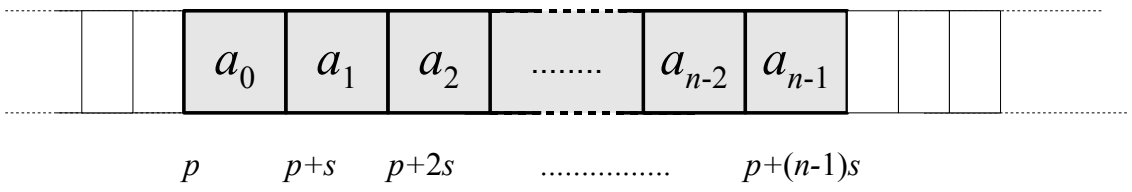
En los casos de las variables de tipos compuestos no recursivos (arrays y registros), podemos decir que una variable (a) incluye otras variables (c), que forman parte de la primera:



Arrays de tamaño fijo

En los diversos lenguajes, los arrays unidimensionales de tamaño fijo se almacenan usando una secuencia consecutiva de bloques de memoria, cada uno con un valor:

En el lenguaje C, los valores se almacenan tal cual:



los valores almacenados (a_i) son de un tipo T, el tamaño total es $s\,n$ donde n es el número de elementos y $s = tam(T)$. El valor n es conocido en tiempo de compilación.

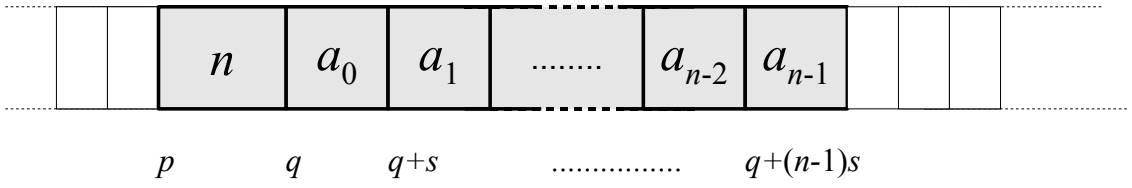
Problemas en los accesos a los arrays

En el lenguaje C, un array de tamaño fijo puede accederse como un array de tamaño variable (dinámico) a través de un puntero al mismo, esto es muy frecuente

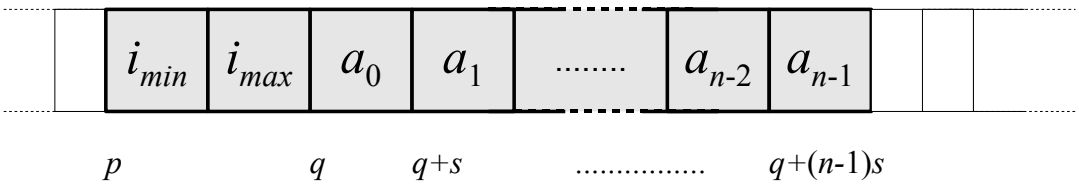
- Esto ocurre, por ejemplo, al pasar un array como parámetro a una función
- Junto con el puntero, es necesario pasar información del tamaño, para que los accesos lo tengan en cuenta
- Esto lo debe de gestionar el programador y puede dar lugar a errores
- Por tanto, en otros lenguaje (p.ej. en Ada o Java), junto con la secuencia de valores se almacena el tamaño siempre (para arrays estáticos y dinámicos)

Representación de arrays en Ada y Java

En Java y C#, la secuencia de valores incluye un prefijo con el número de ellos:



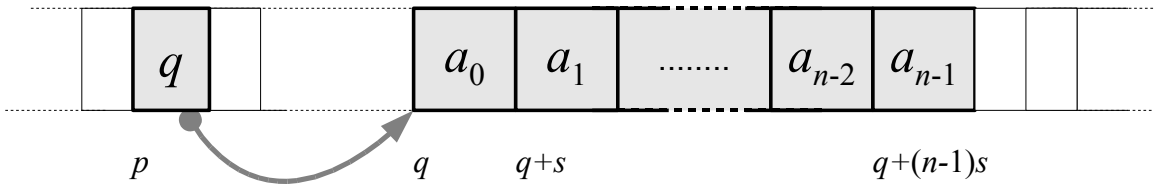
En Ada, siempre se incluye al inicio dos enteros con los indices minimo y máximo del array:



Arrays dinámicos

En la mayoría de los lenguajes, los arrays dinámicos se almacenan usando un puntero (en C) o referencia a la secuencia de valores, que está en bloques de memoria no contiguos a dicha referencia

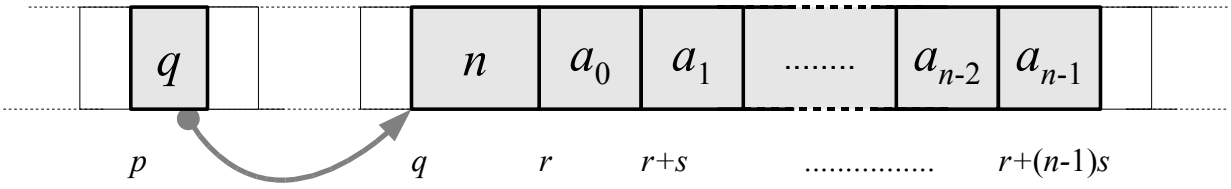
En C/C++ no hay información del tamaño:



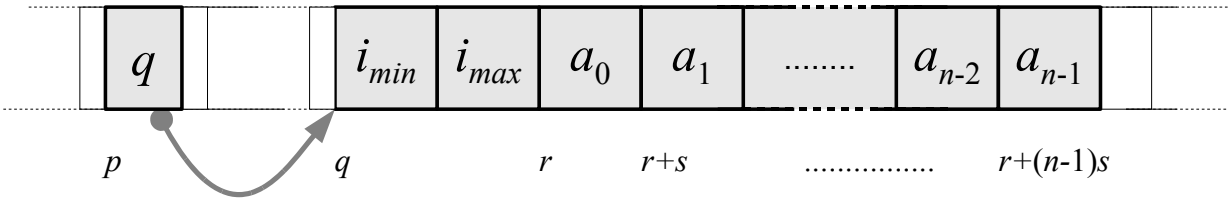
Arrays dinámicos e información de tamaño

En Java y C# se incluye el tamaño, lo cual permite comprobar en cualquier acceso que los índices tienen

valores correctos:



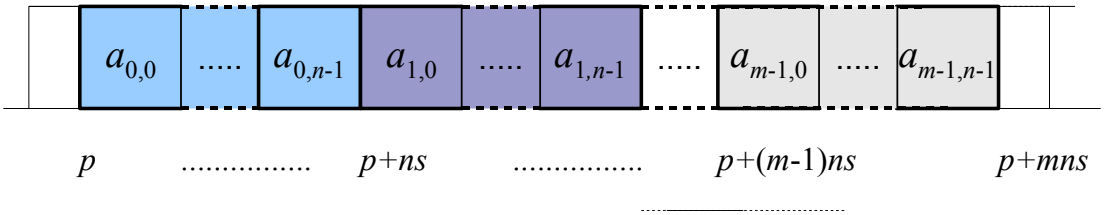
En Ada se incluyen igualmente los valores mínimos y máximos de los índices:



Arrays bidimensionales rectangulares (matrices)

En C/C++ y C# se incorporan los arrays rectangulares, que son vectores cuyos elementos son vectores, todos ellos de igual tamaño. Esto permite almacenarlos en bloques consecutivos de memoria.

En C/C++ el esquema (para una matriz de n columnas y m filas) sería el siguiente:

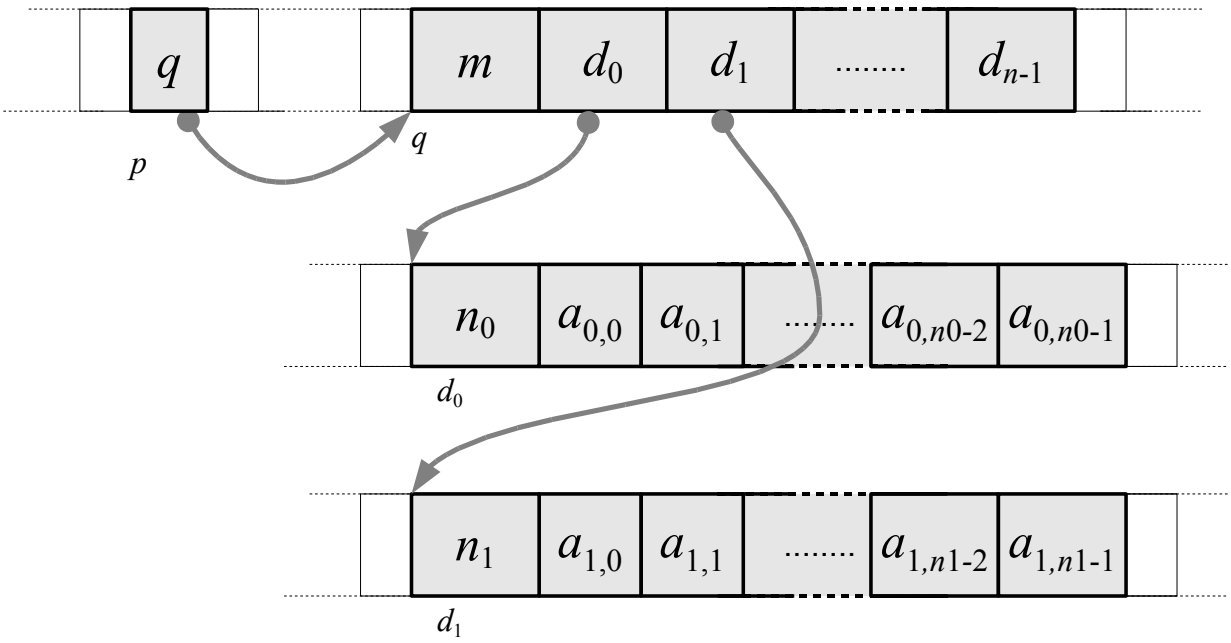


En C# hay una cabecera con el numero de filas y columnas, y en Ada con el rango de indices de las filas y el de las columnas. Esto se puede generalizar a un numero arbitrario de dimensiones.

Vectores de vectores dinámicos (arrays dentados)

Un vector (dinámico), cuyos elementos son a su vez vectores dinámicos se representa como una referencia a

un vector de referencias:



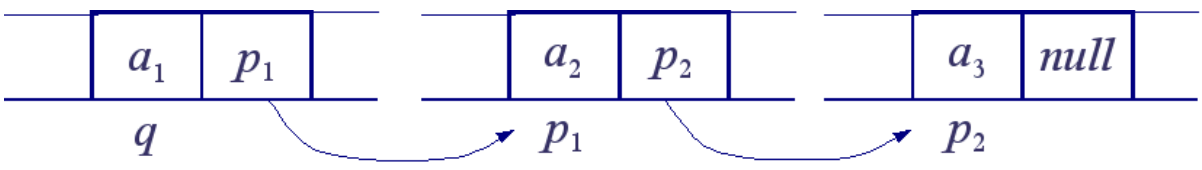
3.2 Almacenamiento de tipos recursivos

Almacenamiento de tipos recursivos

- Una lista o árbol con elementos de tipo T se representa en memoria usando una referencia a un registro.
- El registro (se le suele llamar nodo), tiene un valor de tipo T y una referencia a una lista.
- Para la lista o árbol vacío se usa la referencia nula.

Almacenamiento de tipos recursivos

Como resultado, los tipos recursivos se almacenan en bloques no consecutivos de memoria. Por ejemplo la lista $q = (a_1, a_2, a_3)$ siguiente:



4 Tipos de variables y tiempo de vida

Tipos de variables y tiempo de vida

Las variables (que no forman parte de otras) se pueden clasificar por su tiempo de vida y por su ámbito en varios tipos:

- Variables con nombre
  - Globales
  - Locales (variables en la pila o stack)
  - Estáticas locales
- Variables en el heap (montón).

Variables con nombre.

- Son variables que tienen asociado un nombre o identificador
- Este identificador aparece en texto del programa
- Estas variables se pueden acceder directamente usando dicho nombre, o en algunos casos a través de referencias (aliasing).

Variables globales

- Su ámbito es todo el programa
- Su tiempo de vida coincide con el tiempo de ejecución del programa

Se contemplan en C/C++, pero no en Ada, Java o C#, ya que se considera que aumentan la cohesión y se pueden producir errores

Variables globales (C/C++)

Su ámbito puede ser:

- una única unidad de compilación (archivo .c)
- varias unidades:
  - en una aparece declarada sin la palabra clave `extern`
  - en las demás aparece declarada con la palabra clave `extern`

Ejemplo de vars. globales en C/C++

En este ejemplo, `i` es una variable global, cuyo ámbito es todo el archivo fuente donde aparece:

```
int i ;
void f1() { ..... }
void f2() { i = 3 ; ..... }
int main( int argc, char * argv ) { i = 4 ; .... }
```

Además podriamos usarla desde otros archivos fuente que se enlazaran con el anterior

```
extern int i ;
void f4() { ..... }
void f5() { i = 0 ; ..... }
```

Variables locales (en la pila, stack variables)

- Su ámbito esta restringido a una sentencia o a un subprograma
- Su tiempo de vida coincide con los intervalos de tiempo en los cuales se está ejecutando la sentencia o subprograma
- Se crean y se destruyen automáticamente al entrar o salir del subprograma o sentencia

Se contemplan en la mayoría de los lenguajes

Variables locales en C/C++

- En C van solo al inicio de un bloque

```
if ( x > 3 )
{   int j = x ;
    x = x - j/2 ;
}
```

- En C++ pueden ir donde cualquier sentencia (el ámbito es hasta el final del bloque)

```
if ( x < 3 )
{   x = x -x/2 ;
    int j = x ; ....
}
```

Variables locales en Java y C#

La sintaxis similar a C++, también pueden declararse en lugar de cualquier sentencia (el ámbito es hasta el final del bloque)

```
void p()
{  int a ;
   ...
   while ( ... )
   {  float b ;
      ...
      {  char b ;
         ...
      }
   }
   ...
}
```

Variables locales en Ada

Pueden aparecer como locales a un procedimiento o función

```
procedure p() is
   a : Integer ;
   j : Float ;
begin
   ....
   ....
end ;
```

Variables locales en Ada

o bien locales a cualquier bloque:

```
while ... loop
declare
   b : Float ;
begin
   ...
   declare
      b : Character ;
   begin
      ...
   end ;
   ...
end loop ;
```

Variables locales en Python

- Todas las variables creadas (esto es, asignadas por primera vez) en un subprograma son locales a ese subprograma
- En princpio, cualquier variable global con el mismo nombre que pudiera existir deja de ser accesible desde el subprograma.
- Este mecanismo no esta disponible para bloques en sentencias if o bucles.

Ejemplo de vars. locales en Python

En este ejemplo, hay involucradas dos variables distintas de nombre `c`, una global, y otra local al subprograma `subprog`

```
c = 0               # 'c' global
def subprog():
      c = 10        # 'c' local
      c = c+1       # 'c' local
      print c       # escribe 11

c = c+1             # 'c' global
print c             # escribe 1
```

Variables locales globales en Python

- Si en un subprogramas quiere acceder a una variable global `v` y existe una local con el mismo nombre, en principio es necesario usar la sentencia `global v`
- Esta sentencia, al ejecutarse, indica que la siguiente asignación al nombre `v` en el subprograma tiene como destino la variable global, no la local (si dicha var. global no existe, se crea).
- A partir de la asignación, el nombre queda ligado a la global para lectura y escritura.

Problemas con `global` en Python

El uso de `global` puede dar lugar a comportamiento inesperado:

```
c = 10 # 'c' global

def test5():
   c = 0   # 'c' local = 0 (genera advertencia)
   global c
   c = c+1   # 'c' global = 'c' local + 1
   print "c == ", c # imprime 'c' global ("1")
```

En general, es mejor acceder a las variables globales a traves parámetros (ya veremos como).

Variables estáticas locales

- Su ámbito esta restringido a una sentencia o a un subprograma (en algunos lenguajes, su ámbito puede ser una clase o un módulo, se verá más adelante)
- Su tiempo de vida coincide con el tiempo de ejecución del programa
- Se contemplan en C/C++ (en Ada sólo pueden aparecer en la parte oculta de los packages, en Java y C# sólo en las clases)

Variables estáticas locales en C/C++

- Se declaran anteponiendo la palabra clave `static`
- Se crean antes de la primera vez que el flujo de control llega a la declaración (inmediatamente antes como muy tarde)
- Se destruyen al final del programa

Ejemplo de variables estáticas locales en C/C++

```
void p()
{
   static int a ;
   ....
   while ( ... )
   {
      static float b ;
      ...
      {
         static char b ;
         ...
      }
   }
   ....
}
```

Variables en el montón (heap variables)

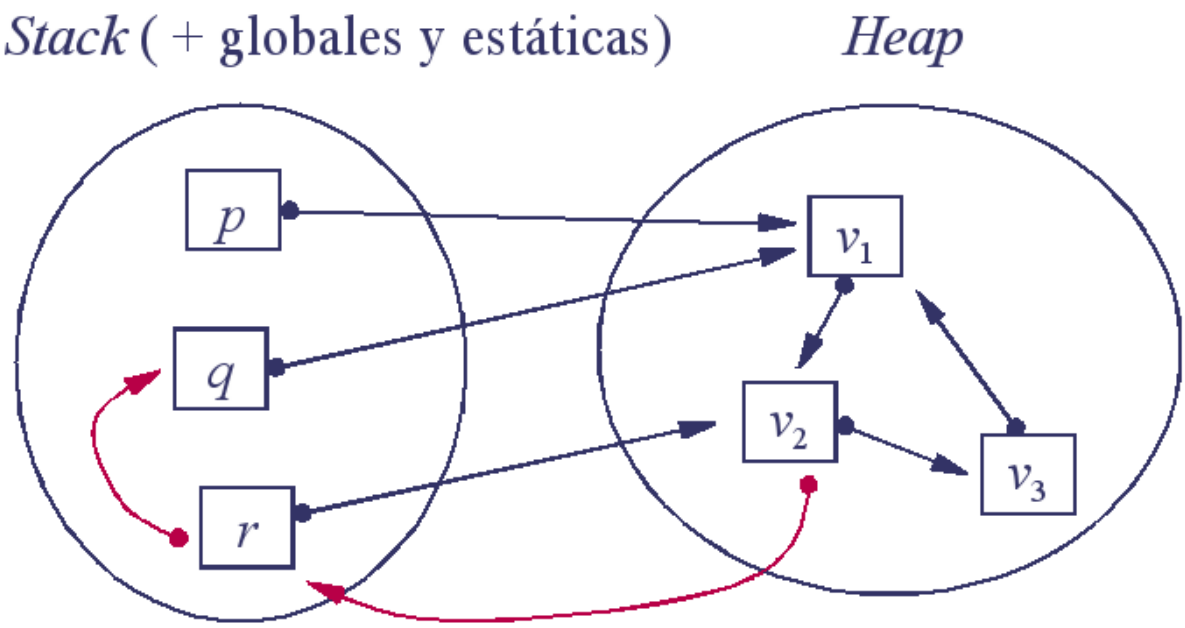
- No tienen nombre, y por tanto no tienen asignado un ámbito.

- Son accedidas exclusivamente usando referencias.
- El acceso se realiza directa o indirectamente a partir de referencias almacenadas en variables con nombre.

Variables en el heap (heap variables)

- Su tiempo de vida comienza cuando se ejecutan sentencias específicas de creación de las variables
- Acaba cuando:
  - Se ejecutan sentencias específicas de destrucción de las variables del heap (C/C++, Pascal)
  - No existen referencias con nombre que permitan acceder a la variable (Ada, Java, C#).

Variables en el heap y en el stack



Esquemas de alojamiento de variables

- En C/C++, Ada y otros lenguajes, el programador elige si una variable está en el heap o en el stack.
- En otros lenguajes no existe tal posibilidad: esto hace a los lenguajes más simples, legibles y fiables (aunque a veces menos eficientes)
- Veremos varios esquemas de alojamiento de variables

Alojamiento exclusivamente en el stack

- No existe el heap, todas las variables están en la pila o son variables globales.
- No existen las referencias
- Corresponde a los primeros lenguajes de programación, como por ejemplo las primeras versiones de Fortran.
- Poco flexible, simple, muy eficiente y muy fiable.

Alojamiento exlusivamente en el heap

- Todas las variables están en el heap, no existe el stack.
- No existen las referencias explícitamente como tipos del lenguaje.
- El acceso a variables se implementa siempre implicitamente mediante referencias
- Los tipos compuestos (arrays, registros y/o clases) se representan mediante conjuntos de referencias a las variables que contienen.

Alojamiento exlusivamente en el heap

- Es el esquema típico en los lenguajes interpretados, como: Python, Perl, PHP, TCL, Smalltalk.
- Esquema simple, fiable, poco eficiente (en memoria y tiempo, para tipos cuya representación ocupa pocos bits).

Alojamiento decidido por el programador

- Para cada variable, el programador especifica si se aloja en el heap o en el stack.
- Se contemplan explícitamente los tipos referencia
- Las variables en el heap se acceden con variables o valores de tipo referencia.
- Las variables que pertenecen a arrays, registros o clases se alojan en la misma zona de memoria que la variable que las contiene.

Alojamiento decidido por el programador

- Esquema que usan muchos lenguajes: Algol, Pascal, C/C++, Ada.
- En algunos casos es poco fiable (C/C++), y en otros es fiable (Ada)
- El esquema es flexible, algo complicado (baja legibilidad y poca facilidad de escritura)
- El programador debe ocuparse de hacerlo eficiente y legible.

Alojamiento determinado por el tipo.

- No existen las referencias
- Para algunos tipos (llamados tipos referencia) todas las variables están en el heap
  - El acceso a variables de tipos referencia (en el heap) se implementa mediante referencias.
  - Si una variable de estos tipos es parte de un array, estructura o clase, en dicha estructura se guarda una referencia a la variable.

Alojamiento determinado por el tipo.

- Para otros tipos (tipos valor), todas las variables se alojan en:
  - El stack, si no son componentes de otras (es decir, si son variables con nombre)
  - El bloque de memoria ocupado por la variable a la que pertenecen, si es un componente de otra variable (es decir, si son variables anónimas).

Alojamiento determinado por el tipo.

- El esquema es simple (mejora la legiblidad y facilidad de escritura) y fiable.
- El programador puede elegir flexibilidad (tipos referencia) o eficiencia en tiempo y memoria (tipos valor).
- Este esquema se introdujo con Java, y C# lo adoptó también (con mejoras).

Alojamiento determinado por el tipo en Java y C#

- Java
  - los tipos primitivos son tipos valor
  - los tipos array y las clases son tipos referencia

- C#
  - los tipos primitivos son tipos valor
  - los tipos arrays son tipos referencia
  - las clases pueden ser tipos referencia (`class`) o tipos valor (`struct`), a elección del programador

Alojamiento determinado por el tipo en Java y C#

- Las variables de tipos-valor solo pueden estar en el heap si son miembros de variables de tipos compuestos (arrays o clases).
- El programador puede situar explícitamente en el heap una variable de un tipo valor (T), mediante el uso de una clase de envoltura (wrapper class) para T

Alojamiento determinado por el tipo en Java y C#

- Una clase de envoltura para T es una clase, de tipo referencia, cuyas variables contienen una única componente de tipo T.
- El lenguaje Java incorpora clases de envoltura predefinidas para todos los tipos primitivos (que son los tipos valor de Java).

Ejemplo de variable en el heap en C

```
{  T * p ;
   ...
   p = (T *) malloc( sizeof( T ) ) ;
             /* poco fiable */

   .... /* variable del heap accesible
           mediante p */
   free( p ) ;
   ....
}
```

Ejemplo de variable en el heap en C++

```
{
   T * p ;
   ...
   p = new T ;
   ... /* variable del heap accesible
          mediante p */
```

```
    delete p  ;
    ....
}
```

Ejemplo de variable array en el heap en C++

```
{
   T * p ;
   ....
   p = new T[N] ;
   .... /* variable del heap accesible
           mediante p */
   delete [] p  ;
   .....
}
```

Ejemplo de variable en el heap en Ada

```
declare

   type RefT is access T ;
   p : RefT ;

begin
   ....
   p := new T ;

   .... -- variable accesible

end
```

Ejemplo de variable array en el heap en Ada

```
declare

   type TipoArray is Array (1..10) of Integer ;
   type RefTipoArray is access TipoArray ;
   ra : RefTipoArray ;

begin
   .....
   ra := new TipoArray ;
   .... - variable accesible
```

```
end
```

Ejemplo de array dinámico en el heap en Ada

```
declare

   type ArrayDin is
       Array (Integer range <>) of Integer ;
   type RefArrayDin is access ArrayDin ;
   r : RefArrayDin ;

begin
    ....
   r := new ArrayDin(0..9) ;
    .... -- variable accesible
end
```

Variables en el heap en Java y C#

```
{
   int[] p ;
   .....
   p = new int[N] ;
   ..... // variable array accesible
}
```

Variables en el heap en Python

En este lenguaje, todos las variables están en el heap:

```
i = 1
x = 34.67

class Persona:
    nombre ="Pepe"
    apellido = "Pérez"

p = Persona()

## i,x y p en el heap.
```

5 Referencias inválidas y recolección automática de basura

Referencias inválidas y recolección automática de basura.

En los ejemplos que hemos visto de variables en el heap en los lenguajes C y C++ hay dos posibilidades de error:

- **Referencias inválidas**: Una referencia puede almacenar una localización donde no hay ninguna variable o donde hay una variable de tipo distinto del esperado.
- **Basura no recolectada**: Pueden existir variables en el heap que no sean accesibles desde ninguna referencia en la pila.

Referencias inválidas

Se producen en varias situaciones:

1. Al hacer incorrectamente una conversión arbitraria de punteros (en C)
2. Al eliminar explícitamente una variable en el heap que es referenciada por otra que sigue existiendo. (Pascal, C, C++)
3. Al acabar el tiempo de vida de una variable en la pila que es referenciada por otra que sigue existiendo (Pascal, C, C++).

Conversión arbitraria de punteros en C/C++

```
#include <string.h>

int main()
{
    int        a  = 5 ;
    double     x  = 0.456 ;

    int *      pi = &a ;
    double *   px = &x ;
    ...

    void *     pvi  = pi ;
    void *     pvx  = px ;
    memcpy(pvx,pvi,8);
}
```

Eliminación de una variable en el heap con referencias a ella

```
int main()
{
  int * p = new int ;
  *p = 5 ; // bien.
  delete p ;

  .....

  *p = 6 ; // fallo
}
```

Eliminación de una variable en el heap con referencias a ella

```
int main()
{
   int * p = new int ;
   *p = 5 ; // bien
   int & r = *p ;
   delete p ;

   ......

   r = 6 ; // fallo
}
```

Fin del tiempo de vida de una variable con referencias

```
int main()
{
   int * p ;

   {
        int a ;
        p = & a ;
        *p = 5 ; // bien
   }

   *p = 6 ; // fallo
}
```

Fin del tiempo de vida de una variable con referencias

```
int *  fun()
{
  int a = 5 ;
  return & a ; // aviso
}

int main()
{
  int * p = fun() ;

  *p = 6 ; // fallo
}
```

Basura no recolectada

- Se produce cuando una variable en el heap deja de ser accesible, pero sigue existiendo (Pascal, C/C++)
- Como la variable en el heap no es accesible, no se pueden usar las instrucciones para destrucción explícita de variables (dispose de Pascal, free de C, delete de C++). Por tanto, la variable existe hasta el final del programa, ocupando inútilmente memoria.

Ejemplos de basura no recolectada en C

```
int main()
{
  {
     int * p = new int ;

     *p = 5 ; // bien.
  }

  // *p existe
  // pero no accesible
 .....
}
```

Ejemplos de basura no recolectada en C

```
void proc()
{
  int * p = new int ;
```

```
  *p = 5 ;   // bien
}

int main()
{
  proc() ;
  // *p existe, no accesible
  ....
}
```

Referencias inválidas y basura no recolectada:

- Los lenguajes más modernos (p.ej.: Ada, Smalltalk, Eiffel, Java, C#) evitan completamente estos problemas con varias estrategias.
- Veremos a continuación los casos de Ada, Java y C#.

Punteros inválidos

El problema se soluciona en los lenguajes modernos (Ada, Java y C#, entre otros)

- El caso (1) se elimina totalmente, al no considerarse el tipo puntero, y al impedirse las conversiones arbitrarias entre referencias a distintos tipos.
- El caso (2) se elimina al no existir instrucciones de destrucción explícita de variables en el heap (se ve más adelante).
- El caso (3) se soluciona imponiendo restricciones a las referencias a variables en la pila.

Restricciones a las referencias a variables en la pila.

Hay dos posibilidades:

- No se permiten referencias a variables en la pila, todas las referencias son a variables en el heap (Ada 83, Java, C#).
- Solo se permiten referencias a variables en la pila si se puede asegurar que el tiempo de vida de la variable no acaba antes que el tiempo de vida de la referencia (Ada 95).

Prevención de referencias inválidas en Ada 95.

- El tipo `access T` solo incluye referencias a variables de tipo `T` en el heap. (igual que en Ada 83)

- El tipo `access all T` incluye referencias a variables de tipo `T` en el heap y en el stack
- Si una variable de la pila, de nombre `v` y de tipo `T` se va a acceder con una referencia, debe declararse con la palabra clave `aliased`, como sigue: `v : aliased T`

Prevención de referencias inválidas en Ada 95.

```
declare
   type RefInt is access Integer ;
   v : Integer := 5 ;
begin
   declare
      r : RefInt ;
   begin
      r := v'access ; -- error!
              -- 'v' no es 'aliased'
              -- 'r' no es 'access all'
   end ;
end ;
```

Prevención de referencias inválidas en Ada 95.

```
declare
   type RefInt is access all Integer ;
   v : aliased Integer := 5 ;
begin
   declare
      r : RefInt ;
   begin
      r := v'access ; -- bien
      .....
   end ; -- fin del t.v. de 'r'
end ;  -- fin del t.v. de 'v'
```

Prevención de referencias inválidas en Ada 95.

```
declare
    type RefInt is access all Integer ;
    r : RefInt ;
begin
    declare
       v : aliased Integer := 5 ;
    begin
       r := v'access ; -- error!
```

```
        ......
    end ; -- fin del t.v. de 'v'
end ; -- fin del t.v. de 'r'
```

Prevención de referencias inválidas en Ada 95.

- En muchos casos, al compilar el programa se puede detectar que es erróneo (como en el caso anterior).
- En otros casos, al compilar un acceso a una variable no se puede saber a priori su tiempo de vida, y por tanto es necesario, durante la ejecución:
  - incorporar información sobre tiempos de vida en la propia variable y/o la referencia
  - realizar una comprobación (que podría fallar)

Basura no recolectada.

El problema se soluciona en los lenguajes modernos (Ada, Java y C#, entre otros), ya que usan el algoritmo que se denomina recolección automática de basura (RAB)

- Es una secuencia de acciones por las cuales se destruyen automáticamente las variables en el heap no accesibles, si hubiese alguna, cuando sea necesario.
- Ocurre como muy tarde justo antes de que falle la creación de una nueva variable en el heap por falta de memoria

Recolección automática de basura.

- Hace innecesarias las instrucciones de destrucción explícita de variables en el heap.
- Libera al programador de ocuparse de la destrucción de variables en el heap.
- Por tanto, hace los programas más legibles y escribibles.

Recolección automática de basura.

- Resuelve el problema de la basura no recolectada
- Resuelve el problema de los punteros inválidos a variables en el heap
- Por tanto, hace los programas más fiables.

Recolección automática de basura.

- Se ejecuta automáticamente, sin intervención del programador.
- Por tanto, introduce retardos en momento no previsibles a priori, aunque esto no suele ser problema en la mayoría de las aplicaciones.

Recolección automática de basura en Ada

```
declare
   type RefInt is access Integer ;
   p : RefInt := new Integer ;
begin
   p.all := 5 ;
   ....
   p := nil ; -- a partir de aquí, se puede usar
              -- el espacio ocupado por 'p.all'
end ;
```

```
declare
    type RefInt is access Integer ;
    r1 : RefInt ;
begin
    declare r2 : RefInt := new Integer ;
    begin
       r2.all := 5 ;
       r1 := r2 ; -- introduce 'aliasing'
       r2 := null ;
    end ;

    ...-- variable del heap aún accesible

end ; -- no accesible a partir de aquí
```

Recolección automática de basura.

Puede hacerse en varios momentos

- Al dejar de existir referencia alguna a una variable en el heap
- Solo cuando sea estrictamente necesario.
- Periódicamente y, además, cuando sea estrictamente necesario.

Métodos para recolección automática de basura

La RAB puede hacerse mediante el método de la cuenta de referencias:

- Cada variable del heap contiene un contador de las referencias que llegan a ella. Cuando es cero se destruye.
- Muy eficiente en tiempo, no sirve para ciclos de referencias.

Métodos para recolección automática de basura

El método de marcar y limpiar (mark and sweep):

- Se recorren y marcan todas las variables accesibles, directa o indirectamente, y a continuación se eliminan las no marcadas (las que no son accesibles).
- Es simple y sirve en cualquier caso.
- Es poco eficiente en tiempo y puede dejar la memoria muy fragmentada.

Métodos para recolección automática de basura

El método de parar y copiar (stop and copy).

- Similar al anterior, pero después de conocer que variables son accesibles, se copian de forma contigua todas ellas a una nueva zona de memoria, con lo cual se eliminan los huecos.
- Necesita más memoria, y es lento, pero no deja la memoria fragmentada.

Recolección automática de basura.

- Existen otros muchos métodos más avanzados, que intentan hacerla más eficiente en tiempo y memoria.
- Hoy en día, se acepta que, a pesar de la carga en tiempo de ejecución y memoria que la R.A.B. conlleva, merece la pena por la ventaja en cuanto a fiabilidad y sencillez de los programas.

Recolección automática de basura.

- El lenguaje C++ puede extenderse con R.A.B. usando clases, genericidad y sobrecarga de operadores.
- En general, la RAB puede suponer el movimiento de variables en el heap, luego esto hace que las referencias deban representarse o implementarse con estructuras de datos más complejas que los punteros de C/C++

fin del capítulo