Punteros. Estructuras dinámicas

Memoria Dinámica y Punteros

Estructuras Dinámicas

- Una estructura de datos se dice *dinámica* si su tamaño cambia en tiempo de ejecución del programa.
- De los tipos vistos:
  - record (case)
  - array
  - set

  ninguno permite construir estructuras dinámicas.
  - El espacio de memoria utilizado se reserva cuando comienza la ejecución del bloque donde están declaradas.
  - Este espacio no varía en toda la ejecución del bloque
  - Observación: array con tope, set y record case **no** son dinámicos (aunque lo parezcan)

El tipo puntero

- Un *puntero* es una variable que apunta o referencia a una *ubicación de memoria* en la cual hay datos
- El contenido del puntero es la *dirección* de esa ubicación
- A través del puntero se puede:
  1. "crear" la ubicación de memoria (`new`)
  2. acceder a los datos en dicha ubicación (`^` desreferenciación)
  3. "destruir" la ubicación de memoria (`dispose`)

Declaración de una variable puntero

Ejemplos de declaraciones

```
type
    ptrint   = ^integer; (* puntero a entero *)
    ptrbool  = ^boolean; (* puntero a boolean *)

    celda = record
                . . .
                . . .
            end;

    ptrcelda = ^celda; (* puntero a un record *)
```

Sintaxis de la declaración

En general:

- **type** *identificador* = ˆ *identificador_de_tipo*

donde:

- *identificador* es el nombre del tipo puntero que se define
- *identificador_de_tipo* corresponde a un tipo predefinido o definido previo a la declaración.
- el tipo asociado con *identificador_de_tipo* puede ser cualquiera de los tipos de Pascal

Valores y operaciones con punteros

- valores iniciales son indefinidos (como toda variable Pascal)
- el valor guardado en una variable puntero es una dirección de memoria, por lo tanto dependiente de la arquitectura del computador
- No es posible leer o escribir valores de tipo puntero (`read` o `write`)
- La única comparación permitida es la igualdad (operador =)

El procedimiento new

Se invoca de la siguiente manera:

```
new(ptr)
```

donde ptr es una variable de tipo ^T.
El efecto de esta operación es:

1. Se crea un espacio nuevo para una variable de tipo T (tanta memoria como requiera el tipo)
2. La dirección de este espacio se guarda en la variable ptr

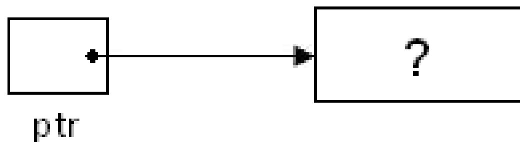

Figure 1:

El procedimiento `new`

El proceso ejecutado por `new` se denomina **asignación dinámica de memoria**

Observar que:

- estas acciones se realizan en tiempo de **ejecución**
- la variable creada de tipo T queda con un valor inicial indefinido
- la variable creada *no tiene nombre* (identificador)

El operador ^

Supongamos que se ejecuta:

```
new(ptr1);
new(ptr2);
```

Es posible hacer uso de las variables creadas usando las expresiones:

- `ptr1^` variable apuntada por `ptr1`
- `ptr2^` variable apuntada por `ptr2`

El operador ^. Observaciones

- La expresion `ptr1^` puede ser usada en cualquier parte donde Pascal admite una variable del tipo correspondiente, por ejemplo:
  - parte izquierda y derecha de una asignación
  - expresiones del tipo apropiado
  - parámetros efectivos (valor y referencia)
- La expresión `ptr^` produce un error si `ptr` no fue inicializado.
- Una variable puntero se inicializa por `new` o por asignación de otro puntero ya inicializado.

Ejemplos

Sean las declaraciones

```
type
   ptrnum = ^integer;
var
   ptr1, ptr2: ptrnum;
```

Las siguientes instruciones son válidas:

```
   new(ptr1);
   new(ptr2);
   ptr1^ := 12;
   ptr2^ := ptr1^ + 4;
   readln(ptr1^);         (* suponga que se ingresa 3 *)
   writeln(ptr2^ * ptr1^) (* ¿qué despliega?          *)
```

Alias de variables

Diferentes punteros pueden apuntar a la misma dirección, generando *alias*.
Alias: expresiones distintas que representan la misma variable.

```
new(ptr1);
ptr1^:= 6;
ptr2:= ptr1;
writeln(ptr2^);      (* --> 6 *)
ptr2^:= ptr1^ + 2;   (* ¿qué pasa aquí? *)
writeln(ptr2^);
writeln(ptr1^);
```

`ptr2` es un *alias* de `ptr1`

Punteros como parámetros por valor

Cuando se pasa un puntero como parámetro por valor, los resultados no son los esperados:

```
type pint = ^integer;
var p: pint;
procedure PP(q1,q2: pint); (*pasaje por valor*)
begin
    q1^:= q2^ * 2;
    q2^:= q1^ + 2;
end;
begin
   new(p);
   p^:= 4;
   PP(p,p);
   writeln(p^); (*¿qué valor despliega?*)
end.
```

Listas

El tipo Lista

- Una **lista** es una *secuencia* o *sucesión* finita de elementos.
- Se puede representar como:
  - **arreglo:**. *Tamaño fijo*, no se pueden agregar y/o quitar elementos.
  - **arreglo con tope:** *Tamaño acotado*, permite agregar y/o quitar elementos. El espacio de memoria ocupado es fijo
  - **listas encadenadas:** utilizando *punteros*. permite agregar y/o quitar elementos. Ocupa un espacio proporcional al tamaño.

Listas encadenadas

- Cada elemento se almacena en una celda
- Cada celda contiene la información del elemento y un puntero a la siguiente celda
- Para acceder a la lista basta con conocer el puntero al primero elemento
- ¿Cómo se representa una lista vacía? ¿cómo se reconoce el último elemento de la lista?

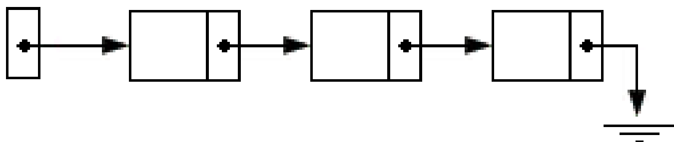

Figure 2:

El puntero nulo

- Existe una constante especial `nil` que es llamada el *puntero nulo*.
- `nil` pertenece a todos los tipos de la forma `^T`
- `nil` no representa una dirección de memoria
- `nil^` da un error en tiempo de ejecución
- El valor `nil` se puede asignar directamente a una variable de tipo puntero:

  ```
  p:= nil
  ```
- ¿Para qué sirve?
  - Representar la lista vacía
  - distinguir el último elemento de una lista encadenada

Listas encadenadas

La definición en Pascal de listas encadenadas:

```
type
    lista = ^celda;

    celda = record
        elemento: T;
        siguiente: lista;
    end;
```

Observar que Pascal admite poner `^celda` aún cuando celda se define después.

Largo de una lista

Calcular la cantidad de elementos de una lista

```
function largo(l: lista): integer;
var
    contador: integer;
    p: lista;
begin
    contador:= 0;
    p:= l;
    while p <> nil do
    begin
       contador:= contador + 1;
       p:= p^.siguiente; (* avanzar a la sig. celda *)
    end;

    largo:= contador;
end;
```

Búsqueda en una lista

Buscar un elemento en una lista

```
function pertenece(elem: T; l: lista): boolean;
var
   p: lista;
begin
   p:= l;
   (* notar: evaluación por circuito corto *)
    while (p <> nil) and (p^.elemento <> elem) do
        p:= p^.siguiente;

    pertenece:= (p <> nil);
end;
```

Agregar elemento al principio

```
procedure agregar_al_principio(var l: lista; elem: T);
var p : lista;
begin
    new(p);                 (*crear nueva celda*)
    p^.elemento:=  elem;    (*cargar el elemento*)

    (* ajuste de punteros *)
    p^.siguiente:= l;
    l:= p;
end;
```

Agregar elemento al final (1)

```
procedure agregar_al_final(var l: lista; elem: T);
var p,q : lista;
begin

...

end;
```

Agregar elemento al final (2)

```
new(p);                    (*crear nueva celda*)
p^.elemento:=  elem;   (*cargar el elemento*)
p^.siguiente:= nil;    (*es el último*)

if l = nil then
    l:= p
else
begin
    (*busco el último de l*)
    q:= l;
    while q^.siguiente <> nil do
       q:= q^.siguiente;

    (*engancho p a continuacion del último*)
    q^.siguiente:= p;
end;
```

El procedimiento `dispose`

- indica al sistema que una celda de memoria ya no será más utilizada
- el sistema recupera el espacio y lo puede reutilizar
- se invoca así:
  `dispose(ptr)`
  donde `ptr` es un puntero a la celda en cuestión. Luego del dispose, `ptr` queda *indefinido*
- se utiliza cuando se borran elementos de una estructura dinámica

Borrar Primero de una Lista

Suponemos la lista no vacía

```
procedure borrar_primero(var l: lista);
var
    p: lista;
begin
    p:= l;
    l:= l^.siguiente;
    dispose(p);
end;
```

Borrar la lista entera

Para liberar todo el espacio ocupado por una lista es necesario liberar celda por celda.

```
procedure borrar_lista(l: lista);
var
   p: lista;
begin
   while l <> nil do
   begin
       p:= l;
       l:= l^.siguiente;
       dispose(p);
   end;
end;
```