Tema 4: Punteros y gestión dinámica de memoria

Tecnología de la Programación

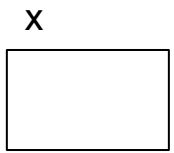
Índice

Índice

Introducción

Hasta ahora:

```
entero x
```



Espacio se reservado por el compilador en tiempo de compilación (gestión **estática**)

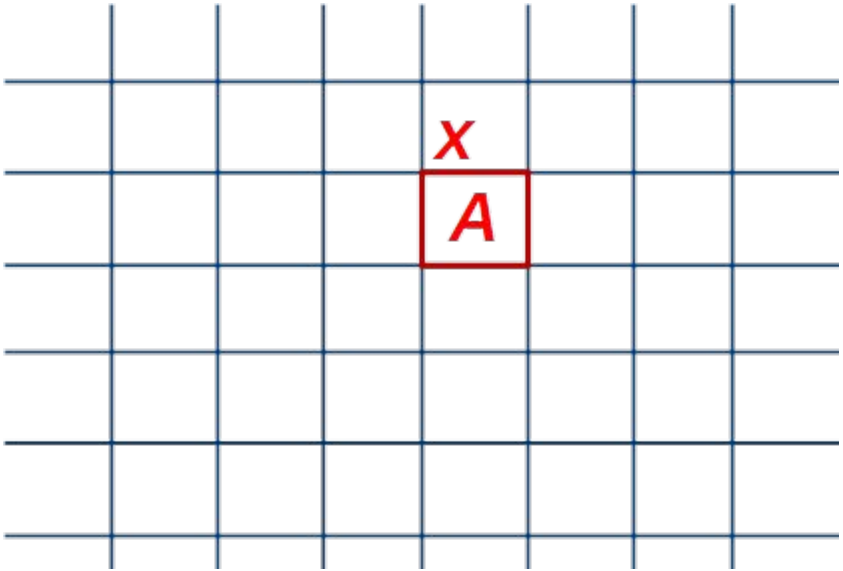
¿Qué ocurre cuando no sabemos de antemano cuántos datos necesitamos almacenar?

- Usamos vectores → limitados por tamaño del vector
- Usamos ficheros → acceso lento

Objetivo: dar herramientas para la gestión **dinámica** de la memoria

Modelo básico de memoria

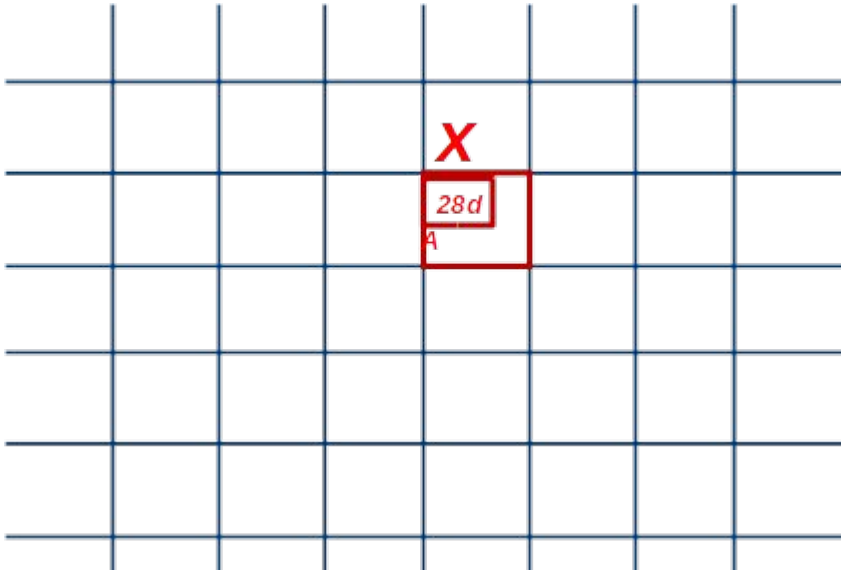
```
caracter x
x = ‘A’
```



Modelo enriquecido de memoria

Identificador → Dirección de memoria → Memoria

```
caracter x
x = ‘A’
dirección de memoria: 28d
direcc(x): 28d
cont(28d): ‘A’
```



Modelo enriquecido de memoria

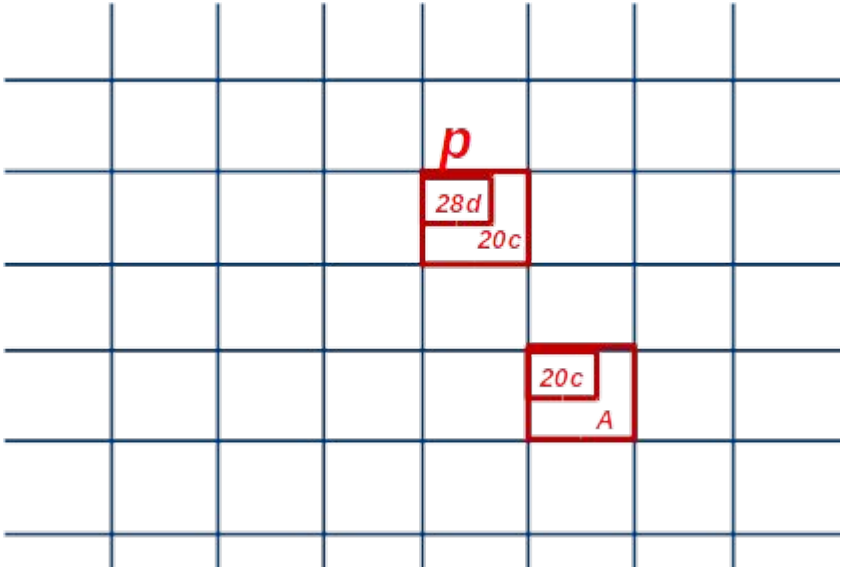
Operadores:

- `direcc(<identificador>)`: dirección de memoria asociada a una variable llamada `<identificador>`
- `cont(<dirección>)`: valor en la celda cuya dirección de memoria es `<dirección>`

Modelo enriquecido de memoria

Idea: las direcciones de memoria son datos y van a poder almacenarse en variables de tipo **puntero**

```
puntero p
direcc(p): 28d
cont(28d): 20c
cont(cont(direcc(p))): ‘A’
```



Modelo enriquecido de memoria

Cuidado: podemos acceder al contenido de la celda `20c` sin hacer referencia a su identificador (más aún, sin conocerlo)

Índice

Concepto de puntero

Definición: un **puntero** es una variable cuyo contenido es una dirección de memoria

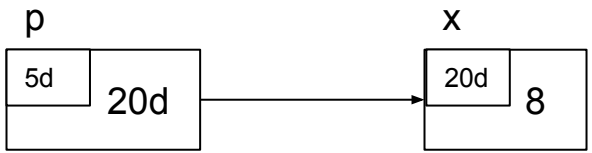
A través de un puntero podemos acceder al dato almacenado en la celda cuya dirección es el contenido del puntero

Nuevo operador: `dest(<puntero>)`: `cont(cont(direcc(<puntero>)))`

Concepto de puntero

Estableciendo “la flecha”

```
entero x
puntero p
x = 8
p = direcc(x)
```



Decimos que p apunta a x

Con `dest(p)` puedo hacer lo mismo que con `x`

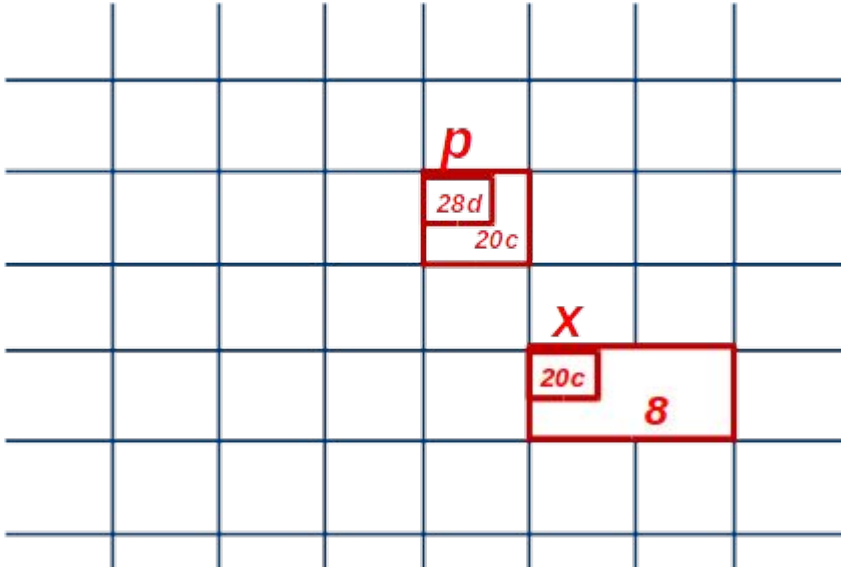
Ejemplo:

```
dest(p) = dest(p) + 1
escribir(x) // x ha pasado a valer 9
```

Concepto de puntero

No se trabaja con punteros genéricos sino con punteros a tipos de datos ya que el operador destino debe saber cuántas celdas tomar

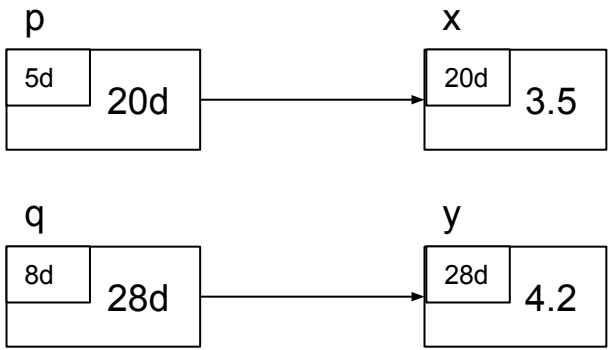
```
puntero a entero p
entero x
p = direcc(x)
dest(p) = 8
```



Concepto de puntero

Diferencia entre punteros y datos a los que apuntan

```
puntero a real p, q
real x = 3.5
real y = 4.2
p = direcc(x)
q = direcc(y)
```



Concepto de puntero

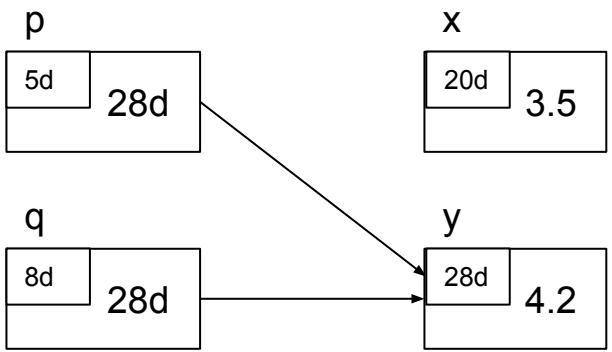
Caso 1)

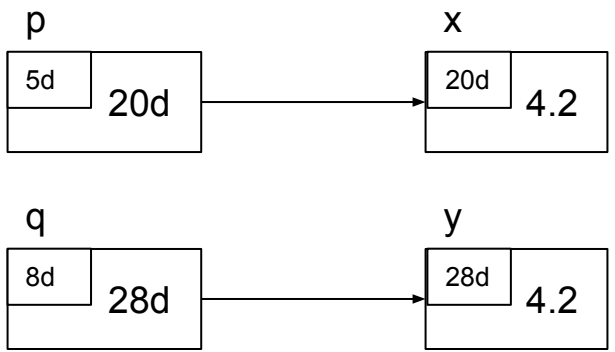
```
puntero a
real x = 3
real y = 4
p = direcc
q = direcc
p = q
```

Caso 2)

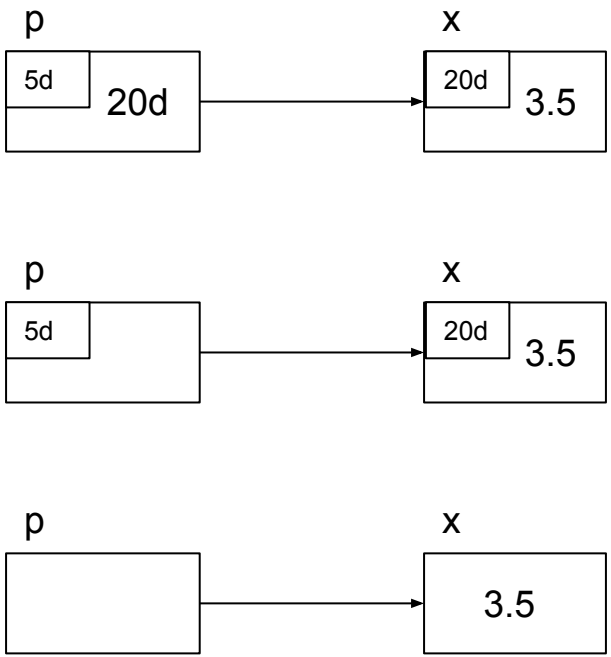
```
                                q
dest(p) = dest(q)
```

¿Qué ocurre si modificamos el contenido de la dirección 28d? ¿Cuál sería el valor de `dest(p)` en cada caso?





Simplificación de la notación



Observaciones

1. No se trabaja con punteros genéricos. Hay que determinar a priori el tipo de dato que se almacenará en su destino
2. A todos los efectos, `dest(p)` se comporta como una variable; es decir, puede evaluarse y modificarse su valor mediante sentencias de asignación
   Ejemplo:

```
entero x
puntero a entero p
p = direcc(x)
dest(p) = 7
x = dest(p)+1
escribir(dest(p))
```

Observaciones

3. El operador destino trabaja solo sobre punteros
4. Una de las características más importantes de los punteros es que se puede modificar el valor de una variable sin que aparezca explícitamente su expresión

Punteros en C++

	Declaración de puntero	Dirección	Destino
Pseudocódigo	`puntero a <tipoDato> p`	`direcc(x)`	`dest(p)`
C++	`<tipoDato> *p;`	`&x`	`*p`

Ejercicio

¿Es el siguiente código correcto?

```
int main(){
    int x,y;
    int *p,*q;
    p = &x;
    *p = 2;
    *q = 3;
    cout<<x+y;
    return 0;
}
```

Índice

Punteros descontrolados

- Es peligroso tener punteros que no sabemos a dónde apuntan
- El operador destino puede cambiar contenido de la celda a la que apuntamos, por lo que podemos modificar valores que no queremos modificar
- Por lo tanto, **siempre** que se declare una variable de tipo puntero hay que **inicializarla**, es decir, hay que hacer que apunte a algún sitio controlado

Inicialización de punteros

Posibilidad 1: que el puntero apunte a alguna variable. Ejemplo:

```
int x;
int *p;
p = &x;
*p = …
```

Posibilidad 2: que el puntero apunte al mismo sitio que otro puntero del mismo tipo. Ejemplo:

```
int x;
int *p,*q;
p = &x;
q = p;
```

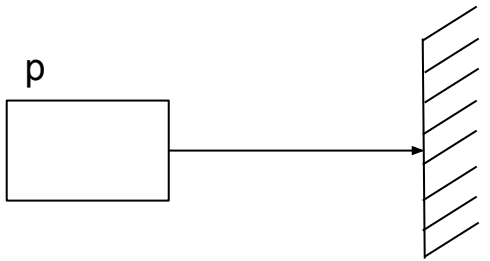
Inicialización de punteros

Posibilidad 3:

- De momento que no apunte a ningún sitio → lo ponemos a “descansar”
- Existe dirección constante que es la dirección nula (`NULL` en C++)
- Esta dirección puede servir como valor inicial de un puntero que de momento no queremos que apunte a ningún sitio en concreto
- Cuando un puntero apunta a `NULL` **es un error** aplicarle el operador destino
- Ejemplo:

```
int *p;
p=NULL;
*p = 7;    ⬅ ERROR
```

Representación:



Inicialización de punteros

Posibilidad 4: reservar una zona de memoria dinámica

Índice

Gestión dinámica de memoria

¿Qué ocurre con la memoria del ordenador durante la ejecución de un programa?

Se crean 4 regiones de memoria lógica diferentes, cada una de las cuales sirve para una función específica

Stack (pila)
Heap (montón)
Variables globales
Código programa

Gestión dinámica de memoria

Stack (pila):

- Mantiene direcciones de retorno de las funciones
- Almacena las variables locales
- Funciones relacionadas con la CPU

Heap (montón):

- Zona utilizada por funciones de asignación dinámica

Variables globales:

- No la utilizamos por los efectos laterales

Código programa:

- Almacena código binario obtenido durante el proceso de compilación

Gestión dinámica de memoria

- Compiladores permiten configurar el tamaño de las distintas zonas
- **Objetivo:** manejar la memoria en tiempo de ejecución de manera que en cada momento se reserve la memoria que necesita el programa
- Los **punteros se utilizan** para crear **estructuras de datos dinámicas:**
  - Su organización (tamaño) no es fija
  - Varía durante la ejecución
  - Cambios que sucedan deben estar previstos de forma que haya memoria para almacenarlos
- **Idea**: cuando necesitemos espacio, pedirlo al compilador y acceder a él a través de un puntero

Operadores: reservar

- `<puntero> = reservar(<tipoDato>[N])`
- Reserva el espacio en memoria de forma consecutiva necesario para almacenar `N` elementos de tipo `<tipoDato>`
- El valor de `N` es opcional, si no se indica toma por defecto el valor 1
- El valor que toma `<puntero>` es la dirección de la primera celda (ver ejemplo en pizarra)
- El operador `reservar` proporciona una **zona de memoria sin identificador** a la que se accede a través del puntero, concretamente como su destino
- Si en una llamada a la operación `reservar` no hay memoria suficiente → el sistema le asigna al puntero el valor `NULL`
  - En este caso por lo tanto sería un error acceder al destino de dicho puntero
  - Por lo tanto **siempre** que se usa el operador `reservar`, **hay que comprobar** si se ha llevado a cabo la reserva

Operadores: reservar

Ejemplo (en pseudocódigo):

```
puntero a entero p
p = reservar(entero)
si (p != NULL) entonces
    …
si_no
    escribir(“No hay espacio suficiente”)
fsi
```

Operadores: reservar

En C++:

- `<puntero> = new(<tipoDato>);`
- `<puntero> = new(<tipoDato>[N]);`

Ejemplo:

```
int *p,*q;
p = new(int);
q = new(int[4])
```

Operadores: liberar

- `liberar(<puntero>), liberar([]<puntero>)`
- Libera la zona de memoria apuntada por `<puntero>` para que pueda reutilizarse
- Hay que tener cuidado porque el puntero queda descontrolado
- Para usar operador liberar sobre un puntero es necesario que la zona a la que apunte haya sido previamente reservada con el operador `reservar`
- Es un **error** perder la referencia a una zona de memoria reservada con el operador `reservar`
  - Se genera basura ya que la única referencia a la zona reservada es el puntero
  - Si perdemos la referencia, la zona sigue ocupada pero no podemos acceder a ella
- Para liberar una zona previamente reservada, se puede:
  - Usar mismo puntero utilizado para reservar la zona
  - Otro puntero que apunte a esa zona

Operadores: liberar

Ejemplo: ¿es correcto el siguiente código?

```
puntero a entero p, q
p = reservar(entero)
q = reservar(entero)
si (p != NULL) AND (q != NULL) entonces
    dest(p) = 5
    dest(q) = 6
    p = q
    liberar(p)
    liberar(q)
fsi
```

Operadores: liberar

En C++:

- `delete(<puntero>)`
- `delete([]<puntero>)`

Ejemplo: ¿es correcto el siguiente código?

```
int x;
int *p,*q;
p = new(int);
if (p != NULL){
    *p = 5; q = p; p = &x;
    delete(q);
}
```

Índice

Listas enlazadas de nodos

Estructura de datos:

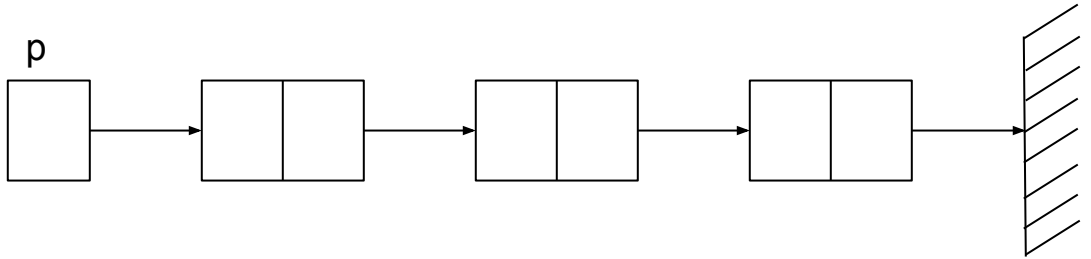
- Se utiliza cuando queremos almacenar varios datos del mismo tipo
- Pero de antemano no sabemos el número de datos que tendremos
- Cada vez que leamos un dato:
  - Reservar espacio para él
  - Ligarlo con el resto de datos: con el anterior y el siguiente es suficiente
- Es decir, para cada dato se reserva espacio para el propio dato y se establece una referencia al siguiente

Listas enlazadas de nodos

Definición:

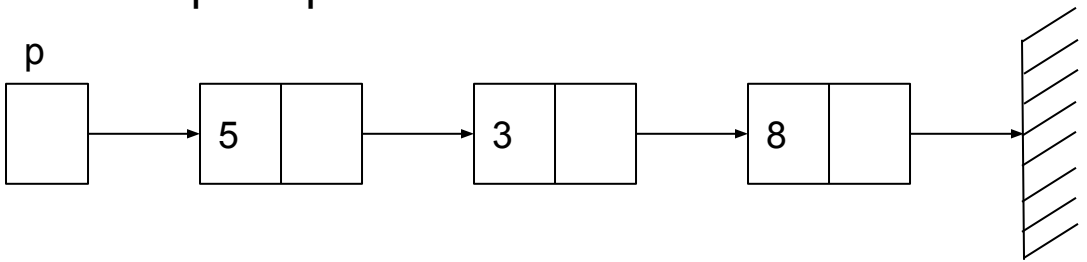
```
tipos
    Nodo = registro
        entero dato
        puntero a Nodo sig
    freg
    lista_enlazada = puntero a Nodo
```

Interpretación gráfica:



Listas enlazadas de nodos

Ejemplo: suponer que tenemos la siguiente lista enlazada de nodos y queremos añadir el elemento 10 al principio de la lista:



Listas enlazadas de nodos

Ejemplo: leer enteros del teclado (terminar con 0) y mostrarlos en orden inverso

1. Crear la lista
2. Mostrar la lista
3. Liberar la lista

Subalgoritmos con parámetro de tipo puntero

Suponer que tenemos el siguiente subalgoritmo

```
acción a1(...)
variables
    puntero a entero p
principio
    p = reservar(entero)
    si (p != NULL) entonces
        dest(p) = 8
        a2(p)
        …
    fsi
fin
```

Subalgoritmos con parámetro de tipo puntero

La acción `a2` puede tomar el puntero como:

1. Parámetro de entrada
2. Parámetro de salida
3. Parámetro de entrada/salida

Después de llamar a `a2`:

Caso 1:

- Se ha podido modificar `dest(p)`
- `p` puede quedar descontrolado:
  - Su valor es el mismo, pero en la acción `a2` se ha podido liberar su destino
  - Realmente parámetro de entrada significa **paso por valor**
  - Paso por valor en punteros no se corresponde con parámetros de entrada

Subalgoritmos con parámetro de tipo puntero

Después de llamar a `a2`:

Casos 2 y 3:

- Se ha podido modificar `dest(p)`
- `p` puede quedar descontrolado o apuntar a otro sitio
- Siempre que se trabaja con punteros como parámetros de subalgoritmos estos son de entrada/salida

Paso por valor y paso por referencia

Paso por valor:

- Copia contenido de la variable que queremos pasar en otra dentro del ámbito local de la subrutina
- Se copia contenido de la memoria del argumento en otra dirección de memoria
- Se obtienen dos valores duplicados pero independientes y la modificación de uno no afecta al otro
- Se suele relacionar con argumentos de entrada
- ¿Qué ocurre con los punteros?
  - Obtenemos dos punteros que apuntan al mismo sitio
  - Por lo tanto modificaciones en uno de ellos afecta al otro

Paso por valor y paso por referencia

Paso por referencia:

- Se proporciona a la subrutina la dirección de memoria del dato
- Se tiene un único valor referenciado desde dos sitios diferentes (programa principal y subrutina)
- Cualquier acción sobre parámetro se realiza sobre misma dirección de memoria
- Se suele relacionar con argumentos de salida o de entrada/salida

Listas enlazadas de nodos

Ejemplo: leer enteros del teclado (terminar con 0) y mostrarlos en orden inverso utilizando subalgoritmos

1. Crear la lista
2. Mostrar la lista
3. Liberar la lista

Listas enlazadas de nodos

Ejercicios:

1. Función `esta?` que comprueba si un entero está en una lista enlazada de nodos de enteros
2. Función `anadirAlFinal` que añade un entero al final de una lista enlazada de nodos de enteros

Punteros y registros

Es posible tener:

- Puntero que apunte a registro
  - Para acceder a campo de registro apuntado por puntero
    - `dest(p).campo` ✓
    - `dest(p.campo)` ✗
  - En C++:
    - `(*p).campo`
    - El operador "`.`" tiene más prioridad que el operador `dest`, por lo que hacen falta los paréntesis
- Registros con campos de tipo puntero

Índice

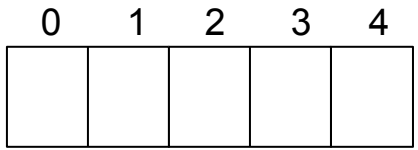
Relación entre vectores y punteros

Vectores:

- Estructura de datos formada por un número fijo de componentes
- Todas las componentes del vector tienen el mismo tipo
- Componentes del vector están indexadas de forma consecutiva
- Se puede acceder directamente a las componentes del vector a través del índice
- Para definir un vector necesitamos dar:
  - Dimensión
  - Tipo de dato de las componentes

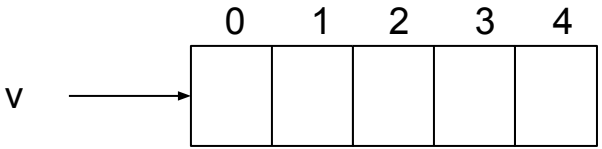
Relación entre vectores y punteros

Al declarar una variable de tipo vector, el compilador reserva el nº de celdas de memoria necesarias de forma consecutiva



Para cada celda tenemos:

- Índice → `0≤i≤4`
- Dato → `v[i]`
- Dirección → `&v[i]`



Relación con punteros:

- `v ≡ &v[0]`
- `v ≡ puntero`
- `*v ≡ v[0]`

Aritmética de punteros

Dado `p : puntero a <TipoDato>` ¿Qué significa `p+1`?

- Puntero a tantas direcciones adelante de `p` como sea necesario para almacenar un dato del `dest` de `p`

Ejemplo:

- `p : puntero a entero`
  - `p = 20d`
  - `p + 1 → 24d`

- `p : puntero a real` (`float` 4 bytes, `double` 8 bytes)
  - `p = 20d`
  - `p + 1 → 24d` (si `float`)
  - `p + 1 → 28d` (si `double`)

Índice

Vectores dinámicos

Volviendo a punteros y vectores:

- `v ≡ &v[0] → v+i ≡ &v[i]`
- `*v ≡ v[0] → *(v+i) ≡ v[i]`

Vectores dinámicos

¿Cómo podemos almacenar datos del mismo tipo?

1. Sabiendo a priori cuántos:
   a. Vectores estáticos
   b. Vectores dinámicos
2. Sin saber a priori cuántos:
   a. Vectores estáticos
   b. Vectores dinámicos

Vectores dinámicos: conociendo nº elementos

Vectores estáticos

```
int i,n;
cin>>n;
int v[n];
for (i=0;i<n;i++){
    cin>>v[i];
}
```

Vectores dinámicos

```
int *p;
int i,n;
cin>>n;
p=new(int[n]);
if (p!=NULL){
    for (i=0;i<n;i++){
        cin>>*(p+i);
    }
}
```

Vectores dinámicos: sin conocer nº elementos

Vectores estáticos

```
int v[100];
int i,x;
cin>>x;
i=0;
while (x!=0){
    v[i]=x;
    i++;
    cin>>x;
}
```

Vectores dinámicos: listas enlazadas

Índice

Representación dinámica de datos definidos por recurrencia

Idea: utilizar punteros y gestión dinámica de memoria para manipular datos cuyo tamaño no se conoce a priori. Un ejemplo son los datos definidos por recurrencia

Cadenas de caracteres:

- Representaciones estáticas:
  1. Registro con dos campos: longitud y vector de caracteres
  2. Vector con una marca final
- Ambas representaciones limitan las cadenas que se pueden representar
- En su lugar utilizar una definición recursiva

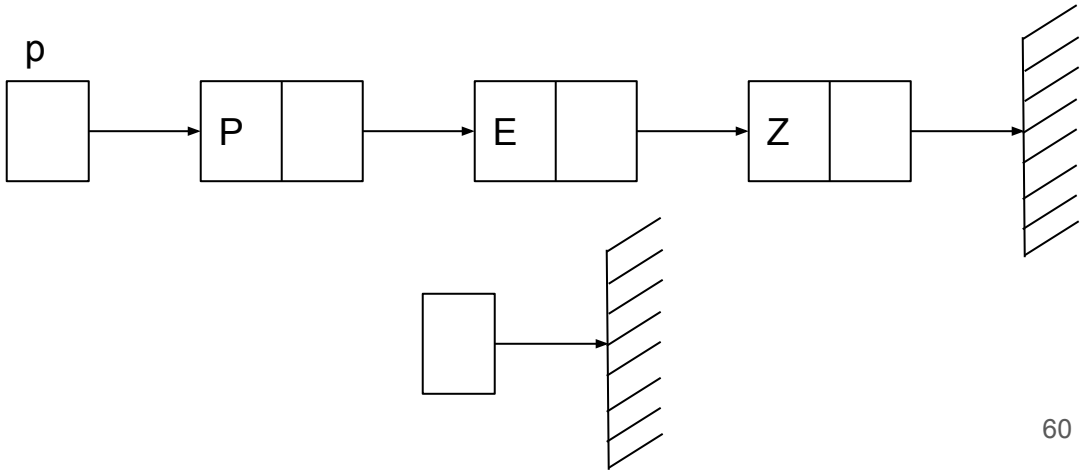
Cadenas de caracteres

Definición. Sea $\mathcal{L} = \{\mathcal{A}, \mathcal{B}, \ldots, \mathcal{Z}\}$ un alfabeto y sea $\mathcal{L}^+$ las cadenas de caracteres definidas del siguiente modo:

- Caso base: $<> \in \mathcal{L}^+$
- Caso recursivo: si $c \in \mathcal{L}$ y $l \in \mathcal{L}^+ \rightarrow cl \in \mathcal{L}^+$

Implementación.

```
Celda = registro
    carácter dato
    puntero a Celda sig
freg
cadena = puntero a Celda
```



Cadenas de caracteres

Una vez elegida una representación, se pueden implementar operaciones con cadenas:

- Longitud
- Añadir
- Esta?
- …

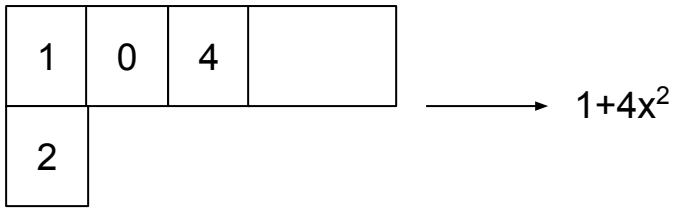
Polinomios

La información que caracteriza a un polinomio es:

- Grado
- Para cada monomio: coeficiente y exponente

Representación estática → registro con dos componentes:

- Grado
- Vector donde índices marcan el grado y la componente es el coeficiente en cada grado



Problemas:

- Limitados por grado
- Hay que fijar dimensión del vector
- Polinomios grado alto con muchos 0s → se desaprovecha espacio

Polinomios

Definición recurrente: $\mathcal{P}(\mathbb{R})$

- Caso base: ax^b `es polinomio con` $a \in \mathbb{R}$ `y` $b \in \mathbb{N}$
- Caso recursivo:

$$\left.\begin{array}{c} ax^b \\ p \in \mathcal{P}(\mathbb{R}) \end{array}\right\} \implies p + ax^b \in \mathcal{P}(\mathbb{R})$$

Representación dinámica → celdas enlazadas en las que cada celda almacena información de un monomio: grado y coeficiente

Polinomios

```
tipos
    monomio = registro
        real coef
        entero grado
        puntero a monomio sig
    freg
    polinomio = puntero a monomio
```

Ejemplo: $8x^5-3x+5$