**Referencias**

**Introdución**

* As estruturas de datos previamente vistas ocupaban en memoria un espazo definido ao declaralas.
* Con todo, é máis óptimo empregar datos cuxa ocupación de memoria cambiase en cada execución do programa. Para isto empréganse **estruturas dinámicas**, creadas e destruídas no momento necesario.
* Estas variables non teñen nome senón que se designan mediante outras variables chamadas punteiros ou referencias.

**Punteiros**

* Os **punteiros** son variables simples que conteñen unha referencia a outras variables
* O seu valor é un enderezo de memoria, sendo a posición doutra variable.
* Os punteiros teñen tipo, que indica o tipo da variable á que apunta.
* Tras declarar un punteiro e unha variable (int \*p, int n) o punteiro p asígnase ao enderezo desta variable mediante a sentencia **p = &n;**
* Os punteiros pódense comparar entre si (p1==p2), e isto devolve true só se apuntan á mesma variable. Devolve false se apuntan a distintas variables aínda que o contido sexa o mesmo.

**Tipos de memoria**

* Durante a compilación, realizarse unha reserva estática da memoria.
* Logo, durante a execución, a memoria reservada denomínase dinámica.
  + A reserva dinámica permite crear tipos de datos que non teñen unha cantidadde de memoria predeterminada, senón que este número varía durante a execución.
  + Desta forma, o número de elementos está limitado pola memoria dispoñible.
  + Esta memoria almacénase no **montón**, ou heap.
  + Para empregar esta memoria, o programa debe ter punteiros e métodos para reservar e liberar memoria.

**Asignación de memoria dinámica**

* Antes de empregar unha variable de tipo punteiro, é preciso asignarlle memoria. Para isto, a forma máis sinxela é a operación:

**p=(tipo\_de\_variable \*)malloc(sizeof(tipo\_de\_variable));**

* Este procedemento crea unha variable dinlamica do tipo de variable ao que apunta o punteiro p (tipo\_de\_variable) e resévaselle a memoria necesaria para unha variable deste tipo (sizeof(tipo\_de\_variable).
* É preciso indicar o tipo de variable que se está gardando porque cada tipo de ddato se representa en memoria de forma distinta.

**Valor NULL**

* Cando unha variable punteiro se declara sen inicializarse, a variable punteiro toma o valor **NULL**, para indicar que non apunta a ningún enderezo de memoria válido.
* Ao comparar se o valor ao que indica é correcto, é recomendable primeiro comprobar se está inicializada o punteiro correctamente, co condicional punteiro!=NULL (non \*punteiro!=NULL)

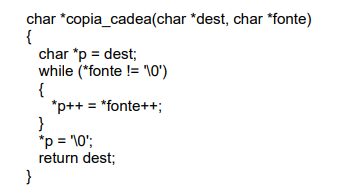
**Compartir, liberar memoria**

* Unha variable dinámica pode estar referenciada por máis dun punteiro. Por exemplo, cando se **copia un punteiro noutro**. (int \*p1, \*p2, p2=p1).
  + p1 e p2 apuntan á mesma variable dinámica.
* Se se fai esto, entón a memoria que fose gardada para p2 queda **perdida**, pois p2 xa non apunta a este enderezo e xa non é posible acceder a esta memoria para eliminala.
  + Para solucionar este problema, antes de asignar p2=p1 débese liberar a memoria que fora ocupada para p2 mediante **free(p2)**.
  + Esta sentencia libera a memoria ocupada previamente por p2, e asígnalle a p2 o valor NULL.

**Aritmética de punteiros**

* Sendo PTR un punteiro a un enteiro que ten o enderezo de memoria 100, ao realizar PTR+1 en realidade sumarase 4 para chegar ata 104, de tal forma que apunta ao seguinte enteiro.
* Do mesmo xeito, se fose un punteiro a un double, avanzaría 8 posicións.
* A **aritmética de punteiros** consiste en avanzar ou retroceder un número de posicións da memoria mediante as operacións de suma e resta.

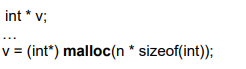
**Punteiros e vectores**

* Se se asigna un punteiro a un vector, da seguinte forma:
* int arr[10]; int \*p1; p1 = arr;
* É equivalente a asignar a p1 o enderezo de memoria do primeiro elemento do vector. É dicir, é equivalente p1=**&arr[0]** que p1=**arr**.
  + Porén, hai que ter en conta que arr é unha constante. Non é correcto realizar arr=p1.
* Se se suma ou resta un valor ao vector, altérase o elemento do mesmo ao que referenciamos. Por exemplo, **p1=arr+3;** é equivalente a **p1=&arr[3];**
* Tamén se aplica esto se modificamos o valor do punteiro cunha operación como **p1++**, o cal faría que o punteiro pasa a referenciar ao seguinte elemento do vector.
* Con este método, podese desinar, por exemplo, un programa para copiar unha string
* 
  + \*fonte++ devolve o valor apuntado por fonte, e despois avanza 1 posición.
  + \*++fonte avanzaría primeiro e despois devolvería o novo valor.
* Ademais, se dous punteiros fan referencia ao mesmo vector, podemos empregar operadores como **p1<p2** para comprobar cal apunta a unha posición máis avanzada do vector.

**Funcións de asignación de memoria**

* O comando **malloc** recibe como parámetro o número de bytes a reservar. Ademais, especifícase primeiro o tipo de punteiro (para un punteiro xenérico escríbese (void\*)).
  + Os punteiros convértense automáticamente ao tipo apropiado, e pódense converter explicitamente.
  + A función devolve un punteiro, pero devolve NULL se non hai espazo libre.
    - É aconsellable comprobar se v==NULL para evitar erros.
* O comando **realloc** permite redimensionar un vector preservando o seu contido previo.
  + Por exemplo, se V xa foi definido previamente, o seguinte comando permite asignarlle un novo tamaño de m:
    - **v = (int\*) realloc(v, m\*sizeof(int));**
* O commando **free(v)**; libera a memoria do punteiro recibido.
  + Só pode liberar a memoria reservada de xeito dinámico, non estático.

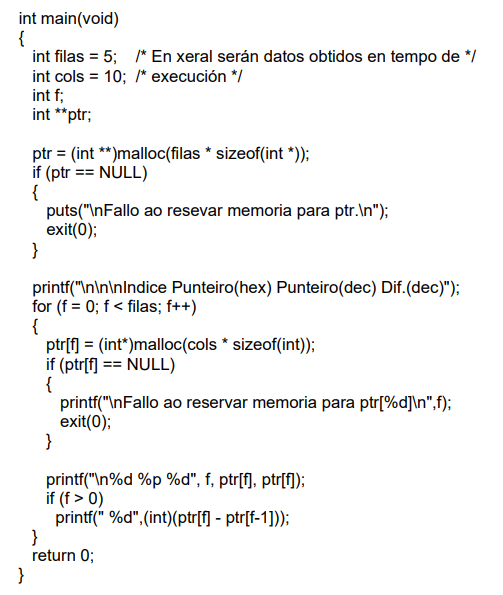
**Vectores dinámicos**

* Coa asignación estática de memoria a un vector, é preciso definir previamente o seu tamaño máximo.
  + A variable representada polo vector almacena o enderezo ocupado polo primeiro elemento do vector.
  + Un vector declarado nunha función non pode ser empregado fóra da mesma.
* Porén, se empregamos un punteiro para reserrvar memoria de forma **dinámica:**
  + A variable de tipo punteiro recibe o enderezo do primeiro elemento.
  + A memoria permanece válida ata que sexa liberada mediante *free*. Por isto, un vector reservado dentro dunha función pódese empregar fóra da mesma.
* Para acceder a un elemento do vector dinámico, efectúase (\*v+k), onde k é a posición á que accedemos.

**Matrices con punteiros**

* Unha matriz, ou un vector do tipo **char** multi[5][10], en realidade trátase dun vector de 5 vectores de 10 caracteres cada un.
* Estos 5 vectores son contiguos en memoria.
* Ao tratar con matrices, ao igual que con vectores, (multi+3) devolve o enderezo da cuarta fila (&multi[3][0])..
* Para acceder ao contido dun elemento, \*( \*(multi+fila)\*ncol +col) é equivalente a multi[fila][col]

**Matrices dinámicas**

* Declarase igual que vectores, por exemplo:
  + float \*mat;
  + mat = (float\*) malloc(m\*n\*sizeof(float))
* Deste xeito,para acceder ao elemento [i][j], emprégase **\*(mat + i\*n + j)**.
* De forma alternativa, tamén se podería definir un vector de punteiros, reservando para cada punteiro tanta memoria como elementos tela cada fila.
* Tamén se poderían empregar dobres punteiros (int \*\*ptr):

**Punteiros e funcións**

* Ao pasar grandes cantidades de datos a un procedemento ou función, no canto de pasar todos os datos como parámetros (moito uso de memoria, pois realízase unha copia) é recomendable pasar o enderezo por referencia.
* A área de memoria dunha variable local só existe durante a execución da función.
  + Se queremos devolver como resultado da función un vector, poderíamos por exemplo pasarlle como parámetro o enderezo dun vector, e que o procedemento garde o resultado neste enderezo.

**Punteiros e estruturas**

* As **estruturas** tamén se poden reservar de xeito dinámico, mediante a función malloc. Exemplo:
  + struct punto\* p;
  + p = (struct punto\*) malloc (sizeof (struct punto));
* Para asignar un valor a unha estrutura definida de forma dinámica, emprégase **->**.
  + p->x = 12.0; asígnalle a p.x o valor 12.0.
* Unha estrutura pode ter un punteiro como campo. Neste caso habería que asignarlle memoria tamén a ese punteiro mediante malloc:
  + Exemplo: se unha estructura ten como campos float \*datos; int dim\_x; e int dim\_; e decimos que dim\_x=N e dim\_y=M:
  + mat->datos = (float\*)malloc(N\*M\*sizeof(float));
* Unha estrutura pode tamén ter unha estrutura como campo, incluso unha do mesmo tipo.