

Unidad III.

PUNTEROS

- 4.1. Fundamentos de punteros
- 4.2. Punteros aplicados cadenas y matrices
- 4.3. Punteros aplicados a listas
- 4.4. Punteros aplicados a árboles
- 4.5. Punteros en otros contextos como FAT e i-nodos
- 4.6. Gestión de la memoria Proyectos con punteros

COMPETENCIA A DESARROLLAR:

El estudiante:

- Programa soluciones aplicadas a la ingeniería, aplicando fundamentos de punteros empleados en cadenas, matrices, listas, árboles; optimizando procesos de gestión de memoria.

4.1. Fundamentos de punteros.

- **Definición:** Un puntero es un dato que contiene una dirección de memoria.

NOTA: Existe una dirección especial que se representa por medio de la constante NULL (definida en <stdlib.h>) y se emplea cuando queremos indicar que un puntero no apunta a ninguna dirección.

- **Declaración:**

<tipo> * <identificador>

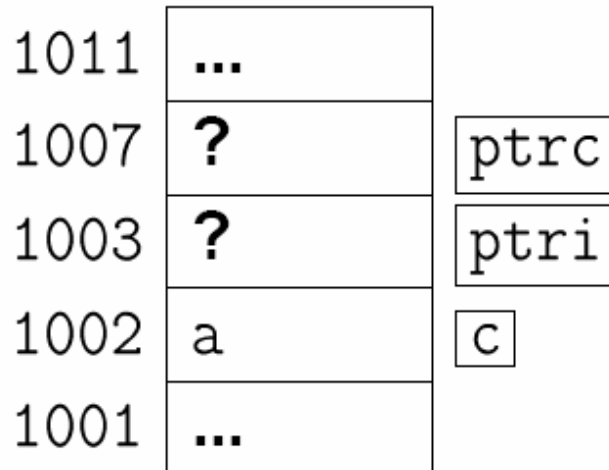
<tipo> Tipo de dato del objeto referenciado por el puntero

<identificador> Identificador de la variable de tipo puntero.

Cuando se declara un puntero se reserva memoria para albergar una dirección de memoria, pero no para almacenar el dato al que apunta el puntero.

El espacio de memoria reservado para almacenar un puntero es el mismo independientemente del tipo de dato al que apunte, es el espacio que ocupa una dirección de memoria.

```
char c = 'a';
char *ptrc;
int *ptri;
```



- **Operaciones básicas con punteros**

Dirección

Operador &

&<identificador> devuelve la dirección de memoria donde comienza el <identificador>.

El operador & se utiliza para asignar valores a datos de tipo puntero.

Ejemplo:

```
int i ;  
int *ptr ;  
...  
ptr = &i;
```

Indirección

Operador *

*<puntero> devuelve el contenido del objeto referenciado por el puntero <puntero>.

El operador * se usa para acceder a los objetos a los que apunta un puntero:

```
char c ;  
  
char *ptr ;  
...  
ptr = &c;  
*ptr = 'A';    // Equivale a escribir: c = 'A'
```

Asignación

Operador =

A un puntero se le puede asignar una dirección de memoria concreta, la dirección de una variable o el contenido de otro puntero.

- **Una dirección de memoria concreta:**

```
int *ptr;  
...  
ptr = 0x1F3CE00A;  
...  
ptr = NULL;
```
- **La dirección de una variable del tipo al que apunta el puntero:**

```
char c;  
char *ptr;  
...  
ptr = &c;
```
- **Otro puntero del mismo tipo:**

```
char c;  
char *ptr1;  
char *ptr2;  
...  
ptr1 = &c;  
ptr2 = ptr1;
```

Como todas las variables, los punteros también contienen "basura" cuando se declaran, por lo que es una buena costumbre inicializarlos con NULL.

Ejemplo:

```
int main ()
{
  int y = 5;
  int z = 3;
  int *nptr;
  int *mptr;
```

```
nptr = &y;
```

```
z = *nptr;
```

```
*nptr = 7;
```

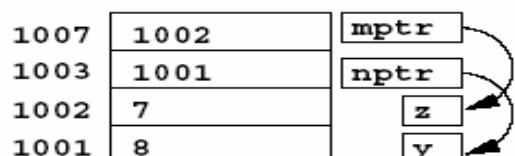
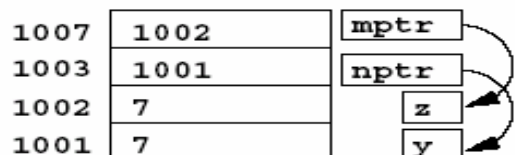
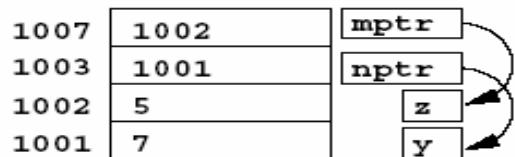
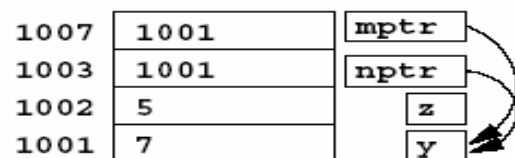
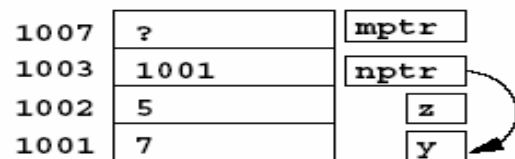
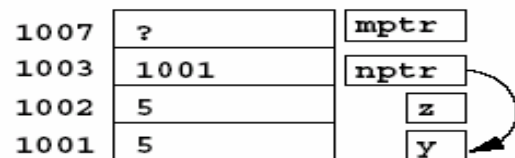
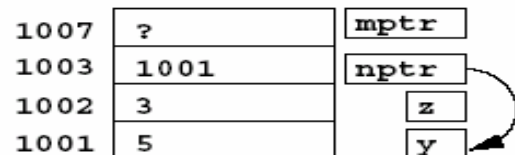
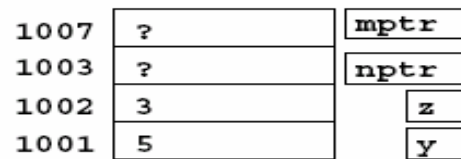
```
mptr = nptr;
```

```
mptr = &z;
```

```
*mptr = *nptr;
```

```
y = (*nptr) + 1;
```

```
return 0;
```



- **Puntero a puntero**

Un puntero a puntero es un puntero que contiene la dirección de memoria de otro puntero


```
int main ()
{
    int a = 5;
    int *p;    // Puntero a entero
    int **q;   // Puntero a puntero

```

1007	?	q
1003	?	p
1001	5	a


```
p = &a;
```

1007	?	q
1003	1001	p
1001	5	a



```
q = &p;
}
```

1007	1003	q
1003	1001	p
1001	5	a



Para acceder al valor de la variable a podemos escribir:

a (forma habitual)

*p (a través del puntero p)

**q (a través del puntero a puntero q)

q contiene la dirección de p, que contiene la dirección de a

4.2. Punteros aplicados a Arreglos

- **Correspondencia entre punteros y vectores**

Cuando declaramos un vector

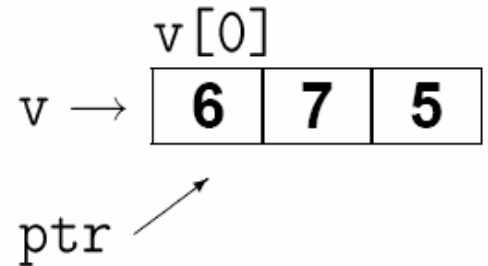
`<tipo> <vector> [tam]`

en realidad:

1. Reservamos memoria para almacenar <tam> elementos de tipo <tipo>.
2. Ahora creamos un puntero <vector> que apunta a la primera posición de la memoria reservada para almacenar los componentes del vector.

Por tanto, el identificador del vector es un puntero.

```
int v[3];
int *ptr;
...
ptr = v; // Equivale a ptr = &v[0]
v[0] = 6; // ≡ *v = 6; ≡ *(&v[0]) = 6
```



- **Aritmética de punteros**

```
<tipo> *ptr;
```

`ptr + <desplazamiento>` devuelve un puntero a la posición de memoria `sizeof(<tipo>) * <desplazamiento>` bytes por encima de `ptr`.

```
int v[];
int *ptr = v; ptr+i apunta a v[i]

*(ptr+i) ≡ v[i]
```

Ejemplo:

```
int suma ( int v[], int N)
{
    int i, suma;
    int *ptr, *ptrfin;

    /* Alternativa 1 */

    suma = 0;
    for (i=0 ; i<N ; i++)
        suma = suma + v[i];

    /* Alternativa
```

4.3 Arreglos Dinámicos

a) **Vectores dinámicos** → Ejemplo de creación, cardado, impresión y ordenamiento de vector dinámico

```
#include <iostream>
using namespace std;

void llenarvector(int *v,int tam);
void mostrarvector(int *v,int tam);
void ordenamiento(int *v,int tam);

int main()
{
    int *vec,tam;
    cout<<"tamaño del vector";
    cin>> tam;
    vec=new int(tam);
    cout<<"\nCARGADO DE VECTOR\n";
    llenarvector(vec,tam);
    cout<<"\nVECTOR ORIGINAL DE VECTOR\n";
    mostrarvector(vec,tam);
    cout<<"\nVECTOR ORDENADO\n";
    ordenamiento(vec,tam);
    mostrarvector(vec,tam);
    cout << endl;
    return 0;
}

void llenarvector(int *v,int tam){
    int i;
    for(i=0;i<tam;i++){
        cout<<"elemento ["<<i<<"]";
        cin>> *(v+i);//v[i];
    }
}

void mostrarvector(int *v,int tam){
    int i;
    for(i=0;i<tam;i++){
        cout<<*(v+i)<<" ";
    }
}
```

```
void ordenamiento(int *v,int tam){
    int i,j,aux;
    for(i=0;i<tam;i++){
        for(j=i+1;j<tam;j++){
            if (*(v+i)>*(v+j)){
                aux=*(v+i);
                *(v+i)=*(v+j);
                *(v+j)=aux;
            }
        }
    }
}
```

b) Matrices dinámicas

```
int **mat;    // mat[50][50]

fil=3
col=3
mat= new int*[fil] //reserva de filas para la matriz

for(i=0;i<fil;i++){
    mat=new int(col); //reserva de columnas para la matriz
}
```

Ejemplo:

c) Ejemplo de creación, cargado e impresión de matriz dinámica

```
#include <iostream>
using namespace std;

//DEFINICION DE PROTOTIPOS DE FUNCIONES
void llenar_matriz(int **mat,int fil,int col);
void mostrar_matriz(int **mat,int fil,int col);
```



```
//PROGRAMA PRINCIPAL

int main()
{int fil,col,i;
  int **puntm;// mat[50][50];
  cout<<"numero de filas?";
  cin>>fil;
  cout<<"numero de columnas?";
  cin>>col;
  puntm=new int*[fil]; //reserva de espacio en la memoria para filas
  for(i=0;i<fil;i++){
    puntm[i]=new int(col); //reserva de espacio en la memoria para columnas
  }
  cout << "LLENADO DE MATRIZ" << endl;
  llenar_matriz(puntm,fil,col);
  cout << " MOSTRARMATRIZ" << endl;
  mostrar_matriz(puntm,fil,col);
  //liberar memoria de las filas
  for(i=0;i<fil;i++){
    delete[] puntm[i];
  }
  //liberar memoria de las columnas
  for(i=0;i<col;i++){
    delete[] puntm[i];
  }
  return 0;
}

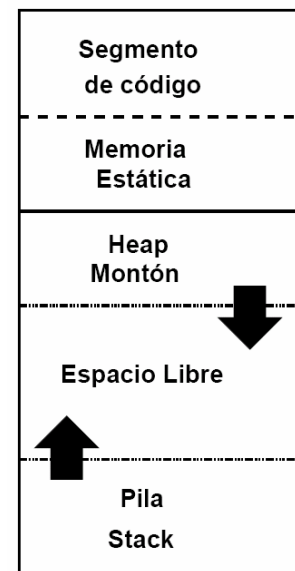
//implementacion de los procedimientos
void llenar_matriz(int **mat,int fil,int col){
  int i,j;
  for(i=0;i<fil;i++){
    for(j=0;j<col;j++){
      cout<<"dato["<<i<<"]["<<j<<"]";
      cin>>*(mat+i+j);//mat[i][j]
    }
  }
}
```

```
void mostrar_matriz(int **mat,int fil,int col){  
    int i,j;  
    for(i=0;i<fil;i++){  
        for(j=0;j<col;j++){  
            cout<<* (mat+i)+j)<<" ";  
        }  
        cout<<endl;  
    }  
}
```

4.4 Gestión de la memoria Proyectos con punteros

- **Organización de la memoria**

- Segmento de código (código del programa).
- Memoria estática (variables globales y estáticas).
- Pila (stack): Variables automáticas (locales).
- Heap ("montón"): Variables dinámicas.



- **Reserva y liberación de memoria**

Cuando se quiere utilizar el heap, primero hay que reservar la memoria que se desea ocupar:

C++: Operador new

Al reservar memoria, puede que no quede espacio libre suficiente, por lo que hemos de comprobar que no se haya producido un fallo de memoria (esto es, ver si la dirección de memoria devuelta es distinta de NULL).

Tras utilizar la memoria reservada dinámicamente, hay que liberar el espacio reservado:

C++: Operador delete

Si se nos olvida liberar la memoria, ese espacio de memoria nunca lo podremos volver a utilizar