

# Tema 4: Memoria dinámica

## Programación 2

---

Grado en Ingeniería Informática  
Universidad de Alicante  
Curso 2024-2025



1. Organización de la memoria
2. Punteros
3. Uso de punteros
4. Referencias
5. Implementación de una pila

# Organización de la memoria

---

# Memoria estática

- Los *datos estáticos* son aquellos cuyo tamaño es fijo y se conoce al escribir el programa
- Las variables que hemos usado hasta ahora son estáticas:

```
int i=0;  
char c;  
float vf[3]={1.0,2.0,3.0};
```

i	c	vf[0]	vf[1]	vf[2]
0		1.0	2.0	3.0
1000	1002	1004	1006	1008

- Permite almacenar grandes volúmenes de datos, cuya cantidad exacta se desconoce al implementar el programa
- Durante la ejecución del programa se ajusta el uso de la memoria a lo que necesita en cada momento
- En C++ se puede hacer uso de la memoria dinámica usando punteros

# Zonas de la memoria

- Durante la ejecución de un programa, se utilizan zonas diferenciadas de la memoria:

Pila ( <i>stack</i> )
Montículo ( <i>heap</i> )
Segmento de datos
Código del programa

- La *pila* almacena los datos locales de una función: parámetros por valor y variables locales
- El *montículo* almacena los datos dinámicos que se van reservando durante la ejecución del programa
- El *segmento de datos* se almacenan los datos de estos tipos, cuyo tamaño se conoce en tiempo de compilación
- El propio código también se almacena en la memoria, como los datos

# Punteros

---

# Definición y declaración

- Un *puntero* almacena la dirección de memoria donde se encuentra otro dato
- Se dice que el puntero “apunta” a ese dato
- Los punteros se declaran usando el carácter \*
- El dato al que apunta el puntero será de un tipo concreto que deberá indicarse al declarar el puntero:

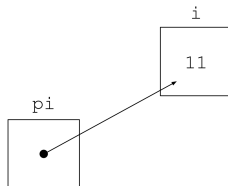
```
int *punteroEntero; // Puntero a entero
char *punteroChar; // Puntero a carácter
int *vecPunterosEntero[20]; // Array de punteros a entero
double **doblePunteroReal; // Puntero a puntero a real
```



## Operadores de punteros (1/2)

- El operador `*` permite acceder al contenido de la variable a la que apunta el puntero
- El operador `&` permite obtener la dirección de memoria en la que está almacenada una variable:

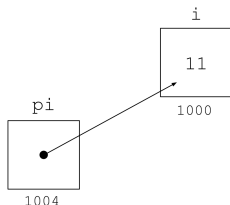
```
int i=3;  
int *pi;  
pi=&i; // pi contiene la dirección de memoria de i  
*pi=11; // Contenido de pi es "11". Por lo tanto i = 11
```



## Operadores de punteros (2/2)

- Suponiendo que `i` está en la posición de memoria 1000 y que `pi` está en la 1004:

```
int i=11;  
int *pi;  
pi=&i;  
cout << pi << endl; // Muestra "1000"  
cout << *pi << endl; // Muestra "11"  
cout << &pi << endl; // Muestra "1004"
```



# Declaración con inicialización

- Como cualquier otra variable, podemos inicializar un puntero en el momento de su declaración:

```
int *pi=&i; // pi contiene la dirección de i
```

- Cuando queremos indicar que un puntero no apunta a ningún dato válido le asignamos el valor `NULL`:

```
int *pi=NULL;
```

- `NULL` es una constante entera con valor cero. A partir del estándar C++ 2011 se puede usar la constante `nullptr` que representa el cero como una dirección de memoria (tipo puntero)

## Ejercicio 1

Indica cuál sería la salida por pantalla de estos fragmentos de código:

```
int e1;  
int *p1,*p2;  
e1=7;  
p1=&e1;  
p2=p1;  
e1++;  
(*p2)+=e1;  
cout << *p1;
```

```
int a=7;  
int *p=&a;  
int **pp=&p;  
cout << **pp;
```

# Uso de punteros

---

## Reserva y liberación de memoria (1/2)

- El operador `new` permite reservar memoria de manera dinámica durante la ejecución del programa
- Devuelve la dirección de inicio de la memoria reservada
- Si no hay suficiente memoria para la reserva, devuelve `NULL`
- Se debe usar un puntero para almacenar la dirección que devuelve `new`:

```
double *pd;  
pd=new double; // Reserva memoria para un double  
if(pd!=NULL){ // Comprueba que se ha podido reservar  
    *pd=4.75;  
    cout << *pd << endl; // Muestra "4.75"  
}
```



## Reserva y liberación de memoria (2/2)

- El operador `delete` permite liberar memoria reservada con `new`:

```
double *pd;  
pd=new double; // Reserva memoria  
...  
delete pd; // Libera la memoria apuntada por pd  
pd=NULL; // Conveniente si vamos a seguir usando pd
```

- Siempre que se reserva con `new` hay que liberar con `delete`
- Un puntero se puede reutilizar tras liberar su contenido y reservar memoria otra vez con `new`:

```
double *pd;  
pd=new double; // Reserva memoria  
...  
delete pd; // Libera la memoria apuntada por pd  
pd=new double; // Reservamos de nuevo memoria  
...
```

## Punteros y arrays (1/3)

- Existe una estrecha relación entre los punteros y los arrays
- La variable de tipo array es en realidad un puntero al primer elemento del array:

```
int vec[5]={4,5,2,8,12};  
cout << vec << endl; // Muestra la dirección de memoria  
                        // del primer elemento del array  
cout << *vec << endl; // Muestra "4"
```

- Siempre apunta al primer elemento del array y no se puede modificar



## Punteros y arrays (2/3)

- Los punteros se pueden usar como accesos directos a componentes de arrays:

```
int vec[20];  
int *pVec=vec; // Ambos son punteros a entero  
*pVec=58; // Equivalente a vec[0]=58;  
pVec=&(vec[7]);  
*pVec=117; // Equivalente a vec[7]=117;
```

## Punteros y arrays (3/3)

- Los punteros también pueden usarse para crear *arrays dinámicos*
- Para reservar memoria para un array dinámico hay que usar corchetes y especificar el tamaño
- Para liberar toda la memoria reservada es necesario también usar corchetes (vacíos):

```
int *pv;  
pv=new int[10]; // Reserva memoria para 10 enteros  
pv[0]=585; // Accedemos como en un array estático  
...  
delete [] pv; // Liberamos toda la memoria reservada
```

# Punteros definidos con typedef

- Como vimos en el *Tema 1*, se pueden definir nuevos tipos de datos con typedef:

```
typedef int entero;  
entero a,b; // Equivalente a int a,b;
```

- Para facilitar la claridad en el código pueden definirse los punteros con typedef:

```
typedef int *tPunteroEntero;  
tPunteroEntero pi; // Variable de tipo puntero a entero  
// No hay que poner * al declararla
```

- Cuando un puntero referencia a un registro, se puede usar el operador `->` para acceder a sus campos:

```
struct TRegistro{
    char c;
    int i;
};
typedef TRegistro *TPunteroRegistro;

TPunteroRegistro pr;
pr=new TRegistro;
pr->c='a'; // Equivalente a (*pr).c='a';
pr->i=88;  // Equivalente a (*pr).i=88;
```

## Punteros como parámetros de funciones (1/2)

- Un puntero, como cualquier otra variable, se puede pasar como parámetro por valor o por referencia a una función:

```
void funcValor(int *p){ // Paso por valor  
    ...  
    p=NULL;  
}  
void funcReferencia(int *&p){ // Paso por referencia  
    ...  
    p=NULL;  
}  
int main(){  
    int i=0;  
    int *p=&i;  
    funcValor(p);  
    // p sigue apuntando a i  
    funcReferencia(p);  
    // p vale NULL  
}
```

## Punteros como parámetros de funciones (2/2)

- El mismo ejemplo de antes usando typedef:

```
typedef int* tPunteroEntero;
void funcValor(tPunteroEntero p) {
    ...
    p=NULL;
}
void funcReferencia(tPunteroEntero &p) {
    ...
    p=NULL;
}
int main() {
    int i=0;
    tPunteroEntero p=&i;
    funcValor(p);
    funcReferencia(p);
}
```

# Errores comunes (1/2)

- No liberar la memoria reservada dinámicamente:

```
void func(){  
    int *pEntero=new int;  
    *pEntero=8;  
    return; // ¡Error! Falta delete pEntero;  
}
```

- Utilizar un puntero que no apunta a nada:

```
int *pEntero;  
*pEntero=7; // ¡Error! pEntero sin inicializar
```

## Errores comunes (2/2)

- Usar un puntero tras haberlo liberado:

```
int *p,*q;  
p=new int;  
...  
q=p;  
delete p;  
*q=7; // ;Error! La memoria ya se había liberado
```

- Liberar memoria no reservada con new:

```
int *pEntero=&i;  
delete pEntero; // ;Error! Apunta a memoria estática
```



## Ejercicio 2

Dado el siguiente registro:

```
struct tCliente{  
    char nombre[32];  
    int edad;  
}tCliente;
```

Realiza un programa que lea un cliente (sólo uno) de un fichero binario, lo almacene en memoria dinámica usando un puntero, imprima su contenido y finalmente libere la memoria reservada.

## Referencias

---

## Referencias (1/4)

- Las referencias de C++ son como punteros pero con una sintaxis menos recargada (*azúcar sintáctica*)
- No hay nada que puedas hacer con referencias que no puedas hacer con punteros

```
int a=10;
int *b=&a; // Variable puntero
*b=20;
cout << a << " " << *b; // Muestra "20 20"
int &c=a; // Variable referencia
c=30;
cout << a << " " << c; // Muestra "30 30"
```

- En el código anterior, `c` se puede considerar como un segundo nombre para `a`

- Las referencias no pueden ser `NULL`, siempre están conectadas a un dato
- Una vez se ha inicializado una referencia, no se puede hacer que se refiera a una posición de memoria diferente, pero esto sí es posible con punteros
- Al crear una referencia siempre hay que inicializarla, pero los punteros se pueden inicializar en cualquier momento tras su declaración

## Referencias (3/4)

- Las referencias simplifican el código de las funciones que tienen parámetros pasados por referencia
- La siguiente función usa punteros para pasar dos parámetros por referencia:

```
void swap(int *x, int *y) {  
    int temp=*x;  
    *x=*y;  
    *y=temp;  
}  
  
int main() {  
    int a=10, b=20;  
    swap(&a, &b);  
    cout << a << " " << b;    // Muestra "20 10"  
}
```

## Referencias (4/4)

- La siguiente función es equivalente a la anterior, pero usa referencias en lugar de punteros:

```
void swap(int &x,int &y){  
    int temp=x;  
    x=y;  
    y=temp;  
}  
  
int main(){  
    int a=10,b=20;  
    swap(a,b);  
    cout << a << " " << b;    // Muestra "20 10"  
}
```

- Esta es la sintaxis que hemos estado usando en la asignatura
- Es más sencilla y cómoda que la del ejemplo anterior

# Implementación de una pila

---

# Implementación de una pila (1/6)

- Una *pila* es una estructura de datos muy usada en programación
- Consiste en una lista de elementos
- Se puede añadir o eliminar elementos a una pila con una restricción: el último elemento añadido (*push*) será el primer elemento en ser sacado (*pop*)
- Ejemplos de pila en el mundo real
  - En una pila de platos, el plato que está encima y que acaba de ser apilado siempre será el primero en ser desapilado
  - Los carritos de la compra del supermercado, donde siempre se coge el último que hayan dejado



## Implementación de una pila (2/6)

- Una pila puede implementarse usando vectores de tamaño fijo, pero esto limitará el número de elementos que pueden apilarse
- Podemos solucionarlo (parcialmente) usando un vector muy grande, pero si apilamos pocos elementos estaremos desperdiciando memoria
- Usar punteros para implementar la pila permite usar solo la memoria que necesitamos en cada momento
- Se puede implementar usando la idea de *lista enlazada*
  - Al apilar un nuevo elemento se reserva dinámicamente espacio en memoria para un registro
  - Este registro contiene los datos a guardar y un puntero al último elemento de la pila
  - Tendremos un puntero (llamado `head`) que siempre apuntará a la cima de la pila

## Implementación de una pila (3/6)

- En la siguiente implementación el puntero `head` se pasa como parámetro a las diferentes funciones
- Se pasa por referencia cuando alguna de estas funciones puede cambiar el puntero para que apunte a otro registro
- Estructura de un elemento (nodo) de la pila:

```
struct Node{  
    int data; // Información que queremos almacenar  
    struct Node *next; // Puntero al siguiente elemento  
};
```

## Implementación de una pila (4/6)

- Funciones para apilar (push) y desapilar (pop) elementos:

```
void push(Node *&head, int newData) {  
    Node *newNode = new Node; // Reservamos memoria  
    newNode->data = newData; // Guardamos los datos  
    newNode->next = head; // Apuntamos al último nodo  
    head = newNode; // head apunta al nuevo nodo  
}  
  
void pop(Node *&head) {  
    Node *ptr;  
    if (head != NULL) { // Nos aseguramos que hay elementos  
        ptr = head->next; // Segundo elemento de la pila  
        delete head; // Borramos la cima  
        head = ptr; // head apunta ahora al segundo elemento  
    }  
}
```

# Implementación de una pila (5/6)

- Funciones para mostrar (display) y vaciar (destroy) la pila:

```
void display(Node *head){
    Node *ptr;
    ptr=head;
    while(ptr!=NULL){ // Hasta recorrer toda la pila
        cout << ptr->data << " "; // Mostramos los datos
        ptr=ptr->next; // Pasamos al siguiente elemento
    }
}

void destroy(Node *&head){
    Node *ptr,*ptr2;
    ptr=head;
    while(ptr!=NULL){ // Hasta recorrer toda la pila
        ptr2=ptr; // Vamos a eliminar el nodo actual
        ptr=ptr->next; // Apuntamos al siguiente elemento
        delete ptr2; // Borramos el nodo actual
    }
    head=NULL; // La pila ya está vacía
}
```

# Implementación de una pila (6/6)

- Ejemplo de función principal usando dos pilas:

```
int main(){
    // Declaramos e inicializamos las dos pilas
    Node *head1=NULL;
    Node *head2=NULL;
    // Añadimos tres elementos a la primera pila
    push(head1,3);
    push(head1,1);
    push(head1,7);
    display(head1); // Muestra "7"
    pop(head1); // Eliminamos la cima
    display(head1); // Muestra "1"
    destroy(head1); // Vaciamos la primera pila
    // Añadimos un elemento a la segunda pila
    push(head2,9);
    display(head2); // Muestra "9"
    destroy(head2); // Vaciamos la segunda pila
}
```