Tema 6. Gestión dinámica de memoria

`http://aulavirtual.uji.es`

José M. Badía, Begoña Martínez, Antonio Morales y José M. Sanchiz

{badia, bmartine, morales, sanchiz}@icc.uji.es

Estructuras de datos y de la información

Universitat Jaume I

Índice

1. Punteros en C++ 5

2. Memoria dinámica en C++ 10

3. Pila con un vector dinámico 13

4. Pila enlazada 26

Bibliografía

- (Nyhoff'99), apartados 2.4, 3.4 y capítulo 7
- (Main y Savitch'01), capítulos 4 y 5 y apartado 7.3

Objetivos

- Recordar el concepto y uso de los punteros y sus operaciones asociadas.
- Recordar el concepto y uso de memoria dinámica.
- Saber implementar en C++ una versión dinámica de un tipo de datos.
- Saber implementar en C++ una versión enlazada de un tipo de datos.

1 Punteros en C++

Puntero: Variable que contiene una dirección de memoria.

$\equiv$ Apunta a una posición en memoria

Declaración de punteros:

```
tipo * puntero [ = dirección ];
```

- Sólo podrá apuntar a variables del `tipo`.
- No se reserva espacio para la variable del `tipo`.

Ejemplo:

```
int i;
double *dPtr, *ePtr; // * delante de cada variable puntero
int *cPtr = &i // & devuelve la dirección de una variable
```

1 Punteros en C++ (II)

- **Asignación:**

```
dPtr = ePtr;
ePtr = &x;
dPtr = 0 // Puntero nulo (NULL)
```

- **Indirección:** (≡ Acceso a la variable apuntada)

```
y = *dPtr;
x = y + *ePtr;
*ePtr = x;
```

- **Comparación:** (sólo entre punteros al mismo tipo base)

```
dPtr == ePtr
ePtr != &x
```

1 Punteros en C++ (III)

Punteros a clases

```
class complejo {
  public:
    complejo & Inicia (double r, double i);
     void Muestra() const;
  private:
    double real, imag;
};
...
complejo c;
complejo *ptrc = &c;
(*ptrc).Inicia(3.0, 1.5);
ptrc->Inicia(1.2, -5.4);
```

El puntero *this*

En las clases, la palabra reservada *this* es un puntero al objeto

Ejemplo

```
complejo & Inicia (double r, double i) {
  real = r;
  imag = i;
  return *this; // Devuelve el objeto
}
...
complejo c;
c.Inicia(3.0, -2.3).Muestra();
```

Aritmética con punteros

```
int vector[4] = { 0, 1, 2, 3 };
int *ptr = vector;             //  ≡ int *ptr = &(vector[0])
cout << *ptr;                  // Muestra vector[0]
ptr++;                         // Añade sizeof(int), NO 1
cout << *ptr;                  // Muestra vector[1]
ptr = vector;
for (int i=0; i<4; i++) {      // Recorrido del vector
  cout << *ptr;
  ptr++;
}
```

2 Memoria dinámica en C++

Problema

Las variables estáticas tienen un tamaño fijo.

Solución

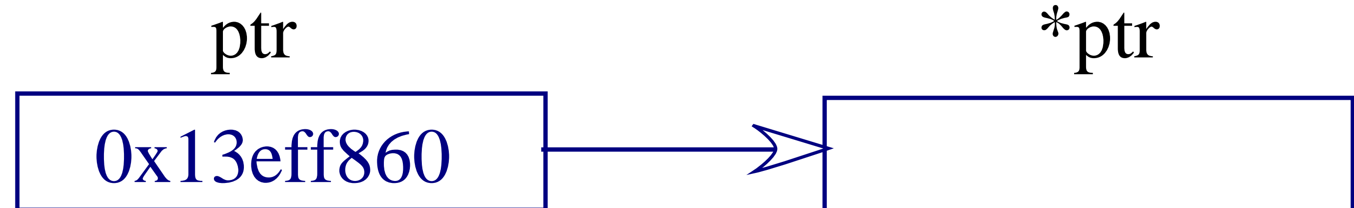
C++ permite reservar y liberar zonas de memoria en tiempo de ejecución.

Reserva de memoria

```
new tipo;
```

Reserva espacio para una variable del `tipo` y devuelve su dirección.

```
int *ptr;
ptr = new int;
```

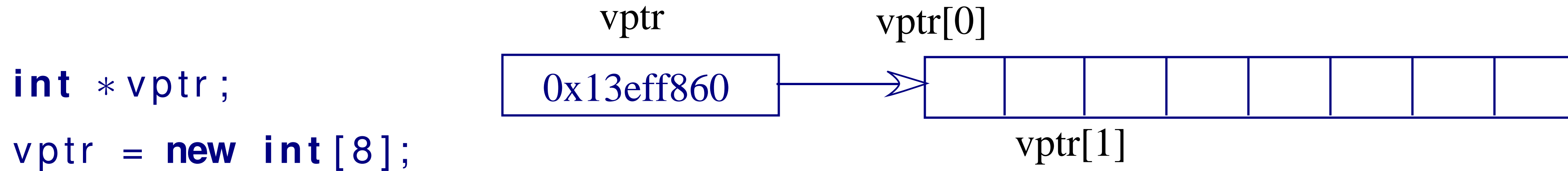


2 Memoria dinámica en C++ (II)

`new tipo[num];`

Reserva espacio para `num` posiciones de `tipo` y devuelve la dirección inicial.

```
int *vptr;
vptr = new int[8];
```



Ejemplo

```
cout << "Numero de elementos? ";
cin >> tam;
int *ptr = new int[tam];
 ...
for (int i=0; i<tam; i++)
  ptr[i] = i;
```

2 Memoria dinámica en C++ (III)

Liberación de memoria

```
delete puntero;
delete [] puntero;
```

La memoria liberada puede reutilizarse en reservas posteriores.

Ejemplo

```
double *ptr = new double[4];
...
ptr[2] = 1.5;
cout << ptr[2];  // Escribe 1.5
delete [] ptr;
cout << ptr[2]; // Valor indefinido. Posible error
```

3 Pila con un vector dinámico

```
const int MaxElem = 128;
typedef int tipobase;
class Stack {
  public:
    // stack, empty, push, pop y top
  private:
    tipobase elementos[MaxElem];
    int tope;
};
```

- Cada vez que se crea un objeto de la clase, se reserva espacio para todos los datos de la misma: vector de elementos y tope.
- La reserva se mantiene hasta salir del ámbito donde está declarado.

3 Pila con un vector dinámico (II)

Ejemplo de gestión estática de la memoria

```
int Fondo(stack  p);
...
int main() {
  stack p;       // Creación de una pila
  cout << "Fondo: " << Fondo(p);      // Copia + destrucción
  if ( !p.empty() ){
    int i;
    stack otra;      // Creación de otra pila
    i=Fondo(otra);      // Copia + destrucción
  }       // Destrucción de la pila otra
}       // Destrucción de la pila p
```

3 Pila con un vector dinámico (III)

Clase Pila con vector dinámico

```
class stack {
  public:
    stack(int numelem = 128);
    ~stack();
    stack(const stack &origen);
    stack & operator= (const stack &origen);
    //  empty, push, pop y top
  private:
      tipobase *elementos;
      int capacidad, tope;
};
```

3 Pila con un vector dinámico (IV)

En clases que utilicen memoria dinámica es necesario crear nuevas funciones miembro y modificar otras:

- **Constructores**: Reservan espacio para los nuevos objetos.
- **Destructores**: Liberan el espacio reservado para los objetos.
- **Constructores de copia**: Construyen objetos como copias de otros.
- **Operadores de asignación**: Asignan unos objetos a otros del mismo tipo.

3 Pila con un vector dinámico (V)

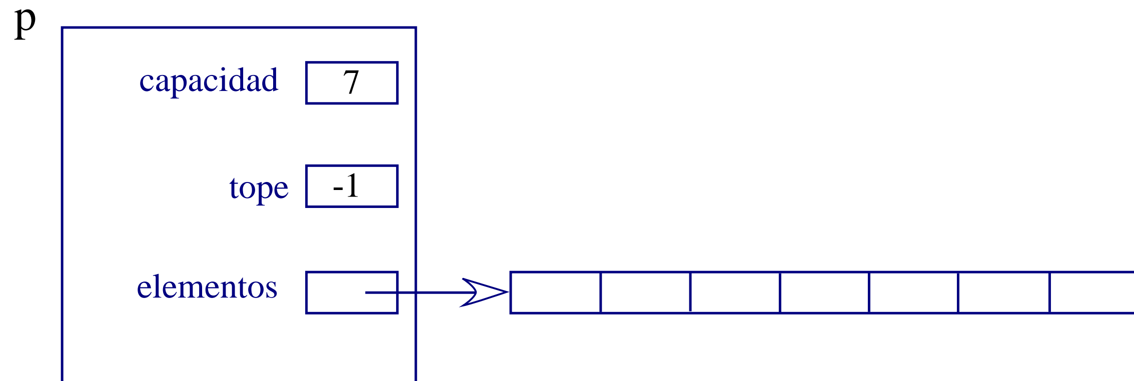
Constructor

```
stack(int numelem) {
  capacidad = numelem;
  elementos = new tipobase[capacidad];
  if (elementos == 0) {
    cerr << "Error al reservar espacio para la pila" << endl;
    exit(-1);
  }
  tope = -1;
}
stack p;  // Reserva espacio para 128 datos de tipobase
cin >> num;
stack q(num);  // Reserva espacio para num datos de tipobase
```

3 Pila con un vector dinámico (VI)

Destructor

```
stack p(7);
```



¿Qué ocurre con el objeto al salir de su ámbito?

El compilador inserta una llamada al destructor del objeto.

- La memoria para los datos estáticos (capacidad, tope y el puntero elementos) se libera automáticamente.
- La memoria dinámica reservada para los elementos NO se libera $\Rightarrow$ Necesitamos un destructor que la libere.

3 Pila con un vector dinámico (VII)

Destructor

```
~stack ( ) {
  delete [] elementos;
}
```

- Nombre de la clase precedido por ~
- No tiene parámetros $\Rightarrow$ Definición única.
- No devuelve nada (ni *void*).

3 Pila con un vector dinámico (VIII)

Constructor de copia

¿Qué ocurre cuando: ?

- Inicializamos un objeto

```
stack pila1(10);  // Rellenamos pila1 por teclado
stack pila2 = pila1;
stack pila3(pila1);
```

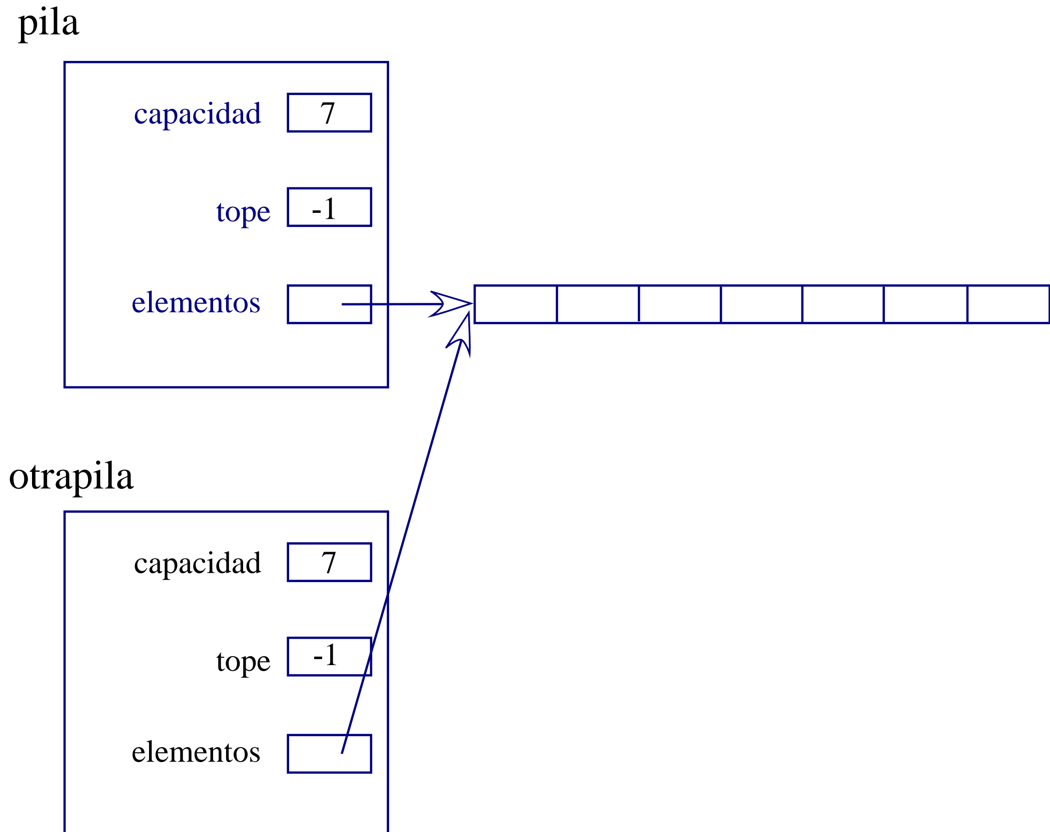
- Pasamos un objeto por valor o lo devuelve una función

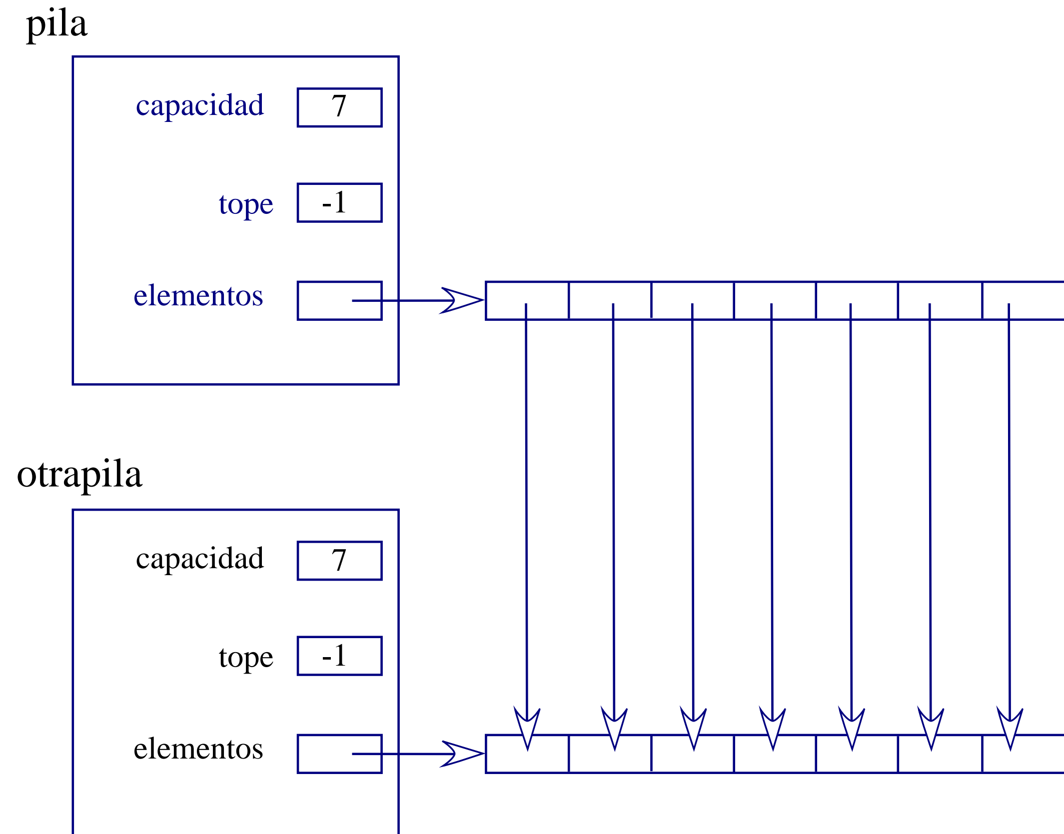
```
stack Invertir(stack p);
```

El constructor de copia por defecto copia los miembros estáticos del objeto byte a byte (capacidad, tope y el puntero elementos).

3 Pila con un vector dinámico (IX)

```
stack otrapila(pila);
```





3 Pila con un vector dinámico (X)

Constructor de copia

```
stack(const stack &origen) {
  capacidad = origen.capacidad;
  elementos = new tipobase[capacidad];
  if (elementos == 0) {
    cerr << "Error al reservar espacio para la pila" << endl;
    exit(-1);
  }
  for (int pos=0; pos<capacidad; pos++)
    elementos[pos] = origen.elementos[pos];
  tope = origen.tope;
}
```

3 Pila con un vector dinámico (XI)

Operador de asignación

Cuando realizamos una asignación:

copiapila = pila ;

Si la clase reserva memoria dinámica, incluir un operador de asignación.

stack & **operator**=(**const** stack &origen);

Similar al operador de copia, pero:

- Debe liberar y reservar la memoria dinámica adecuada para el objeto de la parte izquierda de la asignación.
- Debe tener en cuenta la autoasignación: `pila = pila`.
- Debe devolver la pila que contiene la función: `*this`.

3 Pila con un vector dinámico (XII)

```
stack & operator=(const stack &origen) {
  if (this != &origen) {   // Evita la autoasignación
    delete [] elementos;
    capacidad = origen.capacidad;
    elementos = new tipobase[capacidad];
    if (elementos == 0) {
      cerr << "Error al reservar espacio para la pila" << endl;
      exit(-1);  }
    for (int pos=0; pos<capacidad; pos++)
      elementos[pos] = origen.elementos[pos];
    tope = origen.tope;
  }
  return *this;
}
```

3 Pila con un vector dinámico (XIII)

Ejemplo de uso

```
void Imprime (stack p) {...}
int main() {
  int num;
  cout << "Numero de elementos? ";
  cin >> num;
  stack p(num);         // Constructor
  for (int i=0; i<num; i++) p.push(i);
  stack q = p;          // Constructor de copia
  Imprime(q);           // Constructor de copia + destructor
  stack u;              // Constructor
  u = q;                // Operador de asignación
  Imprime(u);           // Constructor de copia + destructor
}                       // destructor + destructor + destructor
```

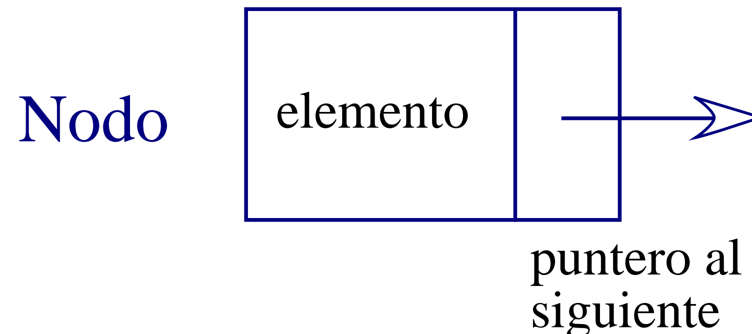
4 Pila enlazada

Consideraciones

- Sólo podemos acceder a la pila a través del tope: única posición visible.
  Pila $\equiv$ Puntero al tope.

- La estructura es ordenada $\Rightarrow$ necesitamos un mecanismo de enlace entre un elemento y el siguiente.

Cada elemento de la pila estará contenido en un nodo con dos componentes:



4 Pila enlazada (II)

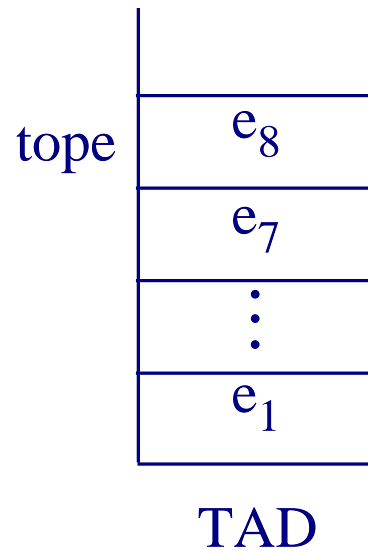
Definición del nodo

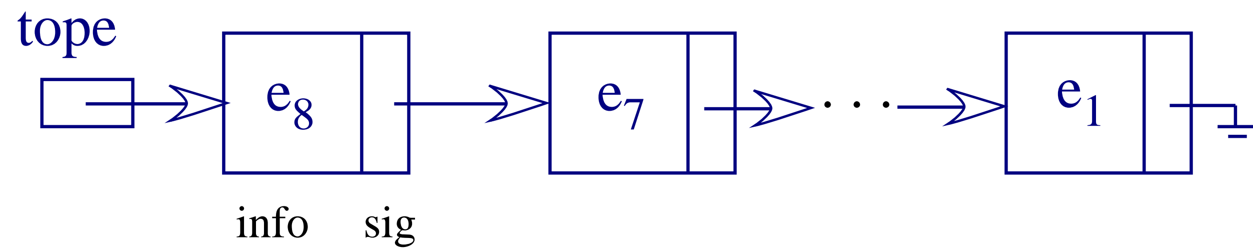
```
class nodo {
  public:
    tipobase info;
    nodo *sig;
    // Constructor
    nodo(const tipobase &valor=tipobase(), nodo *siguiente=NULL):
        info(valor), sig(siguiente) { }
};
```

- **Estructura recursiva:**

  La clase nodo contiene un campo que apunta a un objeto de la misma clase.

Pila enlazada





4 Pila enlazada (IV)

Clase pila enlazada

```
class stack {
  public:
    // Constructores y destructores de la pila
    // Operaciones: empty, pop, push, top
  private:
    class nodo {  // Sólo acceden miembros y amigos de la pila
      // datos y constructor
    };
    nodo * tope;  // Pila ≡ puntero al nodo tope
};
```

4 Pila enlazada (V)

Constructor y destructor

```
stack() {
  tope = NULL;
}

~stack() {
  nodo *aux;
  while (!empty()) {
    aux = tope;
    tope = aux->sig;
    delete aux;
  }
}
```

4 Pila enlazada (VI)

Operaciones de consulta

```
bool empty() const {
  return (tope == NULL);
}

const tipobase & top() const {
  if (!empty())
    return (tope->info);
}
```

4 Pila enlazada (VII)

```
void push(tipobase & valor) {
  nodo *aux = new nodo(valor, tope);
  tope = aux;
}

void pop() {
  nodo *aux;
  if (!empty()) {
    aux = tope;
    tope = tope->sig;
    delete aux;
  }
}
```