# Contenedores, referencias, punteros, pasos por valor

## Por: Karla Verónica Quiros Delgado

### Pasos por valor

Cuando se pasa un argumento por valor, se crea una copia del valor de la variable original y se pasa a la función. Por lo tanto, cualquier modificación realizada en la función no afecta a la variable original.

```
void modificar(int a) {
    a = 20; // Modifica la copia de a, no la variable original
}

int main() {
    int a = 10;
    modificar(a);
    cout << a << endl; // Imprime 10, no 20 porque la modificación se hizo en la
copia de a
    return 0;
}
```

### Referencias

Una referencia es otra forma de acceder a una variable existente mediante otro nombre. Se caracterisan por:

- Ser un alias de una variable existente.
- No poder ser nula.
- Se inicializan al momento de crearse.
- Al modificar una referencia, se modifica la variable a la que hace referencia y viceversa.

La sintaxis para declarar una referencia es: tipo_dato &nombre_referencia = nombre_variable;

Ejemplo:

```
int a = 10;
int &b = a; // b es una referencia de a

/* Al modificar b, se modifica a */
b = 20;
cout << a << endl; // Imprime 20
```

Las referencias son útiles cuando deseas pasar argumentos a funciones por referencia para modificar los valores originales o cuando deseas evitar copias innecesarias de objetos grandes.

```
void duplicar(int &a) {
    a=a*2;// Al modificar a, se modifica la variable original
}

int main() {
    int a = 10;
    duplicar(a);
    cout << a << endl; // Imprime 20
    return 0;
}
```

## Punteros

Un puntero es una variable que almacena la dirección de memoria de otra variable. Se utiliza para acceder y manipular datos almacenados en esa dirección de memoria. Un puntero se declara utilizando el tipo de dato seguido de un asterisco (*).

Características de los punteros:

- Los punteros se utilizan para acceder a una variable mediante su dirección de memoria.
- Se inicializan con nullptr, new o la dirección de memoria de una variable.
- Se debe liberar la memoria reservada con un delete.

Estructura de declaración de un puntero:

```
tipo_dato *nombre_puntero;
```

Estructura de inicialización de un puntero:

```
tipo_dato *nombre_puntero = new tipo_dato (valor_inicial);
```

Ejemplo:

```
int *puntero; // Puntero a un entero
int * num= new int(10); // Puntero a un entero inicializado en 10
Persona * persona = new Persona("Juan", 20); // Puntero a un objeto Persona
```

## Aprendiendo a dereferenciar punteros

Cuando a un puntero se le antepone el operador de desreferenciación (*), se accede al valor almacenado en la dirección de memoria a la que apunta el puntero. Esto se llama desreferenciar el puntero.

```
int a = 10;
int *puntero = &a; // Puntero a la dirección de memoria de a
cout << *puntero << endl; // Imprime 10
```

Aquí, puntero apunta a la dirección de memoria de la variable `a`. Al usar `*puntero`, estás desreferenciando el puntero, lo que significa que estás accediendo al valor almacenado en la dirección de memoria a la que apunta puntero, que es `a`. Por lo tanto, `*puntero`en este caso accede al valor 10 almacenado en la variable a y lo imprime.

### TIPS

- `&variable`: Devuelve la dirección de memoria de la variable.
- `*puntero`: Devuelve el valor almacenado en la dirección de memoria a la que apunta el puntero.

### Punteros nulos

Un puntero nulo es un puntero que no apunta a ninguna dirección de memoria. Se declara con el valor `nullptr` y se utiliza para indicar que el puntero no apunta a ninguna dirección de memoria.

```
int *puntero = nullptr; // Puntero nulo
```

### Liberando memoria con delete

Cuando se reserva memoria con el operador `new`, es necesario liberarla con el operador `delete` para evitar fugas de memoria.

Cuando se libera la memoria de un puntero, antes debe asegurarse de que ese puntero no sea nulo y que su dirección de memoria no este siendo utilizada por otra variable.

**Caso simple**

El siguiente código reserva memoria para un puntero a un entero y luego libera la memoria reservada.

```
int *puntero = new int(10); // Puntero a un entero inicializado en 10
delete puntero; // Libera la memoria reservada
```

**Caso en que la memoria esta siendo utilizada por otra variable**

```
int *puntero = new int(10); // Puntero a un entero inicializado en 10
int *puntero2 = puntero; // puntero2 apunta a la dirección de memoria de puntero
delete puntero; // Libera la memoria reservada
cout << *puntero2 << endl; // Error: puntero2 apunta a una dirección de memoria
que ya no existe
```

En lo anterior, puntero2 apunta a la dirección de memoria de puntero. Cuando se libera la memoria de puntero, puntero2 apunta a una dirección de memoria que ya no existe. Por lo tanto, al desreferenciar puntero2, se produce un error.

## Tiempo de ejecución y compilación

### Tiempo de ejecución

El tiempo de ejecución es el período durante el cual un programa se ejecuta. En este momento, el programa se ejecuta y realiza las tareas que se le han asignado.

### Tiempo de compilación

El tiempo de compilación es el período durante el cual el código fuente se convierte en código de máquina. En este momento, el compilador verifica el código fuente para detectar errores de sintaxis y genera código de máquina.

### Errores de tiempo de ejecución

Los errores de tiempo de ejecución son errores que ocurren durante el tiempo de ejecución. Estos errores no se detectan durante el tiempo de compilación y, por lo tanto, el programa se ejecuta hasta que se encuentra un error al ejecutar alguna instrucción del programa.

### Errores de tiempo de compilación

Los errores de tiempo de compilación son errores de sintaxis que se detectan durante el tiempo de compilación. Estos errores se deben a una sintaxis incorrecta en el código fuente y, por lo tanto, el compilador no puede generar código de máquina para el programa, evitando que se ejecute.

## Contenedores

Los contenedores son objetos que almacenan otros objetos, proporcionan una forma conveniente de almacenar datos y proporcionan métodos para acceder y manipular los datos almacenados.

- Tienen un tamaño.
- Manejan una cantidad de elementos.
- Tiene un tipo de dato.

### Tipos de creación de contenedores

Clase Persona a utilizar en los ejemplos:

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

class Persona{
    private:
        string nombre;
        int edad;
    public:
        Persona();
        Persona(string,int);
        ~Persona();
        string toString();
};
```

```
Persona::Persona(){
    this->nombre="";
    this->edad=0;
}
Persona::Persona(string nombre,int edad){
    this->nombre=nombre;
    this->edad=edad;
}
Persona::~Persona(){
    cout<<"Se destruyo el objeto"<<endl;
}
string Persona::toString(){
    stringstream s;
    s<<"Nombre: "<<nombre<<endl;
    s<<"Edad: "<<edad<<endl;
    return s.str();
}
```

**Forma 1**

La forma 1 fija el tamaño al momento de crear el objeto, por lo que no se puede modificar el tamaño del contenedor, se reserva memoria estática para n elementos, sin embargo, es probable que no se utilice toda la memoria reservada. Por lo tanto, esta forma de crear contenedores no es eficiente.

```
// .h forma 1
#include <iostream>
#include <sstream> // para usar el stringstream
using namespace std;// para no escribir std::cout
class ContenedorUno{
    private:
        int vector[10];
        int cantidad;
        int tamano;
    public:
        ContenedorUno();//constructor
        ~ContenedorUno();//destructor
        void agregar(int);
        void eliminar(int);
        string imprimir();
}

// .cpp forma 1
#include "ContenedorUno.h"
ContenedorUno::ContenedorUno(){
    cantidad=0;
    tamano=10;
    for(int i=0;i<tamano;i++){
        vector[i]=0;
    }
}
```

```
ContenedorUno::~ContenedorUno(){
    cout<<"Se destruyo el objeto"<<endl;
}
void ContenedorUno::agregar(int valor){
    if(cantidad<tamano){
        vector[cantidad++]=valor;
    }
}
void ContenedorUno::eliminar(int valor){
    for(int i=0;i<cantidad;i++){
        if(vector[i]==valor){
            for(int j=i;j<cantidad-1;j++){
                vector[j]=vector[j+1];
            }
            cantidad--;
            break;
        }
    }
}
string ContenedorUno::imprimir(){
    stringstream s;
    for(int i=0;i<cantidad;i++){
        s<<vector[i]<<endl;
    }
    return s.str();
}
```

**Forma 2**

La forma dos es un vector que es dinámico y almacena datos estáticos. La memoria se reserva en el constructor y se libera en el destructor. Se sabe que el vector es dinamico y no sus elementos porque el tamaño del vector es una variable, es decir, el tamaño del vector sera el valor que se le pase al constructor.

```
// .h forma 2

#include <iostream>
#include <sstream> // para usar el stringstream
using namespace std;// para no escribir std::cout
class ContenedorDos{
    private:
        int *vector;// vector dinamico de enteros estaticos
        int cantidad;
        int tamano;
    public:
        ContenedorDos(int tama);//constructor
        ~ContenedorDos();//destructor
        void agregar(int);
        void cambiar(int posicion,int valor);
        string imprimir();
};
```

```
// .cpp forma 2

#include "ContenedorDos.h"
ContenedorDos::ContenedorDos(int tama){
    cantidad=0;
    tamano=tama;
    vector=new int[tamano];// reserva memoria dinamica para el vector de enteros
    for(int i=0;i<tamano;i++){
        vector[i]=0;
    }
}
ContenedorDos::~ContenedorDos(){
    cout<<"Se destruyo el objeto"<<endl;
    delete[] vector;// libera la memoria dinamica reservada para el vector
}
void ContenedorDos::agregar(int valor){
    if(cantidad<tamano){
        vector[cantidad++]=valor;
    }
}
void ContenedorDos::cambiar(int posicion,int valor){
    if(posicion<tamano){
        vector[posicion]=valor;
    }
}
string ContenedorDos::imprimir(){
    stringstream s;
    for(int i=0;i<cantidad;i++){
        s<<vector[i]<<endl;
    }
    return s.str();
}
// ejemplo con objetos Persona

// .h forma 2

#include <iostream>
#include <sstream> // para usar el stringstream
#include "Persona.h"
using namespace std;// para no escribir std::cout
class ContenedorDos{
    private:
        Persona *vector;// vector dinamico de objetos estaticos
        int cantidad;
        int tamano;
    public:
        ContenedorDos(int tama);//constructor
        ~ContenedorDos();//destructor
        void agregar(Persona& p);
        string imprimir();
}

// .cpp forma 2
```

```
#include "ContenedorDos.h"
ContenedorDos::ContenedorDos(int tama){
    cantidad=0;
    tamano=tama;
    vector=new Persona[tamano];
    for(int i=0;i<tamano;i++){
        vector[i]=Persona();
    }
}
ContenedorDos::~ContenedorDos(){
    cout<<"Se destruyo el objeto"<<endl;
    delete[] vector;
}
/* se pasa una referencia de la persona para evitar copias innecesarias, recuerde
que los pasos por valor copian el objeto */
void ContenedorDos::agregar(Persona& p){
    if(cantidad<tamano){
        vector[cantidad++]=p;
    }
}
string ContenedorDos::imprimir(){
    stringstream s;
    for(int i=0;i<cantidad;i++){
        s<<vector[i].toString()<<endl;// accede al metodo toString de la persona
en la posicion i por medio del operador . porque es un objeto estatico
    }
    return s.str();
}
```

**Forma 3**

La forma 3 es un vector estatico, pero que almacenara objetos dinamicos, es decir, se reserva memoria dinamica para cada objeto que se agregue al vector, por lo tanto, se debe liberar la memoria reservada para cada objeto en el destructor. El tamaño del vector es estatico, es decir, le damos un tama al declararlo `vector[tama]` y no se puede modificar.

```
// .h forma 3

#include <iostream>
#include <sstream>
#include "Persona.h"
using namespace std;

class ContenedorTres{
    private:
        Persona * vector[10]; // vector estatico de punteros a persona
        int cantidad;
        int tamano;
```

```
    public:
        ContenedorTres();
        ~ContenedorTres();
        void agregar(Persona * p);
        string imprimir();
};

// .cpp forma 3

#include "ContenedorTres.h"
ContenedorTres::ContenedorTres(){
    cantidad=0;
    tamano=10;
    for(int i=0;i<tamano;i++){
        vector[i]=nullptr;// inicializa el vector de punteros personas en nulo
    }
}
ContenedorTres::~ContenedorTres(){
    cout<<"Se destruyo el objeto"<<endl;
    for(int i=0;i<tamano;i++){
        if(vector[i]!=nullptr){// si el puntero no es nulo
            delete vector[i];// libera la memoria reservada para el objeto
        }
    }
}

void ContenedorTres::agregar(Persona * p){// se pasa un puntero de persona
    if(cantidad<tamano){
        vector[cantidad++]=p;//cambiamos el puntero null por el puntero de persona
ingresado
    }
}

string ContenedorTres::imprimir(){
    stringstream s;
    for(int i=0;i<cantidad;i++){
        s<<vector[i]->toString()<<endl;// -> para acceder a los metodos del objeto
al que apunta el puntero en la posicion i
    }
    return s.str();
}
```

**Forma 4**

La forma 4 es un vector dinamico de punteros a objetos dinamicos, es decir, se reserva memoria dinamica para cada objeto que se agregue al vector, por lo tanto, se debe liberar la memoria reservada para cada objeto en el destructor. El tamaño del vector se define en tiempo de ejecucion, es decir, se le pasa un tama al constructor y se reserva memoria dinamica para el vector de punteros a persona.

```
// .h forma 4
```

```
#include <iostream>
#include <sstream>
using namespace std;
#include "Persona.h"

class ContenedorCuatro{
    private:
        Persona ** vector;// vector dinamico de punteros a persona
        int cantidad;
        int tamano;
    public:
        ContenedorCuatro(int tama);
        ~ContenedorCuatro();
        void agregar(Persona *p);
        string imprimir();
}


// .cpp forma 4

#include "ContenedorCuatro.h"
ContenedorCuatro::ContenedorCuatro(int tama){
    cantidad=0;
    tamano=tama;
    vector=new Persona*[tamano];// reserva memoria dinamica para el vector de
punteros a persona
    for(int i=0;i<tamano;i++){
        vector[i]=nullptr;// inicializa el vector de punteros personas en nulo
    }
}
ContenedorCuatro::~ContenedorCuatro(){
    cout<<"Se destruyo el objeto"<<endl;
    for(int i=0;i<tamano;i++){
        if(vector[i]!=nullptr){// si el puntero no es nulo
            delete vector[i];// libera la memoria reservada para el objeto
        }
    }
    delete[] vector;// libera la memoria reservada para el vector de punteros a
persona
}

void ContenedorCuatro::agregar(Persona * p){// se pasa un puntero de persona
    if(cantidad<tamano){
        vector[cantidad++]=p;//cambiamos el puntero null por el puntero de persona
ingresado
    }
}

string ContenedorCuatro::imprimir(){
    stringstream s;
    for(int i=0;i<cantidad;i++){
        s<<vector[i]->toString()<<endl;// -> para acceder a los metodos del objeto
al que apunta el puntero en la posicion i
```

```
    }
    return s.str();
}
```

Creación de contenedores de las 4 formas en el main

```
#include <iostream>
#include "ContenedorUno.h"
#include "ContenedorDos.h"
#include "ContenedorTres.h"
#include "ContenedorCuatro.h"
#include "Persona.h"
using namespace std;

int main() {
    // Forma 1
    ContenedorUno c1;
    c1.agregar(1);
    c1.agregar(1);
    c1.agregar(1);
    cout << c1.imprimir() << endl;

    // Forma 2

    ContenedorDos c2(4);
    c2.agregar(2);
    c2.agregar(2);
    c2.agregar(2);
    cout << c2.imprimir() << endl;

    // Forma 3
    Persona* per = new Persona("Pedro",12);
    ContenedorTres c3 ;
    c3.agregar(per);
    cout << c3.imprimir() << endl;

    // Forma 4

    Persona* per1= new Persona("Vero", 22);
    ContenedorCuatro* c4 = new ContenedorCuatro(4);
    c4->agregar(per1);
    cout << c4->imprimir() << endl;
    /*Practiquemos referencias y punteros*/
    system("pause");
    system("cls");
    cout << "**********************Practiquemos referencias y
punteros*********************" << endl<<endl;
    cout << "**********************Punteros**********************" << endl;
    cout << "Mostrando per1 y su direccion" << endl;
    cout << per1->toString() << endl;
    cout << "Direccion de memoria que apunta per1 : " <<per1<< endl;
```

```
    cout << "Poniendo a per1 a ver la direccion de per : " << endl;
    per1 = per;
    cout << per1->toString() << endl;
    cout << "Nueva Direccion de memoria que apunta per1 : " << per1 << endl;
    cout << "Veamos si el vector que almacenaba a per1 se modifico" << endl;//no
porque solo las referencias afectan el valor
    cout << c4->imprimir() << endl << endl;
    cout << "********************** Referencias **********************" << endl;
    // Aquí vamos a trabajar con referencias

    Persona& refPersonaEnC4 = *(c4->getPersonaPosicion(0)); // desreferenciamos el
puntero devuelto
    refPersonaEnC4.setNombre("Maria"); // Modificamos la Persona a través de la
referencia

    cout << "Persona en c4 a traves de referencia refPersonaEnC4: " << endl;
    cout<<refPersonaEnC4.toString() << endl;
    cout << "Contenido de c4 despues de modificar la referencia interna: " <<
endl;
    cout << c4->imprimir() << endl;
}
```

## NOTAS

`#pragma once` es una directiva de preprocesador en C++ que se utiliza para evitar la inclusión múltiple de un mismo archivo de encabezado (header) en un programa.