Contratos Inteligentes (Solidity) 🔐

A continuación se presentan los contratos inteligentes clave de CycleWorks en **Solidity**, con comentarios explicativos en inglés para clarificar su propósito y funcionamiento. La arquitectura está diseñada de forma modular:

- **CycleToken (ERC-20):** Token fungible que representa las recompensas por reciclar (CYCLE).
- **CycleCertificate (ERC-721):** NFT que representa certificados de reciclaje o créditos de carbono (emitidos a empresas).
- **CycleWorksCore:** Contrato principal que registra eventos de reciclaje y coordina la emisión de recompensas.
- **Verifier:** Contrato para que una entidad autorizada (verificador) valide eventos de reciclaje en el contrato principal.
- **CycleBridge:** Contrato puente simplificado para transferir tokens CYCLE entre la subred y otra cadena (p.ej. la C-Chain de Avalanche).

Contrato CycleToken (ERC-20)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

/**
 * @title CycleToken
 * @dev ERC-20 token for CycleWorks rewards. Simplified implementation with mint/burn.
 *      This token represents the reward given to users for recycling.
 */
contract CycleToken {
    string public name = "CycleWorks Token";
    string public symbol = "CYCLE";
    uint8 public decimals = 18;
    uint256 public totalSupply;

    // Owner of the token (e.g., CycleWorks contract deployer or a multisig).
    address public owner;

    // Balances and allowances mappings for ERC20
    mapping(address => uint256) public balanceOf;
```

```
mapping(address => mapping(address => uint256)) public allowance;

// Events for transfer and approval (ERC20 standard)
event Transfer(address indexed from, address indexed to, uint256 value);
event Approval(address indexed owner, address indexed spender, uint256 value);

constructor() {
    owner = msg.sender;
}

/**
 * @dev Only owner modifier for functions that should be restricted.
 */
modifier onlyOwner() {
    require(msg.sender == owner, "Not authorized");
    _;
}

/**
 * @dev Transfer tokens from caller to another address.
 */
function transfer(address to, uint256 amount) external returns (bool) {
    require(balanceOf[msg.sender] >= amount, "Insufficient balance");
    balanceOf[msg.sender] -= amount;
    balanceOf[to] += amount;
    emit Transfer(msg.sender, to, amount);
    return true;
}

/**
 * @dev Approve an address to spend caller's tokens.
 */
function approve(address spender, uint256 amount) external returns (bool) {
    allowance[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}

/**
 * @dev Transfer tokens on behalf of another address (with prior approval).
 */
function transferFrom(address from, address to, uint256 amount) external returns (bool) {
    require(balanceOf[from] >= amount, "Insufficient balance");
    require(allowance[from][msg.sender] >= amount, "Allowance exceeded");
```

```
        // Deduct from sender's balance and allowance, then transfer to recipient
        balanceOf[from] -= amount;
        allowance[from][msg.sender] -= amount;
        balanceOf[to] += amount;
        emit Transfer(from, to, amount);
        return true;
    }

    /**
     * @dev Mint new tokens to a specified address. Only owner (CycleWorks contract) can call.
     * @param to The address that will receive the newly minted tokens.
     * @param amount The number of tokens to mint (in smallest unit).
     */
    function mint(address to, uint256 amount) external onlyOwner {
        totalSupply += amount;
        balanceOf[to] += amount;
        emit Transfer(address(0), to, amount);
    }

    /**
     * @dev Burn tokens from the caller's balance, reducing total supply.
     * @param amount The number of tokens to burn from msg.sender.
     */
    function burn(uint256 amount) external {
        require(balanceOf[msg.sender] >= amount, "Insufficient balance to burn");
        balanceOf[msg.sender] -= amount;
        totalSupply -= amount;
        emit Transfer(msg.sender, address(0), amount);
    }
}
```

Contrato CycleCertificate (ERC-721)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

/**
 * @title RecyclingCertificate (CycleWorks NFT)
 * @dev Simplified ERC-721 like token to represent recycling certificates (e.g., carbon or plastic credits).
 *      Each NFT can represent a batch of recycled material or avoided emissions.
 */
contract CycleCertificate {
    string public name = "CycleWorks Certificate";
```

```
    string public symbol = "CYCLE-CERT";

    // Counter for token IDs
    uint256 public nextTokenId;
    // Owner of token ID
    mapping(uint256 => address) public ownerOf;
    // Number of tokens held by address (for ERC721 balance)
    mapping(address => uint256) public balanceOf;

    // Events
    event Transfer(address indexed from, address indexed to, uint256 indexed tokenId);

    // The contract deployer (or CycleWorks main contract) will be the owner (minter)
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not authorized");
        _;
    }

    /**
     * @dev Mint a new certificate NFT to a specified address. Only owner can mint.
     * @param to The address that will receive the NFT.
     * @return tokenId The newly minted token's ID.
     */
    function mint(address to) external onlyOwner returns (uint256 tokenId) {
        tokenId = nextTokenId;
        nextTokenId += 1;
        ownerOf[tokenId] = to;
        balanceOf[to] += 1;
        emit Transfer(address(0), to, tokenId);
    }

    /**
     * @dev Transfer an NFT from one address to another. (Basic implementation, only owner
can directly transfer their NFT)
     * @param to Recipient address.
     * @param tokenId ID of the token to transfer.
     */
    function transferFrom(address from, address to, uint256 tokenId) external {
```

```
        require(ownerOf[tokenId] == from, "From address is not owner");
        require(msg.sender == from, "Only owner of token can transfer (no approval mechanic)");
        // Transfer ownership
        ownerOf[tokenId] = to;
        balanceOf[from] -= 1;
        balanceOf[to] += 1;
        emit Transfer(from, to, tokenId);
    }
}
```

Contrato CycleWorksCore (Principal)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

/**
 * @title CycleWorks (Main Contract)
 * @dev Core smart contract that records recycling events and coordinates rewards.
 *      It interacts with CycleToken (ERC20) to mint rewards and can work with CycleCertificate
(NFT) for credits.
 */
// Assume CycleToken and CycleCertificate contracts are deployed separately
import "CycleToken.sol";
import "CycleCertificate.sol";

contract CycleWorksCore {
    // References to token and certificate contracts
    CycleToken public token;
    CycleCertificate public certificate;
    address public owner;

    uint256 public rewardPerKg;

    // Struct to store a recycling event
    struct RecyclingEvent {
        address user;
        uint256 weight;
        string location;
        uint256 timestamp;
        bool verified;
    }

    // Array of all recycling events
    RecyclingEvent[] public events;
```

```
    // Mapping of event ID to a flag if it has been rewarded (for safety, to avoid double mint)
    // Not strictly necessary if we mint only upon verification once.
    mapping(uint256 => bool) public rewardClaimed;

    // Events
    event WasteDeposited(address indexed user, uint256 indexed eventId, uint256 weight, string
location);
    event WasteVerified(address indexed verifier, uint256 indexed eventId, uint256
rewardTokens);

    constructor(address tokenAddress, address certAddress) {
        owner = msg.sender;
        token = CycleToken(tokenAddress);
        certificate = CycleCertificate(certAddress);
        rewardPerKg = 10; // initial token reward rate per kg (can be adjusted)
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner");
        _;
    }

    /**
     * @dev Function for a user to report a recycling deposit. Records the event on-chain.
     * @param weight Amount of material (in kg or units) deposited.
     * @param location Identifier of the deposit location (e.g., IoT container ID or address).
     */
    function depositWaste(uint256 weight, string memory location) public {
        require(weight > 0, "Weight must be positive");
        // Create new event (unverified initially)
        uint256 eventId = events.length;
        events.push(RecyclingEvent({
            user: msg.sender,
            weight: weight,
            location: location,
            timestamp: block.timestamp,
            verified: false
        }));
        emit WasteDeposited(msg.sender, eventId, weight, location);
        // Note: No tokens minted yet - will mint upon verification to ensure validity
    }

    /**
```

```
     * @dev Verify a recycling event and issue reward tokens. Only authorized verifier (or owner)
can call.
     * @param eventId The index of the event to verify.
     */
    function markEventVerified(uint256 eventId) public {
        // In a real scenario, we'd require(msg.sender == verifierContract or has ROLE_VERIFIER)
        require(msg.sender == owner, "Not authorized to verify"); // simplified authorization
        require(eventId < events.length, "Invalid eventId");
        require(!events[eventId].verified, "Already verified");
        events[eventId].verified = true;
        uint256 reward = events[eventId].weight * rewardPerKg;
        // Mint reward tokens to the user via the CycleToken contract
        token.mint(events[eventId].user, reward);
        emit WasteVerified(msg.sender, eventId, reward);
    }

    /**
     * @dev Change the reward rate (tokens per kg). Only owner (governance) can adjust.
     */
    function setRewardPerKg(uint256 newRate) public onlyOwner {
        rewardPerKg = newRate;
    }

    /**
     * @dev Returns basic info of a recycling event by id.
     */
    function getEvent(uint256 eventId) public view returns (address user, uint256 weight, string
memory location, uint256 timestamp, bool verified) {
        require(eventId < events.length, "Invalid id");
        RecyclingEvent memory e = events[eventId];
        return (e.user, e.weight, e.location, e.timestamp, e.verified);
    }

    /**
     * @dev Returns total number of events recorded.
     */
    function getEventCount() public view returns (uint256) {
        return events.length;
    }
}
```

Contrato Verifier

```
// SPDX-License-Identifier: MIT
```

```
pragma solidity ^0.8.0;

/**
 * @title Verifier
 * @dev Contract that allows an authorized account to verify recycling events on CycleWorksCore.
 *      This can represent an off-chain oracle or trusted party (e.g., a collector or IoT validation service).
 */
contract Verifier {
    CycleWorksCore public cycleWorks;
    address public admin;

    constructor(address cycleWorksAddress) {
        cycleWorks = CycleWorksCore(cycleWorksAddress);
        admin = msg.sender;
    }

    modifier onlyAdmin() {
        require(msg.sender == admin, "Not authorized");
        _;
    }

    /**
     * @dev Verify a specific recycling event on the CycleWorksCore contract.
     * @param eventId The ID of the event to verify.
     */
    function verifyEvent(uint256 eventId) external onlyAdmin {
        // Calls the main contract to mark the event as verified and mint rewards.
        cycleWorks.markEventVerified(eventId);
    }
}
```

Contrato CycleBridge

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

/**
 * @title CycleBridge
 * @dev Simplified bridge contract to lock and unlock tokens for cross-chain transfers.
 *      In a real scenario, this would interface with a bridge oracle or system (e.g., Avalanche Bridge).
 */
```

```
contract CycleBridge {
    CycleToken public token;
    address public admin;

    event TokensLocked(address indexed user, uint256 amount, address indexed target);
    event TokensUnlocked(address indexed user, uint256 amount);

    constructor(address tokenAddress) {
        token = CycleToken(tokenAddress);
        admin = msg.sender;
    }

    modifier onlyAdmin() {
        require(msg.sender == admin, "Not authorized");
        _;
    }

    /**
     * @dev User locks tokens on this chain to be released on another chain.
     * @param amount Amount of tokens to lock (user must have approved this contract).
     * @param targetAddress The address on the target chain that will receive the tokens.
     */
    function lockTokens(uint256 amount, address targetAddress) external {
        // Transfer tokens from user to this contract
        require(token.transferFrom(msg.sender, address(this), amount), "Transfer failed");
        // Optionally burn them to reduce supply on this chain
        token.burn(amount);
        emit TokensLocked(msg.sender, amount, targetAddress);
        // In practice, an off-chain relayer would listen to this event and then mint tokens on target
chain
    }

    /**
     * @dev Admin (bridge operator) unlocks tokens on this chain when tokens have been locked
on the other chain.
     * @param to Address to receive the tokens on this chain.
     * @param amount Amount of tokens to unlock (mint).
     */
    function unlockTokens(address to, uint256 amount) external onlyAdmin {
        // Mint new tokens on this chain to the recipient
        token.mint(to, amount);
        emit TokensUnlocked(to, amount);
    }
}
```