PRACTICE 2

25/03/2025

—

Lydia Muñoz Gallardo

Aybaran Yurtseven
Quality Software

Index

Introduction

The practice is about vehicles, it is a simple practice in C++ that has functions such as displaying vehicle data on the screen, or adding the weight of a certain number of vehicles. It was done in the Object Oriented Programming subject in 2022 by the student, Lydia.

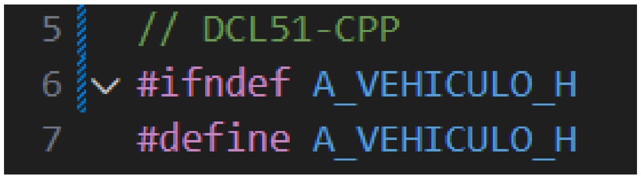
Standard 1

Link: DCL51-CPP. Do not declare or define a reserved identifier
Description:

The DCL51-CPP standard advises against declaring or defining reserved identifiers in C++ to avoid undefined behavior. Reserved identifiers include those with double underscores (__), those starting with an underscore followed by an uppercase letter, and various names from the standard library. It also prohibits redefining or undefining standard library names, keywords, and certain attribute tokens. These restrictions ensure compatibility, prevent conflicts, and maintain proper functionality within the language's standard implementation.

Code:

There is no "_" at the beginning and at the end of the declaration of the "vehiculo.h" file.



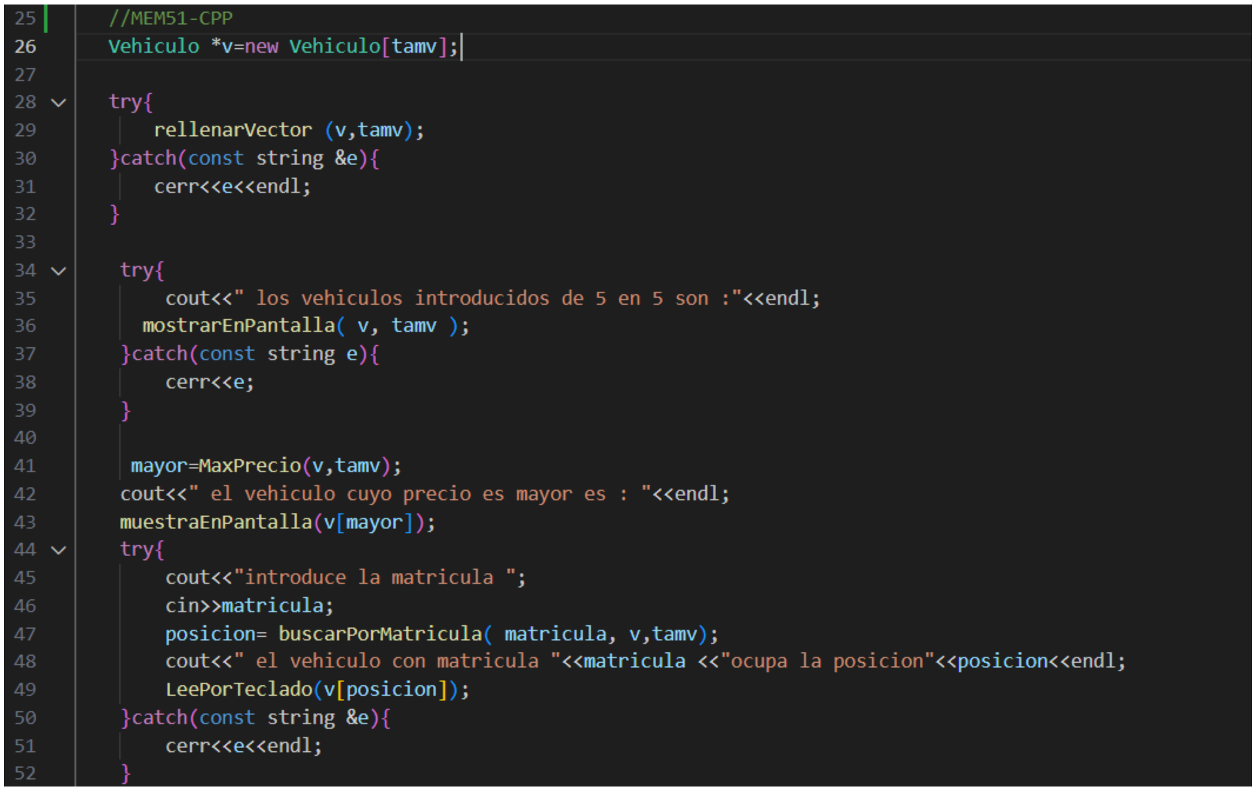
```
// DCL51-CPP
#ifndef A_VEHICULO_H
#define A_VEHICULO_H
```

Standard 2

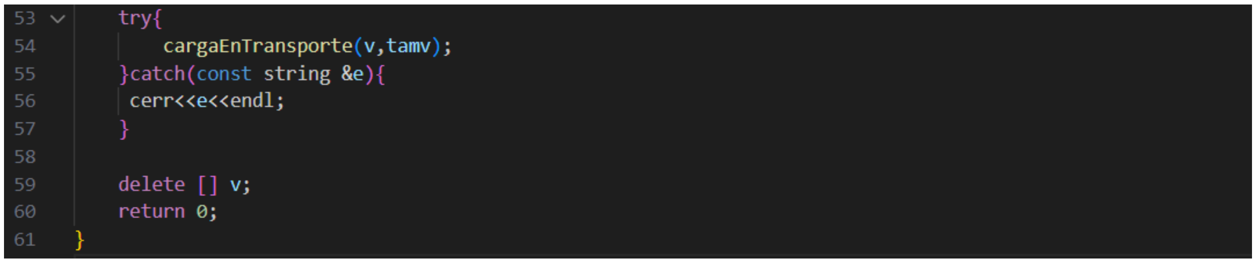
Link: MEM51-CPP. Properly deallocate dynamically allocated resources
Description:

The MEM51-CPP standard emphasizes the importance of correctly releasing dynamically allocated resources in C++. When memory is allocated using new, it must be freed with delete once it is no longer needed. Failing to do so can lead to memory leaks, which may degrade the program's performance and stability. Similarly, any other dynamically allocated resource should be properly managed to prevent issues related to improper usage.

Code:There is the "delete" after using "new" with the pointer “v" in the code.



```
    //MEM51-CPP
    Vehiculo *v=new Vehiculo[tamv];

    try{
        rellenarVector (v,tamv);
    }catch(const string &e){
        cerr<<e<<endl;
    }

     try{
        cout<<" los vehiculos introducidos de 5 en 5 son :"<<endl;
      mostrarEnPantalla( v, tamv );
     }catch(const string e){
        cerr<<e;
     }

      mayor=MaxPrecio(v,tamv);
     cout<<" el vehiculo cuyo precio es mayor es : "<<endl;
     muestraEnPantalla(v[mayor]);
     try{
        cout<<"introduce la matricula ";
        cin>>matricula;
        posicion= buscarPorMatricula( matricula, v,tamv);
        cout<<" el vehiculo con matricula "<<matricula <<"ocupa la posicion"<<posicion<<endl;
        LeePorTeclado(v[posicion]);
     }catch(const string &e){
        cerr<<e<<endl;
     }
```



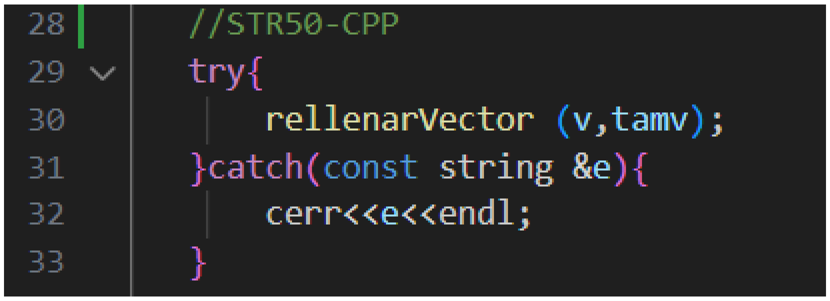
```
     try{
        cargaEnTransporte(v,tamv);
     }catch(const string &e){
      cerr<<e<<endl;
     }

     delete [] v;
     return 0;
}
```

Standard 3

Link: STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator

Description: Copying data into an undersized buffer causes a buffer overflow, often occurring with string manipulation. To prevent this, limit copies through truncation or ensure the destination has enough space. C-style strings require a null terminator, while std::basic_string in C++ does not.

Code:Fill the vector of elements and we do try catch so that if the buffer is full it does not introduce more elements



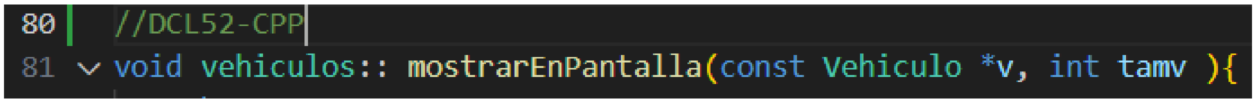
```
    //STR50-CPP
    try{
        rellenarVector (v,tamv);
    }catch(const string &e){
        cerr<<e<<endl;
    }
```

Standard 4

Link: DCL52-CPP. Never qualify a reference type with const or volatile

Description: In C++, reference types cannot be modified, effectively treating all references as const. The C++ Standard states that cv-qualified references are invalid unless introduced via typedef or decltype, in which case the qualifiers are ignored. Only non-reference types can be cv-qualified. This restriction can lead to accidental misuse when trying to apply const to a reference type.

Code: We have got the reference of the object passed by parameter accompanying the object instead of the const



```
//DCL52-CPP
void vehiculos:: mostrarEnPantalla(const Vehiculo *v, int tamv ){
```

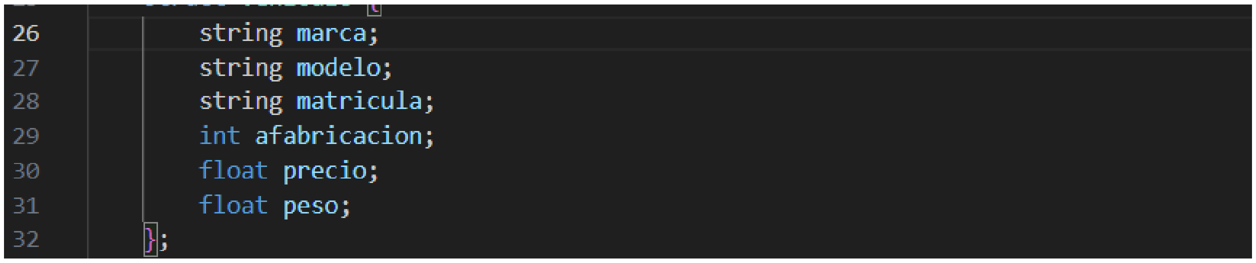
Standard 5

Link: OOP53-CPP. Write constructor member initializers in the canonical order

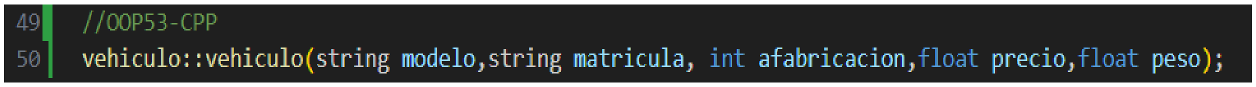
Description:

The OOP53-CPP standard emphasizes initializing class members in constructors following the order in which they are declared in the class definition. Maintaining this sequence improves code clarity, consistency, and readability, making it easier to maintain and understand. This practice is especially beneficial in classes with many members, as it ensures a predictable and structured approach to member initialization.

Code: As we can see we are initializing attributes in the default constructor in the same order we put them in the file.



```
        string marca;
        string modelo;
        string matricula;
        int afabricacion;
        float precio;
        float peso;
    };
```



```
//OOP53-CPP
vehiculo::vehiculo(string modelo,string matricula, int afabricacion,float precio,float peso);
```

Standard 6

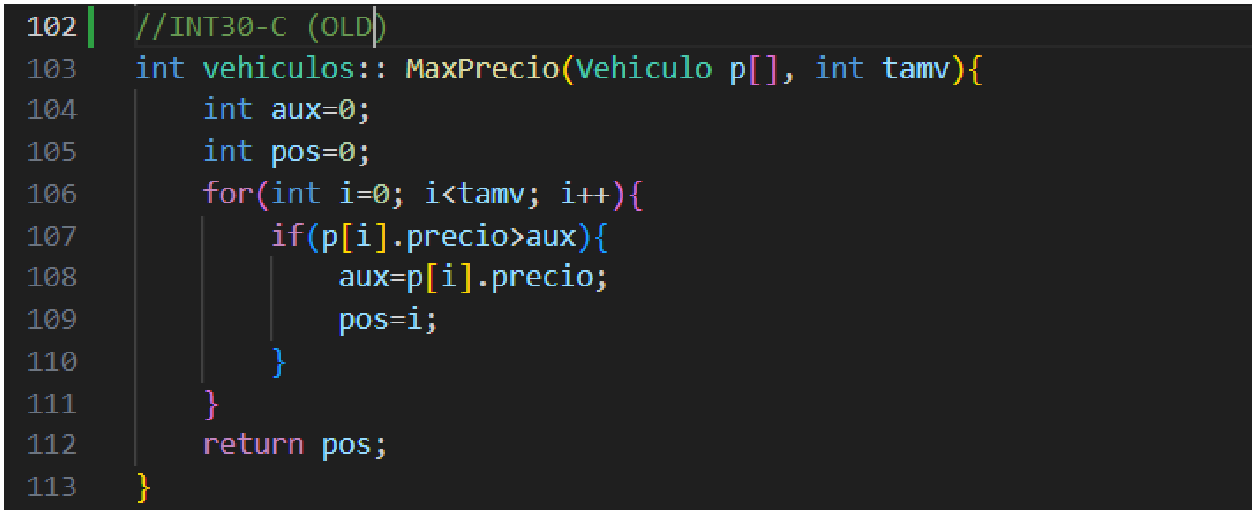
Link: INT30-C. Ensure that unsigned integer operations do not wrap

Description:

The INT30-C standard ensures that unsigned integer operations do not result in overflow or wrapping. Unsigned integers in C are computed modulo 2^N, which means that when the result of an operation exceeds the maximum value representable by the type, it wraps around. To prevent this, programmers should avoid situations where wrapping might occur, especially in pointer arithmetic, array indexing, or security-critical code. The standard also advises understanding integer conversion rules to ensure safe arithmetic operations.

Code:

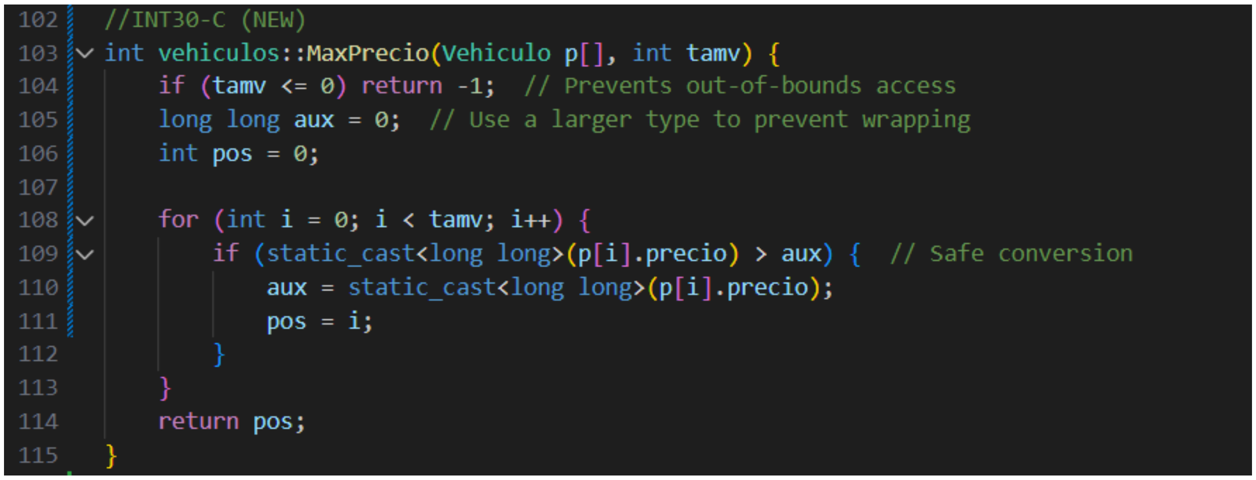
Function partially complies with the INT30-C standard, but there is a potential risk if p[i].precio is a large value and aux is an unsigned integer (unsigned int), which could cause wrapping.



```
//INT30-C (OLD)
int vehiculos:: MaxPrecio(Vehiculo p[], int tamv){
    int aux=0;
    int pos=0;
    for(int i=0; i<tamv; i++){
        if(p[i].precio>aux){
            aux=p[i].precio;
            pos=i;
        }
    }
    return pos;
}
```

To ensure that additions or assignments do not cause wrapping, we add:

1. Use a larger data type (long long or uint64_t if precio is unsigned int).
2. Ensure the value doesn't exceed the range of the data type.



```cpp
//INT30-C (NEW)
int vehiculos::MaxPrecio(Vehiculo p[], int tamv) {
    if (tamv <= 0) return -1;  // Prevents out-of-bounds access
    long long aux = 0;  // Use a larger type to prevent wrapping
    int pos = 0;

    for (int i = 0; i < tamv; i++) {
        if (static_cast<long long>(p[i].precio) > aux) {  // Safe conversion
            aux = static_cast<long long>(p[i].precio);
            pos = i;
        }
    }
    return pos;
}
```

Standard 7

Link: ERR56-CPP. Guarantee exception safety

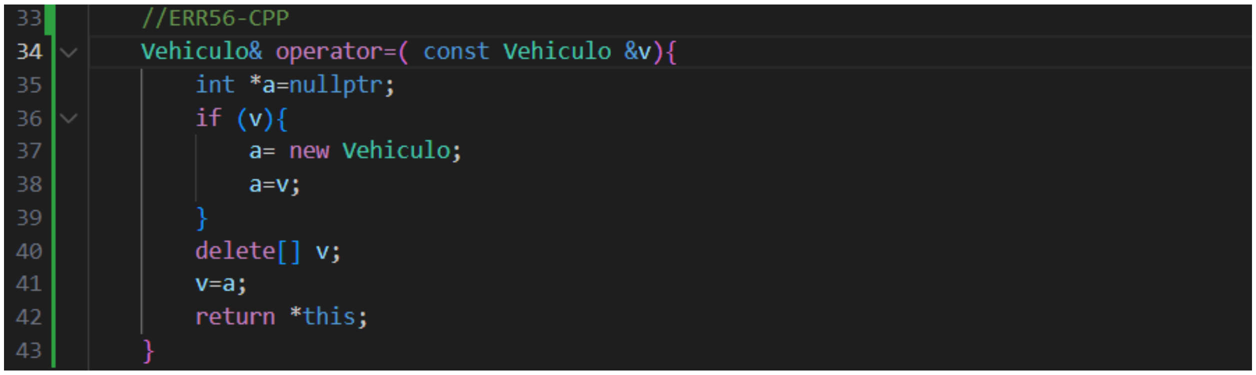
The text explains the importance of properly handling exceptions in C++ to ensure the stability and security of a program. It presents three levels of exception safety:

1. **Strong**: Guarantees that if an exception occurs, there are no changes to the program's state.

2. **Basic**: Prevents resource leaks and maintains program invariants.

3. **None**: Provides no exception safety guarantees, which can lead to an indeterminate state and critical errors.

It also includes an example of unsafe code and how to correct it to ensure strong exception safety. The corrected version first allocates memory for the copy before modifying the original object, preventing an exception from leaving the object in an inconsistent state.

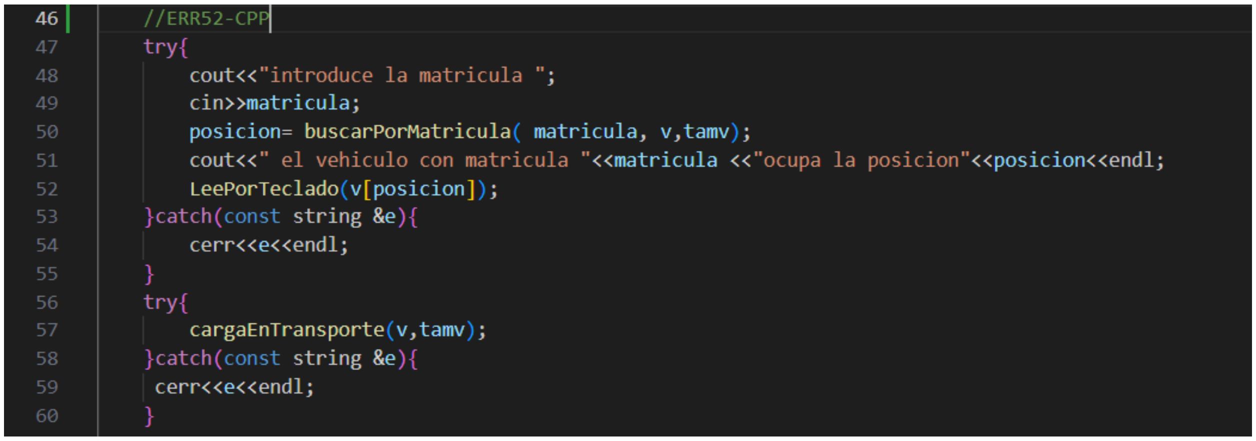


```cpp
    //ERR56-CPP
    Vehiculo& operator=( const Vehiculo &v){
        int *a=nullptr;
        if (v){
            a= new Vehiculo;
            a=v;
        }
        delete[] v;
        v=a;
        return *this;
    }
```

Standard 8

Link: ERR52-CPP. Do not use setjmp() or longjmp()

The C standard library functions **setjmp()** and **longjmp()** can be used to simulate throwing and catching exceptions. However, these functions bypass automatic resource management and can lead to undefined behavior, commonly including memory leaks and denial-of-service attacks.



```cpp
    //ERR52-CPP
    try{
        cout<<"introduce la matricula ";
        cin>>matricula;
        posicion= buscarPorMatricula( matricula, v,tamv);
        cout<<" el vehiculo con matricula "<<matricula <<"ocupa la posicion"<<posicion<<endl;
        LeePorTeclado(v[posicion]);
    }catch(const string &e){
        cerr<<e<<endl;
    }
    try{
        cargaEnTransporte(v,tamv);
    }catch(const string &e){
     cerr<<e<<endl;
    }
```

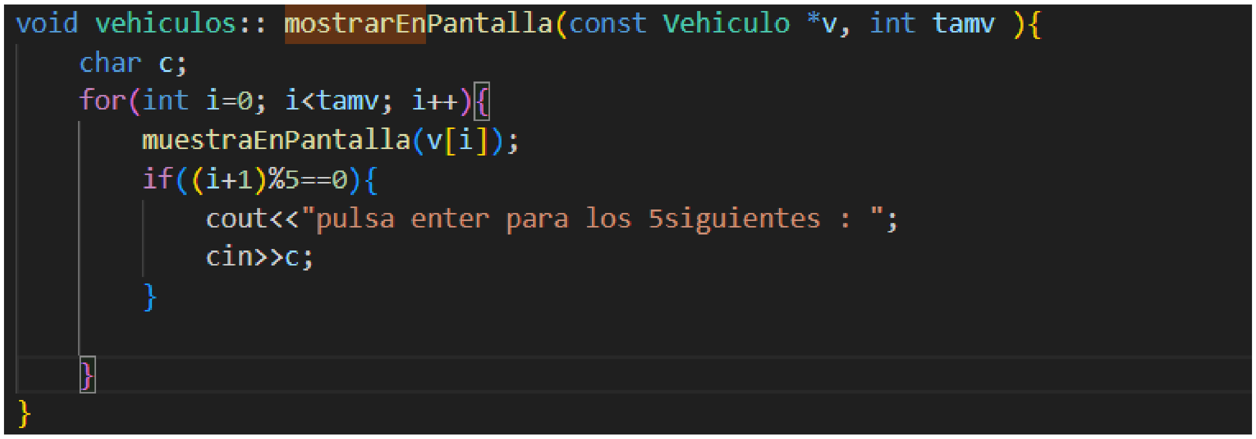
Standard 9

Link: DCL50-CPP. Do not define a C-style variadic function

Variadic functions accept a variable number of arguments and can be defined in C++ using function parameter packs or C-style ellipses. However, C-style variadic functions are unsafe because they lack type checking and argument validation, leading to undefined behavior and potential security risks.

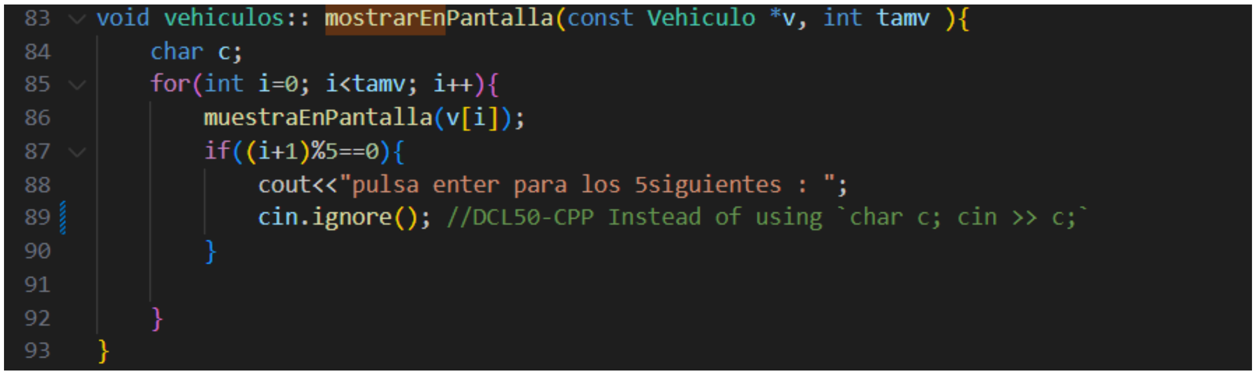
To avoid these issues, do not define C-style variadic functions. Instead, use function parameter packs or alternatives like function currying. For example, C++ replaces C's printf() with the type-safe overloaded std::cout::operator<<().

Before:



```
void vehiculos:: mostrarEnPantalla(const Vehiculo *v, int tamv ){
    char c;
    for(int i=0; i<tamv; i++){
        muestraEnPantalla(v[i]);
        if((i+1)%5==0){
            cout<<"pulsa enter para los 5siguientes : ";
            cin>>c;
        }

    }
}
```

Modified:



```
void vehiculos:: mostrarEnPantalla(const Vehiculo *v, int tamv ){
    char c;
    for(int i=0; i<tamv; i++){
        muestraEnPantalla(v[i]);
        if((i+1)%5==0){
            cout<<"pulsa enter para los 5siguientes : ";
            cin.ignore(); //DCL50-CPP Instead of using `char c; cin >> c;`
        }

    }
}
```

Standard 10

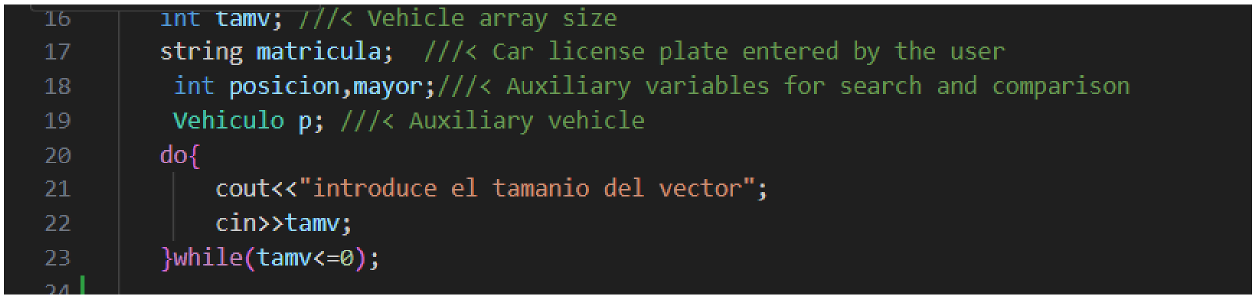
Link: ERR62-CPP. Detect errors when converting a string to a number

Parsing numbers from strings can lead to errors, such as invalid input, out-of-range values, or unexpected extra characters. These errors must be detected when using formatted input streams like std::istream or num_get<>.

Conversion errors can be checked via basic_ios::good(), bad(), and fail() or handled through exceptions. According to the C++ Standard, if conversion fails or the value is out of range, ios_base::failbit is set.

Always check the error state after conversion instead of assuming success. Avoid unsafe functions like std::atoi() and std::scanf() that lack validation.

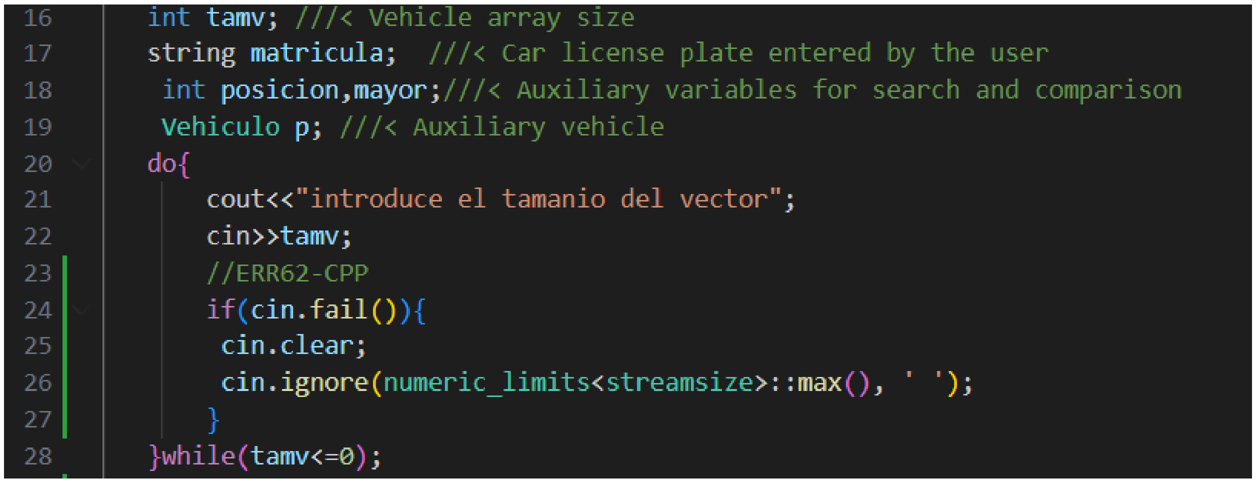
Before:



```
int tamv; ///< Vehicle array size
string matricula;  ///< Car license plate entered by the user
 int posicion,mayor;///< Auxiliary variables for search and comparison
 Vehiculo p; ///< Auxiliary vehicle
do{
    cout<<"introduce el tamanio del vector";
    cin>>tamv;
}while(tamv<=0);
```

In practice, we simply displayed integers according to this standard, ensuring proper error detection and handling during conversion that's why we changed it.

Modified:



```
    int tamv; ///< Vehicle array size
    string matricula;  ///< Car license plate entered by the user
     int posicion,mayor;///< Auxiliary variables for search and comparison
     Vehiculo p; ///< Auxiliary vehicle
    do{
        cout<<"introduce el tamanio del vector";
        cin>>tamv;
        //ERR62-CPP
        if(cin.fail()){
         cin.clear;
         cin.ignore(numeric_limits<streamsize>::max(), ' ');
        }
    }while(tamv<=0);
```