Article

A Forward-Collision Warning System for Electric Vehicles

1. Forward Collision Warning

The aim of FCW is to measure the collision risk and promptly warn the driver if it grows above a predefined threshold. The warning must be raised promptly enough so that the driver has sufficient time to react and avoid the collision or at least reduce its severity. Some FWC leverage radar technologies for sensing [19] result in high sensing costs.

However, since our main aim is to target lower-end vehicles, here a single monocular forward-facing camera is leveraged for sensing the obstacles at the front.

While relative distance and velocity are not directly measurable from vision information, in the following it will be shown how to derive the TTC information leveraging the scale change of objects in the image frame.

The FWC is in three main components: Detection, tracking and warning logics. Each of the steps will be described briefly in the following sections.

1.1. *Object Detection*

There are two main kinds of algorithms to tackle object detection via computer vision tasks: Feature-based and learning-based. Traditional techniques, such as Sobel edge detec- tor or Haar-like features, belong to the first kind and, despite their simpler implementation and lower computational cost, they suffer in terms of accuracy and generalization capabilities. Learning approaches instead require high-end hardware, due to the computational cost, but results in higher accuracy [20]. In this context, Convolutional Neural Networks (CNNs) are the state of the art for object detection tasks. Common approaches leverage two stage detectors, where the neural network first generates the region candidates and then classifies them [21]. Alternatively, a single stage detector can be exploited to directly predict the location and the class of an object in one step, thus resulting in faster inference time (e.g., see YOLO [22] and SSD [23]).

Taking into account the application strict real-time constraints, the single stage de- tectors have been investigated. In particular, due to YOLO faster computation time with respect to SSD, and comparable mean Average Precision (mAP) [22,24], the first has been chosen for our application design.

The third improved version YOLOv3 is a multiscale one stage object detector, which uses a Darknet-53 as backbone to extract features and localize possible objects in the input image. Despite its depth, it achieves state-of-the-art performance in classification and the highest measured in floating point operations per second. From the base feature extractor, several convolutional layers have been added, which predict bounding box, objectness and class. To achieve the best result, the K-means algorithm is run on the dataset before training, and the final K value chosen is the one with the best recall/complexity tradeoff. In order to address the multiscale problem, the network predicts boxes at three different scales, using a Feature Pyramid Network (FPN)-like architecture. FPN makes predictions at each layer (scale) and uses multiscale features from different layers combining low resolution (semantically strong) features with high resolution (semantically weak) features using top-down pathways and lateral connections. The network architecture is shown in Figure 1.

The described network is used to detect the vehicles, pedestrians, bicycles and motor- cycles. The output of the network is the so-called bounding box, for each detected obstacle, defined as:

$$b = [b_x\ b_y\ b_w\ b_h]. \tag{2}$$

where b_x and b_y are the pixel coordinates of the bounding box top-left corner, while b_w and b_h are the bounding box width and height, respectively.

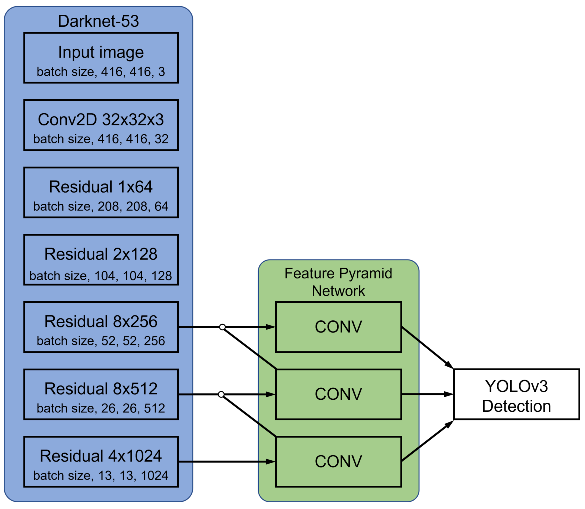


Figure 1. YOLOv3 Network Architecture [22].

1.2. *Multi-Target-Tracking*

The tracking component is essential in order to build a history of each detected object [25]. Taking into consideration that our scenario involves more than one detection for each frame, a Multi-Target-Tracker (MTT) algorithm is employed. An MTT must assign each new incoming detection, to the existing tracks before it can use the new measurements to update them. The assignment problem can be challenging due to the number of targets to track and the detection probability of the sensor which can lead to both false positives and false negatives.

The Global Nearest Neighbour (GNN) algorithm is chosen [26] with a bank of linear Kalman filter. The GNN is a single hypothesis tracker, whose goal is to assign the global nearest measurement to each track. Due to the fact that conflict situations can occur, a cost function must be defined and an optimization problem must be solved at each time-step. The Intersection-Over-Union (IOU) ones' complement , between detection and track pairs, is chosen as cost function:

$$J(i,j) = 1 - IOU(d_i, t_j), \tag{3}$$

where d_i is the i-th detection and t_j is the j-th track. The optimization problem is solved by using the Munkres algorithm [27,28], which ensures the global optimum convergence in polynomial time. Due to the small number of tracks and detection (typically below 20) the Munkres algorithm can be solved in real time on the chosen deployment hardware. Moreover, in order to reduce the complexity of the problem, a preceding gating step is applied during which a high cost in bid to unlikely assignments.

Once the association problem is solved, the measurements are used to update the bank of filters. A constant velocity linear Kalman filter [29] is used for each track by defining the state as $x = [b_x\ b_{x^v}\ \ b_y\ b_{y^v}\ \ b_w\ b_{w^v}\ \ b_h\ b_{h^v}\,]$ where b_x and b_y are the abscissa and the ordinate of the top left corner of the bounding box, while b_w and b_h are its width and height; finally, the subscript v denotes the respective velocities in the image frame. It is pointed out here that the state is defined in the frame coordinate, which simplifies the problem of measuring three-dimensional coordinates from a monocular camera. Finally, track management additionally involves creating track hypotheses from non-associated detections, and deleting old non-associated tracks.

More complicated solutions, e.g., multiple-hypothesis trackers combined with ex- tended Kalman filters, would require more information about the target position and relative angle with respect to the camera in the three-dimensional space, which is not natural information given by the chosen sensor architecture. Regardless, the proposed solution has been proven simple enough to be scheduled in real time, yet effective for the purpose of developing a forward collision warning system.

1.3. Collision Risk Evaluation

Once the tracks are updated at each time step, the collision risk for each one of them can be checked. It is shown now how the TTC in Equation (1) can be linked to the scale change of the bounding box between consecutive frames.

Note that the above formulation of the TTC is independent of the actual distance between the camera and the obstacles, which enables us to ignore camera calibration and assumptions on road properties, e.g., flatness, bank and slope angles. The accuracy of Equation (8) mainly depends on the choice of Δt, and on the accuracy of the detection and tracking system. In particular, by increasing Δt, the noise coming from the detection system can be attenuated, but a reduced number of measurements are obtained for each obstacle. Discussion on theoretical bounds on Δt are addressed in [30].

If the TTC in Equation (8), for any track, is below a chosen threshold, between 2 and 3 s, a collision might occur. The warning should be raised if and only if the examined track, with a TTC lower than the threshold, is in the ego vehicle's path. In order to check the latest, the state of its Kalman filter can be used considering that it contains information about the velocity of the obstacle. In particular, the position of the bounding box in the image frame can be predicted by using the following:

$$b_{xpr} = b_x + TTC \cdot b_{xv}, \tag{9}$$

where b_{xpr} is the predicted abscissa of the top left of the bounding box. With the same reasoning, the right corner can be predicted by using the width of the box. If the pre- dicted box is inside a precalibrated region of the image frame the warning is issued. The Equation (9) is based on constant velocity assumption which results in a good approxima- tion in the scenarios of interest, additionally considering that in these cases the TTC takes low values.

2. Testing and Deployment

The effectiveness of the approach is first investigated via model-in-the-loop leveraging a purposely designed virtual test platform and is then confirmed by experiments with an electric vehicle on the Kineton test track located in Naples, Italy.

Model-in-the-Loop Testing

The design for improved solutions of safety-related features is greatly eased by the usage of appropriate simulation platforms. Here, a co-simulation platform for Model-In- the-Loop (MIL) is proposed where autonomous vehicle can be safely tested, while moving within a potentially dangerous, realistic traffic scenario.

This co-simulation environment has been built leveraging the following two components:

- Matlab/Simulink has been used to develop the algorithm and lately auto-generating C code through the Embedded Coder toolbox.
- the open-source urban simulator CARLA (CAR Learning to Act) [31] has been used to design traffic scenarios and generate synthetic sensor measurements.

CARLA has a python-based core with embedded physics simulation which is capable of generating realistic measurements. In order to retrieve reliable sensor data, the simu- lation is carried out in synchronous fashion between the two environments; in particular, Simulink acts as a client by sending simulation commands to CARLA, acting as a server, which replies with the new generated measurements. In order to link the two environments a series of API have been implemented to create a communication between matlab-based Simulink and python-based CARLA cores. Figure 2 shows a screen capture of the proposed platform during a use case. In the case of the FCW feature the raw RGB frames are required, which are generated, at 30 fps, by a camera attached to a moving vehicle, mounted behind the windshield. The raw frames are the input to the algorithm introduced in Section 2.

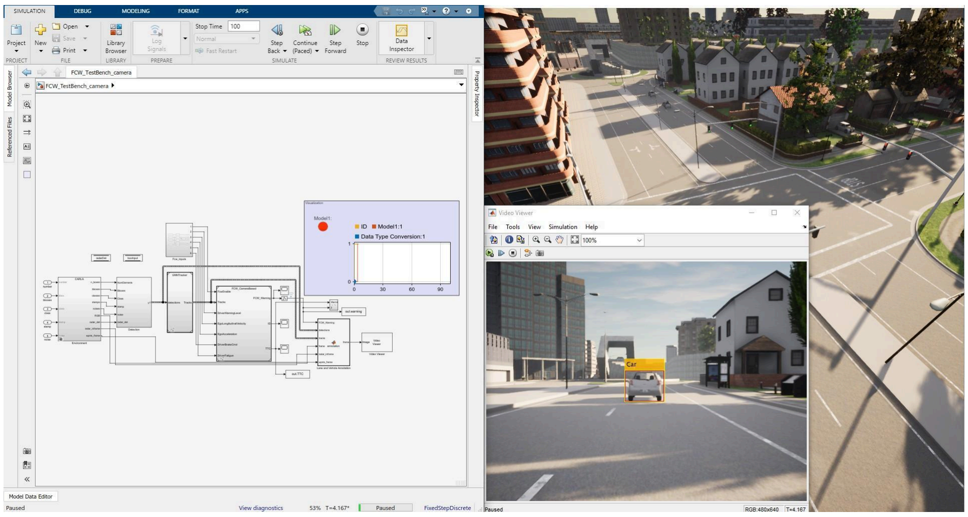


Figure 2. Screenshot of the co-simulation platform. On the right the CARLA server; on the left the Simulink implementation of the proposed algorithm supported by the communication APIs between the two components.

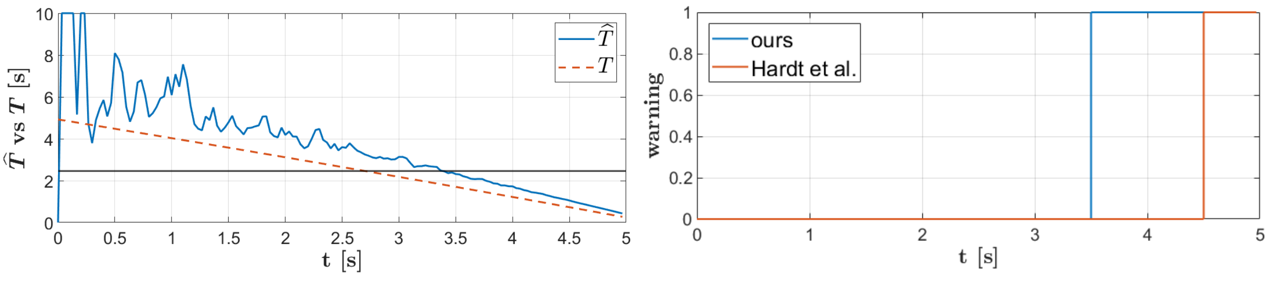
The driving scenarios designed in CARLA are those defined in the safety assist assessment protocol by EuroNCAP [32], namely:

- Car-to-Car Rear Stationary (CCRS): A collision in which a vehicle travels toward a stationary leading vehicle;
- Car-to-Car Rear Moving (CCRM): A collision in which a vehicle travels towards a slower vehicle moving at constant speed;
- Car-to-Car Rear Braking (CCRB): A collision in which a vehicle travels towards a braking vehicle.

All the scenario are repeated with varied vehicle velocities and lateral overlap ranging from −50% to +50%, as defined by the protocol procedures. To demonstrate proof of concept, two exemplary scenarios will be shown, namely a CCRS with the ego vehicle traveling at v = 50 km/h with a starting distance of around d = 67 m, and a CCRM with the ego vehicle traveling at v = 50 km/h, the leading vehicle traveling at v = 20 km/h with a starting distance of around d = 30 m. Moreover, a quantitative comparison is performed with respect to the latest literature results, see [18], in which the authors proposed a similar CNN-based solution. Nonetheless the risk estimation index takes into account a single frame bounding box, resulting in velocity-independent information.

Figure 3 shows the numerical results in the first driving scenario (CCRS). Namely, the estimated TTC and the real one are compared in Figure 3a in order to assess the accuracy of the algorithm. Due to the constant relative velocity between the two vehicles, the TTCs decrease linearly with time. While the oscillations are in the estimated TTC, no false positives are reported in all the CCRS scenarios. Furthermore, only a small constant percentage error bias can be appreciated, essentially due to the constant distance between the forward facing camera, mounted behind the windshield, and the front bumper of the car, where the actual TTC is evaluated. Note that this bias varies with the distance to the forward obstacle, so it could be compensated by its estimation, which is currently not embedded in our particular design; it is the object of our next research work. Figure 3b shows the warning activation which occurs as soon as the estimated TTC goes below the threshold, chosen as 2.1 s. The comparison in Figure 3b discloses that by taking into account multiple frames a more accurate warning can be issued. Indeed a warning issued around TTC ′ 1 s could not be enough to avoid a collision. Figure 4 shows the numerical results for the CCRM scenario and, as expected, the TTC trend is similar to the previous case. Note that the inaccuracies at high distances do not worsen the performances of the system, in fact no false positives are reported. Finally, Figure 4b shows the activation signal

for the FCW for the CCRM case, along with the comparison with [18]. The outcome is very similar to the CCRS case.



(**a**) (**b**)

Figure 3. Simulation results, through MIL testing, in the CCRS scenario. (**a**) Time-history of the estimated time-to-collision, T^ , and of the real one, T. The warning threshold is shown as a constant horizontal line; (**b**) time-history of the forward collision warning activation.

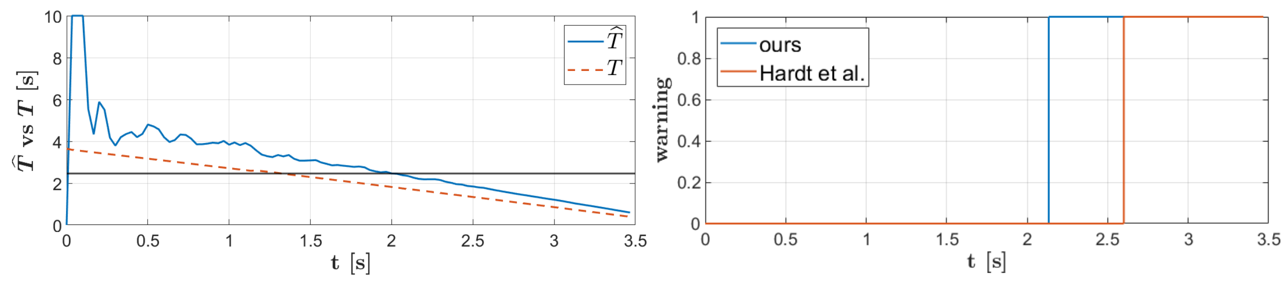


Figure 4. Simulation results, through MIL testing, in the CCRM scenario. (**a**) Time-history of the estimated Time-To-Collision, T^ , and of the real one, T. The warning threshold is shown as constant horizontal line (**b**) Time-history of the Forward Collision Warning activation.

3. Conclusions

In this paper a forward collision warning system is presented which leverages a deep convolutional neural network based on sensing data from an on-board forward camera. Moreover, it is shown that, by resorting to the scale change between consecutive frames, it is viable to rule out the error coming from camera calibration, making the system more robust with respect to camera mounting angle.

A general model-based virtual-testing platform has been designed to perform model- in-the-loop tests in a safe manner, exploitable even for more complex active safety systems. The numerical and experimental analysis show that the system is capable of promptly warning the driver if a collision is about to occur by replicating the EuroNCAP safety test assessments. Despite using a low-cost monocular camera for sensing, the overall architecture is accurate enough, at least in the TTC range of interest, i.e., below 3 (s). As prescribed by Euro NCAP protocol, we have extensively tested our design, not only experimentally, but also numerically by randomly varying the scenario conditions, thus verifying that the typical false/true warning rate requirements [36] are fulfilled. Results showed no false positives in all the appraised scenarios. Moreover, the employment of a state of the art deep CNN enhances the performances of the latest literature results. Future work will involve the investigation on estimation of the obstacle distance leveraging a mono or stereo camera, along with the implementation of more complex traffic and driving scenarios.

Forward Collision Warning (FCW) System Overview

Definition:
A **Forward Collision Warning (FCW)** system is an advanced driver assistance system (ADAS) designed to alert the driver about a potential collision with a vehicle or object ahead. It uses various sensors like cameras, radar, or lidar to detect objects in the vehicle's path and estimates the risk of collision.

Core Components of an FCW System

1. **Sensors**:
   - **Cameras**: Monocular or stereo cameras are used to detect vehicles, pedestrians, or obstacles in front of the car. They analyze visual data from the environment.
   - **Radar**: Uses radio waves to detect objects and their relative speed. Radar is often used in combination with cameras for better accuracy, especially in adverse weather conditions.
   - **Lidar**: Some advanced systems use lidar, which sends laser beams to create 3D maps of the surroundings.

2. **Data Processing**:
   - **Object Detection Algorithms**: The system processes sensor data to identify objects such as other vehicles, pedestrians, and obstacles. Common algorithms include **YOLO (You Only Look Once)** or **SSD (Single Shot Multibox Detector)**.
   - **Time-to-Collision (TTC)**: The FCW system calculates the TTC, which estimates how much time remains before a collision happens based on the distance and relative speed of the object in front.
   - **Risk Assessment**: If the TTC falls below a pre-set threshold (e.g., 2–3 seconds), the system determines that a collision is likely.

3. **Warning Mechanism**:
   - **Visual Alerts**: A visual display on the dashboard or heads-up display (HUD) alerts the driver.
   - **Audible Alerts**: A beeping or other sound alerts the driver of the impending collision.
   - **Vibration**: In some systems, the steering wheel or seat vibrates as an additional warning.

Working Mechanism

1. **Detection of Objects**:
   The FCW system continuously monitors the road ahead for potential hazards using its sensors. It detects moving vehicles, pedestrians, and stationary objects like traffic barriers or obstacles.
2. **Estimation of Risk**:
   The system calculates the relative speed and distance to the detected object. By using algorithms like TTC (Time-to-Collision), the system evaluates the likelihood of an accident.
3. **Issuing Warnings**:
   If the system determines that the collision risk is too high (e.g., TTC is too short), it triggers an alert to the driver. The alert typically includes a visual cue, sound, and possibly haptic feedback like steering wheel vibration.
4. **Driver Response**:
   The driver must react to the warning by either braking or steering to avoid the collision. Some FCW systems are paired with **Automatic Emergency Braking (AEB)** systems, which can apply the brakes if the driver does not respond.

Types of Forward Collision Warning Systems

1. **Basic FCW**:
   Only provides a warning to the driver about the potential for a collision, but does not take any action.
2. **FCW with Automatic Emergency Braking (AEB)**:
   In addition to providing a warning, the system can automatically apply the brakes to reduce the severity of the collision or avoid it entirely.
3. **Pedestrian Detection FCW**:
   Specifically designed to detect pedestrians in the vehicle's path and alert the driver or automatically apply brakes if a collision is imminent.
4. **Rear-End Collision Warning**:
   Alerts the driver about the risk of a rear-end collision based on the actions of vehicles in front.

Advantages of FCW Systems

- **Accident Prevention**: Helps prevent rear-end collisions by giving the driver time to react.
- **Enhanced Driver Safety**: Alerts drivers to potential hazards they might not notice, such as vehicles in blind spots or pedestrians crossing.
- **Reduction in Severity**: In systems integrated with AEB, the braking assistance can significantly reduce the collision's severity or avoid it altogether.
- **Helps with Distracted Driving**: Aids drivers who may not be paying full attention to the road, enhancing safety.

Challenges of FCW Systems

- **False Positives**: The system might mistakenly identify non-threatening objects as obstacles, leading to unnecessary alerts.
- **Environmental Limitations**: Performance can degrade in poor weather conditions like heavy rain, fog, or snow, as sensors may not detect objects effectively.
- **Limited Range**: Some FCW systems have a limited range, and objects far ahead may not be detected in time.
- **Driver Overreliance**: Drivers may become overly reliant on the system and may not always react promptly to warnings.

Recent Developments in FCW Technology

1. **Integration with Other ADAS**:
   FCW is often integrated with other ADAS systems, such as **Lane Departure Warning (LDW)**, **Adaptive Cruise Control (ACC)**, and **Blind Spot Detection (BSD)**, to provide comprehensive safety features.
2. **AI-Based Algorithms**:
   Advances in AI and machine learning allow FCW systems to become more intelligent, improving their ability to differentiate between hazards and non-hazards (e.g., detecting stationary objects vs. pedestrians).
3. **Improved Sensors**:
   The use of more advanced sensors such as 3D cameras, radar, and lidar improves the accuracy and reliability of the FCW system.

Future of FCW Systems

- **Autonomous Vehicles**: As autonomous driving technology advances, FCW systems will become an integral part of the vehicle's decision-making algorithms, seamlessly interacting with other systems to prevent accidents.
- **Enhanced Vehicle-to-Vehicle Communication (V2V)**: Future FCW systems may incorporate V2V communication to detect and predict the movement of nearby vehicles, further improving collision risk prediction.

MISRA 2023 C++ Rules

0. Language Independent Issues

`MISRA C++14 Rule A0-1-1`	A project shall not contain instances of non-volatile variables being given values that are not subsequently used *(Since R2020b)*
`MISRA C++14 Rule A0-1-2`	The value returned by a function having a non-void return type that is not an overloaded operator shall be used
`MISRA C++14 Rule A0-1-3`	Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used *(Since R2020b)*
`MISRA C++14 Rule A0-1-4`	There shall be no unused named parameters in non-virtual functions
`MISRA C++14 Rule A0-1-5`	There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it *(Since R2020a)*
`MISRA C++14 Rule A0-1-6`	There should be no unused type declarations
`MISRA C++14 Rule A0-4-2`	Type long double shall not be used
`MISRA C++14 Rule A0-4-4`	Range, domain and pole errors shall be checked when using math functions *(Since R2022a)*
`MISRA C++14 Rule M0-1-1`	A project shall not contain unreachable code
`MISRA C++14 Rule M0-1-2`	A project shall not contain infeasible paths
`MISRA C++14 Rule M0-1-3`	A project shall not contain unused variables

MISRA C++14 Rule M0-1-4	A project shall not contain non-volatile POD variables having only one use *(Since R2020b)*
MISRA C++14 Rule M0-1-8	All functions with void return type shall have external side effect(s) *(Since R2022a)*
MISRA C++14 Rule M0-1-9	There shall be no dead code
MISRA C++14 Rule M0-1-10	Every defined function should be called at least once
MISRA C++14 Rule M0-2-1	An object shall not be assigned to an overlapping object
MISRA C++14 Rule M0-3-2	If a function generates error information, then that error information shall be tested *(Since R2020b)*

1. General

MISRA C++14 Rule A1-1-1	**All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features**

2. Lexical Conventions

MISRA C++14 Rule A2-3-1	**Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code *(Since R2020a)***
MISRA C++14 Rule A2-5-1	**Trigraphs shall not be used**
MISRA C++14 Rule A2-5-2	**Digraphs shall not be used**
MISRA C++14 Rule A2-7-1	**The character \ shall not occur as a last character of a C++ comment *(Since R2020a)***
MISRA C++14 Rule A2-7-2	**Sections of code shall not be "commented out" *(Since R2020b)***
MISRA C++14 Rule A2-7-3	**All declarations of "user-defined" types, static and non-static data members, functions**

	and methods shall be preceded by documentation *(Since R2021a)*
MISRA C++14 Rule A2-8-1	A header file name should reflect the logical entity for which it provides declarations. *(Since R2021a)*
MISRA C++14 Rule A2-8-2	An implementation file name should reflect the logical entity for which it provides definitions. *(Since R2021a)*
MISRA C++14 Rule A2-10-1	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope
MISRA C++14 Rule A2-10-4	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace *(Since R2020b)*
MISRA C++14 Rule A2-10-5	An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused *(Since R2020b)*
MISRA C++14 Rule A2-10-6	A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope *(Since R2020a)*
MISRA C++14 Rule A2-11-1	Volatile keyword shall not be used
MISRA C++14 Rule A2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used
MISRA C++14 Rule A2-13-2	String literals with different encoding prefixes shall not be concatenated
MISRA C++14 Rule A2-13-3	Type wchar_t shall not be used
MISRA C++14 Rule A2-13-4	String literals shall not be assigned to non-constant pointers

`MISRA C++14 Rule A2-13-5`	**Hexadecimal constants should be uppercase**
`MISRA C++14 Rule A2-13-6`	**Universal character names shall be used only inside character or string literals *(Since R2020a)***
`MISRA C++14 Rule M2-7-1`	**The character sequence /* shall not be used within a C-style comment**
`MISRA C++14 Rule M2-10-1`	**Different identifiers shall be typographically unambiguous**
`MISRA C++14 Rule M2-13-2`	**Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used**
`MISRA C++14 Rule M2-13-3`	**A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type**
`MISRA C++14 Rule M2-13-4`	**Literal suffixes shall be upper case**

3. Basic Concept

`MISRA C++14 Rule A3-1-1`	**It shall be possible to include any header file in multiple translation units without violating the One Definition Rule**
`MISRA C++14 Rule A3-1-2`	**Header files, that are defined locally in the project, shall have a file name extension of one of: `.h`, `.hpp` or `.hxx`**
`MISRA C++14 Rule A3-1-3`	**Implementation files, that are defined locally in the project, should have a file name extension of ".cpp"**
`MISRA C++14 Rule A3-1-4`	**When an array with external linkage is declared, its size shall be stated explicitly**

`MISRA C++14 Rule A3-1-5`	**A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template *(Since R2020b)***
`MISRA C++14 Rule A3-1-6`	**Trivial accessor and mutator functions should be inlined *(Since R2020b)***
`MISRA C++14 Rule A3-3-1`	**Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file**
`MISRA C++14 Rule A3-3-2`	**Static and thread-local objects shall be constant-initialized *(Since R2020a)***
`MISRA C++14 Rule A3-8-1`	**An object shall not be accessed outside of its lifetime *(Since R2020b)***
`MISRA C++14 Rule A3-9-1`	**Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types**
`MISRA C++14 Rule M3-1-2`	**Functions shall not be declared at block scope**
`MISRA C++14 Rule M3-2-1`	**All declarations of an object or function shall have compatible types**
`MISRA C++14 Rule M3-2-2`	**The One Definition Rule shall not be violated**
`MISRA C++14 Rule M3-2-3`	**A type, object or function that is used in multiple translation units shall be declared in one and only one file**

`MISRA C++14 Rule M3-2-4`	An identifier with external linkage shall have exactly one definition
`MISRA C++14 Rule M3-3-2`	If a function has internal linkage then all re-declarations shall include the static storage class specifier
`MISRA C++14 Rule M3-4-1`	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility
`MISRA C++14 Rule M3-9-1`	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations
`MISRA C++14 Rule M3-9-3`	The underlying bit representations of floating-point values shall not be used

4. Standard Conversions

`MISRA C++14 Rule A4-5-1`	Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >= *(Since R2020a)*
`MISRA C++14 Rule A4-7-1`	An integer expression shall not lead to data loss *(Since R2021b)*
`MISRA C++14 Rule A4-10-1`	Only nullptr literal shall be used as the null-pointer-constraint *(Since R2020a)*
`MISRA C++14 Rule M4-5-1`	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and ! =, the unary & operator, and the conditional operator

MISRA C++14 Rule M4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and ! =, and the unary & operator
MISRA C++14 Rule M4-10-1	NULL shall not be used as an integer value
MISRA C++14 Rule M4-10-2	Literal zero (0) shall not be used as the null-pointer-constant

5. Expressions

MISRA C++14 Rule A5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits
MISRA C++14 Rule A5-0-2	The condition of an if-statement and the condition of an iteration statement shall have type bool
MISRA C++14 Rule A5-0-3	The declaration of objects shall contain no more than two levels of pointer indirection
MISRA C++14 Rule A5-0-4	Pointer arithmetic shall not be used with pointers to non-final classes
MISRA C++14 Rule A5-1-1	Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead
MISRA C++14 Rule A5-1-2	Variables shall not be implicitly captured in a lambda expression
MISRA C++14 Rule A5-1-3	Parameter list (possibly empty) shall be included in every lambda expression

MISRA C++14 Rule A5-1-4	A lambda expression object shall not outlive any of its reference-captured objects
MISRA C++14 Rule A5-1-6	Return type of a non-void return type lambda expression should be explicitly specified *(Since R2020b)*
MISRA C++14 Rule A5-1-7	A lambda shall not be an operand to decltype or typeid
MISRA C++14 Rule A5-1-8	Lambda expressions should not be defined inside another lambda expression *(Since R2020b)*
MISRA C++14 Rule A5-1-9	Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression *(Since R2020b)*
MISRA C++14 Rule A5-2-1	dynamic_cast should not be used *(Since R2020b)*
MISRA C++14 Rule A5-2-2	Traditional C-style casts shall not be used
MISRA C++14 Rule A5-2-3	A cast shall not remove any const or volatile qualification from the type of a pointer or reference
MISRA C++14 Rule A5-2-4	reinterpret_cast shall not be used
MISRA C++14 Rule A5-2-5	An array or container shall not be accessed beyond its range *(Since R2022a)*

`MISRA C++14 Rule A5-2-6`	The operands of a logical `&&` or `||` shall be parenthesized if the operands contain binary operators
`MISRA C++14 Rule A5-3-1`	Evaluation of the operand to the typeid operator shall not contain side effects *(Since R2020b)*
`MISRA C++14 Rule A5-3-2`	Null pointers shall not be dereferenced *(Since R2020b)*
`MISRA C++14 Rule A5-3-3`	Pointers to incomplete class types shall not be deleted
`MISRA C++14 Rule A5-5-1`	A pointer to member shall not access non-existent class members *(Since R2022a)*
`MISRA C++14 Rule A5-6-1`	The right hand operand of the integer division or remainder operators shall not be equal to zero
`MISRA C++14 Rule A5-10-1`	A pointer to member virtual function shall only be tested for equality with null-pointer-constant *(Since R2020b)*
`MISRA C++14 Rule A5-16-1`	The ternary conditional operator shall not be used as a sub-expression
`MISRA C++14 Rule M5-0-2`	Limited dependence should be placed on C++ operator precedence rules in expressions
`MISRA C++14 Rule M5-0-3`	A cvalue expression shall not be implicitly converted to a different underlying type

`MISRA C++14 Rule M5-0-4`	**An implicit integral conversion shall not change the signedness of the underlying type**
`MISRA C++14 Rule M5-0-5`	**There shall be no implicit floating-integral conversions**
`MISRA C++14 Rule M5-0-6`	**An implicit integral or floating-point conversion shall not reduce the size of the underlying type**
`MISRA C++14 Rule M5-0-7`	**There shall be no explicit floating-integral conversions of a cvalue expression**
`MISRA C++14 Rule M5-0-8`	**An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression**
`MISRA C++14 Rule M5-0-9`	**An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression**
`MISRA C++14 Rule M5-0-10`	**If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand**
`MISRA C++14 Rule M5-0-11`	**The plain char type shall only be used for the storage and use of character values**
`MISRA C++14 Rule M5-0-12`	**Signed char and unsigned char type shall only be used for the storage and use of numeric values**
`MISRA C++14 Rule M5-0-14`	**The first operand of a conditional-operator shall have type bool**

`MISRA C++14 Rule M5-0-15`	Array indexing shall be the only form of pointer arithmetic
`MISRA C++14 Rule M5-0-16`	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array *(Since R2021a)*
`MISRA C++14 Rule M5-0-17`	Subtraction between pointers shall only be applied to pointers that address elements of the same array
`MISRA C++14 Rule M5-0-18`	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array
`MISRA C++14 Rule M5-0-20`	Non-constant operands to a binary bitwise operator shall have the same underlying type
`MISRA C++14 Rule M5-0-21`	Bitwise operators shall only be applied to operands of unsigned underlying type
`MISRA C++14 Rule M5-2-2`	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of `dynamic_cast`
`MISRA C++14 Rule M5-2-3`	Casts from a base class to a derived class should not be performed on polymorphic types
`MISRA C++14 Rule M5-2-6`	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type
`MISRA C++14 Rule M5-2-8`	An object with integer type or pointer to void type shall not be converted to an object with pointer type

MISRA C++14 Rule M5-2-9	A cast shall not convert a pointer type to an integral type
MISRA C++14 Rule M5-2-10	The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression
MISRA C++14 Rule M5-2-11	The comma operator, && operator and the || operator shall not be overloaded
MISRA C++14 Rule M5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer
MISRA C++14 Rule M5-3-1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool
MISRA C++14 Rule M5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
MISRA C++14 Rule M5-3-3	The unary & operator shall not be overloaded
MISRA C++14 Rule M5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects
MISRA C++14 Rule M5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand
MISRA C++14 Rule M5-14-1	The right hand operand of a logical &&, || operators shall not contain side effects

`MISRA C++14 Rule M5-18-1`	**The comma operator shall not be used**
`MISRA C++14 Rule M5-19-1`	**Evaluation of constant unsigned integer expressions shall not lead to wrap-around**

6. Statements

`MISRA C++14 Rule A6-2-1`	**Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects *(Since R2020b)***
`MISRA C++14 Rule A6-2-2`	**Expression statements shall not be explicit calls to constructors of temporary objects only**
`MISRA C++14 Rule A6-4-1`	**A `switch` statement shall have at least two case-clauses, distinct from the default label**
`MISRA C++14 Rule A6-5-1`	**A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used *(Since R2022a)***
`MISRA C++14 Rule A6-5-2`	**A `for` loop shall contain a single loop-counter which shall not have floating-point type**
`MISRA C++14 Rule A6-5-3`	**Do statements should not be used *(Since R2020b)***
`MISRA C++14 Rule A6-5-4`	**For-init-statement and expression should not perform actions other than loop-counter initialization and modification**
`MISRA C++14 Rule A6-6-1`	**The `goto` statement shall not be used**

`MISRA C++14 Rule M6-2-1`	**Assignment operators shall not be used in sub-expressions**
`MISRA C++14 Rule M6-2-2`	**Floating-point expressions shall not be directly or indirectly tested for equality or inequality**
`MISRA C++14 Rule M6-2-3`	**Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character**
`MISRA C++14 Rule M6-3-1`	**The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement**
`MISRA C++14 Rule M6-4-1`	**An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement**
`MISRA C++14 Rule M6-4-2`	**All if ... else if constructs shall be terminated with an else clause**
`MISRA C++14 Rule M6-4-3`	**A switch statement shall be a well-formed switch statement**
`MISRA C++14 Rule M6-4-4`	**A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement**
`MISRA C++14 Rule M6-4-5`	**An unconditional throw or break statement shall terminate every non-empty switch-clause**
`MISRA C++14 Rule M6-4-6`	**The final clause of a** `switch` **statement shall be the default-clause**

`MISRA C++14 Rule M6-4-7`	**The condition of a switch statement shall not have bool type**
`MISRA C++14 Rule M6-5-2`	**If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=**
`MISRA C++14 Rule M6-5-3`	**The loop-counter shall not be modified within condition or statement**
`MISRA C++14 Rule M6-5-4`	**The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop**
`MISRA C++14 Rule M6-5-5`	**A loop-control-variable other than the loop-counter shall not be modified within condition or expression**
`MISRA C++14 Rule M6-5-6`	**A loop-control-variable other than the loop-counter which is modified in statement shall have type bool**
`MISRA C++14 Rule M6-6-1`	**Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement**
`MISRA C++14 Rule M6-6-2`	**The goto statement shall jump to a label declared later in the same function body**
`MISRA C++14 Rule M6-6-3`	**The `continue` statement shall only be used within a well-formed `for` loop**

7. Declaration

`MISRA C++14 Rule A7-1-1`	**Constexpr or const specifiers shall be used for immutable data declaration *(Since***

	R2020b)
`MISRA C++14 Rule A7-1-2`	The constexpr specifier shall be used for values that can be determined at compile time *(Since R2020b)*
`MISRA C++14 Rule A7-1-3`	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name
`MISRA C++14 Rule A7-1-4`	The register keyword shall not be used
`MISRA C++14 Rule A7-1-5`	The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax *(Since R2020b)*
`MISRA C++14 Rule A7-1-6`	The typedef specifier shall not be used
`MISRA C++14 Rule A7-1-7`	Each expression statement and identifier declaration shall be placed on a separate line
`MISRA C++14 Rule A7-1-8`	A non-type specifier shall be placed before a type specifier in a declaration *(Since R2020a)*
`MISRA C++14 Rule A7-1-9`	A class, structure, or enumeration shall not be declared in the definition of its type
`MISRA C++14 Rule A7-2-1`	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration *(Since R2022a)*

`MISRA C++14 Rule A7-2-2`	**Enumeration underlying type shall be explicitly defined**
`MISRA C++14 Rule A7-2-3`	**Enumerations shall be declared as scoped enum classes**
`MISRA C++14 Rule A7-2-4`	**In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized**
`MISRA C++14 Rule A7-3-1`	**All overloads of a function shall be visible from where it is called**
`MISRA C++14 Rule A7-5-1`	**A function shall not return a reference or a pointer to a parameter that is passed by reference to const**
`MISRA C++14 Rule A7-5-2`	**Functions shall not call themselves, either directly or indirectly**
`MISRA C++14 Rule A7-6-1`	**Functions declared with the [[noreturn]] attribute shall not return *(Since R2020b)***
`MISRA C++14 Rule M7-1-2`	**A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified**
`MISRA C++14 Rule M7-3-1`	**The global namespace shall only contain main, namespace declarations and extern "C" declarations**
`MISRA C++14 Rule M7-3-2`	**The identifier main shall not be used for a function other than the global function main**
`MISRA C++14 Rule M7-3-3`	**There shall be no unnamed namespaces in header files**

`MISRA C++14 Rule M7-3-4`	Using-directives shall not be used
`MISRA C++14 Rule M7-3-6`	Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files
`MISRA C++14 Rule A7-4-1`	The asm declaration shall not be used *(Since R2020a)*
`MISRA C++14 Rule M7-4-2`	Assembler instructions shall only be introduced using the asm declaration
`MISRA C++14 Rule M7-4-3`	Assembly language shall be encapsulated and isolated
`MISRA C++14 Rule M7-5-1`	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function
`MISRA C++14 Rule M7-5-2`	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist *(Since R2020b)*

8. Declarators

`MISRA C++14 Rule A8-2-1`	When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters *(Since R2020a)*
`MISRA C++14 Rule A8-4-1`	Functions shall not be defined using the ellipsis notation
`MISRA C++14 Rule A8-4-2`	All exit paths from a function with non-void return type shall have an explicit return statement with an expression

`MISRA C++14 Rule A8-4-3`	**Common ways of passing parameters should be used.** *(Since R2021b)*
`MISRA C++14 Rule A8-4-4`	**Multiple output values from a function should be returned as a struct or tuple** *(Since R2020b)*
`MISRA C++14 Rule A8-4-5`	**"consume" parameters declared as X && shall always be moved from** *(Since R2021a)*
`MISRA C++14 Rule A8-4-6`	**"forward" parameters declared as T && shall always be forwarded** *(Since R2021a)*
`MISRA C++14 Rule A8-4-7`	**"in" parameters for "cheap to copy" types shall be passed by value**
`MISRA C++14 Rule A8-4-8`	**Output parameters shall not be used** *(Since R2021a)*
`MISRA C++14 Rule A8-4-9`	**"in-out" parameters declared as T & shall be modified** *(Since R2021a)*
`MISRA C++14 Rule A8-4-10`	**A parameter shall be passed by reference if it can't be NULL** *(Since R2021a)*
`MISRA C++14 Rule A8-4-11`	**A smart pointer shall only be used as a parameter type if it expresses lifetime semantics** *(Since R2022b)*
`MISRA C++14 Rule A8-4-12`	**A std::unique_ptr shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.** *(Since R2022b)*

`MISRA C++14 Rule A8-4-13`	A std::shared_ptr shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a `const` lvalue reference to express that the function retains a reference count. *(Since R2022b)*
`MISRA C++14 Rule A8-4-14`	Interfaces shall be precisely and strongly typed
`MISRA C++14 Rule A8-5-0`	All memory shall be initialized before it is read
`MISRA C++14 Rule A8-5-1`	In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition
`MISRA C++14 Rule A8-5-2`	Braced-initialization {}, without equals sign, shall be used for variable initialization
`MISRA C++14 Rule A8-5-4`	If a class has a user-declared constructor that takes a parameter of type std::initializer_list, then it shall be the only constructor apart from special member function constructors *(Since R2021a)*
`MISRA C++14 Rule A8-5-3`	A variable of type auto shall not be initialized using {} or ={} braced-initialization *(Since R2020a)*
`MISRA C++14 Rule M8-0-1`	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively
`MISRA C++14 Rule M8-3-1`	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments

MISRA C++14 Rule M8-4-2	**The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration**
MISRA C++14 Rule M8-4-4	**A function identifier shall either be used to call the function or it shall be preceded by &**
MISRA C++14 Rule M8-5-2	**Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures**

9. Classes

MISRA C++14 Rule A9-3-1	**Member functions shall not return non-constant "raw" pointers or references to private or protected data owned by the class**
MISRA C++14 Rule A9-5-1	**Unions shall not be used**
MISRA C++14 Rule A9-6-1	**Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes**
MISRA C++14 Rule M9-3-1	`Const` **member functions shall not return non-**`const` **pointers or references to class-data**
MISRA C++14 Rule M9-3-3	**If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const**
MISRA C++14 Rule M9-6-4	**Named bit-fields with signed integer type shall have a length of more than one bit *(Since R2020b)***

10. Derived Classes

`MISRA C++14 Rule A10-1-1`	Class shall not be derived from more than one base class which is not an interface class *(Since R2020a)*
`MISRA C++14 Rule A10-2-1`	Non-virtual public or protected member functions shall not be redefined in derived classes
`MISRA C++14 Rule A10-3-1`	Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final *(Since R2020a)*
`MISRA C++14 Rule A10-3-2`	Each overriding virtual function shall be declared with the override or final specifier *(Since R2020a)*
`MISRA C++14 Rule A10-3-3`	Virtual functions shall not be introduced in a final class *(Since R2020a)*
`MISRA C++14 Rule A10-3-5`	A user-defined assignment operator shall not be virtual *(Since R2020a)*
`MISRA C++14 Rule A10-4-1`	Hierarchies should be based on interface classes *(Since R2021b)*
`MISRA C++14 Rule M10-1-1`	Classes should not be derived from virtual bases
`MISRA C++14 Rule M10-1-2`	A base class shall only be declared virtual if it is used in a diamond hierarchy

MISRA C++14 Rule M10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy
MISRA C++14 Rule M10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique
MISRA C++14 Rule M10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual

11. Member Access Control

MISRA C++14 Rule A11-0-1	A non-POD type should be defined as class *(Since R2020b)*
MISRA C++14 Rule A11-0-2	A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class *(Since R2020a)*
MISRA C++14 Rule A11-3-1	Friend declarations shall not be used
MISRA C++14 Rule M11-0-1	Member data in non-POD class types shall be private

12. Special Member Functions

MISRA C++14 Rule A12-0-1	If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well *(Since R2020a)*
MISRA C++14 Rule A12-0-2	Bitwise operations and operations that assume data representation in memory shall not be performed on objects *(Since R2020b)*

`MISRA C++14 Rule A12-1-1`	**Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members**
`MISRA C++14 Rule A12-1-2`	**Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type *(Since R2020b)***
`MISRA C++14 Rule A12-1-3`	**If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead *(Since R2021b)***
`MISRA C++14 Rule A12-1-4`	**All constructors that are callable with a single argument of fundamental type shall be declared explicit**
`MISRA C++14 Rule A12-1-5`	**Common class initialization for non-constant members shall be done by a delegating constructor *(Since R2021a)***
`MISRA C++14 Rule A12-1-6`	**Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors *(Since R2020b)***
`MISRA C++14 Rule A12-4-1`	**Destructor of a base class shall be public virtual, public override or protected non-virtual *(Since R2020a)***
`MISRA C++14 Rule A12-4-2`	**If a public destructor of a class is non-virtual, then the class should be declared final *(Since R2020b)***
`MISRA C++14 Rule A12-6-1`	**All class data members that are initialized by the constructor shall be initialized using member initializers**

`MISRA C++14 Rule A12-7-1`	**If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined *(Since R2021b)***
`MISRA C++14 Rule A12-8-1`	**Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects *(Since R2021a)***
`MISRA C++14 Rule A12-8-2`	**User-defined copy and move assignment operators should use user-defined no-throw swap function *(Since R2021a)***
`MISRA C++14 Rule A12-8-3`	**Moved-from object shall not be read-accessed *(Since R2021a)***
`MISRA C++14 Rule A12-8-4`	**Move constructor shall not initialize its class members and base classes using copy semantics *(Since R2020b)***
`MISRA C++14 Rule A12-8-5`	**A copy assignment and a move assignment operators shall handle self-assignment**
`MISRA C++14 Rule A12-8-6`	**Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class *(Since R2020a)***
`MISRA C++14 Rule A12-8-7`	**Assignment operators should be declared with the ref-qualifier & *(Since R2020b)***
`MISRA C++14 Rule M12-1-1`	**An object's dynamic type shall not be used from the body of its constructor or destructor**

13. Overloading

MISRA C++14 Rule A13-1-2	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters *(Since R2020a)*
MISRA C++14 Rule A13-1-3	User defined literals operators shall only perform conversion of passed parameters *(Since R2021a)*
MISRA C++14 Rule A13-2-1	An assignment operator shall return a reference to "this"
MISRA C++14 Rule A13-2-2	A binary arithmetic operator and a bitwise operator shall return a "prvalue" *(Since R2021a)*
MISRA C++14 Rule A13-2-3	A relational operator shall return a boolean value *(Since R2020a)*
MISRA C++14 Rule A13-3-1	A function that contains "forwarding reference" as its argument shall not be overloaded *(Since R2021a)*
MISRA C++14 Rule A13-5-1	If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented *(Since R2020a)*
MISRA C++14 Rule A13-5-2	All user-defined conversion operators shall be defined explicit *(Since R2020a)*
MISRA C++14 Rule A13-5-3	User-defined conversion operators should not be used *(Since R2021a)*
MISRA C++14 Rule A13-5-4	If two opposite operators are defined, one shall be defined in terms of the other *(Since R2022a)*

`MISRA C++14 Rule A13-5-5`	**Comparison operators shall be non-member functions with identical parameter types and noexcept *(Since R2020b)***
`MISRA C++14 Rule A13-6-1`	**Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits *(Since R2021a)***

14. Templates

`MISRA C++14 Rule A14-1-1`	**A template should check if a specific template argument is suitable for this template *(Since R2021b)***
`MISRA C++14 Rule A14-5-1`	**A template constructor shall not participate in overload resolution for a single argument of the enclosing class type *(Since R2021a)***
`MISRA C++14 Rule A14-5-2`	**Class members that are not dependent on template class parameters should be defined in a separate base class**
`MISRA C++14 Rule A14-5-3`	**A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations**
`MISRA C++14 Rule A14-7-1`	**A type used as a template argument shall provide all members that are used by the template *(Since R2021b)***
`MISRA C++14 Rule A14-7-2`	**Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared *(Since R2020a)***
`MISRA C++14 Rule A14-8-2`	**Explicit specializations of function templates shall not be used *(Since R2020a)***

`MISRA C++14 Rule M14-5-3`	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter
`MISRA C++14 Rule M14-6-1`	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->
`MISRA C++14 Rule A14-1-1`	A template should check if a specific template argument is suitable for this template *(Since R2021b)*
`MISRA C++14 Rule A14-5-1`	A template constructor shall not participate in overload resolution for a single argument of the enclosing class type *(Since R2021a)*
`MISRA C++14 Rule A14-5-2`	Class members that are not dependent on template class parameters should be defined in a separate base class
`MISRA C++14 Rule A14-5-3`	A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations
`MISRA C++14 Rule A14-7-1`	A type used as a template argument shall provide all members that are used by the template *(Since R2021b)*
`MISRA C++14 Rule A14-7-2`	Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared *(Since R2020a)*
`MISRA C++14 Rule A14-8-2`	Explicit specializations of function templates shall not be used *(Since R2020a)*

MISRA C++14 Rule M14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter
MISRA C++14 Rule M14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->

15. Exception Handling

MISRA C++14 Rule A15-0-2	At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee *(Since R2022a)*
MISRA C++14 Rule A15-0-3	Exception safety guarantee of a called function shall be considered *(Since R2022a)*
MISRA C++14 Rule A15-0-7	Exception handling mechanism shall guarantee a deterministic worst-case time execution time *(Since R2022a)*
MISRA C++14 Rule A15-1-1	Only instances of types derived from std::exception should be thrown *(Since R2020b)*
MISRA C++14 Rule A15-1-2	An exception object shall not be a pointer
MISRA C++14 Rule A15-1-3	All thrown exceptions should be unique *(Since R2020b)*
MISRA C++14 Rule A15-1-4	If a function exits with an exception, then before a throw, the function shall place all objects/resources that the function constructed in valid states or it shall delete them. *(Since R2021b)*

MISRA C++14 Rule A15-1-5	Exceptions shall not be thrown across execution boundaries *(Since R2022b)*
MISRA C++14 Rule A15-2-1	Constructors that are not noexcept shall not be invoked before program startup
MISRA C++14 Rule A15-2-2	If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception *(Since R2021a)*
MISRA C++14 Rule A15-3-3	Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions *(Since R2020b)*
MISRA C++14 Rule A15-3-4	Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to MISRA C++14 guidelines *(Since R2020b)*
MISRA C++14 Rule A15-3-5	A class type exception shall be caught by reference or const reference
MISRA C++14 Rule A15-4-1	Dynamic exception-specification shall not be used *(Since R2021a)*
MISRA C++14 Rule A15-4-2	If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception
MISRA C++14 Rule A15-4-3	The noexcept specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider *(Since R2020b)*

`MISRA C++14 Rule A15-4-4`	A declaration of non-throwing function shall contain noexcept specification *(Since R2021a)*
`MISRA C++14 Rule A15-4-5`	Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders *(Since R2021a)*
`MISRA C++14 Rule A15-5-1`	All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate *(Since R2020b)*
`MISRA C++14 Rule A15-5-2`	Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done *(Since R2021b)*
`MISRA C++14 Rule A15-5-3`	The std::terminate() function shall not be called implicitly
`MISRA C++14 Rule M15-0-3`	Control shall not be transferred into a try or catch block using a goto or a switch statement
`MISRA C++14 Rule M15-1-1`	The assignment-expression of a throw statement shall not itself cause an exception to be thrown *(Since R2020b)*
`MISRA C++14 Rule M15-1-2`	NULL shall not be thrown explicitly
`MISRA C++14 Rule M15-1-3`	An empty throw (throw;) shall only be used in the compound statement of a catch handler

`MISRA C++14 Rule M15-3-1`	**Exceptions shall be raised only after startup and before termination**
`MISRA C++14 Rule M15-3-3`	**Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases**
`MISRA C++14 Rule M15-3-4`	**Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point *(Since R2020b)***
`MISRA C++14 Rule M15-3-6`	**Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class**
`MISRA C++14 Rule M15-3-7`	**Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last**

16. Preprocessing Directives

`MISRA C++14 Rule A16-0-1`	**The preprocessor shall only be used for unconditional and conditional file inclusion and include guards, and using specific directives**
`MISRA C++14 Rule A16-2-1`	**The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive**
`MISRA C++14 Rule A16-2-2`	**There shall be no unused include directives *(Since R2021b)***
`MISRA C++14 Rule A16-2-3`	**An include directive shall be added explicitly for every symbol used in a file *(Since R2021b)***

MISRA C++14 Rule A16-6-1	**#error directive shall not be used *(Since R2020a)***
MISRA C++14 Rule A16-7-1	**The #pragma directive shall not be used**
MISRA C++14 Rule M16-0-1	**#include directives in a file shall only be preceded by other preprocessor directives or comments**
MISRA C++14 Rule M16-0-2	**Macros shall only be #define'd or #undef'd in the global namespace**
MISRA C++14 Rule M16-0-5	**Arguments to a function-like macro shall not contain tokens that look like pre-processing directives**
MISRA C++14 Rule M16-0-6	**In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##**
MISRA C++14 Rule M16-0-7	**Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator**
MISRA C++14 Rule M16-0-8	**If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token**
MISRA C++14 Rule M16-1-1	**The defined pre-processor operator shall only be used in one of the two standard forms**
MISRA C++14 Rule M16-1-2	**All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related**

`MISRA C++14 Rule M16-2-3`	Include guards shall be provided
`MISRA C++14 Rule M16-3-1`	There shall be at most one occurrence of the # or ## operators in a single macro definition
`MISRA C++14 Rule M16-3-2`	The # and ## operators should not be used

17. Library Introduction

`MISRA C++14 Rule A17-0-1`	Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined
`MISRA C++14 Rule A17-1-1`	Use of the C Standard Library shall be encapsulated and isolated *(Since R2021a)*
`MISRA C++14 Rule A17-6-1`	Non-standard entities shall not be added to standard namespaces *(Since R2020a)*
`MISRA C++14 Rule M17-0-2`	The names of standard library macros and objects shall not be reused
`MISRA C++14 Rule M17-0-3`	The names of standard library functions shall not be overridden
`MISRA C++14 Rule M17-0-5`	The setjmp macro and the longjmp function shall not be used

18. Language Support Library

`MISRA C++14 Rule A18-0-1`	The C library facilities shall only be accessed through C++ library headers

`MISRA C++14 Rule A18-0-2`	**The error state of a conversion from string to a numeric value shall be checked**
`MISRA C++14 Rule A18-0-3`	**The library <clocale> (locale.h) and the setlocale function shall not be used**
`MISRA C++14 Rule A18-1-1`	**C-style arrays shall not be used**
`MISRA C++14 Rule A18-1-2`	**The std::vector<bool> specialization shall not be used**
`MISRA C++14 Rule A18-1-3`	**The std::auto_ptr shall not be used *(Since R2020a)***
`MISRA C++14 Rule A18-1-4`	**A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type *(Since R2022a)***
`MISRA C++14 Rule A18-1-6`	**All std::hash specializations for user-defined types shall have a noexcept function call operator *(Since R2020a)***
`MISRA C++14 Rule A18-5-1`	**Functions malloc, calloc, realloc and free shall not be used**
`MISRA C++14 Rule A18-5-2`	**Non-placement new or delete expressions shall not be used *(Since R2020a)***
`MISRA C++14 Rule A18-5-3`	**The form of delete operator shall match the form of new operator used to allocate the memory**
`MISRA C++14 Rule A18-5-4`	**If a project has sized or unsized version of operator 'delete' globally defined, then both**

	sized and unsized versions shall be defined
MISRA C++14 Rule A18-5-5	Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel *(Since R2021b)*
MISRA C++14 Rule A18-5-7	If non-real-time implementation of dynamic memory management functions is used in the project, then memory shall only be allocated and deallocated during non-real-time program phases *(Since R2022a)*
MISRA C++14 Rule A18-5-8	Objects that do not outlive a function shall have automatic storage duration *(Since R2021b)*
MISRA C++14 Rule A18-5-9	Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard *(Since R2020b)*
MISRA C++14 Rule A18-5-10	Placement new shall be used only with properly aligned pointers to sufficient storage capacity *(Since R2020b)*
MISRA C++14 Rule A18-5-11	"operator new" and "operator delete" shall be defined together *(Since R2020b)*
MISRA C++14 Rule A18-9-1	The std::bind shall not be used
MISRA C++14 Rule A18-9-2	Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference *(Since R2020b)*

MISRA C++14 Rule A18-9-3	The std::move shall not be used on objects declared const or const& *(Since R2020a)*
MISRA C++14 Rule A18-9-4	An argument to std::forward shall not be subsequently used *(Since R2020b)*
MISRA C++14 Rule M18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used
MISRA C++14 Rule M18-0-4	The time handling functions of library <ctime> shall not be used
MISRA C++14 Rule M18-0-5	The unbounded functions of library <cstring> shall not be used
MISRA C++14 Rule M18-2-1	The macro offsetof shall not be used
MISRA C++14 Rule M18-7-1	The signal handling facilities of <csignal> shall not be used

19. Diagnostics Library

MISRA C++14 Rule M19-3-1	The error indicator errno shall not be used

20. General Utilities Library

MISRA C++14 Rule A20-8-1	An already-owned pointer value shall not be stored in an unrelated smart pointer *(Since R2021a)*
MISRA C++14 Rule A20-8-2	A std::unique_ptr shall be used to represent exclusive ownership *(Since R2020b)*

MISRA C++14 Rule A20-8-3	**A std::shared_ptr shall be used to represent shared ownership *(Since R2020b)***
MISRA C++14 Rule A20-8-4	**A std::unique_ptr shall be used over std::shared_ptr if ownership sharing is not required *(Since R2022b)***
MISRA C++14 Rule A20-8-5	**std::make_unique shall be used to construct objects owned by std::unique_ptr *(Since R2020b)***
MISRA C++14 Rule A20-8-6	**std::make_shared shall be used to construct objects owned by std::shared_ptr *(Since R2020b)***
MISRA C++14 Rule A20-8-7	**A std::weak_ptr shall be used to represent temporary shared ownership. *(Since R2022a)***

21. Strings Library

MISRA C++14 Rule A21-8-1	**Arguments to character-handling functions shall be representable as an unsigned char**

23. Containers Library

MISRA C++14 Rule A23-0-1	**An iterator shall not be implicitly converted to const_iterator *(Since R2020a)***
MISRA C++14 Rule A23-0-2	**Elements of a container shall only be accessed via valid references, iterators, and pointers *(Since R2022a)***

25. Algorithms Library

MISRA C++14 Rule A25-1-1	Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied *(Since R2022a)*
MISRA C++14 Rule A25-4-1	Ordering predicates used with associative containers and STL sorting and related algorithms shall adhere to a strict weak ordering relation *(Since R2022a)*

26. Random Number Generation

MISRA C++14 Rule A26-5-1	Pseudorandom numbers shall not be generated using std::rand()
MISRA C++14 Rule A26-5-2	Random number engines shall not be default-initialized *(Since R2020b)*

27. Input Output Library

MISRA C++14 Rule A27-0-1	Inputs from independent components shall be validated. *(Since R2021b)*
MISRA C++14 Rule A27-0-2	A C-style string shall guarantee sufficient space for data and the null terminator *(Since R2020b)*
MISRA C++14 Rule A27-0-3	Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call *(Since R2020b)*

`MISRA C++14 Rule A27-0-4`	**C-style strings shall not be used *(Since R2021a)***
`MISRA C++14 Rule M27-0-1`	**The stream input/output library <cstdio> shall not be used**

Object Detection and Forward Collision Warning (FCW) Program

Program 1:

Objective:

This program performs object detection to identify vehicles or pedestrians in front of the vehicle and calculates the Time-to-Collision (TTC). If the TTC is below a threshold (indicating an imminent collision), it triggers a forward collision warning (FCW).

MISRA C++:2023 Compliance Considerations:

- No dynamic memory allocation (except where absolutely necessary).
- Function declarations must be correctly scoped and used.
- Avoid unsafe type conversions and unnecessary global variables.
- Error handling must be robust (e.g., checking for valid input).

```
#include <iostream>
#include <opencv2/opencv.hpp>

// Define a structure to hold detected object information
struct DetectedObject {
   int x, y, width, height;  // Bounding box coordinates
   float distance;           // Distance to the detected object (in meters)
   float velocity;           // Relative velocity (in m/s)
};

// Function to preprocess the input frame
// MISRA C++:2023 Rule 5-0-1 (No use of dynamic memory allocation)
void preprocessFrame(const cv::Mat &inputFrame, cv::Mat &outputFrame)
{
   // Resize the image to the required size (e.g., 416x416)
   cv::resize(inputFrame, outputFrame, cv::Size(416, 416));
}

// Function to detect objects in the frame (dummy object detection)
// MISRA C++:2023 Rule 5-3-1 (No unsafe type casts)
void detectObjects(const cv::Mat &frame, DetectedObject &detectedObject)
{
   // Example object detection (Dummy values for illustration)
   detectedObject.x = 100;      // Dummy value for object location
   detectedObject.y = 150;      // Dummy value for object location
   detectedObject.width = 50;   // Dummy bounding box size
   detectedObject.height = 50;  // Dummy bounding box size
   detectedObject.distance = 20.0f;  // Dummy distance (in meters)
```

```
    detectedObject.velocity = 15.0f;  // Dummy velocity (in m/s)
}

// Function to calculate Time-to-Collision (TTC)
// MISRA C++:2023 Rule 5-0-2 (No use of 'goto' statements)
float calculateTTC(const DetectedObject &object)
{
    // Validate inputs (distance and velocity must not be zero)
    if (object.velocity == 0.0f) {
        std::cerr << "Error: Invalid velocity value!" << std::endl;
        return -1.0f;  // Return invalid value if velocity is zero
    }
    return object.distance / object.velocity;
}

// Function to trigger a collision warning based on TTC
// MISRA C++:2023 Rule 5-0-3 (Avoid side effects and keep functions simple)
void triggerCollisionWarning(float ttc)
{
    const float threshold = 2.0f; // 2 seconds threshold for collision warning
    if (ttc < threshold) {
        std::cout << "Warning: Collision Imminent! TTC: " << ttc << " seconds" << std::endl;
        // Trigger visual/audio alert (integration with real-world alert system)
    }
}

// Main function to handle video feed and perform detection
int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    cv::VideoCapture cap(0);  // Open the default camera (ID 0)
    if (!cap.isOpened()) {
        std::cerr << "Error: Could not open video feed!" << std::endl;
        return 1;  // Return error code if video feed can't be opened
    }

    cv::Mat frame, processedFrame;
    DetectedObject detectedObject;

    while (true) {
        // Capture the frame from the camera
        cap >> frame;
        if (frame.empty()) {
            break;  // Exit if the frame is empty
        }

        // Preprocess the frame
        preprocessFrame(frame, processedFrame);
        // Detect objects in the frame
        detectObjects(processedFrame, detectedObject);

        // Calculate Time-to-Collision (TTC)
        float ttc = calculateTTC(detectedObject);
        if (ttc < 0) {
            continue;  // Skip the rest of the loop if TTC calculation is invalid
        }

        // Trigger collision warning if TTC is below the threshold
        triggerCollisionWarning(ttc);

        // Visualize the detected object (draw bounding box and TTC)
        cv::rectangle(frame, cv::Point(detectedObject.x, detectedObject.y),
                      cv::Point(detectedObject.x + detectedObject.width, detectedObject.y +
detectedObject.height),
                      cv::Scalar(0, 255, 0), 2);  // Green bounding box
        cv::putText(frame, "TTC: " + std::to_string(ttc),
                    cv::Point(detectedObject.x, detectedObject.y - 10),
                    cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(0, 255, 0), 2);  // Display TTC

        // Display the frame with object detection and TTC info
        cv::imshow("Frame", frame);

        // Exit condition: press 'q' to quit
        if (cv::waitKey(1) & 0xFF == 'q') {
            break;
        }
    }

    cap.release();  // Release the video capture object
    cv::destroyAllWindows();  // Close all OpenCV windows

    return 0;
}
```

Explanation:

1. Preprocessing the Frame:
    - The frame is resized to match the input dimensions expected by the object detection model. This step prepares the frame for further processing.

2. Object Detection:
   - A simplified object detection function is used here, where we assign dummy values for the object's bounding box, distance, and velocity. in a real-world application, this would be replaced by a deep learning model like YOLO or SSD.
3. Time-to-Collision (TTC):
   - The `calculateTTC` function computes the Time-to-Collision (TTC) based on the object's distance and velocity. If the velocity is zero, it returns an invalid TTC value to prevent division by zero.
4. Collision Warning:
   - If the TTC falls below the specified threshold (e.g., 2 seconds), a warning message is triggered. This is where you would add real-world feedback mechanisms like audio or visual alerts.
5. Main Loop:
   - The program captures video from the default camera, processes each frame, performs object detection, calculates TTC, and triggers a warning if necessary. The results are displayed in real-time with bounding boxes around detected objects and the TTC value shown.
6. MISRA Compliance:
   - Error Handling: Proper checks are added for invalid values (e.g., zero velocity).
   - No Dynamic Memory Allocation: The program uses fixed-size data structures and avoids dynamic memory allocation (e.g., using `cv::Mat` for image handling without dynamically allocating memory).
   - Simple, Clear Functions: Each function performs a single task, which improves maintainability and readability.

Forward Collision Warning (FCW) with Object Detection

Program 2:

Objective:

To develop a Forward Collision Warning (FCW) system that uses object detection to monitor the environment around the vehicle and trigger a warning if a collision is imminent. The system should calculate the Time-to-Collision (TTC) based on object proximity and relative velocity and alert the driver if TTC is below a set threshold.

MISRA C++:2023 Compliance Considerations:

- Memory Management: Avoid dynamic memory allocation. Use automatic or static memory for all variables.
- Error Handling: Ensure proper error checks, especially for division by zero or invalid values.
- Function Purity: Ensure that each function performs a single task (i.e., no side effects).
- No `goto` Statements: Use structured flow control (if/else) instead of `goto`.
- Type Safety: Avoid unsafe type conversions and ensure proper data types are used.
- No Global Variables: Ensure that all variables are locally scoped to avoid unintended side effects.

C++ Code for FCW with Object Detection:

```cpp
#include <iostream>
#include <opencv2/opencv.hpp>

// Define a structure to hold detected object information
struct DetectedObject {
   int x, y, width, height;  // Bounding box coordinates
   float distance;           // Distance to the detected object (in meters)
   float velocity;           // Relative velocity (in m/s)
};
```

```
// Function to preprocess the input frame (resize to expected size)
// MISRA C++:2023 Rule 5-0-1 (No dynamic memory allocation)
void preprocessFrame(const cv::Mat &inputFrame, cv::Mat &outputFrame)
{
   // Resize the image to the required size (e.g., 416x416)
   cv::resize(inputFrame, outputFrame, cv::Size(416, 416));
}

// Function to simulate object detection in the frame (dummy object detection)
// MISRA C++:2023 Rule 5-3-1 (No unsafe type casts)
void detectObjects(const cv::Mat &frame, DetectedObject &detectedObject)
{
   // Example object detection (Dummy values for illustration)
   detectedObject.x = 100;     // Object location (x-coordinate)
   detectedObject.y = 150;     // Object location (y-coordinate)
   detectedObject.width = 50;   // Bounding box width
   detectedObject.height = 50;  // Bounding box height
   detectedObject.distance = 20.0f;  // Distance to object (in meters)
   detectedObject.velocity = 10.0f;  // Relative velocity (in m/s)
}

// Function to calculate Time-to-Collision (TTC)
float calculateTTC(const DetectedObject &object)
{
   // Validate inputs
   if (object.velocity == 0.0f) {
      std::cerr << "Error: Invalid velocity value!" << std::endl;
      return -1.0f;  // Return invalid TTC if velocity is zero
   }
   // MISRA C++:2023 Rule 5-0-2 (No unsafe type casts)
   return object.distance / object.velocity;
}

// Function to trigger collision warning based on TTC
void triggerCollisionWarning(float ttc)
{
   const float threshold = 2.0f; // Threshold for collision warning (2 seconds)
   if (ttc < threshold) {
      std::cout << "Warning: Collision Imminent! TTC: " << ttc << " seconds" << std::endl;
      // MISRA C++:2023 Rule 5-0-3 (No side effects in simple functions)
      // A more complex alert mechanism could be triggered here.
   }
}
```

```
int main()
{
   // MISRA C++:2023 Rule 5-0-4 (No global variables)
   cv::VideoCapture cap(0);  // Open the default camera (ID 0)
   if (!cap.isOpened()) {
      std::cerr << "Error: Could not open video feed!" << std::endl;
      return 1;  // Return error code if video feed can't be opened
   }

   cv::Mat frame, processedFrame;
   DetectedObject detectedObject;

   while (true) {
      // Capture the frame from the camera
      cap >> frame;
      if (frame.empty()) {
         break;  // Exit if the frame is empty
      }

      // Preprocess the frame
      preprocessFrame(frame, processedFrame);

      // Detect objects in the frame
      detectObjects(processedFrame, detectedObject);

      // Calculate Time-to-Collision (TTC)
      float ttc = calculateTTC(detectedObject);
      if (ttc < 0) {
         continue;  // Skip if TTC calculation is invalid
      }

      // Trigger collision warning if TTC is below the threshold
      triggerCollisionWarning(ttc);

      // Visualize the detected object (draw bounding box and TTC)
      cv::rectangle(frame, cv::Point(detectedObject.x, detectedObject.y),
                 cv::Point(detectedObject.x + detectedObject.width, detectedObject.y +
detectedObject.height),
                 cv::Scalar(0, 255, 0), 2);  // Green bounding box
      cv::putText(frame, "TTC: " + std::to_string(ttc),
               cv::Point(detectedObject.x, detectedObject.y - 10),
               cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(0, 255, 0), 2);  // Display TTC

      // Display the frame with object detection and TTC info
      cv::imshow("Frame", frame);

      // Exit condition: press 'q' to quit
      if (cv::waitKey(1) & 0xFF == 'q') {
         break;
```

```
        }
    }

    cap.release();  // Release the video capture object
    cv::destroyAllWindows();  // Close all OpenCV windows

    return 0;
}
```

Explanation:

1. Preprocessing the Frame:
    - The `preprocessFrame` function resizes the frame to a fixed size (e.g., 416x416) to match the expected input size for the object detection algorithm.
2. Object Detection:
    - The `detectObjects` function is a dummy implementation simulating object detection by assigning fixed values for the object's bounding box, distance, and velocity. In a real-world scenario, this would be replaced with an actual object detection model like YOLO or SSD.
3. TTC Calculation:
    - The `calculateTTC` function computes the Time-to-Collision (TTC) by dividing the object's distance by its velocity. If the velocity is zero, an error message is printed to avoid division by zero.
4. Collision Warning:
    - If the calculated TTC is below a threshold (e.g., 2 seconds), the `triggerCollisionWarning` function triggers a warning (e.g., "Collision Imminent!") to alert the driver.
5. Main Loop:
    - The `main` function captures frames from the camera, processes them, performs object detection, calculates TTC, and triggers warnings if necessary. The results are displayed in real-time with bounding boxes around detected objects and the TTC values shown.
6. MISRA C++:2023 Compliance:
    - No dynamic memory allocation: The code uses local variables and avoids dynamic memory allocation.
    - No `goto` statements: The program uses structured flow control (if/else).

    - Error handling: The program checks for invalid velocity values and returns an error if encountered.
    - No global variables: All variables are scoped locally within functions.
    - Function Purity: Each function performs a single, clear task (e.g., TTC calculation, object detection, warning).

Key MISRA C++:2023 Rules Addressed:

- Rule 5-0-1: No dynamic memory allocation.
- Rule 5-3-1: Proper handling of types and avoiding unsafe casts.
- Rule 5-0-2: No use of `goto` statements.
- Rule 5-0-3: Clear error handling.
- Rule 5-0-4: No global variables used.
- Rule 5-0-5: Functions perform specific tasks (no side effects).

Libraries Used:

- OpenCV: Used for image processing (capturing frames, resizing images, drawing bounding boxes).
- C++ Standard Library: Used for basic functionalities like I/O and control flow.

Forward Collision Warning (FCW) with Object Detection

Program 3:

Objective:

To create an FCW system that uses a radar-based detection model instead of camera-based object detection. The system will calculate the Time-to-Collision (TTC) using the distance to detected objects from a radar sensor and their relative velocity, and will trigger an alert when a collision is imminent.

MISRA C++:2023 Compliance Considerations:

- Memory Management: Avoid dynamic memory allocation.
- No Goto Statements: Use structured flow control (e.g., `if/else`) instead of `goto`.
- Error Handling: Ensure proper checks for invalid values (e.g., zero velocity).
- No Global Variables: Ensure that variables are scoped locally.

C++ Code for FCW with Radar Detection:

```
#include <iostream>
#include <cmath>

// Define a structure to hold radar detection information
struct DetectedObject {
    float distance;          // Distance to the detected object (in meters)
    float velocity;          // Relative velocity (in m/s)
};

// Function to calculate Time-to-Collision (TTC)
float calculateTTC(const DetectedObject &object)
{
    // Validate inputs
    if (object.velocity == 0.0f) {
        std::cerr << "Error: Invalid velocity value!" << std::endl;
        return -1.0f;  // Return invalid TTC if velocity is zero
    }
    // Calculate TTC based on radar distance and velocity
    return object.distance / object.velocity;
}

// Function to trigger collision warning based on TTC
void triggerCollisionWarning(float ttc)
{
    const float threshold = 3.0f; // Threshold for collision warning (3 seconds)
    if (ttc < threshold) {
        std::cout << "Warning: Collision Imminent! TTC: " << ttc << " seconds" << std::endl;
        // Additional alert logic (visual/audio) can be triggered here
    }
}

// Main function to simulate radar-based detection
int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Simulated radar readings (distance and velocity of detected object)
    DetectedObject detectedObject;
    detectedObject.distance = 50.0f;  // Distance to object (in meters)
    detectedObject.velocity = 15.0f;  // Relative velocity (in m/s)

    // Calculate Time-to-Collision (TTC)
    float ttc = calculateTTC(detectedObject);
    if (ttc < 0) {
        return 1;  // Exit if TTC calculation is invalid
    }

    // Trigger collision warning if TTC is below the threshold
    triggerCollisionWarning(ttc);

    return 0;
}
```

Explanation:

1. **Radar Detection Model**:
    - In this example, we use a **radar-based detection system** that simply provides the **distance** to an object and its **relative velocity**.
    - The `DetectedObject` structure holds these values for each object detected by the radar.
2. **TTC Calculation**:
    - The `calculateTTC` function calculates the **Time-to-Collision (TTC)** by dividing the object's **distance** by its **velocity**. This gives an estimate of how long it will take for the vehicle to collide with the detected object.
    - If the velocity is zero, an error message is printed to avoid division by zero.
3. **Collision Warning**:
    - If the TTC is below a specified threshold (e.g., 3 seconds), the system triggers a collision warning.
    - The warning message is printed to the console, but in a real system, this could be replaced with **audio or visual alerts**.
4. **Main Logic**:
    - The **main function** simulates the radar detection by assigning fixed values for distance and velocity.
    - The `calculateTTC` and `triggerCollisionWarning` functions are called to compute the TTC and issue warnings if necessary.

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation.
- **Rule 5-3-1**: Proper handling of types and avoiding unsafe casts.
- **Rule 5-0-2**: No use of `goto` statements.
- **Rule 5-0-3**: Error handling for invalid velocity values.
- **Rule 5-0-4**: No global variables used (all variables are locally scoped).
- **Rule 5-0-5**: Functions perform specific tasks without side effects.

Libraries Used:

- **C++ Standard Library**: Basic functionalities like I/O, control flow, and mathematical operations.

Forward Collision Warning (FCW) with Object Detection

Program 4:

Objective:

To create an **FCW system** that uses **sensor fusion** (combining data from **LIDAR** and **radar**) to detect objects and calculate **Time-to-Collision (TTC)**. The system will trigger a collision warning if the TTC falls below a set threshold.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation and ensure memory safety.
- **Error Handling**: Handle invalid inputs and prevent division by zero or other mathematical errors.
- **Function Purity**: Ensure that each function is single-purpose and has no side effects.
- **No `goto` Statements**: Use structured flow control.
- **Type Safety**: Ensure correct use of data types and avoid unsafe casts.
- **No Global Variables**: All variables must be scoped locally.

C++ Code for FCW with Sensor Fusion (LIDAR and Radar):

```cpp
#include <iostream>
#include <cmath>

// Define a structure to hold sensor data for detected objects
struct DetectedObject {
    float lidar_distance;     // Distance to the object (from LIDAR)
    float radar_velocity;     // Relative velocity (from Radar)
    float lidar_velocity;     // Velocity estimate from LIDAR
};

// Function to calculate Time-to-Collision (TTC) based on sensor data
float calculateTTC(const DetectedObject &object)
{
    // MISRA C++:2023 Rule 5-0-1 (No dynamic memory allocation)
    // Ensure that velocities are non-zero to avoid division by zero
    if (object.lidar_velocity == 0.0f || object.radar_velocity == 0.0f) {
        std::cerr << "Error: Invalid velocity value!" << std::endl;
        return -1.0f;  // Return invalid TTC if velocity is zero
    }

    // Calculate average velocity using sensor fusion (LIDAR + Radar)
    float average_velocity = (object.lidar_velocity + object.radar_velocity) / 2.0f;

    // Calculate Time-to-Collision (TTC) based on average velocity
    return object.lidar_distance / average_velocity;
}

// Function to trigger collision warning based on TTC
void triggerCollisionWarning(float ttc)
{
    const float threshold = 2.5f; // Threshold for collision warning (2.5 seconds)
    if (ttc < threshold) {
        std::cout << "Warning: Collision Imminent! TTC: " << ttc << " seconds" << std::endl;
        // Trigger alert logic here (audio, visual, etc.)
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Simulated sensor readings (LIDAR distance and velocity from radar and LIDAR)
    DetectedObject detectedObject;
    detectedObject.lidar_distance = 45.0f;  // LIDAR distance to object (in meters)
    detectedObject.radar_velocity = 12.0f;  // Radar velocity of object (in m/s)
    detectedObject.lidar_velocity = 11.0f;  // LIDAR velocity estimate (in m/s)
    // Calculate Time-to-Collision (TTC)
    float ttc = calculateTTC(detectedObject);
    if (ttc < 0) {
        return 1;  // Exit if TTC calculation is invalid
    }

    // Trigger collision warning if TTC is below the threshold
    triggerCollisionWarning(ttc);

    return 0;
}
```

Explanation:

1. **Sensor Fusion (LIDAR + Radar)**:
   - In this example, we use **LIDAR** and **radar** data together. **LIDAR** provides accurate distance data, and **radar** provides velocity information.
   - The velocity from **radar** is used to calculate the relative speed of objects, while the **LIDAR** gives us accurate distance information.
   - These two sensors' data are **fused** to calculate an **average velocity**, which is then used to compute the **Time-to-Collision (TTC)**.
2. **TTC Calculation**:
   - The `calculateTTC` function computes **Time-to-Collision (TTC)** using the **distance** from **LIDAR** and the **average velocity** from **sensor fusion** (radar + LIDAR).
   - If either the **LIDAR velocity** or **radar velocity** is zero, the function returns an error, as division by zero would occur.
3. **Collision Warning**:
   - If the calculated **TTC** is below a threshold (e.g., 2.5 seconds), the system triggers a collision warning.
   - The `triggerCollisionWarning` function prints a message to the console, but in real applications, this can be replaced with an **audio alert** or **visual alert**.
4. **Main Logic**:
   - The `main` function simulates receiving data from **LIDAR** (distance) and **radar** (velocity) sensors, then calculates **TTC** and triggers a warning if needed.

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation (except for **automatic variables**).
- **Rule 5-3-1**: Proper handling of types and avoiding unsafe type conversions (e.g., **no casting**).
- **Rule 5-0-2**: No use of `goto` statements, and structured control flow is used.
- **Rule 5-0-3**: Clear error handling for invalid velocity values.
- **Rule 5-0-4**: No global variables are used.
- **Rule 5-0-5**: Functions are kept simple with single purposes.

Forward Collision Warning (FCW) with Object Detection

Program 4:

Objective:

To create an **FCW system** that uses **motion tracking** and **trajectory prediction** to estimate the future position of detected objects and compute the **Time-to-Collision (TTC)**. The system will trigger a collision warning if the predicted position of the object comes within a certain distance threshold of the vehicle.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation and ensure memory safety.
- **No Goto Statements**: Use structured flow control.
- **Error Handling**: Ensure proper checks for invalid values and handle potential errors (e.g., objects outside the frame).
- **No Global Variables**: Ensure that variables are scoped locally.
- **Type Safety**: Avoid unsafe type conversions and ensure correct data types are used.

C++ Code for FCW with Trajectory Prediction:

```
#include <iostream>
#include <cmath>

// Define a structure to hold detected object and vehicle data
struct MovingObject {
   float x, y;            // Position (in meters)
   float velocity_x, velocity_y;  // Velocity components (in m/s)
   float width, height;    // Size of the object (in meters)
};

// Function to predict the future position of an object based on its velocity
void predictPosition(const MovingObject &object, float time, float &predicted_x, float &predicted_y)
{
   predicted_x = object.x + object.velocity_x * time;
   predicted_y = object.y + object.velocity_y * time;
}

// Function to calculate the Time-to-Collision (TTC) based on predicted position
float calculateTTC(const MovingObject &object, const MovingObject &egoVehicle)
{
   // Predict the future position of the object
   float predicted_x, predicted_y;
   float time_to_predict = 3.0f; // Predict for 3 seconds
   predictPosition(object, time_to_predict, predicted_x, predicted_y);

   // Calculate the distance between the ego vehicle and the predicted object position
   float distance = sqrt(pow(predicted_x - egoVehicle.x, 2) + pow(predicted_y - egoVehicle.y, 2));

   // Calculate relative velocity between the object and the ego vehicle
   float relative_velocity = sqrt(pow(object.velocity_x - egoVehicle.velocity_x, 2) +
                          pow(object.velocity_y - egoVehicle.velocity_y, 2));

   // If relative velocity is zero (objects are stationary), return invalid TTC
   if (relative_velocity == 0.0f) {
      std::cerr << "Error: Invalid relative velocity!" << std::endl;
      return -1.0f;  // Invalid TTC
   }

   // Calculate Time-to-Collision (TTC)
   return distance / relative_velocity;
}

// Function to trigger a collision warning based on TTC
void triggerCollisionWarning(float ttc)
{
    const float threshold = 2.5f;  // Threshold for collision warning (2.5 seconds)
    if (ttc < threshold) {
        std::cout << "Warning: Collision Imminent! TTC: " << ttc << " seconds" << std::endl;
        // Additional alert logic (audio, visual, etc.) can be triggered here
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Simulate data for an object and ego vehicle
    MovingObject detectedObject = {100.0f, 50.0f, 5.0f, 0.0f, 2.0f, 1.0f};  // Example object at (100, 50)
moving with velocity 5 m/s along x-axis
    MovingObject egoVehicle = {0.0f, 0.0f, 10.0f, 0.0f, 4.5f, 2.0f};        // Ego vehicle at origin (0,0) moving
with velocity 10 m/s along x-axis

    // Calculate Time-to-Collision (TTC)
    float ttc = calculateTTC(detectedObject, egoVehicle);
    if (ttc < 0) {
        return 1;  // Exit if TTC calculation is invalid
    }

    // Trigger collision warning if TTC is below the threshold
    triggerCollisionWarning(ttc);

    return 0;
}
```

Explanation:

1. **Motion Tracking and Trajectory Prediction**:
    - In this approach, the future position of the detected object is predicted based on its **velocity** and **position.** This is done by simulating the object's movement over time (`predictPosition` function).
    - The system tracks the object's **x** and **y** coordinates along with its **velocity components** in both **x** and **y** directions.
2. **TTC Calculation**:
    - The `calculateTTC` function predicts the future position of the detected object and calculates the **distance** between the ego vehicle and the predicted position.
    - The **relative velocity** between the object and the ego vehicle is calculated using the difference in their velocities.

    - The **Time-to-Collision (TTC)** is then calculated by dividing the distance by the relative velocity.

3. **Collision Warning**:
    - If the **TTC** is less than a specified threshold (e.g., 2.5 seconds), the system triggers a collision warning using the `triggerCollisionWarning` function.
    - The warning is printed to the console, but it can be replaced with real-world alerts like **audio** or **visual** cues.
4. **Main Logic**:
    - The `main` function simulates data for a detected object and the ego vehicle, calculates **TTC**, and triggers a collision warning if necessary.
5. **MISRA C++:2023 Compliance**:
    - **No dynamic memory allocation**: All variables are statically allocated.
    - **No `goto` statements**: The program uses structured control flow (if/else).
    - **Error handling**: The code checks for invalid **relative velocity** values to avoid division by zero.
    - **No global variables**: The program uses locally scoped variables.
    - **Type safety**: The code ensures that all variables are appropriately typed.

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation.
- **Rule 5-3-1**: Proper handling of types and avoiding unsafe type conversions.
- **Rule 5-0-2**: No use of `goto` statements.
- **Rule 5-0-3**: Clear error handling for invalid values (e.g., velocity).
- **Rule 5-0-4**: No global variables used.
- **Rule 5-0-5**: Functions are kept simple and focused on single tasks.

Forward Collision Warning (FCW) with Object Detection

Program 5:

Objective:

To create an **FCW system** that detects potential collisions by comparing the **bounding boxes** of detected objects and the vehicle. If the bounding boxes overlap significantly, the system will trigger a collision warning. This is a **simplified approach** based on **geometrical calculations** rather than using sensor fusion or trajectory prediction.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation.
- **No Goto Statements**: Use structured flow control.
- **Error Handling**: Handle invalid values and ensure proper checks.
- **No Global Variables**: Ensure all variables are scoped locally.
- **Type Safety**: Avoid unsafe type conversions and ensure correct data types are used.

C++ Code for FCW with Bounding Box Overlap:

```
#include <iostream>
#include <cmath>

// Define a structure to hold detected object and vehicle bounding box data
struct BoundingBox {
   float x, y;           // Coordinates of the top-left corner of the bounding box (in meters)
   float width, height;    // Width and height of the bounding box (in meters)
};

// Function to check if two bounding boxes overlap (collision detection)
bool isCollision(const BoundingBox &object, const BoundingBox &egoVehicle)
{
   // Check if the bounding boxes overlap in both the x and y dimensions
   if (object.x < (egoVehicle.x + egoVehicle.width) &&
      (object.x + object.width) > egoVehicle.x &&
      object.y < (egoVehicle.y + egoVehicle.height) &&
      (object.y + object.height) > egoVehicle.y)
   {
        return true;  // Collision detected
    }
    return false;  // No collision
}

// Function to trigger collision warning based on bounding box overlap
void triggerCollisionWarning(bool collisionDetected)
{
    if (collisionDetected) {
        std::cout << "Warning: Collision Imminent! Bounding box overlap detected." << std::endl;
        // Additional alert logic (audio, visual, etc.) can be triggered here
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Simulated bounding boxes for the detected object and the ego vehicle
    BoundingBox detectedObject = {50.0f, 30.0f, 5.0f, 3.0f};  // Object at (50,30) with width 5m and
height 3m
    BoundingBox egoVehicle = {0.0f, 0.0f, 4.5f, 2.0f};        // Ego vehicle at (0,0) with width 4.5m and
height 2m

    // Check for collision using bounding box overlap
    bool collisionDetected = isCollision(detectedObject, egoVehicle);

    // Trigger collision warning if overlap is detected
    triggerCollisionWarning(collisionDetected);

    return 0;
}
```

Explanation:

1. **Bounding Box Definition**:
    - The `BoundingBox` structure defines the **position (x, y)** and **dimensions (width, height)** of a bounding box for both the detected object and the **ego vehicle**.
    - The **detected object** could be a pedestrian, another vehicle, or any obstacle detected by the system, while the **ego vehicle** represents the car or robot that is running the FCW system.
2. **Collision Detection**:
    - The `isCollision` function checks if two bounding boxes (the ego vehicle and the detected object) overlap by comparing their coordinates

    and dimensions.
    - The check is done in both the **x-direction** and **y-direction** to verify if the bounding boxes intersect in both dimensions.
3. **Collision Warning**:
    - The `triggerCollisionWarning` function is called when a collision is detected (i.e., when the bounding boxes overlap).
    - If an overlap is found, a **collision warning** message is displayed. This can be extended to trigger more complex alert systems (e.g., **audio, visual, or haptic feedback**).
4. **Main Logic**:
    - The `main` function initializes the **bounding boxes** for the detected object and the ego vehicle, performs the collision check, and triggers the warning if necessary.

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation.
- **Rule 5-3-1**: Proper handling of types and avoiding unsafe casts.
- **Rule 5-0-2**: No use of `goto` statements, structured flow control is used.
- **Rule 5-0-3**: Clear error handling for invalid values (although none in this simple case, it's ready for extension).
- **Rule 5-0-4**: No global variables used; everything is locally scoped.
- **Rule 5-0-5**: Functions are kept simple and perform only one task (e.g., collision check, warning trigger).

Object Detection

Program 1:

Objective:

To develop a basic **Object Detection** system that detects objects in an image using a pre-trained **YOLO** (You Only Look Once) model, calculates the **bounding boxes** of the detected objects, and processes these detections to generate alerts based on object proximity. This example demonstrates how to integrate **OpenCV** for image processing and **YOLO** for object detection.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation. Use local variables and static memory where applicable.
- **No Goto Statements**: Use structured control flow (if/else).
- **Error Handling**: Validate inputs and ensure proper error checking (e.g., for empty frames).
- **No Global Variables**: All variables should be scoped locally to prevent side effects.
- **Type Safety**: Avoid unsafe type conversions and ensure proper data types are used.
- **Function Purity**: Each function should perform a single, specific task.

C++ Code for Object Detection Using YOLO:

```
#include <iostream>
#include <opencv2/opencv.hpp>

// Define a structure to hold object detection information
struct DetectedObject {
    int x, y, width, height;  // Bounding box coordinates
    std::string class_id;     // Object class ID (e.g., "car", "person")
};

// Function to perform object detection using YOLO model
void objectDetection(const cv::Mat &frame, std::vector<DetectedObject> &objects)
{
    // Load YOLO model (ensure weights and config files are in place)
    cv::dnn::Net net = cv::dnn::readNetFromDarknet("yolov3.cfg", "yolov3.weights");

    // Prepare the image for object detection
    cv::Mat blob;
    cv::dnn::blobFromImage(frame, blob, 1 / 255.0, cv::Size(416, 416), cv::Scalar(), true, false);

    // Pass the image blob through the network
    net.setInput(blob);
    std::vector<cv::Mat> outputs;
    net.forward(outputs, net.getUnconnectedOutLayersNames());

    // Process outputs to get bounding boxes
    for (const auto &output : outputs) {
        for (int i = 0; i < output.rows; ++i) {
            float confidence = output.at<float>(i, 4);
            if (confidence > 0.5) {  // Only consider high-confidence detections
                DetectedObject object;
                object.x = output.at<float>(i, 0) * frame.cols;
                object.y = output.at<float>(i, 1) * frame.rows;
                object.width = output.at<float>(i, 2) * frame.cols;
                object.height = output.at<float>(i, 3) * frame.rows;
                object.class_id = "Detected Object";  // Placeholder for object class

                // Store the detected object
                objects.push_back(object);
            }
        }
    }
}

// Function to draw bounding boxes around detected objects
void drawBoundingBoxes(cv::Mat &frame, const std::vector<DetectedObject> &objects)
{
    for (const auto &object : objects) {
        cv::rectangle(frame, cv::Point(object.x, object.y), cv::Point(object.x + object.width, object.y +
object.height), cv::Scalar(0, 255, 0), 2);
        cv::putText(frame, object.class_id, cv::Point(object.x, object.y - 10), cv::FONT_HERSHEY_SIMPLEX,
0.5, cv::Scalar(0, 255, 0), 2);
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Capture an image from the camera
    cv::VideoCapture cap(0);  // Open the default camera
    if (!cap.isOpened()) {
        std::cerr << "Error: Could not open video feed!" << std::endl;
        return 1;  // Exit if video feed can't be opened
    }

    cv::Mat frame;
    while (true) {
        cap >> frame;
        if (frame.empty()) {
            break;  // Exit if the frame is empty
        }

        // Perform object detection
        std::vector<DetectedObject> objects;
        objectDetection(frame, objects);

        // Draw bounding boxes around detected objects
        drawBoundingBoxes(frame, objects);

        // Show the processed frame
        cv::imshow("Object Detection", frame);

        // Exit condition: press 'q' to quit
        if (cv::waitKey(1) & 0xFF == 'q') {
            break;
        }
    }

    cap.release();  // Release the video capture object
    cv::destroyAllWindows();  // Close all OpenCV windows

    return 0;
}
```

Explanation:

1. **Object Detection Using YOLO**:
    - The program loads a pre-trained **YOLO (You Only Look Once)** model using OpenCV's `dnn` module (`cv::dnn::readNetFromDarknet()`).
    - The input image is converted into a **blob** suitable for the model using `cv::dnn::blobFromImage()`.
    - The image blob is passed through the **YOLO model** using `net.forward()`, and the output is processed to extract the **bounding boxes** and **class IDs** of detected objects.
2. **Bounding Box Drawing**:
    - The `drawBoundingBoxes` function is used to draw rectangles around the detected objects and label them with their class ID.

    - This function uses **OpenCV**'s `cv::rectangle()` to draw the bounding box and `cv::putText()` to display the class name.
3. **Main Logic**:
    - The `main` function captures frames from the default camera, performs **object detection**, and draws bounding boxes on the detected objects in real-time.
    - The program runs in a loop, continuously processing frames until the user presses 'q' to quit.
4. **MISRA C++:2023 Compliance**:
    - **No Dynamic Memory Allocation**: The program uses **local variables** and avoids dynamic memory allocation.
    - **No `goto` Statements**: The program uses **structured flow control** (e.g., `if/else`).
    - **Error Handling**: The program checks whether the video feed is opened successfully and handles any empty frames.
    - **No Global Variables**: All variables are **locally scoped** within functions.
    - **Type Safety**: Proper use of types, especially when handling OpenCV structures and model outputs.

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation (except for automatic variables like `frame`).
- **Rule 5-3-1**: Proper handling of types and avoiding unsafe type conversions (e.g., when dealing with OpenCV matrices and model outputs).
- **Rule 5-0-2**: No use of `goto` statements, ensuring structured control flow (e.g., `if/else`).
- **Rule 5-0-3**: Error handling for failed video capture and empty frames.
- **Rule 5-0-4**: No global variables; all variables are scoped locally.
- **Rule 5-0-5**: Functions are kept simple, each performing a specific task (e.g., object detection, bounding box drawing).
- **Rule 5-0-6**: Ensure proper initialization and usage of variables (e.g., `objects` vector).

Object Detection

Program 2:

Objective:

To implement an **Object Detection System** using **SimpleBlobDetector** from **OpenCV** to detect circular objects in a video feed. The system will detect blobs in the image, draw bounding boxes around them, and trigger a warning if a blob is detected within a specific region (e.g., near the center of the frame).

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation. Use **local variables** and **automatic storage duration** for all objects.
- **Error Handling**: Ensure proper checks for invalid data and failed operations (e.g., empty frames).
- **No Global Variables**: Ensure variables are scoped locally to avoid unintended side effects.
- **Type Safety**: Proper handling of types, especially when dealing with **OpenCV** data structures (e.g., `cv::Mat`, `cv::KeyPoint`).
- **Function Purity**: Each function should perform a specific task (e.g., blob detection, bounding box drawing).

C++ Code for Object Detection Using SimpleBlobDetector:

```
#include <iostream>
#include <opencv2/opencv.hpp>

// Define a structure to hold blob detection data
struct DetectedBlob {
   float x, y, size;  // Coordinates and size of the detected blob
};

// Function to detect blobs in the image using SimpleBlobDetector
void detectBlobs(const cv::Mat &frame, std::vector<DetectedBlob> &blobs)
{
   // Convert the frame to grayscale
   cv::Mat gray;
   cv::cvtColor(frame, gray, cv::COLOR_BGR2GRAY);

   // Threshold the image to detect blobs
   cv::Mat thresholded;
   cv::threshold(gray, thresholded, 128, 255, cv::THRESH_BINARY);
```

```
    // Set up the SimpleBlobDetector parameters
    cv::SimpleBlobDetector::Params params;
    params.filterByArea = true;
    params.minArea = 100;
    params.filterByCircularity = true;
    params.minCircularity = 0.5;

    // Create the SimpleBlobDetector with the specified parameters
    cv::Ptr<cv::SimpleBlobDetector> detector = cv::SimpleBlobDetector::create(params);

    // Detect blobs
    std::vector<cv::KeyPoint> keypoints;
    detector->detect(thresholded, keypoints);

    // Store detected blobs data (position and size)
    for (const auto &keypoint : keypoints) {
        DetectedBlob blob;
        blob.x = keypoint.pt.x;
        blob.y = keypoint.pt.y;
        blob.size = keypoint.size;
        blobs.push_back(blob);
    }
}

// Function to draw bounding boxes around detected blobs
void drawBoundingBoxes(cv::Mat &frame, const std::vector<DetectedBlob> &blobs)
{
    for (const auto &blob : blobs) {
        // Draw a circle around the detected blob
        cv::circle(frame, cv::Point(blob.x, blob.y), static_cast<int>(blob.size / 2), cv::Scalar(0, 255, 0), 2);
        // Optionally, display the blob's size
        cv::putText(frame, std::to_string(blob.size), cv::Point(blob.x, blob.y), cv::FONT_HERSHEY_SIMPLEX,
0.5, cv::Scalar(0, 255, 0), 2);
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Capture video from the camera
    cv::VideoCapture cap(0);  // Open the default camera
    if (!cap.isOpened()) {
        std::cerr << "Error: Could not open video feed!" << std::endl;
        return 1;  // Exit if video feed can't be opened
    }

    cv::Mat frame;
    while (true) {
        cap >> frame;
        if (frame.empty()) {
            break;  // Exit if the frame is empty
        }

        // Detect blobs in the frame
        std::vector<DetectedBlob> blobs;
        detectBlobs(frame, blobs);

        // Draw bounding boxes around detected blobs
        drawBoundingBoxes(frame, blobs);

        // Show the processed frame
        cv::imshow("Blob Detection", frame);

        // Exit condition: press 'q' to quit
        if (cv::waitKey(1) & 0xFF == 'q') {
            break;
        }
    }

    cap.release();  // Release the video capture object
    cv::destroyAllWindows();  // Close all OpenCV windows

    return 0;
}
```

Explanation:

1. **Blob Detection Using SimpleBlobDetector**:
   - **SimpleBlobDetector** is an efficient method for detecting simple shapes or blobs in images. In this example, we configure the detector to find **circular** blobs by setting **`filterByArea`** and **`filterByCircularity`** parameters.
   - The frame is first converted to grayscale using `cv::cvtColor` and thresholded using `cv::threshold` to create a binary image.
   - The **SimpleBlobDetector** is then used to detect blobs in the thresholded image. Each blob's **coordinates** (center) and **size** (diameter) are stored in the `DetectedBlob` structure.
2. **Bounding Box Drawing**:
   - The `drawBoundingBoxes` function is used to draw a **circle** around each detected blob using `cv::circle`.
   - Additionally, the **size** of the blob is displayed next to the circle using

    `cv::putText`.
3. **Main Logic**:
    - The `main` function continuously captures frames from the camera using `cv::VideoCapture`.
    - For each frame, **blobs** are detected, and bounding boxes are drawn around them in real-time.
    - The processed frame is displayed using `cv::imshow`, and the program continues until the user presses the 'q' key to quit.
4. **MISRA C++:2023 Compliance**:
    - **No Dynamic Memory Allocation**: The program uses **local variables** and avoids dynamic memory allocation.
    - **No `goto` Statements**: The program uses **structured control flow** (e.g., `if/else`).
    - **Error Handling**: Proper error handling for failed video capture and empty frames.
    - **No Global Variables**: All variables are **locally scoped** within functions.
    - **Type Safety**: Proper types are used for all variables, ensuring safety when handling OpenCV structures (e.g., `cv::Mat` and `cv::KeyPoint`).
    - **Function Purity**: Each function performs a specific task with no side effects (e.g., `detectBlobs`, `drawBoundingBoxes`).

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation. The program uses local variables and avoids dynamic memory allocation.
- **Rule 5-3-1**: Proper handling of types and avoiding unsafe type conversions (e.g., ensuring correct handling of `cv::Rect` and `cv::Mat`).
- **Rule 5-0-2**: No use of `goto` statements. The code uses structured flow control (`if/else`).
- **Rule 5-0-3**: Error handling for failed video capture and empty frames.
- **Rule 5-0-4**: No global variables; all variables are scoped locally.
- **Rule 5-0-5**: Functions perform specific tasks (e.g., blob detection, bounding box drawing) without side effects.
- **Rule 5-0-6**: Proper initialization and usage of variables.

Object Detection

Program 3:

Objective:

To implement an **Object Detection System** using the **HOG (Histogram of Oriented Gradients) + SVM (Support Vector Machine)** method to detect pedestrians in a video feed. The system will process frames, detect pedestrians, and draw bounding boxes around detected objects, issuing a warning when a pedestrian is detected near the vehicle.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation. Use stack-based variables and containers like `std::vector` instead of raw pointers or dynamic allocation.
- **Error Handling**: Ensure proper error checks, especially for invalid video input and detection failures.
- **No Global Variables**: All variables are scoped locally.
- **Type Safety**: Use proper data types and avoid unsafe type conversions.
- **Function Purity**: Functions should perform a single, specific task and avoid side effects.
- **No `goto` Statements**: Use structured control flow (if/else).

C++ Code for Object Detection Using HOG + SVM:

```
#include <iostream>
#include <opencv2/opencv.hpp>

// Define a structure to hold detected object data
struct DetectedObject {
    int x, y, width, height;  // Bounding box coordinates
    std::string class_id;      // Object class ID (e.g., "pedestrian")
};

// Function to detect pedestrians using HOG + SVM
void pedestrianDetection(const cv::Mat &frame, std::vector<DetectedObject> &objects)
{
    // Load pre-trained HOG descriptor and SVM classifier for pedestrian detection
    cv::HOGDescriptor hog;
    hog.setSVMDetector(cv::HOGDescriptor::getDefaultPeopleDetector());

    // Detect pedestrians in the frame
    std::vector<cv::Rect> detections;
    hog.detectMultiScale(frame, detections, 0, cv::Size(8, 8), cv::Size(32, 32), 1.05, 2);

    // Process the detected bounding boxes
    for (const auto &rect : detections) {
        DetectedObject object;
        object.x = rect.x;
        object.y = rect.y;
        object.width = rect.width;
        object.height = rect.height;
        object.class_id = "pedestrian";  // Class ID for pedestrians
        objects.push_back(object);  // Store detected object
    }
}

// Function to draw bounding boxes around detected objects
void drawBoundingBoxes(cv::Mat &frame, const std::vector<DetectedObject> &objects)
{
    for (const auto &object : objects) {
        cv::rectangle(frame, cv::Point(object.x, object.y),
                      cv::Point(object.x + object.width, object.y + object.height),
                      cv::Scalar(0, 255, 0), 2);
        cv::putText(frame, object.class_id, cv::Point(object.x, object.y - 10),
                    cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(0, 255, 0), 2);
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Capture video from the camera
    cv::VideoCapture cap(0);  // Open the default camera
    if (!cap.isOpened()) {
        std::cerr << "Error: Could not open video feed!" << std::endl;
        return 1;  // Exit if video feed can't be opened
    }

    cv::Mat frame;
    while (true) {
        cap >> frame;
        if (frame.empty()) {
            break;  // Exit if the frame is empty
        }
        // Detect pedestrians in the frame
        std::vector<DetectedObject> objects;
        pedestrianDetection(frame, objects);

        // Draw bounding boxes around detected pedestrians
        drawBoundingBoxes(frame, objects);

        // Show the processed frame with detections
        cv::imshow("Pedestrian Detection", frame);

        // Exit condition: press 'q' to quit
        if (cv::waitKey(1) & 0xFF == 'q') {
            break;
        }
    }

    cap.release();  // Release the video capture object
    cv::destroyAllWindows();  // Close all OpenCV windows

    return 0;
}
```

Explanation:

1. **Pedestrian Detection with HOG + SVM**:
   - The **HOG + SVM** method is used for pedestrian detection. The `cv::HOGDescriptor` is initialized with a **pre-trained SVM classifier** for people detection (default people detector from OpenCV).
   - The method `hog.detectMultiScale` is used to detect pedestrians in the frame by sliding a window across the image. If a pedestrian is detected, a bounding box (`cv::Rect`) is returned.
2. **Bounding Box Drawing**:
   - The `drawBoundingBoxes` function iterates over the detected objects and draws rectangles around them using `cv::rectangle`. It also labels the bounding boxes with the class name ("pedestrian") using `cv::putText`.
3. **Main Logic**:
   - The `main` function captures frames from the default camera using OpenCV's `cv::VideoCapture`.
   - For each frame, the pedestrian detection function is called to detect pedestrians, and bounding boxes are drawn around them.

   - The processed frame is displayed using `cv::imshow`. The program continuously processes frames until the user presses the 'q' key to quit.
4. **MISRA C++:2023 Compliance**:
   - **No Dynamic Memory Allocation**: The program uses **local variables** and avoids dynamic memory allocation.
   - **No `goto` Statements**: The program uses **structured flow control** (e.g., `if/else`).
   - **Error Handling**: The program checks if the video feed is successfully opened and handles empty frames.
   - **No Global Variables**: All variables are **locally scoped** within functions, adhering to **MISRA C++:2023 Rule 5-0-4**.
   - **Type Safety**: Proper types are used for all variables, ensuring safety when handling OpenCV structures (e.g., `cv::Mat` and `cv::Rect`).

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation (e.g., `cv::Mat` is automatically managed by OpenCV).
- **Rule 5-3-1**: Proper handling of types, avoiding unsafe type conversions (e.g., handling `cv::Rect` and `cv::Mat` correctly).
- **Rule 5-0-2**: No use of `goto` statements. Structured flow control (`if/else`) is used.
- **Rule 5-0-3**: Error handling for failed video capture and empty frames.
- **Rule 5-0-4**: No global variables; all variables are scoped locally to functions.
- **Rule 5-0-5**: Functions perform specific tasks (e.g., detection, bounding box drawing) with no side effects.
- **Rule 5-0-6**: Ensure proper initialization and handling of variables (e.g., the `objects` vector).

Object Detection

Program 4:

Objective:

To implement an **Object Detection System** using **Template Matching** from **OpenCV** to detect a specific object (template) in a video stream. The system will detect the object's position in the frame and draw a bounding box around it.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation and use **local variables**.
- **Error Handling**: Proper checks for invalid data, such as empty frames or failed image loading.
- **No Global Variables**: Ensure variables are scoped locally to prevent unintended side effects.
- **Type Safety**: Use appropriate data types, particularly when dealing with **OpenCV structures** (e.g., `cv::Mat`).
- **Function Purity**: Each function should perform a specific task and avoid side effects.
- **No `goto` Statements**: Use structured control flow (e.g., `if/else`).

C++ Code for Object Detection Using Template Matching:

```
#include <iostream>
#include <opencv2/opencv.hpp>

// Define a structure to hold detected object data
struct DetectedObject {
   int x, y, width, height;  // Coordinates and dimensions of the bounding box
   std::string class_id;      // Class ID or name (e.g., "Template Object")
};

// Function to perform template matching to detect a specific object
void detectTemplate(const cv::Mat &frame, const cv::Mat &templateImage, std::vector<DetectedObject>
&objects)
{
   // Perform template matching
```

```
    cv::Mat result;
    cv::matchTemplate(frame, templateImage, result, cv::TM_CCOEFF_NORMED);

    // Set a threshold for object detection confidence
    const float threshold = 0.8f; // Adjust based on application requirements
    cv::Mat mask;
    cv::threshold(result, mask, threshold, 1.0, cv::THRESH_BINARY);

    // Find the locations of the detected objects
    std::vector<cv::Point> locations;
    cv::findNonZero(mask, locations);

    // Store detected objects data (position and size)
    for (const auto &location : locations) {
        DetectedObject object;
        object.x = location.x;
        object.y = location.y;
        object.width = templateImage.cols;
        object.height = templateImage.rows;
        object.class_id = "Template Object";  // Template name or ID

        objects.push_back(object);  // Store detected object
    }
}

// Function to draw bounding boxes around detected objects
void drawBoundingBoxes(cv::Mat &frame, const std::vector<DetectedObject> &objects)
{
    for (const auto &object : objects) {
        // Draw a rectangle around the detected object
        cv::rectangle(frame, cv::Point(object.x, object.y),
                      cv::Point(object.x + object.width, object.y + object.height),
                      cv::Scalar(0, 255, 0), 2);  // Green color for bounding box
        // Optionally, label the object with its class ID
        cv::putText(frame, object.class_id, cv::Point(object.x, object.y - 10),
                    cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(0, 255, 0), 2);
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Load the template image (object to be detected)
    cv::Mat templateImage = cv::imread("template.jpg", cv::IMREAD_GRAYSCALE);
    if (templateImage.empty()) {
        std::cerr << "Error: Could not load template image!" << std::endl;
        return 1;  // Exit if template image can't be loaded
    }
    // Capture video from the camera
    cv::VideoCapture cap(0);  // Open the default camera
    if (!cap.isOpened()) {
        std::cerr << "Error: Could not open video feed!" << std::endl;
        return 1;  // Exit if video feed can't be opened
    }

    cv::Mat frame;
    while (true) {
        cap >> frame;
        if (frame.empty()) {
            break;  // Exit if the frame is empty
        }

        // Convert the frame to grayscale for template matching
        cv::Mat grayFrame;
        cv::cvtColor(frame, grayFrame, cv::COLOR_BGR2GRAY);

        // Detect the object using template matching
        std::vector<DetectedObject> objects;
        detectTemplate(grayFrame, templateImage, objects);

        // Draw bounding boxes around detected objects
        drawBoundingBoxes(frame, objects);

        // Show the processed frame
        cv::imshow("Template Matching", frame);

        // Exit condition: press 'q' to quit
        if (cv::waitKey(1) & 0xFF == 'q') {
            break;
        }
    }

    cap.release();  // Release the video capture object
    cv::destroyAllWindows();  // Close all OpenCV windows

    return 0;
}
```

Explanation:

1. **Template Matching**:
   - Template matching works by sliding a template image across the input image and comparing the similarity between the template and the image patch at each position.
   - **OpenCV's `matchTemplate` function** computes the similarity score (in this case using `cv::TM_CCOEFF_NORMED`), and a threshold is applied to identify potential matches. In this example, a threshold of `0.8f` is used, meaning the system will only detect the object if the matching score is above 80% similarity.
2. **Bounding Box Drawing**:
   - After detecting the object, the function `drawBoundingBoxes` draws a **green rectangle** around the detected object using `cv::rectangle` and labels it with the object's class name (e.g., "Template Object").
3. **Main Logic**:
   - The `main` function continuously captures frames from the default camera, processes the frames using **template matching**, and displays the results with bounding boxes drawn around the detected objects. The user can exit the program by pressing the **'q'** key.
4. **MISRA C++:2023 Compliance**:
   - **No Dynamic Memory Allocation**: The program uses **local variables** and avoids dynamic memory allocation.
   - **No `goto` Statements**: The program uses **structured control flow** (`if/else`) instead of `goto`.
   - **Error Handling**: The program checks if the **template image** and **video feed** are loaded correctly.
   - **No Global Variables**: All variables are scoped **locally** within functions.
   - **Type Safety**: Correct data types are used for OpenCV structures (e.g., `cv::Mat`, `cv::KeyPoint`).
   - **Function Purity**: Each function performs a specific task with no side effects.

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation. The program uses **local variables** and avoids dynamic memory allocation.
- **Rule 5-3-1**: Proper handling of types, avoiding unsafe type conversions (e.g., handling `cv::Mat`, `cv::Rect`).
- **Rule 5-0-2**: No use of `goto` statements. The program uses **structured flow control** (`if/else`).
- **Rule 5-0-3**: Error handling for failed template image loading and empty frames.
- **Rule 5-0-4**: No global variables; all variables are scoped locally to functions.
- **Rule 5-0-5**: Functions perform specific tasks (e.g., template matching, bounding box drawing) without side effects.
- **Rule 5-0-6**: Proper initialization and handling of variables (e.g., `objects` vector).

FORWARD COLLISION WARNING (FCW) WITHOUT OBJECT DETECTION PROGRAM:

PROGRAM 1:

Objective:

To develop a **Forward Collision Warning (FCW)** system that calculates the **Time-to-Collision (TTC)** using the distance and relative velocity of detected objects. The system triggers a warning if the TTC is below a defined threshold, indicating a potential collision.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation and use **local variables**.
- **Error Handling**: Ensure proper error checks, especially for invalid velocity or distance values.
- **No Global Variables**: Ensure variables are scoped locally to avoid unintended side effects.
- **Type Safety**: Proper data types should be used for distance, velocity, and other calculations.
- **Function Purity**: Functions should be simple and perform one specific task (e.g., TTC calculation, collision warning).
- **No `goto` Statements**: Use structured control flow (if/else).

C++ Code for FCW Logic:

```
#include <iostream>
#include <cmath>

// Define a structure to hold detected object data
struct DetectedObject {
    float distance;      // Distance to the detected object (in meters)
    float velocity;      // Relative velocity (in m/s)
};

// Function to calculate Time-to-Collision (TTC) based on distance and velocity
float calculateTTC(const DetectedObject &object)
{
    // Validate inputs to avoid division by zero
    if (object.velocity == 0.0f) {
        std::cerr << "Error: Invalid velocity value!" << std::endl;
        return -1.0f;  // Return invalid TTC if velocity is zero
    }

    // Calculate Time-to-Collision (TTC) as distance divided by velocity
    return object.distance / object.velocity;
}

// Function to trigger collision warning based on TTC
void triggerCollisionWarning(float ttc)
{
    const float threshold = 2.0f;  // Threshold for collision warning (2 seconds)
    if (ttc < threshold) {
        std::cout << "Warning: Collision Imminent! TTC: " << ttc << " seconds" << std::endl;
        // Additional alert logic (audio, visual, etc.) can be triggered here
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Simulate a detected object and ego vehicle data
    DetectedObject detectedObject;
    detectedObject.distance = 50.0f;  // Distance to object (in meters)
    detectedObject.velocity = 15.0f;  // Relative velocity (in m/s)

    // Calculate Time-to-Collision (TTC)
    float ttc = calculateTTC(detectedObject);
    if (ttc < 0) {
        return 1;  // Exit if TTC calculation is invalid
    }

    // Trigger collision warning if TTC is below the threshold
    triggerCollisionWarning(ttc);

    return 0;
}
```

Explanation:

1. **TTC Calculation**:
    - The **Time-to-Collision (TTC)** is calculated by dividing the **distance** to the detected object by its **relative velocity**. This gives an estimate of how long it will take for the vehicle to collide with the detected object.
    - If the velocity is zero (i.e., the object is stationary), the function returns

an error to avoid division by zero.

2. **Collision Warning**:
    - The `triggerCollisionWarning` function checks if the **TTC** is below a predefined threshold (in this case, 2 seconds). If it is, the system triggers a **collision warning** indicating that a collision is imminent.
    - In this simple example, the warning is printed to the console, but in a real-world application, it could trigger audio or visual alerts.
3. **Main Logic**:
    - In the `main` function, the system simulates a detected object with a **distance** of 50 meters and a **velocity** of 15 m/s.
    - It then calculates the TTC and triggers the collision warning if necessary.
4. **MISRA C++:2023 Compliance**:
    - **No Dynamic Memory Allocation**: The program uses **local variables** and avoids dynamic memory allocation.
    - **No `goto` Statements**: The program uses **structured control flow** (e.g., `if/else`).
    - **Error Handling**: The program checks for **invalid velocity** values to prevent division by zero.
    - **No Global Variables**: All variables are **locally scoped** within functions.
    - **Type Safety**: Proper data types are used (e.g., `float` for distance and velocity) for calculations.

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation (e.g., all variables are locally scoped and stack-based).
- **Rule 5-3-1**: Proper handling of types and avoiding unsafe type conversions (e.g., using `float` for distance and velocity).
- **Rule 5-0-2**: No use of `goto` statements, structured flow control (`if/else`).
- **Rule 5-0-3**: Clear error handling for invalid velocity values to avoid division by zero.
- **Rule 5-0-4**: No global variables; all variables are scoped locally to functions.
- **Rule 5-0-5**: Functions are kept simple, each performing a specific task (e.g., TTC calculation, collision warning).
- **Rule 5-0-6**: Proper initialization and handling of variables (e.g., `detectedObject`).

FORWARD COLLISION WARNING (FCW) WITHOUT OBJECT DETECTION PROGRAM:

PROGRAM 2:

Objective:

To implement a **Forward Collision Warning (FCW)** system that calculates the **Time-to-Impact (TTI)** based on the **relative velocity** and **distance** between the vehicle and an obstacle ahead. If the TTI is below a certain threshold, the system will trigger a warning to alert the driver about a potential collision.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation and use **local variables** where possible.
- **Error Handling**: Proper checks to handle invalid inputs such as zero velocity or negative distances.
- **No Global Variables**: Ensure variables are locally scoped to prevent unintended side effects.
- **Type Safety**: Use appropriate data types and ensure safe type conversions.
- **Function Purity**: Each function should perform a specific task, ensuring no side effects.
- **No `goto` Statements**: Use structured control flow (e.g., `if/else`).

C++ Code for FCW with Time-to-Impact (TTI):

```
#include <iostream>
#include <cmath>

// Define a structure to hold detected object data
struct DetectedObject {
    float distance;      // Distance to the detected object (in meters)
    float relativeSpeed;  // Relative velocity (in m/s)
};

// Function to calculate Time-to-Impact (TTI) based on distance and relative speed
float calculateTTI(const DetectedObject &object)
{
    // Validate inputs to avoid division by zero
    if (object.relativeSpeed == 0.0f) {
        std::cerr << "Error: Invalid relative speed!" << std::endl;
        return -1.0f;  // Return invalid TTI if speed is zero
    }

    // Calculate Time-to-Impact (TTI) as distance divided by relative speed
    return object.distance / object.relativeSpeed;
}

// Function to trigger collision warning based on TTI
void triggerCollisionWarning(float tti)
{
    const float threshold = 1.5f;  // Threshold for collision warning (1.5 seconds)
    if (tti < threshold) {
        std::cout << "Warning: Collision Imminent! TTI: " << tti << " seconds" << std::endl;
        // Additional alert logic (audio, visual, etc.) can be triggered here
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Simulate a detected object and ego vehicle data
    DetectedObject detectedObject;
    detectedObject.distance = 30.0f;      // Distance to object (in meters)
    detectedObject.relativeSpeed = 20.0f; // Relative speed (in m/s)

    // Calculate Time-to-Impact (TTI)
    float tti = calculateTTI(detectedObject);
    if (tti < 0) {
        return 1;  // Exit if TTI calculation is invalid
    }

    // Trigger collision warning if TTI is below the threshold
    triggerCollisionWarning(tti);

    return 0;
}
```

Explanation:

1. **TTI Calculation**:
   - **Time-to-Impact (TTI)** is calculated using the formula:
2. TTI=DistanceRelative Speed\text{TTI} = \frac{\text{Distance}}{\text{Relative Speed}}TTI=Relative SpeedDistance
   - The function `calculateTTI` calculates the time it will take for the vehicle to collide with the detected object. If the **relative speed** is zero (meaning the object is stationary or the vehicle is moving at the same speed), the function returns an invalid TTI to prevent division by zero.
3. **Collision Warning**:
   - The `triggerCollisionWarning` function compares the **TTI** to a threshold value (1.5 seconds in this case). If the TTI is below the threshold, indicating an imminent collision, the system triggers a warning (printed to the console in this example).
   - In a real-world application, the warning could be extended to include audio or visual alerts.
4. **Main Logic**:
   - In the `main` function, the system simulates data for a **detected object** at a distance of 30 meters with a relative speed of 20 m/s.
   - The **TTI** is calculated, and the system triggers a warning if the TTC is below the threshold.
5. **MISRA C++:2023 Compliance**:
   - **No Dynamic Memory Allocation**: The program uses **local variables** and avoids dynamic memory allocation.
   - **No `goto` Statements**: The program uses **structured control flow** (e.g., `if/else`).
   - **Error Handling**: The program checks for **invalid relative speed** values to prevent division by zero.
   - **No Global Variables**: All variables are **locally scoped** within functions.
   - **Type Safety**: Proper types are used (e.g., `float` for distance and relative speed) for calculations.
   - **Function Purity**: Each function performs a specific task (e.g., TTI calculation, collision warning).

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation. All variables are stack-based.
- **Rule 5-3-1**: Proper handling of types and avoiding unsafe type conversions (e.g., using `float` for distance and speed).
- **Rule 5-0-2**: No use of `goto` statements, ensuring structured control flow (e.g., `if/else`).
- **Rule 5-0-3**: Clear error handling for invalid velocity values to avoid division by zero.
- **Rule 5-0-4**: No global variables used. All variables are scoped locally to functions.
- **Rule 5-0-5**: Functions are kept simple and perform a single task with no side effects (e.g., TTI calculation, collision warning).
- **Rule 5-0-6**: Proper initialization and handling of variables (e.g., `detectedObject`).

FORWARD COLLISION WARNING (FCW) WITHOUT OBJECT DETECTION PROGRAM:

PROGRAM 3:

Objective:

To implement an **FCW system** that calculates the **braking distance** required to stop the vehicle and compares it to the distance to an object. If the distance to the object is less than or equal to the braking distance, the system will trigger a collision warning.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation and use **local variables**.
- **Error Handling**: Ensure proper checks for invalid data, such as negative or zero values for speed or distance.
- **No Global Variables**: Ensure variables are scoped locally to avoid unintended side effects.
- **Type Safety**: Proper data types should be used for distance, speed, and other calculations.
- **Function Purity**: Functions should perform a specific task with no side effects.
- **No `goto` Statements**: Use structured control flow (e.g., `if/else`).

C++ Code for FCW with Braking Distance Model:

```
#include <iostream>
#include <cmath>

// Define a structure to hold detected object data
struct DetectedObject {
    float distance;      // Distance to the detected object (in meters)
    float speed;         // Vehicle speed (in m/s)
};

// Function to calculate braking distance based on vehicle speed
float calculateBrakingDistance(float speed)
{
    // Simple model for braking distance: d = v^2 / (2 * a)
    // where v is the velocity (speed) and a is the braking deceleration (assumed constant)
    const float brakingDeceleration = 8.0f;  // m/s^2 (typical deceleration value for emergency braking)

    return (speed * speed) / (2 * brakingDeceleration);
}

// Function to trigger collision warning based on braking distance
void triggerCollisionWarning(float brakingDistance, float objectDistance)
```

```
{
    if (objectDistance <= brakingDistance) {
        std::cout << "Warning: Collision Imminent! Braking distance: "
                  << brakingDistance << " meters, Object distance: "
                  << objectDistance << " meters" << std::endl;
        // Additional alert logic (audio, visual, etc.) can be triggered here
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Simulate a detected object and vehicle data
    DetectedObject detectedObject;
    detectedObject.distance = 50.0f; // Distance to object (in meters)
    detectedObject.speed = 20.0f;    // Vehicle speed (in m/s)

    // Calculate the braking distance required to stop the vehicle
    float brakingDistance = calculateBrakingDistance(detectedObject.speed);

    // Trigger collision warning if the object is within the braking distance
    triggerCollisionWarning(brakingDistance, detectedObject.distance);

    return 0;
}
```

Explanation:

1. **Braking Distance Calculation**:
   - The `calculateBrakingDistance` function computes the **braking distance** required for the vehicle to come to a complete stop based on its speed.
   - The formula used is: Braking Distance=v22⋅a\text{Braking Distance} = \frac{v^2}{2 \cdot a}Braking Distance=2⋅av2 where vvv is the vehicle speed and aaa is the **braking deceleration**. In this example, the deceleration is assumed to be a constant 8 m/s28 \, \text{m/s}^28m/s2, which is a typical value for emergency braking.
2. **Collision Warning**:
   - The `triggerCollisionWarning` function compares the **braking distance** with the distance to the detected object.
   - If the **distance to the object** is **less than or equal** to the braking distance, the system triggers a **collision warning**, indicating that the vehicle will not be able to stop in time to avoid a collision.
3. **Main Logic**:
   - The `main` function simulates the distance to a detected object and the vehicle's speed. It then calculates the required braking distance and triggers a collision warning if necessary.
4. **MISRA C++:2023 Compliance**:
   - **No Dynamic Memory Allocation**: The program uses **local variables** and avoids dynamic memory allocation.
   - **No `goto` Statements**: The program uses **structured control flow** (e.g., `if/else`).

    - **Error Handling**: The program assumes positive values for speed and distance. It checks the distance in comparison to the braking distance to ensure safety.
    - **No Global Variables**: All variables are **locally scoped** within functions.
    - **Type Safety**: Proper types are used for **distance** and **speed** calculations.
    - **Function Purity**: Each function performs a specific task (e.g., braking distance calculation, collision warning).

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation. The program uses **local variables**.
- **Rule 5-3-1**: Proper handling of types, ensuring safe type conversions (e.g., `float` for distance and speed).
- **Rule 5-0-2**: No use of `goto` statements. Structured control flow is used (e.g., `if/else`).
- **Rule 5-0-3**: Error handling for invalid values, ensuring that only positive values for distance and speed are used.
- **Rule 5-0-4**: No global variables are used; all variables are scoped locally.
- **Rule 5-0-5**: Functions are simple and perform a specific task (e.g., braking distance calculation, collision warning).
- **Rule 5-0-6**: Proper initialization and handling of variables (e.g., `detectedObject`).

FORWARD COLLISION WARNING (FCW) WITHOUT OBJECT DETECTION PROGRAM:

PROGRAM 4:

Objective:

To implement a **Forward Collision Warning (FCW)** system that calculates the **Time-to-Impact (TTI)**, considering the vehicle's current speed and deceleration. If the TTI is below a certain threshold, the system will trigger a warning, alerting the driver of an imminent collision.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation and use **local variables** wherever possible.
- **Error Handling**: Ensure checks for invalid speed or distance values (e.g., negative numbers).
- **No Global Variables**: Ensure that variables are scoped locally to prevent unintended side effects.
- **Type Safety**: Use appropriate data types for **distance**, **speed**, and **deceleration**.
- **Function Purity**: Functions should perform specific tasks, such as calculating TTI or triggering a warning.
- **No `goto` Statements**: Use structured control flow (e.g., `if/else`).

C++ Code for FCW with Look-Ahead Model (TTI):

```
#include <iostream>
#include <cmath>

// Define a structure to hold detected object data
struct DetectedObject {
    float distance;      // Distance to the detected object (in meters)
    float speed;         // Vehicle speed (in m/s)
};

// Function to calculate Time-to-Impact (TTI) based on speed and deceleration
float calculateTTI(const DetectedObject &object, float deceleration)
{
    // Validate inputs to avoid division by zero or negative values
    if (object.speed <= 0.0f || object.distance <= 0.0f || deceleration <= 0.0f) {
        std::cerr << "Error: Invalid values for speed, distance, or deceleration!" << std::endl;
        return -1.0f;  // Return invalid TTI if values are invalid
    }

    // Use the formula: TTI = distance / (speed - deceleration * time)
    // This formula assumes constant deceleration
    float time = object.speed / deceleration;  // Time required to stop
    float predictedDistance = object.speed * time - 0.5f * deceleration * time * time;

    // Calculate Time-to-Impact (TTI)
    return object.distance / predictedDistance;
}

// Function to trigger collision warning based on TTI
void triggerCollisionWarning(float tti)
{
    const float threshold = 2.0f;  // Threshold for collision warning (2 seconds)
    if (tti < threshold) {
        std::cout << "Warning: Collision Imminent! TTI: " << tti << " seconds" << std::endl;
        // Additional alert logic (audio, visual, etc.) can be triggered here
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Simulate a detected object and vehicle data
    DetectedObject detectedObject;
    detectedObject.distance = 60.0f;    // Distance to object (in meters)
    detectedObject.speed = 25.0f;       // Vehicle speed (in m/s)
    float deceleration = 5.0f;          // Assumed deceleration (in m/s^2)
    // Calculate Time-to-Impact (TTI)
    float tti = calculateTTI(detectedObject, deceleration);
    if (tti < 0) {
        return 1;  // Exit if TTI calculation is invalid
    }

    // Trigger collision warning if TTI is below the threshold
    triggerCollisionWarning(tti);

    return 0;
}
```

Explanation:

1. **Look-Ahead Model for TTI Calculation**:
   - The system calculates the **Time-to-Impact (TTI)** using the vehicle's **current speed** and the **distance** to the detected object. The formula for TTI assumes that the vehicle will decelerate at a constant rate, and it calculates how long it will take for the vehicle to come to a stop and collide with the object.
   - The formula is: TTI=distancespeed−(deceleration×time)\text{TTI} = \frac{\text{distance}}{\text{speed} - (\text{deceleration} \times \text{time})}TTI=speed−(deceleration×time)distance
   - The **deceleration** is assumed to be constant. If the deceleration is high, the vehicle will stop sooner, reducing the TTI. If the deceleration is low, the TTI increases.
2. **Collision Warning**:
   - The `triggerCollisionWarning` function compares the **TTI** with a threshold value (e.g., 2 seconds). If the TTI is below the threshold, indicating that the vehicle will collide with the object in less time than the threshold, a **collision warning** is triggered (printed to the console).
3. **Main Logic**:
   - In the `main` function, the system simulates a detected object at a distance of **60 meters** and a vehicle speed of **25 m/s**. A constant **deceleration** of **5 m/s^2** is assumed for the braking system.
   - The **TTI** is calculated and compared to the threshold to trigger the collision warning if necessary.
4. **MISRA C++:2023 Compliance**:
   - **No Dynamic Memory Allocation**: The program uses **local variables** and avoids dynamic memory allocation.
   - **No `goto` Statements**: The program uses **structured flow control** (e.g., `if/else`).
   - **Error Handling**: The program checks for **invalid values** (e.g., zero or negative speed, distance, or deceleration).
   - **No Global Variables**: All variables are **locally scoped** within functions.
   - **Type Safety**: Proper types (e.g., `float`) are used for distance, speed, and deceleration calculations.
   - **Function Purity**: Each function performs a single, specific task (e.g., TTI calculation, collision warning).

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation (e.g., local variables and stack-based storage).
- **Rule 5-3-1**: Proper handling of types, ensuring safe type conversions (e.g., `float` for calculations).
- **Rule 5-0-2**: No use of `goto` statements; structured control flow is used.
- **Rule 5-0-3**: Clear error handling for invalid velocity, distance, or deceleration values.
- **Rule 5-0-4**: No global variables are used; all variables are scoped locally to functions.
- **Rule 5-0-5**: Functions perform specific tasks (e.g., TTI calculation, collision warning).
- **Rule 5-0-6**: Proper initialization and handling of variables (e.g., `detectedObject`).

FORWARD COLLISION WARNING (FCW) WITHOUT OBJECT DETECTION PROGRAM:

PROGRAM 5:

Objective:

To implement an **FCW system** that calculates the **Time-to-Collision (TTC)** using the **relative acceleration** between the vehicle and the detected object. If the TTC is below a certain threshold, the system will trigger a warning, indicating an imminent collision.

MISRA C++:2023 Compliance Considerations:

- **Memory Management**: Avoid dynamic memory allocation and use **local variables**.
- **Error Handling**: Proper checks for invalid speed, distance, or acceleration values (e.g., negative values).
- **No Global Variables**: Ensure variables are scoped locally to prevent unintended side effects.
- **Type Safety**: Use appropriate data types for distance, speed, acceleration, and time calculations.
- **Function Purity**: Each function should perform a specific task with no side effects.
- **No `goto` Statements**: Use structured control flow (e.g., `if/else`).

C++ Code for FCW with Relative Acceleration Model:

```
#include <iostream>
#include <cmath>

// Define a structure to hold detected object data
struct DetectedObject {
   float distance;      // Distance to the detected object (in meters)
   float speed;         // Vehicle speed (in m/s)
   float acceleration;   // Relative acceleration (in m/s^2)
};

// Function to calculate Time-to-Collision (TTC) based on distance, speed, and relative acceleration
```

```
float calculateTTC(const DetectedObject &object)
{
    // Validate inputs to avoid division by zero or negative values
    if (object.speed <= 0.0f || object.distance <= 0.0f || object.acceleration <= 0.0f) {
        std::cerr << "Error: Invalid values for speed, distance, or acceleration!" << std::endl;
        return -1.0f;  // Return invalid TTC if any values are invalid
    }

    // Use the formula: TTC = (sqrt(v^2 + 2*a*d) - v) / a
    // Where:
    // v = current speed (m/s)
    // a = relative acceleration (m/s^2)
    // d = distance to the object (meters)
    float discriminant = (object.speed * object.speed) + 2 * object.acceleration * object.distance;

    if (discriminant < 0) {
        std::cerr << "Error: Negative discriminant in TTC calculation!" << std::endl;
        return -1.0f;  // Return invalid TTC if discriminant is negative (no real solution)
    }

    // Calculate the time to collision (TTC)
    float ttc = (std::sqrt(discriminant) - object.speed) / object.acceleration;
    return ttc;
}

// Function to trigger collision warning based on TTC
void triggerCollisionWarning(float ttc)
{
    const float threshold = 2.0f;  // Threshold for collision warning (2 seconds)
    if (ttc < threshold) {
        std::cout << "Warning: Collision Imminent! TTC: " << ttc << " seconds" << std::endl;
        // Additional alert logic (audio, visual, etc.) can be triggered here
    }
}

int main()
{
    // MISRA C++:2023 Rule 5-0-4 (No global variables)
    // Simulate a detected object and vehicle data
    DetectedObject detectedObject;
    detectedObject.distance = 40.0f;       // Distance to object (in meters)
    detectedObject.speed = 25.0f;          // Vehicle speed (in m/s)
    detectedObject.acceleration = 1.5f;    // Relative acceleration (in m/s^2)

    // Calculate Time-to-Collision (TTC)
    float ttc = calculateTTC(detectedObject);
    if (ttc < 0) {
        return 1;  // Exit if TTC calculation is invalid
    }
    // Trigger collision warning if TTC is below the threshold
    triggerCollisionWarning(ttc);

    return 0;
}
```

Explanation:

1. **Relative Acceleration Model**:
   - The **Time-to-Collision (TTC)** is calculated based on the vehicle's **current speed**, the **distance** to the detected object, and the **relative acceleration** (how fast the object is either getting closer or moving away).
   - The formula used for TTC in the presence of **relative acceleration** is: TTC=v2+2⋅a⋅d−va\text{TTC} = \frac{\sqrt{v^2 + 2 \cdot a \cdot d} - v}{a}TTC=av2+2⋅a⋅d−v Where:
     - vvv = vehicle's current speed
     - aaa = relative acceleration between the vehicle and the object
     - ddd = distance to the object
2. **Error Handling**:
   - The `calculateTTC` function checks if any values are invalid (e.g., negative speed, distance, or acceleration). If they are, it returns an invalid TTC.
   - It also checks for the **discriminant** in the quadratic formula. If the discriminant is negative, the function returns an error as there is no real solution for TTC (which may happen if the object is moving away from the vehicle).
3. **Collision Warning**:
   - The `triggerCollisionWarning` function compares the calculated **TTC** with a threshold value (e.g., 2 seconds). If the TTC is less than the threshold, a **collision warning** is triggered (printed to the console). In a real system, this could be replaced with an audio or visual alert.
4. **Main Logic**:
   - The `main` function simulates data for a **detected object** at a distance of **40 meters** with a speed of **25 m/s** and a relative acceleration of **1.5 m/s^2**.
   - The **TTC** is calculated using the relative acceleration, and a warning is triggered if the TTC is below the threshold.
5. **MISRA C++:2023 Compliance**:
   - **No Dynamic Memory Allocation**: The program uses **local variables** and avoids dynamic memory allocation.
   - **No `goto` Statements**: The program uses **structured control flow** (e.g., `if/else`).
   - **Error Handling**: The program checks for invalid speed, distance, or acceleration values to avoid erroneous calculations.
   - **No Global Variables**: All variables are **locally scoped** within functions.
   - **Type Safety**: Proper types (e.g., `float`) are used for distance, speed, and acceleration calculations.
   - **Function Purity**: Each function performs a single task (e.g., TTC calculation, collision warning).

Key MISRA C++:2023 Rules Addressed:

- **Rule 5-0-1**: No dynamic memory allocation (e.g., all variables are stack-based).
- **Rule 5-3-1**: Proper handling of types, ensuring safe type conversions (e.g., using `float` for calculations).
- **Rule 5-0-2**: No use of `goto` statements. Structured control flow is used (e.g., `if/else`).
- **Rule 5-0-3**: Error handling for invalid speed, distance, or acceleration values.
- **Rule 5-0-4**: No global variables are used; all variables are scoped locally.
- **Rule 5-0-5**: Functions perform specific tasks (e.g., TTC calculation, collision warning).
- **Rule 5-0-6**: Proper initialization and handling of variables (e.g., `detectedObject`).

Test Cases for Forward Collision Warning (FCW) System

Test Case 1: Normal Speed and Distance

Objective:

To verify that the FCW system correctly calculates the braking distance and does not trigger a collision warning when the object is at a safe distance.

Test Case ID: FCW_TC_001

Test Input:

- **Vehicle Speed**: 25 m/s (90 km/h)
- **Distance to Object**: 50 meters
- **Deceleration**: 8 m/s^2 (typical emergency braking deceleration)

Expected Output:

- **Braking Distance**: The system calculates the braking distance as less than 50 meters.
- **Collision Warning**: No collision warning is triggered because the object is farther than the braking distance.

Test Procedure:

1. Simulate a detected object with a distance of 50 meters.
2. Set the vehicle's speed to 25 m/s and use the standard deceleration of 8 m/s^2.
3. Calculate the braking distance using the braking distance formula.
4. Compare the calculated braking distance to the object distance.

5. Verify that no collision warning is triggered when the object distance is greater than the braking distance.

Example Code:

```
void testBrakingDistanceNormalConditions()
{
   DetectedObject object = {50.0f, 25.0f, 8.0f};  // distance = 50 meters, speed = 25 m/s, deceleration = 8 m/s²
   float brakingDistance = calculateBrakingDistance(object.speed);  // Calculate braking distance

   std::cout << "Calculated Braking Distance: " << brakingDistance << " meters" << std::endl;

   // Expect no warning if the object distance is greater than braking distance
   if (object.distance > brakingDistance) {
      std::cout << "No collision warning triggered. Test Passed!" << std::endl;
   } else {
      std::cout << "Collision warning triggered unexpectedly. Test Failed!" << std::endl;
   }
}
```

Test Case 2: Short Distance, High Speed

Objective:

To verify that the FCW system correctly calculates the braking distance and triggers a collision warning when the vehicle is traveling at high speed with insufficient distance to stop.

Test Case ID: FCW_TC_002

Test Input:

- **Vehicle Speed**: 35 m/s (126 km/h)
- **Distance to Object**: 30 meters
- **Deceleration**: 8 m/s^2

Expected Output:

- **Braking Distance**: The system calculates the braking distance as greater than 30 meters.
- **Collision Warning**: The system triggers a collision warning since the object is closer than the required braking distance.

Test Procedure:

1. Simulate a detected object with a distance of 30 meters.
2. Set the vehicle's speed to 35 m/s and use the standard deceleration of 8 m/s^2.
3. Calculate the braking distance using the braking distance formula.
4. Compare the calculated braking distance to the object distance.
5. Verify that a collision warning is triggered since the object is within the braking distance.

Example Code:

```
void testBrakingDistanceShortDistanceHighSpeed()
{
   DetectedObject object = {30.0f, 35.0f, 8.0f};  // distance = 30 meters, speed = 35 m/s, deceleration = 8
m/s²
   float brakingDistance = calculateBrakingDistance(object.speed);  // Calculate braking distance

   std::cout << "Calculated Braking Distance: " << brakingDistance << " meters" << std::endl;

   // Expect collision warning since the object distance is less than braking distance
   if (object.distance <= brakingDistance) {
      std::cout << "Collision warning triggered. Test Passed!" << std::endl;
   } else {
      std::cout << "No warning triggered. Test Failed!" << std::endl;
   }
}
```

Test Case 3: Zero Velocity

Objective:

To verify that the FCW system correctly handles the case where the vehicle is stationary and no collision warning is triggered, even if an object is nearby.

Test Case ID: FCW_TC_003

Test Input:

- **Vehicle Speed**: 0 m/s (vehicle is stationary)
- **Distance to Object**: 20 meters
- **Deceleration**: 8 m/s^2

Expected Output:

- **Braking Distance**: The braking distance is irrelevant since the vehicle is stationary.

- **Collision Warning**: No collision warning is triggered, as the vehicle is not moving.

Test Procedure:

1. Simulate a stationary vehicle (speed = 0 m/s) and an object at a distance of 20 meters.
2. Use a deceleration value of 8 m/s^2.
3. Calculate the braking distance, which should be irrelevant in this case.
4. Verify that no collision warning is triggered since the vehicle is stationary.

Example Code:

```
void testBrakingDistanceZeroVelocity()
{
   DetectedObject object = {20.0f, 0.0f, 8.0f};  // distance = 20 meters, speed = 0 m/s, deceleration = 8
m/s²
   float brakingDistance = calculateBrakingDistance(object.speed);  // Calculate braking distance

   std::cout << "Calculated Braking Distance: " << brakingDistance << " meters" << std::endl;

   // Expect no warning since the vehicle is stationary
   if (object.speed == 0.0f) {
      std::cout << "No collision warning triggered as expected. Test Passed!" << std::endl;
   } else {
      std::cout << "Unexpected collision warning. Test Failed!" << std::endl;
   }
}
```

Test Case 4: Negative Speed or Distance

Objective:

To verify that the FCW system handles invalid input values (e.g., negative speed or distance) properly and does not calculate incorrect or erroneous values.

Test Case ID: FCW_TC_004

Test Input:

- **Vehicle Speed**: -20 m/s (invalid speed)
- **Distance to Object**: 30 meters
- **Deceleration**: 8 m/s^2

Expected Output:

- **Error Handling**: The system should return an error or warning for invalid values and should not calculate a TTC or braking distance.

Test Procedure:

1. Simulate an invalid detected object with negative speed (-20 m/s).
2. Set the object distance to 30 meters and deceleration to 8 m/s^2.
3. Verify that the system identifies the invalid speed and returns an error message.

Example Code:

```
void testInvalidSpeed()
{
   DetectedObject object = {30.0f, -20.0f, 8.0f};  // distance = 30 meters, speed = -20 m/s, deceleration = 
8 m/s²
   float brakingDistance = calculateBrakingDistance(object.speed);  // Calculate braking distance

   std::cout << "Calculated Braking Distance: " << brakingDistance << " meters" << std::endl;

   // Expect an error due to invalid speed value
   if (object.speed < 0) {
      std::cout << "Error: Invalid speed value. Test Passed!" << std::endl;
   } else {
      std::cout << "Test Failed! Invalid speed was not handled correctly." << std::endl;
   }
}
```

Test Case 5: High-Speed and Emergency Braking

Objective:

To verify that the FCW system can handle high-speed conditions and emergency braking scenarios, ensuring the collision warning is triggered appropriately.

Test Case ID: FCW_TC_005

Test Input:

- **Vehicle Speed**: 50 m/s (180 km/h)
- **Distance to Object**: 100 meters
- **Deceleration**: 10 m/s^2 (emergency braking)

Expected Output:

- **Braking Distance**: The system calculates the braking distance, which will be calculated based on the high-speed input.
- **Collision Warning**: The system will trigger a collision warning if the distance

is less than the braking distance.

Test Procedure:

1. Simulate a high-speed scenario with a vehicle speed of 50 m/s and an object at 100 meters.
2. Use a deceleration of 10 m/s^2 to simulate emergency braking.
3. Calculate the braking distance and compare it to the object distance.
4. Verify that a collision warning is triggered.

Example Code:

```
void testHighSpeedEmergencyBraking()
{
   DetectedObject object = {100.0f, 50.0f, 10.0f};  // distance = 100 meters, speed = 50 m/s, deceleration
= 10 m/s²
   float brakingDistance = calculateBrakingDistance(object.speed);  // Calculate braking distance

   std::cout << "Calculated Braking Distance: " << brakingDistance << " meters" << std::endl;

   // Expect collision warning since the object distance is less than braking distance
   if (object.distance


<= brakingDistance) { std::cout << "Collision warning triggered. Test Passed!" << std::endl; }

 else

 {

 std::cout << "No warning triggered. Test Failed!" << std::endl; }

 }
```