# Enums

A white text with black text

Description automatically generated

Enumerated types, often referred to as enums, are a powerful feature in C++ that allow developers to define a new data type with a restricted set of values. Here's a comprehensive explanation of enums based on the provided text:

1. **Purpose of Enums**:
   * Enums are used to define a set of named integral constants, where each constant represents a specific value within the set.
   * They provide a way to create symbolic names for values, enhancing code readability and maintainability.
2. **Syntax**:
   * Enums are declared using the **enum** keyword followed by the name of the enum type and a list of enumerator names enclosed in curly braces.
   * Each enumerator name represents a constant value of the enum type.
   * By default, the value of the first enumerator is 0, and subsequent enumerators have values incremented by 1.
3. **Implicit Conversion to Integers**:
   * Enumerators are internally represented as integral values, typically **int**.
   * While enumerators can be implicitly converted to integers, the reverse is not true without explicit casting.
4. **Scope**:
   * Enumerators are visible only within the scope in which they are defined.
   * They can be accessed using the scope resolution operator (::) if declared within a namespace or class.
5. **Example Usage**:

A white background with black text

Description automatically generated

1. **Error Handling**:
   * Assigning an integer outside the defined range of enumerators to an enum variable will result in a compiler error.
   * Casting an integer to an enum type using **static\_cast** is possible but should be done carefully to avoid undefined behavior.

Overall, enums provide a convenient and type-safe way to work with a fixed set of related values, improving code clarity and robustness.

Enumerated types, commonly known as enums:

1. **Purpose of Enums**:
   * Enums are used to define a set of named integral constants, where each constant represents a specific value within the set.
   * They enhance code readability and maintainability by providing symbolic names for values.
2. **Basic Syntax**:
   * Enums are declared using the **enum** keyword followed by the name of the enum type and a list of enumerator names enclosed in curly braces.
   * Each enumerator name represents a constant value of the enum type.
   * By default, the value of the first enumerator is 0, and subsequent enumerators have values incremented by 1.
3. **Implicit Conversion to Integers**:
   * Enumerators are internally represented as integral values, typically **int**.
   * Enumerators can be implicitly converted to integers, but the reverse is not true without explicit casting.
4. **Scoped Enums**:
   * C++11 introduced scoped enums, where the enumerator names are scoped within the enum type itself.
   * Scoped enums prevent naming conflicts and provide better encapsulation.
   * Scoped enums are declared using the **enum class** syntax instead of just **enum**.
5. **Underlying Type**:
   * The underlying type of an enum is **int** by default, but it can be explicitly specified to any integral type using the colon syntax (**enum class MyEnum : underlying\_type**).
6. **Enum Initialization**:
   * Enumerators can be initialized with specific values, which don't have to be sequential.
   * Subsequent enumerators will have values incremented by 1 unless explicitly specified.
7. **Example Usage**:

A computer code with text and symbols

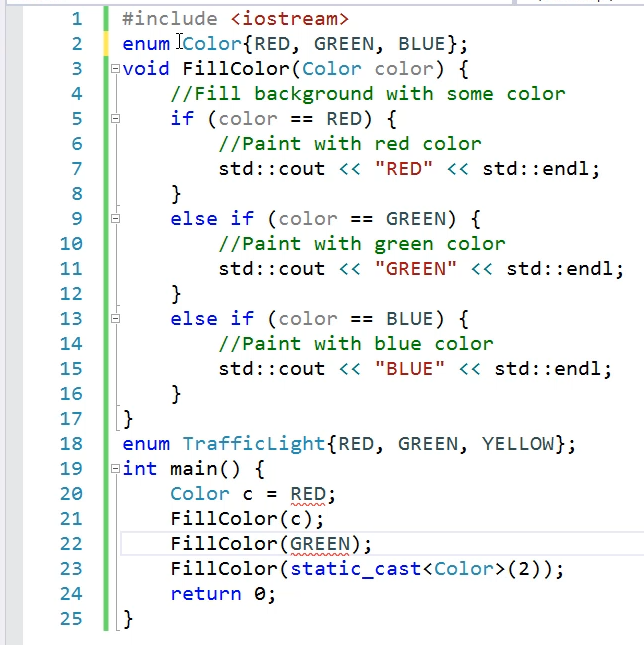
Description automatically generated

1. **Usage and Advantages**:
   * Enums restrict the range of values that can be assigned to variables, improving type safety.
   * Scoped enums provide better encapsulation and prevent naming conflicts.
   * Enums are preferred over raw integers for clarity and robustness in code.

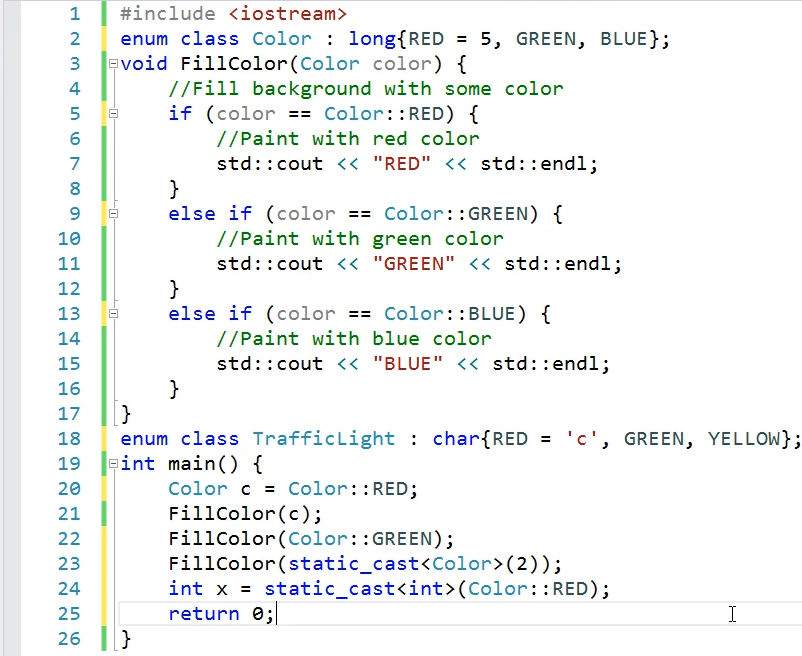
Overall, enums are a valuable tool in C++ for creating custom types with a defined set of values, improving code organization and reliability

# Normal enum use

Can not used redefine enum with same name in same scope so we can use in c++ scoped enum by using keyword class



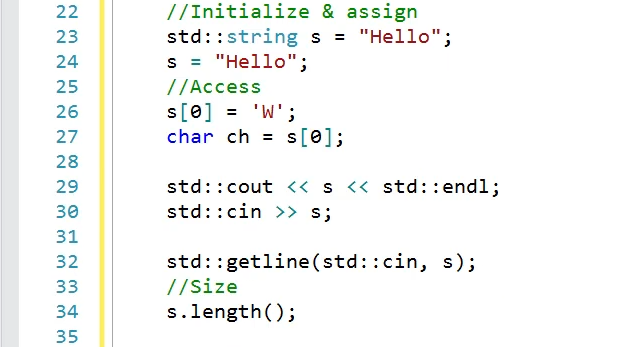
# Scoped enum



# Strings (Raw Strings & stdstring)

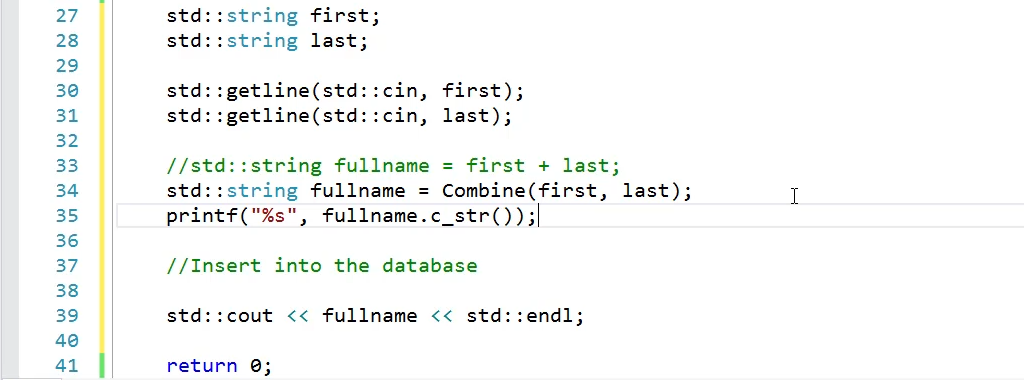
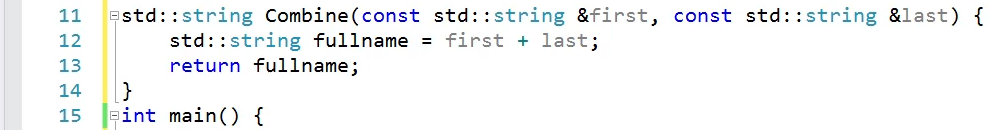
To use string features in c++ by include #include <string>

The string class provides a lot of constructors. It provide a default constructor, copy constructor, or constructor through which the string object can be initialized with a raw string and other constructors.

A screenshot of a computer program

Description automatically generated

A screen shot of a computer code

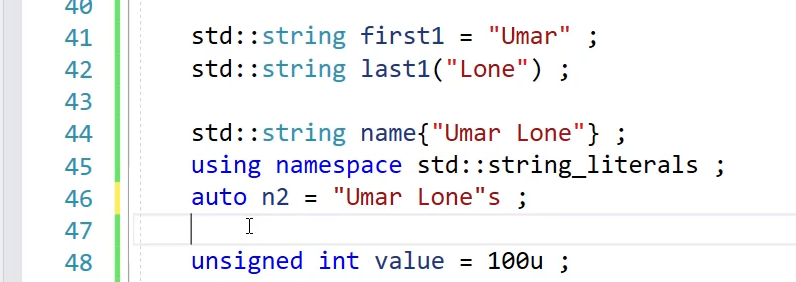
Description automatically generated 

In the provided example, C-style strings are used to read the first name and last name of a user and then combine them into a full name. Here's an overview of how C-style strings are utilized in the program, along with the associated issues and solutions:

1. **Declaration and Initialization**:
   * Character arrays are declared to store the first name (**firstName**) and last name (**lastName**).
   * The size of these arrays should be large enough to accommodate the expected maximum length of names.
2. **Reading Input**:
   * The **getline()** function is used to read the first name and last name from the user input and store them in the respective character arrays.
   * The size of the input buffer is specified to prevent buffer overflow.
3. **Combining Names**:
   * A third character array (**fullName**) is used to store the combined full name.
   * The **strcpy()** function is used to copy the first name into **fullName**.
   * The **strcat()** function is used to concatenate the last name onto **fullName**.
4. **Memory Management**:
   * Memory for the **fullName** array is allocated dynamically using the **new** operator to ensure that it can accommodate the combined full name.
   * The memory allocated for **fullName** should be deallocated using the **delete[]** operator after its use to prevent memory leaks.
5. **Null Termination**:
   * C-style strings are null-terminated, meaning they must be terminated by a null character (**'\0'**).
   * Memory allocation for **fullName** should include an additional byte to accommodate the null terminator.
6. **Error Handling**:
   * Failure to include a null terminator can lead to buffer overflow and undefined behavior.
   * The program may crash if memory corruption is detected during deallocation.
7. **Complexity and Error-Prone Nature**:
   * Working with C-style strings involves managing memory allocation, null terminators, buffer sizes, and manual deallocation.
   * Forgetting to handle these details correctly can lead to bugs, crashes, and memory leaks.
8. **Consideration of Alternatives**:
   * While C-style strings provide low-level control, they are error-prone and complex to work with.
   * C++ provides the **std::string** class, which simplifies string manipulation by handling memory management internally.
   * Using **std::string** eliminates the need for manual memory allocation and deallocation, making code safer and more robust.

In summary, while C-style strings offer control over memory management, they require careful handling to avoid errors and memory-related issues. In modern C++, using **std::string** is recommended for safer and more efficient string manipulation.

# Initialize string



# Strings (String Streams)

A close-up of a text

Description automatically generated

* Use string streams so that we can Concatenate both
* Stringstreams are just like IO streams. Using IO streams, We can read some input from the keyboard and write some output to the console. In string streams,
* We can do the same thing but instead of reading from the keyboard or writing to the console, we perform these operations on a string buffer.
* The stringstream internally maintains the string buffer and all the operations are performed in this string buffer.
* So there are three classes that we can use with stringstreams. There is:

1. Stringstream.
2. Istringstream.
3. Ostringstream.

* The stringstream class provides both insertion and extraction operators.

A close-up of a text

Description automatically generated

1. `std::stringstream`:

* This class provides the functionality to treat strings as input or output streams.
* It is capable of both input and output operations.
* You can use it to manipulate strings just like you would manipulate data using input/output streams.

2. `std::istringstream`:

* This class is specifically used for input operations.
* It allows you to treat a string as an input stream.
* You can extract values from the string as if reading from `std::cin`.

3.`std::ostringstream`:

* This class is specifically used for output operations.
* It allows you to treat a string as an output stream.
* You can insert values into the string as if writing to `std::cout`.

These classes are very useful for various string manipulation tasks, such as parsing input strings, formatting output strings, or converting between different data types and strings.

* All these strings and the integer variables will be inserted into the stringstream. The stringstream internally manages a string buffer and we can access that buffer through a member function of the string stream.
* Called str(). This function has two overloads.
* One overload returns the copy of the buffer as a standard string object.



* The other overload accepts a string.

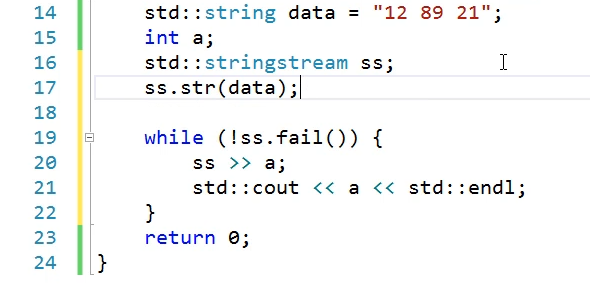


* Function called to\_string() and this function is overloaded for all the primitive types and it returns the primitive types as string.

A computer screen shot of a code

Description automatically generated

* We can also read from the stringstream. And that is useful for parsing strings, if we have a string that contains a bunch of numbers, then we can parse the string and extract the individual numbers.
* ‎**std::stringstream::fail()** is a member function that checks whether the most recent ‎input operation failed. This failure typically occurs when an attempt is made to extract ‎data of one type but the actual data in the stream cannot be converted to that type.‎
* **Checking for fail bit:** Before extracting data from a stringstream, it's a good ‎practice to check if the operation is successful using the fail() function. If the ‎operation fails, it means the data couldn't be extracted as expected.‎



* The extraction operator of the stringstream returns a reference of the stringstream object itself.
* The stringstream class also contains the bool operator which is overloaded.
* And we can use that bool operator to check for the fail bit.
* So instead of writing the statement here, we can write it within the conditional expression itself.
* So when this expression extracts the last number and in the next iteration when it tries to read again, it will set the fail bit.
* This expression will return the string stream on which the bool operator will be invoked and that will return a false

A computer screen shot of a code

Description automatically generated

**Conclusion**

**Transitioning from C-Style Strings to std::string in C++**

**Introduction:** delved into the complexities of working with C-style strings and the inherent challenges they pose in terms of memory management and error-prone operations. Recognizing these limitations, C++ provides a robust solution in the form of the **std::string** class. This tutorial serves as a comprehensive guide to mastering the usage of **std::string**, offering insights into its advantages, basic operations, and advanced techniques.

**Advantages of std::string:** Unlike their C-style counterparts, **std::string** encapsulates string manipulation operations within a streamlined, object-oriented framework. By abstracting away memory management intricacies, such as allocation and deallocation, **std::string** simplifies string handling and enhances code reliability.

**Getting Started with std::string:** To harness the power of **std::string**, we first explore its fundamental operations. This includes initialization and assignment, along with accessing individual characters within the string using the subscript operator **[]** and querying the size of the string using the **size()** function.

**Concatenation and Insertion:** One of the strengths of **std::string** lies in its ability to facilitate string concatenation and insertion effortlessly. By overloading operators like **+** and **+=**, developers can concatenate strings with ease. Additionally, the **insert()** function enables precise insertion of substrings at specified positions within a target string.

**Comparison and Search:** **std::string** provides intuitive mechanisms for comparing strings and searching for substrings. With overloaded comparison operators such as **==**, **<**, **>**, and **!=**, along with the versatile **find()** function, developers can efficiently compare strings for equality and perform substring searches with ease.

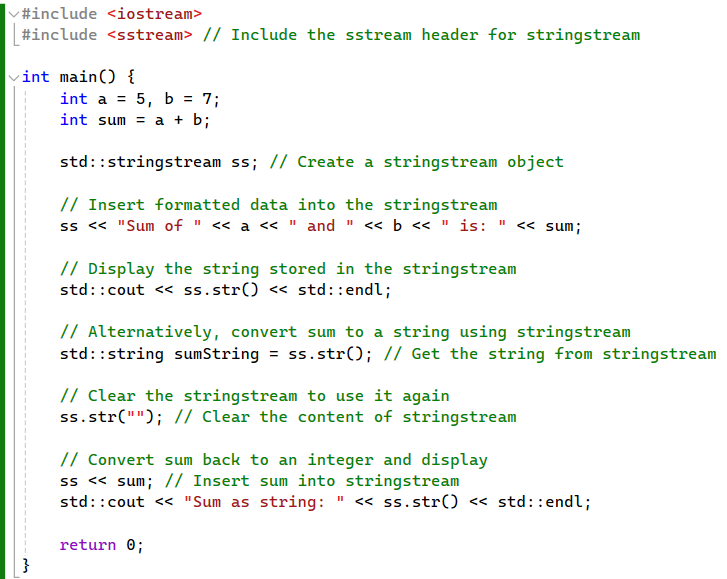
**Usage Best Practices:** Embracing **std::string** over C-style strings is advocated for its enhanced readability, maintainability, and robustness. By leveraging the automatic memory management capabilities of **std::string**, developers can mitigate common pitfalls associated with manual memory handling.

**Advanced Techniques:** As C++ evolves, so do its language features. The introduction of user-defined literals in C++11 revolutionized string initialization syntax, offering a cleaner and more expressive approach. By harnessing custom literals, developers can further enhance code clarity and reduce verbosity in string initialization.

**Conclusion:** In conclusion, transitioning from C-style strings to **std::string** marks a significant stride in simplifying string manipulation in C++. By mastering the capabilities of **std::string** and adhering to best practices, developers can unlock the full potential of string handling in their C++ projects. From basic operations to advanced techniques, **std::string** stands as a testament to the evolution of string manipulation paradigms in modern C++ programming.

Stringstreams in C++ provide a versatile way to manipulate strings as if they were streams of input/output. They allow you to perform operations like extraction and insertion of data into a string buffer, similar to how you would read from or write to the console with standard input/output streams.

Here's an example of using stringstreams to add two numbers and display their sum as a formatted string:

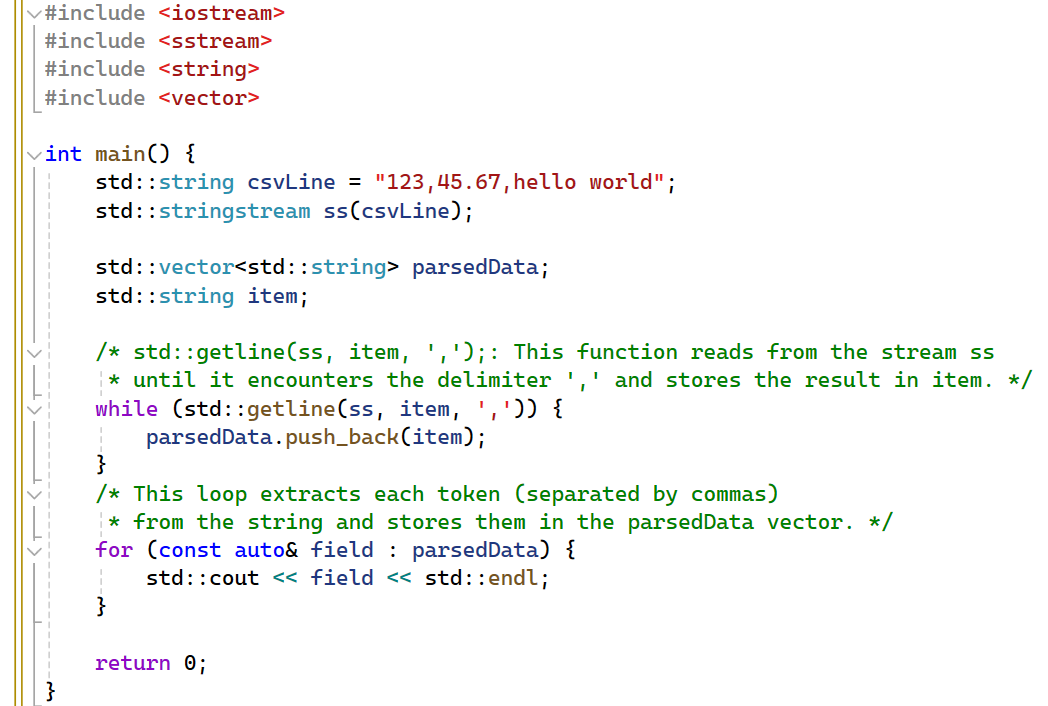


In this example:

* We first create a stringstream object **ss**.
* We use the insertion operator (**<<**) to insert formatted data into the stringstream, just like we would with **cout**.
* We retrieve the string stored in the stringstream using the **str()** member function.
* We can also convert an integer (**sum**) to a string using the stringstream.
* After using the stringstream, we can clear its content using **str("")** to reuse it for other operations.

Stringstreams provide a convenient way to manipulate strings and convert between different data types, making them useful for tasks like formatting output or parsing input strings. However, it's worth noting that C++11 introduced functions like **std::to\_string()** and **std::stoi()** for simpler conversion between primitive types and strings.

**Some Examples**



A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

**Summary**

* **Stream**: Represents a sequence of characters for input or output.
* **<< Operator**: Inserts data into a stream.
* **>> Operator**: Extracts data from a stream.
* **std::stringstream**: Used for both input and output on strings.
* **std::istringstream**: Used for input on strings.
* **std::ostringstream**: Used for output on strings.
* **str() Method**: Retrieves the contents of the stream as a string.
* **getline() Function**: Reads a line or a portion of a string up to a delimiter.

A white background with black text

Description automatically generated

A screenshot of a computer code

Description automatically generated

A white background with black text

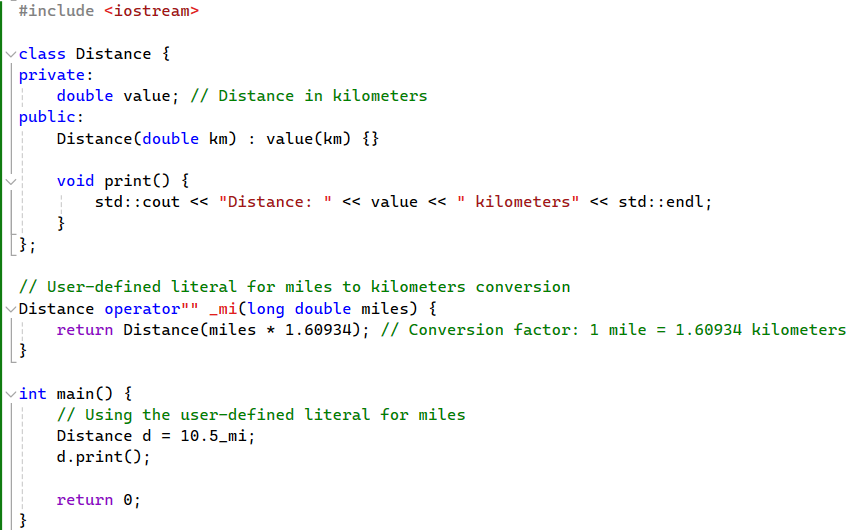
Description automatically generated

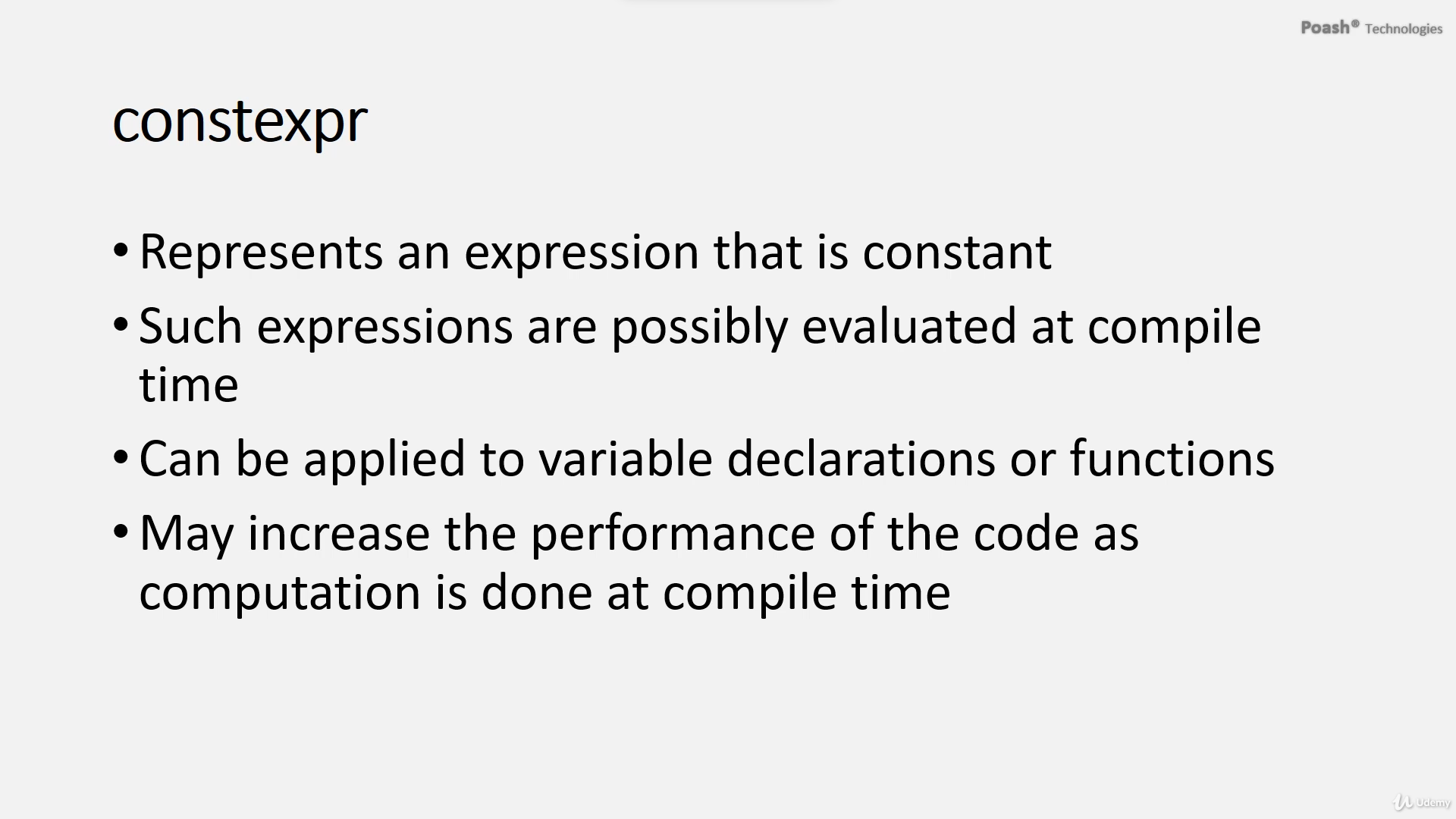
User-defined literals in C++ provide a way to extend the language's syntax to define custom literal suffixes for various types. This allows developers to create more expressive and readable code by introducing syntactic shortcuts for representing specific values.

Here's an overview of user-defined literals and how they work:

1. **Definition**: To create a user-defined literal, you define a function using the **operator** keyword followed by a pair of double underscores (**\_\_**). The function takes a single parameter representing the literal value and returns the desired type.
2. **Syntax**: User-defined literals always begin with an underscore (**\_**). This **is to distinguish** them from standard library literals, which do not start with an underscore. Attempting to define a literal without the leading underscore may result in a compiler warning or error.
3. **Types**: User-defined literals can be applied to certain types only. These include unsigned long long (for integer literals), long double (for floating-point literals), const char\* (for string literals), and char (for character literals).
4. **Global Functions**: Literal operator functions cannot be member functions of a class; **they must be global functions.**
5. **Suffixes**: User-defined literals are applied to literals by suffixing them with the defined identifier. For example, if you define a literal operator function **\_km** for kilometers, you can use it as **5\_km**.
6. **Type Safety**: User-defined literals can enhance type safety by allowing developers to specify units or representations more explicitly. For example, instead of manually converting between kilometers and miles, you can define literals for each unit and use them directly in your code.

Here's a summarized example demonstrating the use of a user-defined literal for converting miles to kilometers:

When used in the **main()** function as **10.5\_mi**, it automatically converts the specified distance from miles to kilometers using the conversion factor and initializes a **Distance** object with the converted value.



A white paper with black text

Description automatically generated

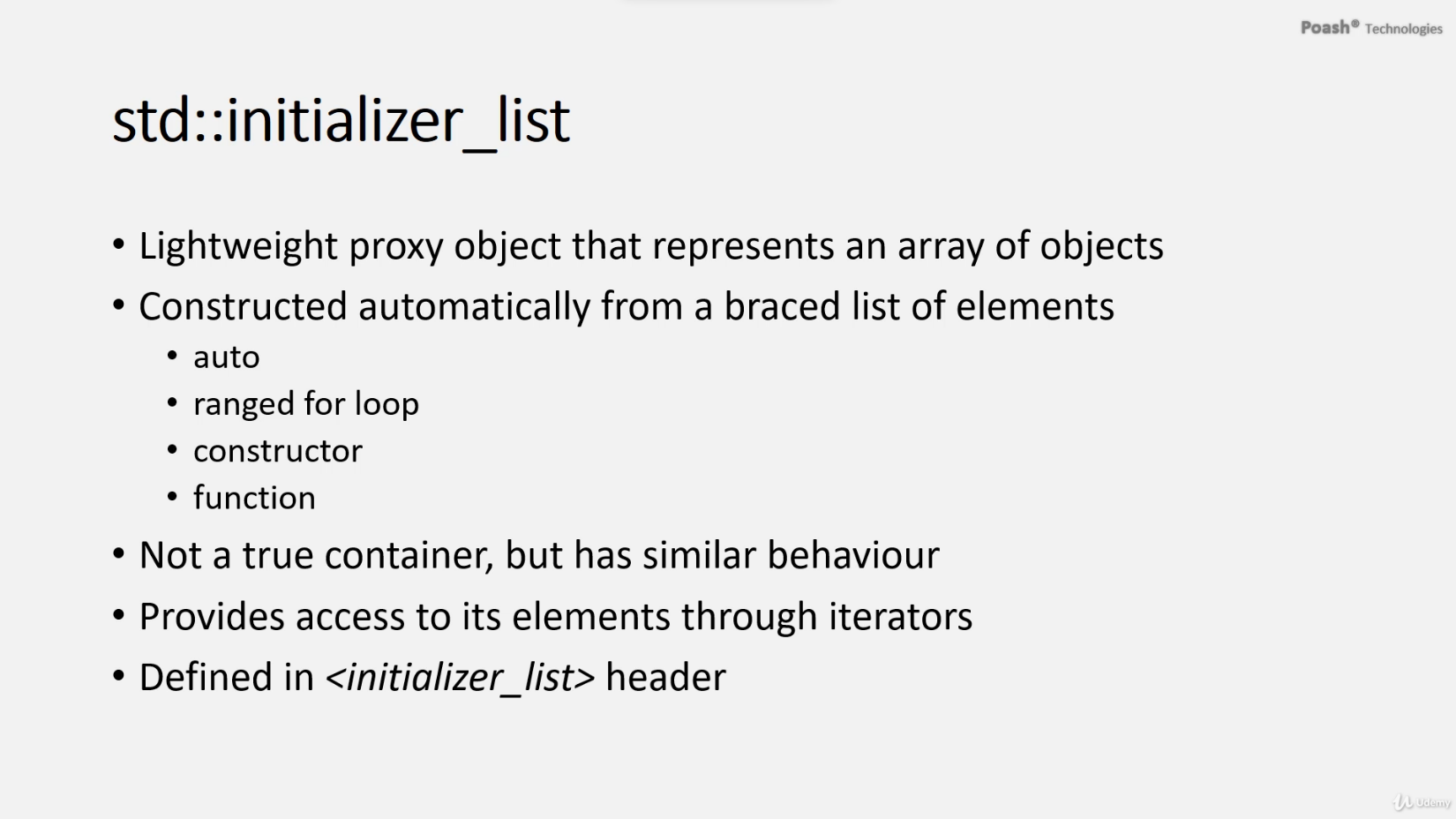
A screenshot of a computer program

Description automatically generated

The **constexpr** keyword in C++ signifies a constant expression. Here's a breakdown of its usage and implications:

1. **Constant Expressions**: **constexpr** is used to denote expressions that can **be evaluated at compile time.** This can lead to improved performance as the expressions are computed during compilation.
2. **Applicability**: **constexpr** can be applied to variable declarations and functions. When applied to a **variable**, it ensures that **the value is computed at compile time**. When applied to a **function**, it specifies **that the function can be evaluated at compile time** **if its arguments are known at compile time.**
3. **Compile-Time Evaluation**: Expressions marked with **constexpr** are evaluated at compile time whenever possible. This is useful for initializing variables or defining functions whose results are known at compile time.
4. **Performance**: Using **constexpr** can potentially improve performance by computing values at compile time rather than at runtime.
5. **Differences from const**: While both **constexpr** and **const** can be used to create **constants**, there are important differences. **const** variables **can be initialized at runtime**, while **constexpr** variables must be **initialized at compile time.** Additionally, **constexpr** functions can only accept and return literal types, and their bodies are restricted **to a single line statement that must be a return statement**. In contrast, **const** functions **do not have these restrictions**.
6. **Literal Types**: **constexpr** functions **can only accept and return literal types**. Literal types include scalar types (such as integer, float, and char), references, arrays, and classes with **constexpr** constructors.
7. **Usage**: **constexpr** functions can be invoked in both **constexpr** and non-**constexpr** contexts. In a **constexpr** context, the function is evaluated at compile time if possible, otherwise, it behaves like a regular function and is evaluated at runtime.
8. **C++14 Relaxations**: C++14 relaxed some of the restrictions on **constexpr** functions, allowing conditional statements inside them. However, not all compilers may support these features.
9. **Implicitly Inline**: All **constexpr** functions are implicitly inline, meaning they can be defined in header files.
10. **Choosing Between const and constexpr**: Use **const** to indicate that a value cannot be modified and **constexpr** for expressions that need to be evaluated at compile time, such as declarations and functions.

In summary, **constexpr** enables the creation of constant expressions that are evaluated at compile time, potentially improving performance and providing compile-time guarantees. It is particularly useful for creating compile-time constants and functions with known results at compile time.



**{ }**

* The purpose of the initializer list is to initialize user-defined objects, Just like an array is initialized especially if the user-defined object is a container.
* The initializer is a lightweight proxy object and it represents an array of objects. An initializer\_list object will be constructed automatically from a brace list of elements.
* For example, when you use automatic type inference for initializing a brace list of elements then an initializer list object is automatically created.
* The initializer list object is also created automatically when you use the brace list of elements in a range-based for loop. It is also automatically created if a constructor or some other function accepts a brace list of elements.
* An initializer list is not a true container but it has similar behavior to that of containers.
* It provides access to its elements through iterators. To use the initializer list, You have to include the header file, initializer\_list.

An initializer list is a lightweight proxy object introduced in C++11 to represent an array of objects. It's important not to confuse initializer lists with member initializer lists, which serve a different purpose.

Here's a summary of initializer lists and their usage:

1. **Uniform Initialization Syntax**: C++11 introduced uniform initialization syntax, which allows for consistent initialization of scalar and array types using curly braces **{}**. This syntax can also be used with user-defined objects.
2. **Initializer List Class**: The uniform initialization introduced a new class template called **initializer\_list**, which is used to store an array of objects of a specified type.
3. **Usage with Container Classes**: Initializer lists are commonly used with container classes, which are classes that can hold objects of other classes.
4. **Creating Custom Container Class**: For example, you can create a custom container class like a **Bag**, which behaves like an array and can hold objects of other classes. In this case, you might initialize the **Bag** with a set of predefined values using an initializer list.
5. **Constructor Accepting Initializer List**: To initialize objects with a brace-enclosed list of elements, you can define a constructor in your class that accepts an **initializer\_list** as an argument.
6. **Accessing Elements**: Elements in the initializer list can be accessed using iterators. The **begin()** and **end()** functions provide iterators to the beginning and end of the list, respectively.
7. **Automatic Type Inference**: If you use automatic type inference for initializing a brace-enclosed list of elements, an initializer list object is automatically created.
8. **Range-Based For Loop**: Initializer lists can also be used with range-based for loops, where the loop iterates over each element in the list.
9. **Header File**: To use initializer lists, you need to include the **<initializer\_list>** header file.

In summary, initializer lists provide a convenient way to initialize objects, especially when working with container classes or when initializing arrays of objects. They are automatically created from brace-enclosed lists of elements and behave similarly to containers, providing access to their elements through iterators.

A screenshot of a computer code

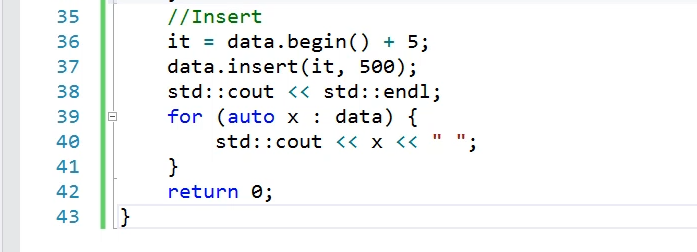
Description automatically generatedA computer code with green and blue text

Description automatically generated

# Vector class

A screenshot of a computer program

Description automatically generated



The **vector** class provided by the standard library in C++ offers a convenient way to work with dynamic arrays without the need for manual memory management. Here's a breakdown of how to use the **vector** class and its common operations:

1. **Including the Header File**: To use the **vector** class, you need to include the **<vector>** header file.
2. **Class Template**: The **vector** class is a class template, meaning it can store objects of any type. When creating a **vector**, you specify the type of objects it will store within angular brackets, such as **vector<int>** for storing integers.
3. **Initialization**: You can initialize a **vector** using uniform initialization syntax, providing initial data if desired. For example, **vector<int> myVector = {1, 2, 3};** initializes a vector with three integers.
4. **Adding Elements**: You can add elements to the end of a **vector** using the **push\_back()** member function. This function automatically resizes the vector if necessary to accommodate the new element.
5. **Accessing Elements**: Elements in a **vector** can be accessed using the subscript operator **[]**. The **size()** function returns the number of elements in the vector, allowing you to iterate over them using a loop.
6. **Modifying Elements**: Elements in a **vector** can be modified using the subscript operator **[]**. For example, **myVector[0] = 100;** modifies the first element of the vector.
7. **Iterators**: You can also access elements in a **vector** using iterators. The **begin()** function returns an iterator pointing to the beginning of the vector, and the **end()** function returns an iterator pointing to the end. Iterators provide flexible ways to traverse and manipulate the elements of the vector.
8. **Removing Elements**: The **erase()** function allows you to remove elements from a **vector** at a specified position, which is indicated by an iterator.
9. **Inserting Elements**: The **insert()** function allows you to insert elements into a **vector** at a specified position, which is indicated by an iterator. The function takes two arguments: the iterator pointing to the position where the element should be inserted, and the value of the new element.

Overall, the **vector** class provides a more convenient and safer alternative to manual memory management when working with dynamic arrays in C++. It automatically manages memory allocation and resizing, making it easier to work with collections of data.

A screenshot of a computer program

Description automatically generated

Unions have several disadvantages.

* The first disadvantage is that there is no way to know which type it holds That is, which is the active member.
* Secondly, if the nested member has a known default constructor, then that deletes the default constructor of the union.
* Additionally, the destructor also becomes deleted, so you will have to manually implement these functions in the union.
* Note that: before C++ 11, it was illegal for a union to have a member that contains user-defined constructors and destructo, but in C++ 11, it is allowed as long as you provide implementations of the constructors and destructor in the union.
* If the union has a user-defined type as a member, then you cannot directly initialize it or even assign it to it. Instead, you have to use the placement new operator.
* The user-defined types in the union are not destroyed implicitly, you have to manually invoke their destructor.
* A union cannot have a base class. You cannot inherit from a union. Also, a union cannot contain virtual functions.
* In C++, a union can have a constructor and a destructor, so I can add a constructor here and a destructor.
* In the constructor, you can initialize only one member, you cannot initialize both members.

Unions in C++ provide a way to represent different types of data in the same memory space, which can be advantageous for saving space, especially on embedded platforms. Unlike structures or classes, where each member has its own separate memory, in unions, the different members share the same memory.

However, unions come with several disadvantages:

1. **No Type Information**: There's no built-in way to know which type (or member) of the union is currently active. This lack of type information can lead to errors if you access the wrong member.
2. **Constructor and Destructor Limitations**: If a union contains a member with a known default constructor, the default constructor of the union is deleted. Similarly, the destructor of the union is deleted, requiring manual implementation if needed.
3. **Initialization and Assignment**: If a union contains a user-defined type as a member, you cannot directly initialize or assign values to it. Instead, you have to use the placement new operator.
4. **No Base Class or Virtual Functions**: Unions cannot have a base class, and they cannot contain virtual functions.

Despite these limitations, unions have some advantages. For example, they can help save memory by storing different types of data in the same space. The size of a union is determined by the size of its largest member, and all members share the same storage.

Here's an example of using a union in C++:

A screen shot of a computer code

Description automatically generated

In this example, **Test** is a union containing an integer and a character. The union's size is determined by the size of its largest member. You can access and manipulate the union's members, but you need to be careful about which member is currently active. Accessing a non-active member can lead to undefined behavior.

Using user-defined objects inside a union in C++ comes with several challenges and limitations. Let's break down the main points from the provided text:

1. **Default Constructor and Destructor**: If the classes stored within the union (**A** and **B** in this case) have user-defined default constructors or destructors, the default constructor and destructor of the union become deleted. This means you must manually define a constructor and destructor for the union.
2. **Initialization**: You cannot use the default constructor or assignment operator to initialize objects stored within the union. Instead, you must use the placement new operator, which initializes the memory without allocating it. This is necessary because the memory for the union is already allocated based on the size of its largest member.
3. **Manual Invocation of Destructor**: Objects stored within the union are not implicitly destroyed, so you must manually invoke their destructors when necessary.

Here's a summary of the steps to use user-defined objects within a union:

* Manually define a constructor and destructor for the union.
* Use the placement new operator to initialize objects stored within the union.
* Manually invoke the destructor of objects stored within the union when needed.

The provided example demonstrates these concepts by initializing objects of classes **A** and **B** within a union and manually invoking their constructors and destructors.

While unions have limitations when dealing with user-defined types, they can still be useful in certain scenarios. Additionally, C++17 introduced **std::variant**, which provides type-safe union-like functionality and can be a preferable alternative in many cases.