Agenda

- Singleton class
- Factory Design Pattern
- nested class and local class
- constexpr
- noexcept
- nullptr
- nested namespace
- Smart Pointer

Singleton class

- It is a design pattern
- Design patterns are a standard solution to well-known problem
- It enables to use a single object of the class through out the application

Factory Design Pattern

- It is a creational design that offers a way to produce objects in a basesclass while letting derived classes change the kind of objects that are created.
- It is useful when there is a need to create multiple objects of the same type, but the type of the objects is not known until runtime.
- It is implemented using a factory function, which is a method that returns an object of the specified type.

Local Class

- If inside a function you declare a class then such classes are called as local classes.
- Inside local class you can access static and global members but you cannot access the local members declared inside the function where the class is declared.
- Inside local classes we cannot declare static data member however defining static member functions are allowed.
- As every local variable declared in function goes on its individual stack frame, such variables cannot be accessed in the local class functions.
- static and global variales gets space on data section which are designed to be shared.
- In a class static data members are designed to be accessed using classname and :: , the static member are designed to be shareed however in local classes we cannot access them ouside the function in ehich they are declared.
- Hence their is no purpose of keeping static data members inside local classes

Nested class

- A class declared inside another class is called as nested class
- Inside Nested class we can access private static members of outer class directly.
- A nested class can access all the private and public members of outer class directly on the outer class object

- Inside Nested class you can access static and global variables.
- As static and global variables are designed to shared they are accessiable inside the nested class.
- Data members gets the memory only when object is created and hence the nested class cannot access outer class data non static data members directly as they do not get memory inside them.

noexcept operator

- The noexcept operator performs a compile-time check that returns true if an expression is declared to not throw any exceptions.
- It can be used within a function template's noexcept specifier to declare that the function will throw exceptions for some types but not others.

```
void may_throw(); // it will throw exception
void no_throw() noexcept; // it will not throw exception

int main()
{
cout<<"may_throw() is noexcept(" << noexcept(may_throw())<<")"<<endl;
cout<<"no_throw() is noexcept(" << noexcept(no_throw())<<")"<<endl;
}

// output
// may_throw() is noexcept(false)
// no_throw() is noexcept(true)
```

constexpr

- constexpr is a feature added in C++ 11.
- The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time.
- Such variables and functions can then be used where only compile time constant expressions are allowed (provided that appropriate function arguments are given).
- A constexpr specifier used in an object declaration or non-static member function(until C++14) implies const.
- A constexpr specifier used in a function or static data member(since C++17) declaration implies inline.

constexpr vs inline

- The constexpr keyword is used to declare that a function or variable can be evaluated at compile-time.
- It guarantees that the value or result of the function can be computed at compile-time if all arguments are known at compile-time.
- constexpr functions must have a return type that is literal and must consist of a single return statement.
- Variables declared as constexpr must be initialized by constant expressions.
- constexpr functions can be used in contexts where constant expressions are required, such as array sizes, template arguments,etc.

- The inline keyword is used to suggest to the compiler that a function should be expanded in place at each call site rather than being called like a regular function.
- It is primarily used to improve performance by reducing the overhead of function calls, especially for small, frequently called functions.
- The compiler may choose to ignore the inline keyword and not inline the function if it determines that inlining would not be beneficial.
- inline functions may still have a separate definition in a translation unit, and they can contain any code that a regular function can have.

Rules for constexpr

- In C++ 11, a constexpr function should contain only one return statement. C++ 14 allows more than one statement.
- constexpr function should refer only to constant global variables.
- constexpr function can call only other constexpr functions not simple functions.
- The function should not be of a void type.

nullptr

- A nullptr is a keyword introduced in C++11 to represent a null pointer.
- It provides a type-safe and clearer alternative to using NULL or 0 for null pointer constants.
- It's recommended to use nullptr in modern C++ code.
- nullptr helps avoid ambiguities in function overloading and template specialization scenarios.

```
void f1(int *n)
{
    cout << "Function with int* " << endl;
}
void f1(int n)
{
    cout << "Function with int " << endl;
}

int main()
{
    int *ptr1 = 0;
    int *ptr2 = NULL;
    int *ptr3 = nullptr;
    cout << "ptr1 = " << ptr1 << endl;
    cout << "ptr2 = " << ptr2 << endl;
    cout << "ptr3 = " << ptr3 << endl;

    f1(0); // fun with int
    // f1(NULL); // ambugity
    f1(nullptr); // fun with int*
```

```
    return 0;
}
```

- Smart pointers are objects that behave like pointers but provide automatic memory management.
- Smart pointers enable automatic, exception-safe, object lifetime management.
- They help prevent memory leaks and manage the lifetime of dynamically allocated objects.
- Smart pointers are part of the C++ Standard Library and are implemented as template classes.
- The main smart pointers in C++ are:

1. std::unique_ptr

- `std::unique_ptr` is a smart pointer that owns a dynamically allocated object and ensures that the object is deleted when the `unique_ptr` goes out of scope or is reset.
- It is unique in that it cannot be copied or shared. It can only be moved.
- It is lightweight and efficient because it does not incur the overhead of reference counting.
- The object is disposed of, using the associated deleter when either of the following happens:
    1. the managing unique_ptr object is destroyed.
    2. the managing unique_ptr object is assigned another pointer via operator= or reset().

2. std::shared_ptr

- `std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer.
- Several shared_ptr objects may own the same object.
- The object is destroyed and its memory deallocated when either of the following happens:
    1. the last remaining shared_ptr owning the object is destroyed;
    2. the last remaining shared_ptr owning the object is assigned another pointer via operator= or reset().
- It manages the ownership of a dynamically allocated object using reference counting.

3. std::weak_ptr

- `std::weak_ptr` is a smart pointer that provides a non-owning reference to an object managed by `std::shared_ptr`.
- It does not participate in reference counting and does not keep the object alive.
- It is used to break cyclic dependencies between `std::shared_ptr` objects.
- It must be converted to `std::shared_ptr` in order to access the referenced object.
- It models temporary ownership when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else