## Concurrency

A white text with black text

Description automatically generated

Concurrency in programming, particularly in C++, is a crucial concept that allows applications to perform multiple operations simultaneously, improving efficiency and responsiveness. Here is a more detailed breakdown of the concepts and practical usage:

**1. Understanding Concurrency**

Concurrency is the ability of a system to handle multiple tasks at the same time. In a concurrent system, various tasks or processes progress in overlapping periods. This does not necessarily mean they are executing at the exact same time (simultaneously), but rather that their execution is interleaved, providing the appearance of simultaneous execution.

**2. Importance of Concurrency**

Concurrency is vital for large applications with several components that need to run tasks concurrently. It helps in:

* **Improving CPU Utilization**: By overlapping tasks, an application can make better use of CPU resources, especially on multi-core systems.
* **Enhancing Responsiveness**: Applications, particularly those with user interfaces, can remain responsive to user input while performing background tasks.
* **Parallel Processing**: On multi-core processors, true parallel execution can occur, further enhancing performance.

**3. Concurrency in C++ (C++11 and later)**

In C++, concurrency is commonly achieved through multi-threading. The C++11 standard introduced a dedicated threading library to simplify and standardize thread management. This library provides various tools and features:

**Threads**

A thread is the smallest unit of execution in a program. In C++, threads can be created using the std::thread class.  
When a long-running task, such as downloading a file, is performed in a single thread, it can lead to blocking behavior in the application.

**Detailed Explanation**

**1. Downloading the File**

When the Download() function is invoked, it simulates the process of downloading a file from the internet. This function might involve time-consuming operations, such as pushing a large number of elements into a list. These operations represent the time it takes to read the file data and store it.

**2. Main Thread Blocked**

In the main() function, a message indicating the start of the download operation is printed. However, because the Download() function is blocking due to its time-consuming tasks, the subsequent message indicating the start of another operation is not printed until Download() completes its execution. This results in the main thread being blocked.

**3. User Interaction Blocked**

Since the main thread is busy executing the Download() function, the application cannot respond to user input during this time. This means that while the file is being downloaded, the user cannot interact with the application.

**4. Primary Thread Responsibility**

In most applications, including this example, the primary thread (main thread) is responsible for executing all operations and handling user input. If the primary thread is occupied with a long-running task like downloading a file, it cannot process user input until the task is complete.

**Example**

Here is an example to illustrate this blocking behavior in a single-threaded C++ application:

A screenshot of a computer program

Description automatically generated

**Output**

A close up of black text

Description automatically generated

In this example:

* The Download() function is a blocking call, simulating a time-consuming task by pushing a large number of elements into a vector.
* The message "Download operation completed!" is not printed until after the Download() function completes, demonstrating that the main thread is blocked during the download.
* The application remains unresponsive to user input during the download.

**Importance of Concurrency and Asynchronous Execution**

To avoid blocking behavior and keep the application responsive, concurrency and asynchronous execution are crucial. By offloading long-running tasks to separate threads, the main thread can remain responsive to user interactions. Here's an improved version of the example using threads:

**Improved Example with Concurrency**

A screenshot of a computer code

Description automatically generated

**Output**

A white background with black text

Description automatically generated

In this improved version:

* The Download() function is executed in a separate thread using std::thread.
* The main thread prints the message "Download operation started!" and continues to the next operation without waiting for the download to finish.
* The application remains responsive to user input while the file is being downloaded.

By using concurrency, the application can handle long-running tasks without blocking the main thread, ensuring a better user experience.



**User Interface**: The Qt application features a clean and intuitive user interface with several interactive elements that enhance user engagement.

**Push Button**: A push button triggers the on\_pushButton\_clicked callback function, displaying a message box, showcasing the application's responsiveness.

**File Download Scenario**: When integrating a file download process, initiating it via the push button causes a noticeable delay before the message appears.

**Responsiveness Issue**: During the download process, the application becomes unresponsive, with a "Not Responding" message appearing in the title bar, indicating that the main thread is blocked.

**Optimization Need**: To improve responsiveness and prevent the application from freezing, it is crucial to handle long-running operations by offloading them to background threads or using asynchronous techniques.

**Conclusion**: Prioritizing responsive design and efficient handling of long-running tasks ensures the application remains functional and engaging, providing a positive user experience.

A white background with black text

Description automatically generated

**Thread Class Introduction**: The threading library's thread class, found in the standard namespace, enables concurrent operations through multi-threading.

**Callable Entities**: The thread class constructor accepts callable entities, such as function pointers, function objects, or lambda functions, which it executes within a distinct thread.

**Non-blocking Instantiation**: When instantiated, the thread class constructor does not halt execution to wait for the thread creation; it immediately returns control to the caller, allowing the program to continue running independently.

**Header File Inclusion**: To use the thread class, including the thread header file is necessary, providing developers with a comprehensive set of tools for managing concurrent operations in their applications.

# Thread Creation (stdthread)

A screenshot of a computer code

Description automatically generated

* **Separate Thread for Download**: To perform the download operation in a separate thread, include the <thread> header file and create a thread by passing the callable (e.g., the Download() function).
* **Concurrent Execution**: This creates a new thread where the Download() function executes, allowing the main() function to continue its execution concurrently.
* **Two Threads**: The application now runs with two threads: the main thread and the child thread.
* **Synchronization with Join**: Without synchronization, the main thread may finish execution and terminate before the child thread completes. To prevent this, the main thread should wait for the child thread by calling **join** on it.
* **Importance of Join**: It is essential to invoke join on joinable threads to avoid premature termination of the program. If join is not called, the thread object may be destroyed without completing its execution, leading to immediate program termination.
* **Effective Concurrency Management**: Running the program shows that the main thread prints messages immediately, remaining unblocked while the Download() function runs in the background, demonstrating effective concurrency management.

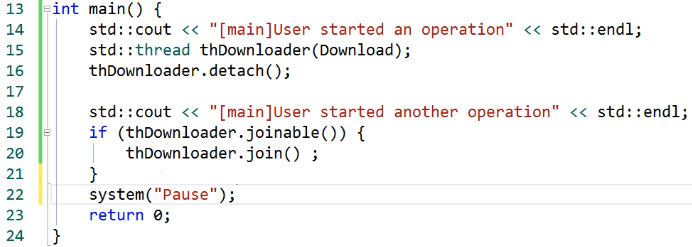
# detached threads

* Additionally, detached threads offer a way to execute tasks in the background without waiting for them to finish. We can create a detached thread by calling the **detach()** member function on its thread object. However, a detached thread cannot be joined, so calling **join** on it is not allowed. To check if a thread can be joined, we can use the **joinable()** function.

A screen shot of a computer code

Description automatically generated

* It's important to note that once a thread is detached, it can never become joinable again. A detached thread automatically releases its resources when the thread object is destroyed, ensuring proper resource management in multi-threaded applications.
* Upon execution, you'll observe that the application may start and terminate quickly if the main thread doesn't wait for the child thread to complete its execution. To pause the main thread's execution, we can use system functions before running it again. Ultimately, a detached thread automatically releases its resources upon destruction, ensuring proper resource management in the application



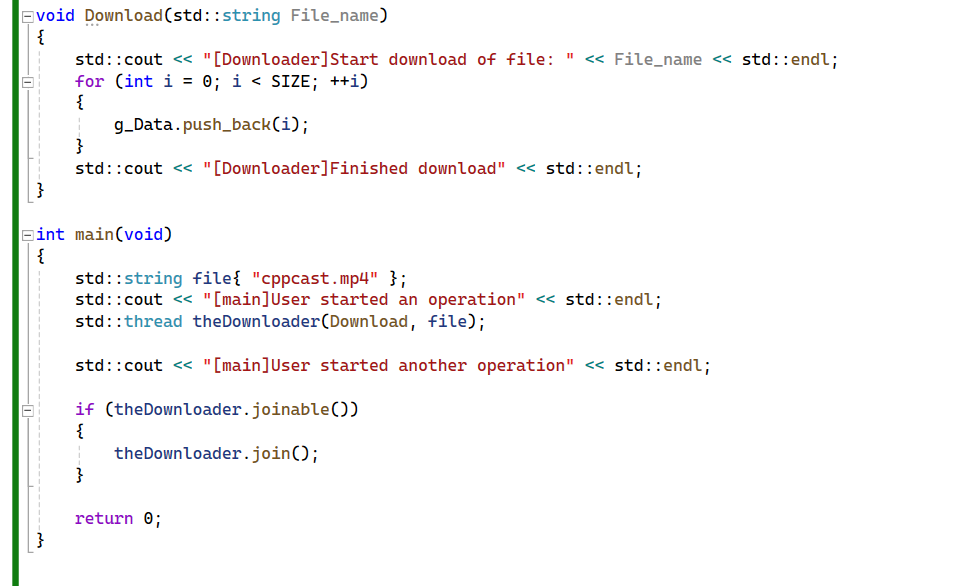
# Pass arguments to thread functions

To pass arguments to thread functions in C++, you can provide them directly to the constructor of the **std::thread** class. The constructor of the thread function is a variadic function, meaning it can accept any number of arguments.

Here's a step-by-step guide:

1. Define the thread function, such as **Download()**.
2. Specify the arguments you want to pass to the thread function.
3. Pass the callable (thread function) as the first argument to the constructor of **std::thread**.
4. Specify the additional arguments as subsequent arguments to the constructor.

For example, if you want to specify the name of the file to download, you can create a variable for the file name and pass it as the second argument to the constructor of **std::thread**. This argument will then be passed to the **Download()** function.



In this example, **filename** is passed as the second argument to the constructor of **std::thread**, and it is then passed to the **Download()** function when the thread is created.

You can also pass arguments by reference if needed, by specifying them as references in the thread function and passing them accordingly to the **std::thread** constructor.

# Passing Arguments To Threads

A screenshot of a computer program

Description automatically generated

# std::ref() and std::cref().

To pass arguments to thread functions by reference or as constant references, you can use helper functions like **std::ref()** and **std::cref()**. **These functions ensure that the arguments are passed by reference correctly**, avoiding unnecessary copies.

Here's how you can do it:

1. **Create a custom string class:** Since you want to track the number of instances of the string created, you create a custom string class. This class will have constructors, copy constructor, assignment operator, and destructor with **cout** statements to log their invocations.
2. **Replace standard string with custom string:** Replace instances of **std::string** with your custom string class in the thread function and main function.
3. **Use std::ref() or std::cref():** To pass the string argument to the thread function by reference or as a constant reference, use **std::ref()** or **std::cref()** respectively.

Here's a simplified example:

A screenshot of a computer code

Description automatically generated

In this example, the **String** class represents a custom string class with logging statements in its constructors, copy constructor, assignment operator, and destructor. Then, the **ProcessString** function takes a **const String&** argument, and we use **std::ref()** and **std::cref()** to pass **myString** by reference or as a constant reference to the thread function.

 **std::ref()**: Used to pass an object by reference to a thread function.

 **std::cref()**: Used to pass an object by const reference to a thread function.Top of Form

# Thread Synchronization (stdmutex)

A screenshot of a computer program

Description automatically generated

Mutexes, short for mutual exclusion, are synchronization primitives used in **multithreaded** programming to **prevent race conditions** and **ensure that only one thread can access a shared resource** at any given time. They are crucial for maintaining data consistency and avoiding conflicts when multiple threads try to access shared data concurrently.

Here's how mutexes typically work:

1. **Locking**: When a thread wants to access a shared resource protected by a mutex, it first acquires the mutex lock. If the lock is available (i.e., not held by another thread), the thread successfully acquires the lock and can proceed with accessing the shared resource. If the lock is already held by another thread, the requesting thread will be blocked until the lock becomes available.
2. **Unlocking**: After a thread finishes its operation on the shared resource, it releases the mutex lock by unlocking it. This allows other threads waiting to acquire the lock to proceed with their operations.

Mutexes ensure that only one thread can access the shared resource at any given time, thus preventing race conditions where multiple threads access or modify the resource simultaneously. By enforcing mutual exclusion, mutexes maintain data integrity and consistency in multithreaded programs.

It's important to handle exceptions and edge cases properly when using mutexes to avoid deadlocks, where threads are indefinitely blocked waiting for locks that will never become available. Deadlocks can occur if a thread fails to release the mutex lock in certain scenarios, such as when an exception occurs or when a thread terminates prematurely. Proper handling of mutex locks and unlocks helps prevent deadlocks and ensures the correct synchronization of shared resources in multithreaded applications.

Top of Form

# stdlock\_guard

The Resource Acquisition Is Initialization (RAII) idiom is a powerful technique in C++ to ensure that resources are properly managed, especially in multithreaded environments. Using std::lock\_guard simplifies mutex management by automatically locking a mutex upon creation and unlocking it when the std::lock\_guard object goes out of scope. This ensures that the mutex is always unlocked, even if exceptions occur or the function returns early.

**Key Points:**

1. **RAII Idiom**: This ensures that resource management (like locking a mutex) is tied to the lifecycle of an object. When the object is created, the resource is acquired (mutex is locked), and when the object is destroyed, the resource is released (mutex is unlocked).
2. **std::lock\_guard Class**: This class template is part of the C++11 standard library and is specifically designed to manage mutexes using the RAII idiom. When a std::lock\_guard object is created, it locks the mutex, and when the object is destroyed, it unlocks the mutex.
3. **Automatic Locking and Unlocking**: By using std::lock\_guard, you avoid the need to manually call lock and unlock on the mutex. This reduces the risk of errors, such as forgetting to unlock the mutex or accidentally double-locking it.
4. **Synchronization**: Using std::lock\_guard in combination with mutexes ensures that only one thread can access a critical section of code at a time, preventing data races and ensuring data integrity.

**Example:**

Here is an example of how to use std::lock\_guard to manage a mutex within a thread function:

A screenshot of a computer program

Description automatically generated

**Explanation:**

1. **Mutex Declaration**: A std::mutex object g\_Mutex is declared globally.
2. **Download Function**:

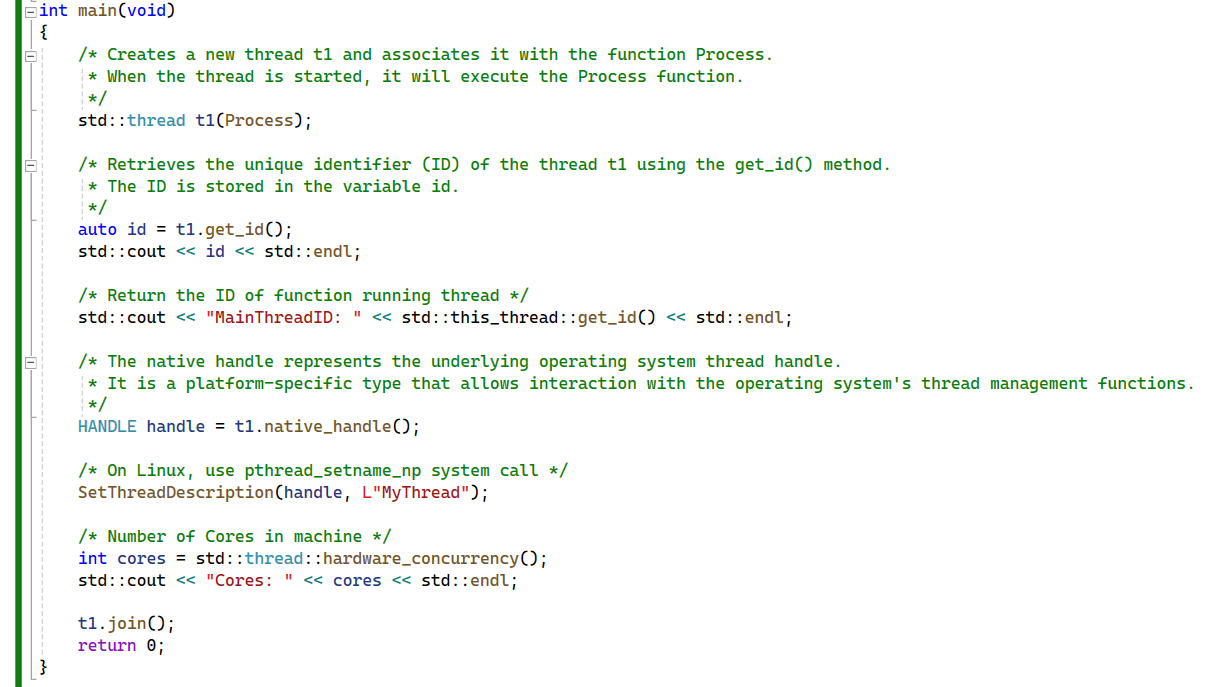
* A **std::lock\_guard<std::mutex>** object named guard is created and passed g\_Mutex.
* The mutex g\_Mutex is locked when guard is constructed and will be automatically unlocked when guard goes out of scope (at the end of the function).
* This ensures that the mutex is properly unlocked even if an exception occurs within the Download function.

1. **Main Function**:

* A thread is created to execute the Download function.
* The main thread waits for the downloader thread to finish by calling join.

Using std::lock\_guard makes the code cleaner and less error-prone by automating the locking and unlocking of the mutex, thus following the RAII idiom.

# stdthread Functions & stdthis\_thread Namespace



1. **std::thread t1(Process);**: This line creates a new thread **t1** and associates it with the function **Process**. When the thread is started, it will execute the **Process** function.
2. **auto id = t1.get\_id();**: This line retrieves the unique identifier (ID) of the thread **t1** using the **get\_id()** method. The ID is stored in the variable **id**.
3. The code **t1.native\_handle();** retrieves the native handle of the **std::thread** object **t1**.

* The native handle represents the underlying operating system thread handle. It is a platform-specific type that allows interaction with the operating system's thread management functions.
* However, it's important to note that directly manipulating the native handle is generally discouraged because it can lead to platform-dependent and unsafe code. In most cases, you should use the C++ standard library facilities provided by **std::thread** to manage threads

1. The code **t1.join();** is used to wait for the thread represented by the **std::thread** object **t1** to finish execution.

* When you call **join()** on a thread object, the current thread (usually the main thread) will wait at that point until the thread represented by **t1** completes its execution. Once the thread **t1** finishes its execution, the **join()** function returns, and the program continues its execution.
* This is a way to synchronize the execution of different threads, ensuring that certain parts of your program are executed sequentially or that resources are safely accessed by only one thread at a time.

A screen shot of a computer code

Description automatically generated

summary of the information provided about **std::thread** and related functions:

1. **get\_id() Function**:

* Returns a unique identifier of the thread.
* Represented by an instance of the **std::thread::id** class.
* Useful for identifying threads and debugging purposes.
* The ID can be printed or used for other identification purposes.

1. **native\_handle() Function**:

* Returns a type representing the underlying thread on the platform.
* On Windows, it returns a **HANDLE**, and on Linux, it returns a **pthread\_t**.
* Useful for interacting with the thread at a lower level using platform-specific APIs.

1. **SetThreadDescription() / pthread\_setname\_np() Functions**:

* Allows assigning a name to the thread for easier identification during debugging.
* **SetThreadDescription()** is used in Windows, while **pthread\_setname\_np()** is used in Linux.
* Names assigned through these functions are visible in debugging tools and logs.

1. **hardware\_concurrency() Function**:

* Static member function of **std::thread**.
* Returns the number of CPU cores present in the system.
* Helpful for determining the optimal number of threads for performance, usually matching the number of CPU cores.

1. **this\_thread Namespace**:

* Contains functions related to the current thread.
* **get\_id()** retrieves the ID of the current running thread.
* **sleep\_for()** introduces a delay in the execution of a thread.
* Time durations for **sleep\_for()** can be specified using **std::chrono** durations like seconds, milliseconds, microseconds, etc.

These functions and techniques provide useful capabilities for managing and interacting with threads in C++ programs, including identification, debugging, and thread control.

# Task Based Concurrency

In C++, both low-level and high-level concurrency mechanisms are available to handle multi-threading and parallelism effectively.

**Low-Level Concurrency**

Low-level concurrency involves directly working with threads. The C++ Standard Library provides the std::thread class to create and manage threads. Here is a simple example of creating and using a thread:

A computer code with text

Description automatically generated with medium confidence

In this example, std::thread is used to create a new thread that runs threadFunction. The join() method ensures that the main thread waits for the new thread to finish before exiting.

**High-Level Concurrency (Task-Based Concurrency)**

High-level concurrency abstracts away direct thread management and allows for a more intuitive approach to concurrent programming. C++ provides the <future> and <async> facilities to handle tasks.

Here is an example using std::async and std::future:

A computer code with text

Description automatically generated

In this example, std::async is used to launch taskFunction asynchronously. The std::future object result is used to retrieve the result of the task once it is completed.

**Benefits of High-Level Concurrency**

* **Simplicity**: Abstracts thread management, reducing the complexity of concurrent programming.
* **Task-Based Model**: Allows for a focus on the tasks rather than on thread management.
* **Automatic Thread Management**: The system can manage the underlying threads, optimizing resource usage.

**Conclusion**

While low-level concurrency gives fine-grained control over threads, high-level concurrency offers a more straightforward and intuitive approach for most applications. By leveraging high-level constructs like std::async and std::future, developers can write cleaner, more maintainable code while still achieving the benefits of concurrent execution.

**Without using thread management:**

A screenshot of a computer code

Description automatically generated

A white background with black text

Description automatically generated

A screenshot of a computer code

Description automatically generated

A white text on a white background

Description automatically generated

A close-up of a note

Description automatically generated

A close-up of a white background

Description automatically generated

Explains the usage of high-level concurrency features in C++, focusing on the **std::async** function and associated concepts:

1. **std::async Function**:
   * It allows executing a callable (such as a function, object, or lambda) in a separate thread.
   * Returns a **std::future** object, providing access to the result of the callable.
   * The header file **<future>** needs to be included to use **std::async**.
2. **Overloads of std::async**:
   * One overload accepts a callable and arguments, which are automatically passed to the task.
   * Another overload accepts a launch policy, which decides the behavior of **std::async**.
3. **Launch Policies**:
   * Two launch policies: **std::launch::deferred** and **std::launch::async**.
   * **std::launch::deferred** executes the task **synchronously**
   * while **std::launch::async** creates a **new** **thread**.
   * If no launch policy is specified, the behavior depends on the compiler implementation.
4. **Passing Arguments to Tasks**:
   * Arguments can be passed to the task through **std::async**, always by value.
   * To pass arguments by reference or constant reference, use **std::ref** or **std::cref** wrappers.
5. **Returning Values from Tasks**:
   * Values can be easily returned from tasks by returning them from the corresponding function.
   * The returned value is available through the **std::future** object associated with the task.
6. **std::future and std::promise**:
   * **std::future** is used for communication between threads, providing access to a shared state.
   * Internally, **std::future** is created through a **std::promise** object.
   * **std::promise** and **std::future** are always used as a pair, representing an **input and output channel**, respectively.
   * When a value is returned from the task, it is set in the **std::promise**, and then available in a different thread through the associated **std::future**.
   * This mechanism allows safe data sharing between threads without explicit synchronization.

In summary, **std::async**, along with **std::future** and **std::promise**, provides a convenient and safe way to execute tasks asynchronously in C++, facilitating parallelism and concurrent programming.

In C++, the **async** function from the **<future>** header allows us **to execute a function as a task in a separate thread,** without the need to manually create a thread. This function is particularly useful for task-based concurrency.

Here's how **async** works:

1. **Callable Function**: We can pass any callable function to **async**, such as a regular function, function object, or lambda expression. In the example provided, **Downloader()** is used as the callable.
2. **Return Type**: **async** returns an object of type **std::future**, which represents the result of the asynchronous operation.
3. **Getting the Result**: It's important to capture the **std::future** object returned by **async**, as not doing so can lead to unexpected behavior. If the result of the asynchronous operation needs to be retrieved, the **get()** function can be called on the **std::future** object. This function will block the main thread until the result is available, similar to the **join()** function of the **std::thread** class.

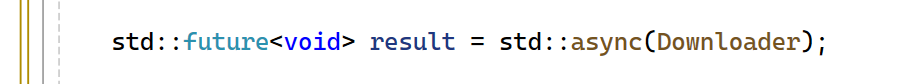
A computer code on a white background

Description automatically generated

Here's a summary of the steps involved:

1. Use the **async** function to execute a function as a task in a separate thread.
2. Capture the returned **std::future** object to ensure proper synchronization.
3. If needed, call the **get()** function on the **std::future** object to retrieve the result of the asynchronous operation.

By using **async**, we can initiate tasks to be executed asynchronously without having to manage threads explicitly, improving code readability and maintainability.



performs the following actions:

1. It calls the **std::async** function, which is part of the **<future>** header in C++. This function allows for the asynchronous execution of a function or callable object.
2. The function **Downloader** is passed as an argument to **std::async**. This function will be executed asynchronously in a separate thread.
3. The return type of **std::async** is **std::future<void>**, indicating that the asynchronous operation will not return a value (void).
4. The returned **std::future<void>** object, which represents the result of the asynchronous operation, is stored in the variable **result**.

In summary, this line of code asynchronously executes the **Downloader** function in a separate thread and captures the future object representing the result of that operation in the variable **result**.

A close up of a text

Description automatically generated

performs the following action:

1. It calls the **get()** member function of the **std::future<void>** object **result**.
2. This function call blocks the current thread until the asynchronous operation associated with the **std::future<void>** object **result** has completed and its result is available.

In summary, **result.get()** blocks the current thread until the asynchronous operation initiated by **std::async(Downloader)** completes. It waits for the result of the operation to become available.

# Launch Policies

A computer code with text

Description automatically generated

Code demonstrates the usage of **std::async** to execute a task asynchronously in a separate thread and retrieve the result using a **std::future** object. Here's a summary of the code:

1. **Operation Function**:

* The **Operation** function now accepts an integer parameter **count**, which determines the number of iterations for the sum calculation.
* The loop iterates **count** times instead of a fixed number (10).
* It simulates a delay using **std::this\_thread::sleep\_for** to make the operation take some time.
* The result of the operation, which is the sum, is returned.

1. **Main Function**:

* Inside the **main** function, **std::async** is used to asynchronously execute the **Operation** function in a separate thread.
* The result of the asynchronous operation is captured in a **std::future<int>** object named **result**.
* std::async is called with two arguments: the Operation function and the value 10, ‎representing the count parameter.‎
* A message **"main() Thread continues execution"** is printed to indicate that the main thread continues its execution without waiting for the asynchronous operation to finish.
* **result.valid()** is used to check if the future object is valid.
* If the future object is valid, **result.get()** is called to block the current thread until the result of the asynchronous operation becomes available.
* Once the result is available, it is printed to the console.

Overall, the code demonstrates how to perform an asynchronous operation using **std::async**, retrieve the result using a **std::future** object, and synchronize the main thread with the asynchronous operation using **std::future::get()**. Additionally, it showcases the use of a delay to simulate a time-consuming operation.

A computer screen shot of a program

Description automatically generated

The code line is using **std::async** with a specified launch policy of **std::launch::async**. This launch policy ensures that the operation represented by the function **Operation** will **be executed asynchronously** in a **separate thread**.

**std::future<int> result = std::async(std::launch::async, Operation, 10);**

**Explanation:**

* **std::async** is a function template that creates a **std::future** representing the result of an asynchronous operation.
* It accepts the launch policy as its first argument, which specifies how the operation should be executed.
* In this case, **std::launch::async** is used as the launch policy. This policy ensures that the operation will be **executed asynchronously in a separate thread**.
* The second argument is the function to be executed asynchronously, which is **Operation** in this case.
* Any additional arguments required by the function **Operation** are passed after the function name. Here, **10** is passed as the **count** parameter to the **Operation** function.

By specifying **std::launch::async** as the launch policy, the code guarantees that the **Operation** function will be executed asynchronously in a separate thread, regardless of the current system load or other factors. This helps improve the responsiveness and performance of the program by utilizing concurrency.

A computer screen shot of a computer code

Description automatically generated

The **std::async** function in C++ allows for the asynchronous execution of tasks, providing flexibility in managing concurrency. Two launch policies, **std::launch::async** and **std::launch::deferred**, dictate how the task is executed:

1. **Asynchronous Launch Policy (std::launch::async):**

* When **std::launch::async** is specified, the task is executed **asynchronously** in a **separate** thread.
* The task begins execution immediately upon invocation of **std::async**, independent of when its result is accessed.
* This policy is suitable for scenarios where the task's execution should not delay the caller's operation.

1. **Deferred Launch Policy (std::launch::deferred):**

* With the **std::launch::deferred** policy, the task is executed **synchronously**, potentially in the **same** thread as the caller.
* The execution of the task is deferred until its result is explicitly requested using the **get()** function on the associated **std::future** object.
* This policy is useful when delaying the task's execution until its result is needed, optimizing resource utilization.

In summary,

* **std::launch::async:** **ensures immediate execution of the task in a separate thread,**
* **std::launch::deferred:** **postpones execution until the task's result is accessed.**

The choice between these launch policies depends on the desired behavior and resource constraints of the application.

# stdfuture Wait Functions

A screen shot of a computer code

Description automatically generated

In this example, we explore the usage of wait functions associated with **std::future**. These functions allow for effective management of asynchronous tasks without blocking the main thread. Here's a breakdown of the key points discussed:

1. **wait() Function:**

* The **wait()** function pauses the current thread until the shared state of the associated future is ready.
* Unlike **get()**, it does not return the shared state but simply waits for it to be ready.
* Before calling **wait()**, it's essential to ensure that the future object is valid.

1. **wait\_for() Function:**

* **wait\_for()** waits for a specific duration for the shared state to become ready.
* It returns an enum **std::future\_status**, indicating whether the future is ready, still running, or has timed out.
* A timeout occurs if the shared state is not ready within the specified duration.

1. **wait\_until() Function:**

* **wait\_until()** is similar to **wait\_for()** but accepts a specific timepoint instead of a duration.
* The timepoint represents a future time at which the wait operation should cease.

1. **Usage Examples:**

* We demonstrate scenarios where **wait\_for()** and **wait\_until()** are useful, such as periodically checking for the readiness of the shared state without blocking the main thread entirely.
* By using these wait functions judiciously, the main thread can remain responsive while monitoring the progress of asynchronous tasks.

By leveraging these wait functions, developers can design more efficient and responsive concurrent applications, ensuring smooth execution and effective resource utilization.

A screen shot of a computer code

Description automatically generated

In this segment of the code, we use the **wait\_for()** function to wait for a maximum duration of 4 seconds for the shared state of the **result** future to become ready. The status of the future's readiness is then checked using a switch statement:

* **std::future\_status::ready:** If the shared state becomes ready within the specified duration, this case is triggered, indicating that the task associated with the future has completed, and its result is available.
* **std::future\_status::timeout:** If the shared state does not become ready within the specified duration, this case is triggered, indicating that the task is still running and has not completed within the allotted time.
* **std::future\_status::deferred:** This case is triggered if the task associated with the future has been deferred and will execute synchronously when its result is requested. In this scenario, the task does not run asynchronously but rather executes in the same thread as the caller.

These status messages provide valuable insights into the progress and behavior of asynchronous tasks, allowing for appropriate actions or responses based on the task's state.

A screen shot of a computer code

Description automatically generated

In this part of the code, we create a timepoint representing the current time using **std::chrono::system\_clock::now()**. We then add a duration of 1 second (**1s**) to this timepoint, effectively setting a point in time 1 second into the future. This timepoint is used as the deadline for waiting for the future's shared state to become ready.

The **wait\_until()** function is called on the **result** future, which waits until the specified timepoint is reached or the shared state becomes ready, whichever occurs first. The status of the future's readiness is then checked using a switch statement:

* **std::future\_status::ready:** If the shared state becomes ready before the specified timepoint is reached, this case is triggered, indicating that the task associated with the future has completed, and its result is available.
* **std::future\_status::timeout:** If the specified timepoint is reached before the shared state becomes ready, this case is triggered, indicating that the task is still running and has not completed before the deadline.
* **std::future\_status::deferred:** This case is triggered if the task associated with the future has been deferred and will execute synchronously when its result is requested. In this scenario, the task does not run asynchronously but rather executes in the same thread as the caller.

These status messages provide useful information about the progress and behavior of asynchronous tasks, enabling appropriate actions or responses based on the task's state and timing.

Top of Form

# Using stdpromise

A white background with black text

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

1. **Promise**:
   * A promise is an object that represents a value that may be available in the future.
   * It serves as a mechanism for storing a value or an exception, known as the shared state of the promise.
   * You can set a value or an exception in the promise, making the shared state available.
   * Once the shared state is set, it becomes accessible to other threads through a future.
   * A promise can only be used once; after setting the shared state, it cannot be modified.
2. **Future**:
   * A future is an object that represents the result of an asynchronous operation.
   * It is used to retrieve the shared state stored in a promise from another thread.
   * Think of it as the receiving end of the communication channel established by the promise.
   * Once the shared state is available, the future provides access to it.
   * After retrieving the shared state, the future becomes invalid and cannot be reused.
3. **Communication Channel**:
   * Promises and futures act as endpoints of a **communication channel**.
   * One operation sets the **shared** state in a **promise**, while another operation **retrieves** **it** asynchronously through a **future**.
   * This communication is synchronized, ensuring thread safety without the need for explicit synchronization mechanisms.
4. **Thread Safety**:
   * Promises and futures provide **a thread-safe way to share** data between different threads.
   * Data is set in one thread using a promise and retrieved in another thread using a future.
   * This ensures that concurrent access to the shared state is handled safely without the risk of data races or undefined behavior.

Overall, promises and futures are powerful tools for asynchronous programming, allowing you to write efficient and thread-safe code by separating the production and consumption of data in concurrent applications.

# using namespace std::chrono\_literals

The line **using namespace std::chrono\_literals;** allows you to use the literals defined in the **std::chrono** namespace directly in your code without explicitly qualifying them with **std::chrono::**.

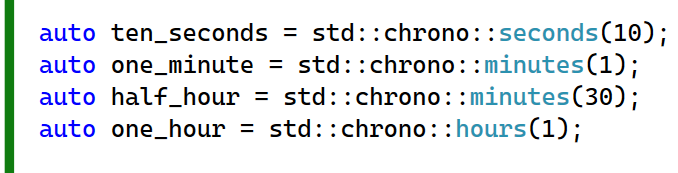
In C++11 and later, the **<chrono>** header provides facilities for working with time durations and time points. It introduces user-defined literals for representing time durations in a human-readable way.

Here's how you can use it:

A computer screen shot of a code

Description automatically generated

Without the **using namespace std::chrono\_literals;** line, you would need to qualify the literals with **std::chrono::** like this:



Using the **using namespace std::chrono\_literals;** directive makes the code more concise and readable by allowing you to use the literals directly. However, it's important to use it judiciously to avoid potential naming conflicts.

Top of Form

# Using stdpromise

A screenshot of a computer program

Description automatically generated

A detailed explanation of how to use **std::promise** and **std::future** to share data between threads asynchronously. Here's a summary of the steps involved:

1. Create a **std::promise** object to hold the future value.
2. Pass the **std::promise** object to the function (**Operation**) that will eventually set the value.
3. Use **std::ref** to pass the **std::promise** object by reference.
4. In the function (**Operation**), get the **std::future** object from the **std::promise**.
5. Wait for the value to be set using **future.get()**. This call will block until the value is available.
6. Set the value in the **std::promise** object after some delay.
7. The function (**Operation**) will continue its work after the value is set.
8. Once the value is available, **future.get()** returns, and the shared state can be acquired.

This approach allows for asynchronous communication between threads without manual thread synchronization, improving the efficiency and readability of the code. Additionally, it demonstrates the use of promises and futures for low-level concurrency in C++.



The line **auto f = data.get\_future();** is a common usage pattern when working with **std::promise** and **std::future** in C++.

Here's what it does:

1. **data** is likely an instance of **std::promise**, which is used to set a value or an exception in a thread-safe manner. A **std::promise** is associated with a **std::future**, which represents the result of an asynchronous operation.
2. **get\_future()** is a member function of **std::promise** that **returns a std::future object** associated with the promise. This future can be used to retrieve the value or exception set by the promise at a later point in time.
3. **auto f = data.get\_future();** declares a variable **f** using the **auto** keyword, which allows the compiler to deduce the type of **f** from the return type of **get\_future()**. In this case, **f** will be of type **std::future**.

By calling **get\_future()** on a **std::promise**, you create a corresponding **std::future** that can be used to asynchronously retrieve the result of the operation that the promise represents. This is a fundamental mechanism for communication between threads in C++ when dealing with asynchronous operations.

Top of Form

  
The line **auto count = f.get();** retrieves the value that was set in the associated **std::promise** object. Here's what it does:

1. **f** is a **std::future** object obtained from a **std::promise** using the **get\_future()** function, as you mentioned earlier. A **std::future** represents a value or an exception that will be available in the future.
2. **get()** is a member function of **std::future** that blocks the current thread until the value becomes available. When the value is available, **get()** returns the stored value.
3. **auto count = f.get();** declares a variable **count** using the **auto** keyword, which allows the compiler to deduce the type of **count** from the type of the value returned by **get()**.
4. After this line executes, **count** will contain the value that was set in the associated **std::promise** object earlier. If the value has not been set yet, the call to **get()** will block until the value becomes available.

This mechanism allows you to synchronize between different threads by allowing one thread to set a value using a **std::promise**, while another thread can wait for and retrieve that value using a corresponding **std::future**



This line of code demonstrates the usage of **std::async** to asynchronously execute a function (**Operation**) with a specified launch policy (**std::launch::async**) and pass arguments to it. Here's a breakdown:

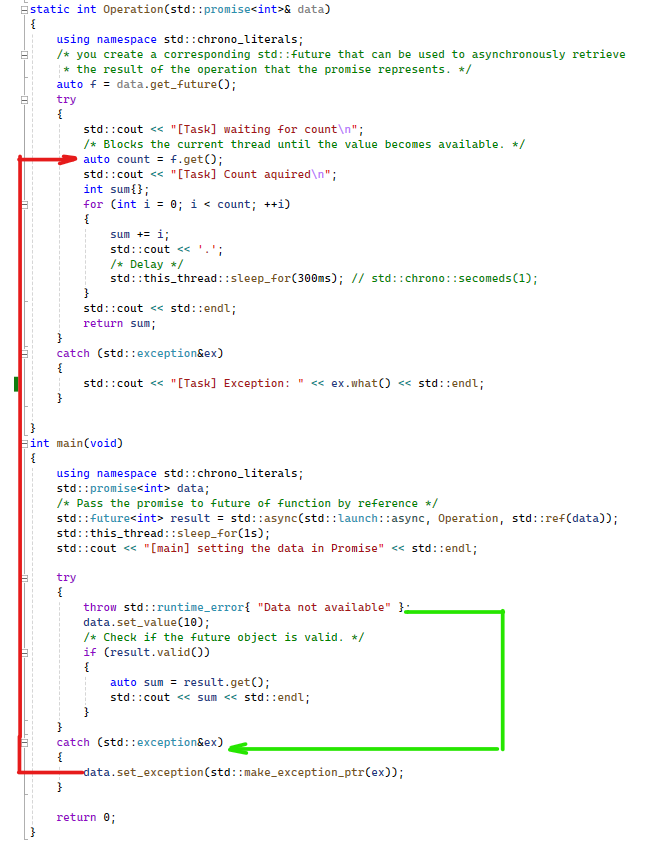
1. **std::future<int>**: This declares a future object named **result** that will eventually hold the result of the asynchronous operation. In this case, it is expected to hold an integer value.
2. **std::async**: This function is used to asynchronously execute a function. It takes the following arguments:

* **std::launch::async**: This is the launch policy indicating that the function should be executed asynchronously.
* **Operation**: This is the function to be executed asynchronously.
* **std::ref(data)**: This is an argument to be passed to the function **Operation**. **std::ref** is used to wrap **data** as a **std::reference\_wrapper** so that it can be passed by reference.

1. **Operation**: This is the function that will be executed asynchronously. It should accept **data** as an argument.
2. After executing this line, **result** will hold a future that will eventually contain the result of the execution of **Operation** once it completes asynchronously.

Overall, **std::async** is a convenient way to perform asynchronous operations in C++, allowing you to execute functions concurrently and obtain their results via future objects.

# Propagating Exceptions Across Threads



This excerpt explains how **std::promise** can be used to propagate exceptions between threads. Here's a summary of the steps involved:

1. Create a **std::promise** object to hold the result or exception.
2. Pass the **std::promise** object to the function (**Operation**) that may throw an exception.
3. Use **std::ref** to pass the **std::promise** object by reference.
4. In the function (**Operation**), write the code that may throw an exception inside a try-catch block.
5. If an exception is caught, create an exception pointer using **std::make\_exception\_ptr(exception)** and set it inside the promise using **set\_exception(exception\_ptr)**.
6. When waiting on the future associated with the promise, if an exception is set in the promise, **future.get()** will throw the exception.
7. The exception can be caught and handled in the calling thread.

This approach allows exceptions to be propagated between threads without manual synchronization. Additionally, it demonstrates the use of promises and futures for asynchronous exception handling in C++.

Promises provide a mechanism for communicating exceptions between threads. If a value that needs to be set in a promise is not available, an exception can be thrown, and the promise can propagate that exception to another thread through a future object.

To achieve this, you can use a try-catch block in the task where you catch the exception. Then, you create an exception pointer using **std::make\_exception\_ptr()** and set it inside the promise using the **set\_exception()** function. This exception pointer serves as a shared pointer for exceptions, allowing the exception to be shared between different threads.

When waiting on the shared state of the promise, if an exception is set, calling **get()** on the future will throw that exception, which can then be caught and handled in the receiving thread.

The reverse scenario is also possible, where the task itself throws an exception. In this case, the exception is automatically propagated to the main thread when calling **get()** on the future, without the need to manually create an exception pointer inside the task.

In summary, promises facilitate the sharing of data between threads, including exceptions, without the need for manual thread synchronization, making concurrent programming more manageable and efficient.