MQF 633: C++ For Financial Engineering

Lecture 8: C++ Async, Multi-threading

Part I: Asynchronous Programming and Multi-threading basics

Synchronous programming

In synchronous programming, tasks are executed one after another, in a sequential, blocking manner. If any step takes a long time, you're essentially standing there, doing nothing else, just waiting for that step to complete. In a program, this means the entire application might freeze, become unresponsive, or block the execution of other useful work.

Asynchronous programming

In asynchronous programming, tasks can be initiated and then the program can move on to other tasks while the initiated task runs in the background. When the background task is done, it notifies the main program, and the program can then process the result.
It refers to the occurrence of events independent of the main program flow and ways to deal with such events. These may be "outside" events such as the arrival of signals, or actions instigated by a program that take place concurrently with program execution, without the program hanging to wait for results. Asynchronous input/output is an example of the latter case of asynchrony, and lets programs issue commands to storage or network devices that service these requests while the processor continues executing the program. Doing so provides a degree of parallelism.
A common way for dealing with asynchrony in a programming interface is to provide subroutines that return a future or promise that represents the ongoing operation, and a synchronising operation that blocks until the future or promise is completed.

Examples of asynchrony include the following:

- Asynchronous procedure call, a method to run a procedure concurrently, a lightweight alternative to threads.
- Ajax is a set of client-side web technologies used by the client to create asynchronous I/O web applications.
- Asynchronous method dispatch (AMD), a data communication method used when there is a need for the server side to handle a large number of long lasting client requests.Using synchronous method dispatch (SMD), this scenario may turn the server into an unavailable busy state resulting in a connection failure response caused by a network connection request timeout. The servicing of a client request is immediately dispatched to an available thread from a pool of threads and the client is put in a blocking state. Upon the completion of the task, the server is notified by a callback. The server unblocks the client and

transmits the response back to the client. In case of thread starvation, clients are blocked waiting for threads to become available.

Thread vs Process

In operating systems, both **processes** and **threads** are fundamental units of execution, but they differ significantly in terms of their independence, resource ownership, and how they share resources. Understanding their distinctions is crucial for designing efficient and robust concurrent applications.

Process

A **process** is an independent execution environment that represents a running instance of a program. When you launch an application (e.g., open a web browser, start a word processor), the operating system creates a new process for it.

Key Characteristics of a Process:

1. **Independent Memory Space (Virtual Address Space):** Each process has its own isolated and dedicated virtual memory space. This means that one process cannot directly access the memory of another process without explicit inter-process communication (IPC) mechanisms. This isolation provides strong security and stability, as a crash in one process typically does not affect others.
2. **Resource Ownership:** A process owns its own set of resources, including:
   - **Code Segment:** The executable program instructions.
   - **Data Segment:** Global variables and static data.
   - **Heap:** Dynamically allocated memory during runtime.
   - **Stack:** Used for local variables and function calls (though each thread within the process has its own stack).
   - **File Descriptors/Handles:** Open files, network connections, etc.
   - **Process ID (PID):** A unique identifier assigned by the OS.
   - **Security Context:** User and group IDs, permissions.
3. **Heavyweight**:
   - **Creation Overhead:** Creating a new process is relatively "heavy" or expensive in terms of time and resources because the OS needs to allocate a completely new, independent memory space and set up all the associated resources.

- **Context Switching Overhead:** Switching between processes (when the OS decides to run a different process) is also expensive. It involves saving the entire state of the current process (CPU registers, memory maps, etc.) and loading the state of the new process. This often involves flushing CPU caches (like the Translation Lookaside Buffer - TLB) because the virtual memory mapping changes.

4. **Inter-Process Communication (IPC):** Since processes are isolated, they need specific mechanisms to communicate with each other, such as:
    - Pipes (named and unnamed)
    - Sockets (for network communication, even on the same machine)
    - Shared Memory
    - Message Queues
    - Semaphores

Analogy: Think of a process as a **separate house.** Each house has its own rooms (memory), its own furniture (resources), and its own set of rules. People in one house cannot directly walk into another house and take things without a deliberate form of communication (like sending a letter or calling).

Thread

A **thread** (or "thread of execution") is the smallest unit of execution within a process. Every process has at least one thread, known as the main thread. A process can have multiple threads running concurrently.

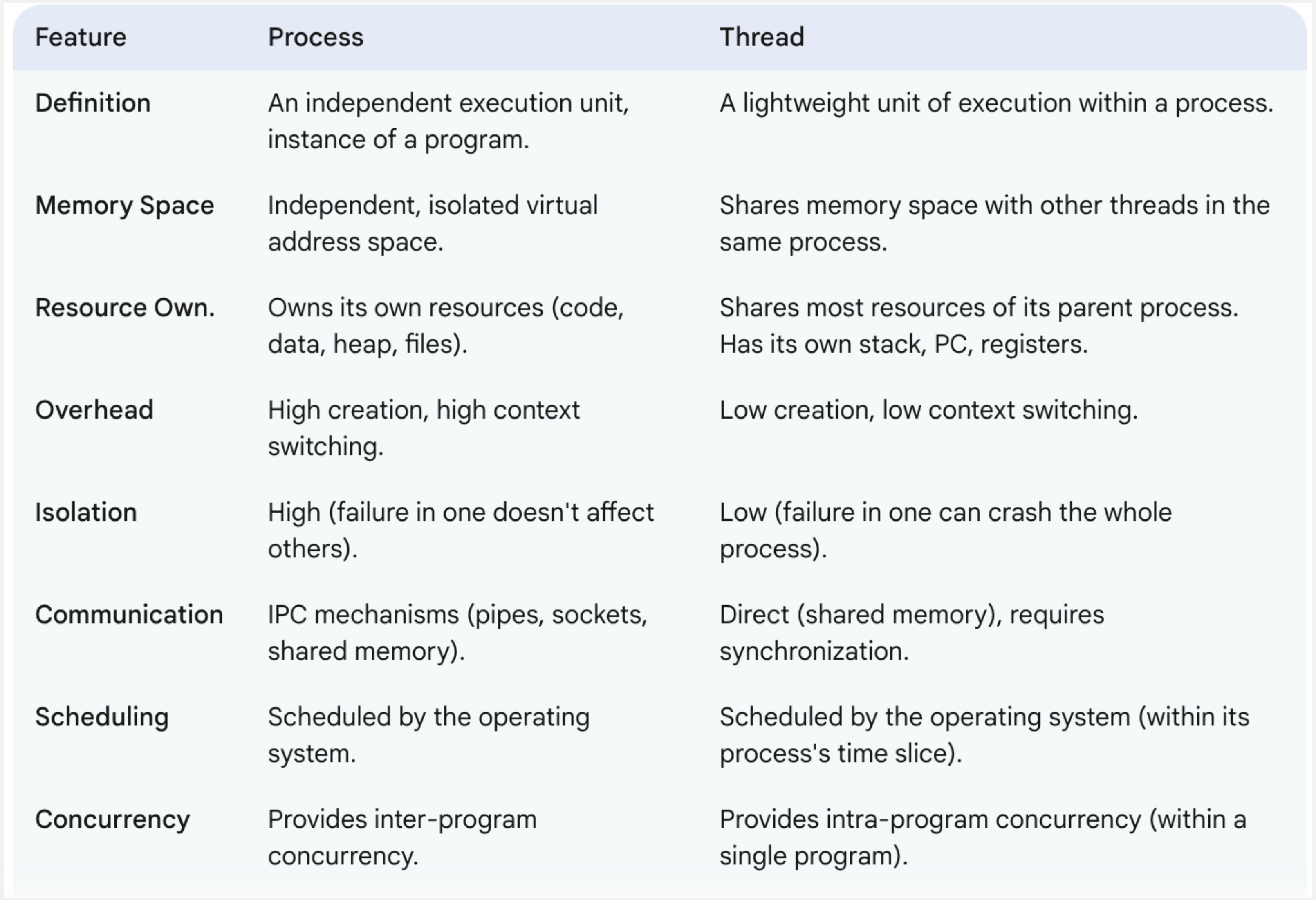
Key Characteristics of a Thread:

1. **Shared Memory Space:** All threads within the same process share the process's memory space. This includes the code segment, data segment, and heap. This shared memory makes communication between threads very efficient, as they can directly access shared variables.
2. **Individual Components:** While threads share most of the process's resources, each thread has its own:
    - **Program Counter (PC):** Keeps track of the next instruction to execute.
    - **Stack:** Used for local variables and function calls within that specific thread.
    - **Registers:** CPU registers (including stack pointer).
    - **Thread ID (TID):** A unique identifier within the process.

3. **Lightweight**:
    - **Creation Overhead:** Creating a new thread is much "lighter" or less expensive than creating a process because threads share the parent process's resources and memory. The OS only needs to set up a new stack, program counter, and registers.
    - **Context Switching Overhead:** Switching between threads within the same process is much faster than switching between processes. The memory context (virtual address space) doesn't change, so CPU caches (like TLB) don't need to be flushed as often.
4. **Easy Communication:** Because threads share the same memory space, communication between them is direct and efficient. They can simply read and write to shared variables. However, this also introduces the need for **synchronisation mechanisms** (e.g., mutexes, semaphores, condition variables) to prevent race conditions and ensure data consistency when multiple threads access shared resources simultaneously.
5. **Failure Impact:** If one thread within a process crashes due to an unhandled error (e.g., a segmentation fault), it typically brings down the entire process, affecting all other threads within that process.

Analogy: Think of threads as **individuals living in the same house (process)**. They share the house's common areas (shared memory like kitchen, living room, and shared utilities) and resources (like the main electricity supply or water). Each person has their own bedroom (stack) and their own personal to-do list (program counter) but can easily talk to others in the house or access shared items. If one person accidentally burns down the kitchen, the whole house is affected.

Refer to thread.cpp example



Feature	Process	Thread
Definition	An independent execution unit, instance of a program.	A lightweight unit of execution within a process.
Memory Space	Independent, isolated virtual address space.	Shares memory space with other threads in the same process.
Resource Own.	Owns its own resources (code, data, heap, files).	Shares most resources of its parent process. Has its own stack, PC, registers.
Overhead	High creation, high context switching.	Low creation, low context switching.
Isolation	High (failure in one doesn't affect others).	Low (failure in one can crash the whole process).
Communication	IPC mechanisms (pipes, sockets, shared memory).	Direct (shared memory), requires synchronization.
Scheduling	Scheduled by the operating system.	Scheduled by the operating system (within its process's time slice).
Concurrency	Provides inter-program concurrency.	Provides intra-program concurrency (within a single program).

std::mutex

std::mutex is a fundamental synchronisation primitive in C++ (introduced in C++11 as part of the **<mutex>** header). Its name, "mutex," is short for mutual exclusion. Its primary purpose is to protect shared data from being simultaneously accessed by multiple threads, thereby preventing race conditions and ensuring thread safety.

How std::mutex Works

std::mutex provides an exclusive, non-recursive ownership semantic. This means:

1. **Exclusive Ownership:** Only one thread can "own" (lock) a std::mutex at any given time.
2. **Blocking:** If a thread tries to lock a std::mutex that is already owned by another thread, the requesting thread will block (pause its execution) until the owning thread unlocks the mutex.
3. **Non-Recursive:** A std::mutex cannot be locked multiple times by the same thread without first unlocking it. Attempting to do so by the same thread will lead to undefined behaviour (often a deadlock). For recursive locking, you would use std::recursive_mutex.

By strategically placing std::mutex locks around sections of code that access shared data (known as critical sections), you ensure that only one thread can execute that critical section at a time.

Refer to mutex.cpp

Using std::mutex to solve racing conditions

What is Racing Condition?
A racing condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads **at any time**, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Refer to racing.cpp

```
void some_task(int& x, int&y) {
    if (x == 5) // The "Check"
    {
        y = x * 2; // The "Act"
        // If another thread changed x in between "if (x == 5)" and "y = x * 2" above, y will not be equal to 10.
    }
    cout<< std::this_thread::get_id()<<":" << "x: " << x << ", y: " << y <<endl;
}

void ThreadA(int& x, int& y)
{
    std::cout << "Thread A is executing" << std::endl;
    x = 5;
    some_task(x, y);
}

void ThreadB(int& x, int& y)
{
    std::cout << "Thread B is executing" << std::endl;
    some_task(x, y);
}
```

The some_task() will check x value and calculate Y if x ==5. When this task is invoke from thread A or thread B from main thread, the x value can be either 0 or 5, since we don't know exactly which thread is executing and change the value of x first. As a result, the Y value might not be the appears as the code. This is one example of racing condition. In order to prevent race conditions from occurring, you would typically put a lock around the shared data, x, to ensure only one thread can access the data at a time.

Using std::cout directly in a multi-threaded environment without synchronization will lead to race conditions, resulting in garbled or interleaved output. This happens because std::cout (which is an instance of std::ostream) is not inherently thread-safe at the

character-by-character or even line-by-line level for concurrent writes. Multiple threads might write to the underlying buffer at the same time, leading to corrupted output.
To avoid racing when printing with std::cout in multi-threaded C++, you need to protect the access to the stream using a synchronization mechanism. The most common and idiomatic ways are:

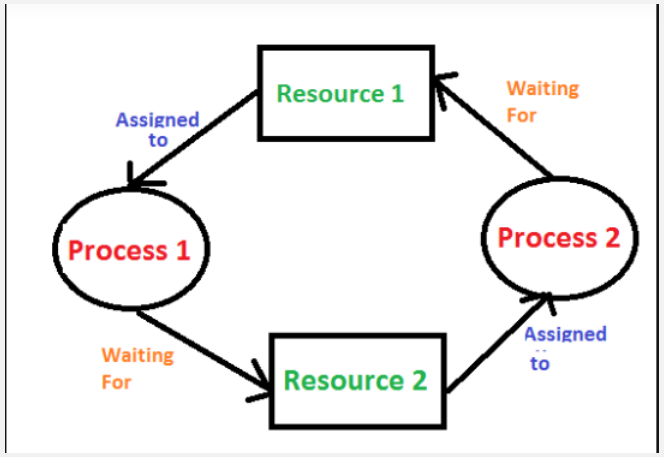
1. **Using a std::mutex** with std::lock_guard (C++11 and later): This is the most direct and widely applicable method.
2. **Using std::osyncstream** (C++20 and later): This is the modern, built-in solution specifically designed for synchronized output.

Let's look at examples for both.
1. Using std::mutex with std::lock_guard (C++11/14/17 compatible)
This is the most common approach for older C++ standards or when you need fine-grained control.

Dead lock

A process in operating system uses resources in the following way.

- Requests a resource
- Use the resource
- Releases the resource



In concurrent programming (where multiple threads or processes run seemingly simultaneously), a deadlock is a specific and highly problematic situation where two or more competing actions are each waiting for the other to finish, and thus none of them ever finish. It's like a perpetual stalemate, leading to a program or system becoming unresponsive or "hung."
Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

Refer to dead_lock.cpp example

How to Prevent This Deadlock (Common Solutions)

The most common way to prevent this specific type of deadlock is to **establish a consistent ordering for acquiring locks**.

Solution 1: Consistent Lock Ordering
Ensure that all threads acquire the mutexes in the *same* fixed order.

Refer to dead_lock1.cpp example

In this version, if Thread 1 acquires g_mutex1 first, Thread 2 will block when it tries to acquire g_mutex1. Thread 1 will eventually release both locks, allowing Thread 2 to proceed. There's no circular waiting.

Solution 2: Using std::scoped_lock (C++17 and later)
std::scoped_lock is designed specifically to prevent deadlocks when acquiring multiple mutexes by locking them all at once (or none at all) using an internal algorithm to avoid circular dependencies.

Refer to dead_lock2.cpp example

Thread Safety

In multi-threaded computer programming, a function is thread-safe when it can be invoked or accessed concurrently by multiple threads without causing unexpected behaviour, race conditions, or data corruption. As in the multi-threaded context where a program executes several threads simultaneously in a shared address space and each of those threads has access to all every other thread's memory, thread-safe functions need to ensures all those threads behave properly and fulfils their design specifications without unintended interaction.

The most commonly use thread-safety terminology are:

1. **Not thread safe**: Data structures should not be accessed simultaneously by different threads.
2. **Thread safe, serialisation**: Use a single mutex for all resources to guarantee the thread to be free of race conditions when those resources are accessed by multiple threads simultaneously.
3. **Thread safe, MT-safe**: Use a mutex for every single resource to guarantee the thread to be free of race conditions when those resources are accessed by multiple threads simultaneously.

The core goal of thread safety is to prevent data races and ensure that program invariants (conditions that should always be true about your data) are maintained, regardless of how threads are scheduled by the operating system.

Key Tools and Concepts for Thread Safety in C++

C++ (especially since C++11) provides a rich set of tools in its Standard Library to achieve thread safety:

1. **Mutual Exclusion (Mutexes):**
   - std::mutex: The most fundamental synchronization primitive. It provides exclusive ownership. Only one thread can acquire (lock) a std::mutex at a time. If another thread tries to lock it, it blocks until the mutex is released (unlocked).
   - **RAII Wrappers (std::lock_guard, std::unique_lock, std::scoped_lock):** These are crucial for safe mutex usage. They acquire the mutex upon construction and guarantee its release upon destruction (when they go out of scope), even if an exception occurs. This prevents common errors like forgetting to unlock or deadlocks due to exceptions.
     - std::lock_guard: Simple, non-movable, non-copyable. Acquires lock on construction, releases on destruction.
     - std::unique_lock: More flexible, movable, allows deferred locking, timed locking, and explicit unlock/relock.

        - std::scoped_lock (C++17): Designed to lock multiple mutexes simultaneously without causing deadlocks, by using a deadlock-avoidance algorithm (e.g., ordering the locks internally).
    - **Other Mutex Types:**
        - std::recursive_mutex: Allows the same thread to acquire the mutex multiple times.
        - std::timed_mutex, std::recursive_timed_mutex: Allow attempts to lock with a timeout.

2. **Atomic Operations (<atomic> header):**
    - std::atomic<T>: Provides atomic operations for fundamental data types (like int, bool, pointers, etc.) without requiring explicit locks. An operation on an std::atomic variable is guaranteed to complete in a single, indivisible step from the perspective of other threads.
    - Useful for simple counters, flags, or constructing more complex lock-free algorithms.
    - Example: std::atomic<int> counter = 0; then counter++; (this ++ is now an atomic operation).
    - Atomic operations are typically faster than mutexes for simple operations on small data types because they often map directly to special CPU instructions and avoid operating system calls for context switching.
    - They also involve complex memory ordering semantics (e.g., memory_order_relaxed, memory_order_acquire, memory_order_release, memory_order_seq_cst) that control how memory operations are reordered by the compiler and CPU, affecting visibility of changes across threads.
3. **Condition Variables (std::condition_variable):**
    - Used to synchronize threads based on a condition. One thread can wait on a condition variable until another thread signals that the condition has been met.
    - Always used in conjunction with a std::unique_lock and a shared mutable state (protected by a mutex).
    - Use case: Producer-consumer patterns, waiting for a task to complete, etc.
4. **Futures and Promises (std::future, std::promise, std::async):**
    - Provide a higher-level abstraction for managing asynchronous operations and retrieving their results.
    - A std::promise sets a value or an exception, and a std::future retrieves that value or exception.
    - std::async allows you to run a function asynchronously and get a std::future to its result.
5. **std::call_once and std::once_flag:**
    - Guarantees that a function (e.g., for initialization) is called exactly once, even if multiple threads try to call it concurrently. Useful for lazy initialization of singletons or shared resources.
6. **Thread-Local Storage (thread_local keyword):**

    - Declares a variable that has independent storage for each thread. Each thread gets its own copy of the variable, so there are no shared state issues to worry about for that variable.
    - Example: thread_local int thread_specific_counter = 0;

Designing for Thread Safety

Achieving thread safety often involves a combination of these techniques and careful design:

- **Identify Shared Mutable State:** The first step is to identify all data that can be accessed and modified by multiple threads. This is where race conditions can occur.
- **Minimize Shared State:** Design your architecture to reduce the amount of shared mutable state as much as possible. Prefer passing data by value, using immutable data structures, or making data thread-local when feasible.
- **Encapsulate Synchronization:** Encapsulate shared data along with the necessary synchronization mechanisms (e.g., a mutex) within a class. This hides the synchronization details from the users of the class, making it easier to use correctly.
- **Consistent Locking Policy:** If using mutexes, ensure all threads acquire locks in a consistent order to prevent deadlocks. std::scoped_lock can help with this for multiple locks.
- **Avoid Excessive Locking:** While synchronization is necessary, too much locking can reduce concurrency and lead to performance bottlenecks. Use the most appropriate and least restrictive synchronization primitive for the task.
- **Consider Lock-Free Programming:** For very high-performance scenarios or specific data structures, lock-free programming using atomics might be considered. However, this is significantly more complex and error-prone.
- **Static Analysis Tools:** Use static analysis tools (like Clang Thread Safety Analysis) to help identify potential race conditions and deadlocks at compile time.
- **Testing**: Thoroughly test multi-threaded code under various load conditions to uncover race conditions, which are often non-deterministic and hard to reproduce.

Thread safety is a complex but essential aspect of modern C++ development for building robust and performant concurrent applications.

Lvalue and Rvalue

Originally, **“L-value”** refers to a memory location that identifies an object. **“R-value”** refers to the data value that is stored at some address in memory. References in C++ are nothing but the alternative to the already existing variable. They are declared using the ‘&’ before the name of the variable.

In C++03, an expression is either an rvalue or an lvalue. In C++11, an expression can be an:

- rvalue

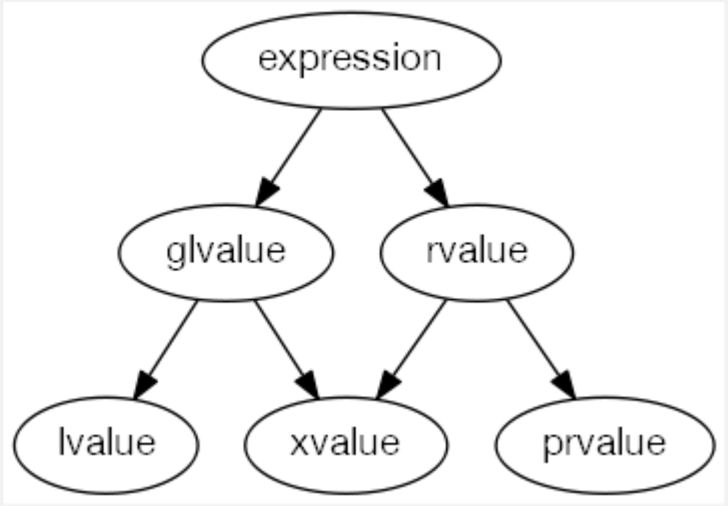
An rvalue (so-called, historically, because rvalues could appear on the right-hand side of an assignment expression) is an xvalue, a temporary object or sub object thereof, or a value that is not associated with an object.

- lvalue

An lvalue (so-called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [Example: If E is an expression of pointer type, then *E is an lvalue expression referring to the object or function to which E points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue.]

- xvalue

An xvalue (an “**eXpiring**” value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references. [Example: The result of calling a function whose return type is an rvalue reference is an xvalue.]


$$\begin{tikzcd}
 & & \text{expression} \arrow[dl] \arrow[dr] & & \\
 & \text{glvalue} \arrow[dl] \arrow[dr] & & \text{rvalue} \arrow[dl] \arrow[dr] & \\
\text{lvalue} & & \text{xvalue} & & \text{prvalue}
\end{tikzcd}$$

- glvalue

A glvalue (“generalized” lvalue) is an lvalue or an xvalue.

- prvalue

A prvalue (“pure” rvalue) is an rvalue that is not an xvalue. [Example: The result of calling a function whose return type is not a reference is a prvalue]

Example: basic use of lvalue and rvalue

```
// C++ program to illustrate the lvalue and rvalue
#include <iostream>
using namespace std;

// Driver Code
int main()
{
        int a = 10;

        // Declaring lvalue reference (i.e variable a)
        int& lref = a;

        // Declaring rvalue reference
        int&& rref = 20;

        // Print the values
        cout << "lref = " << lref << endl;
        cout << "rref = " << rref << endl;

        // Value of both a and lref is changed
        lref = 30;

        // Value of rref is changed
        rref = 40;
        cout << "lref = " << lref << endl;
        cout << "rref = " << rref << endl;

        // This line will generate an error as l-value cannot be assigned to the r-value references
        // int &&ref = a;
        return 0;
}
```

Example 2: use of the lvalue references:

1. lvalue references can be used to alias an existing object.
2. They can also be used to implement pass-by-reference semantics.

```
// C++ program to illustrate lvalue
#include <iostream>
using namespace std;

// Creating the references of the parameter passed to the function
void swap(int& x, int& y)
{
        int temp = x;
        x = y;
        y = temp;
}

// Driver Code
int main()
{
        // Given values
        int a{ 10 }, b{ 20 };
        cout << "a = " << a << " b = " << b << endl;

        // Call by Reference
        swap(a, b);

        // Print the value
        cout << "a = " << a << " b = " << b << endl;
        return 0;
}
```

Uses of rvalue references:

1. They are used in working with the move constructor and move assignment.
2. Cannot bind non-const lvalue reference of type ‘int&‘ to an rvalue of type ‘int’.
3. Cannot bind rvalue references of type ‘int&&‘ to lvalue of type ‘int’.

```
// C++ program to illustrate rvalue
#include <iostream>
using namespace std;

// lvalue reference to the lvalue passed as the parameter
void printReferenceValue(int& x)
{
        cout << x << endl;
}
// rvalue reference to the rvalue passed as the parameter
void printReferenceValue2(int&& x)
{
        cout << x << endl;
}

int main()
{
        // Given value
        int a{ 10 };

        // Function call is made lvalue & can be assigned to lvalue reference
        printReferenceValue(a);

        // Works fine as the function is called with rvalue
        printReferenceValue2(100);

        return 0;
}
```

Part II: Async in C++

std::async in STL

As the name indicates, C++ std::async is a function template, which takes functions or function objects as arguments (basically called callbacks) and runs them asynchronously. It returns the std:: the future object which is used to keep the result of the above function. The result is stored in the shared state. In order to exact the value from it, its member future:: get needs to be called by the programmer.

In C++, async functions are used in 2 ways, i.e. with or without specifying the policies in the function arguments. When specifying the launch policy, the first argument is the policy itself which defines the asynchronous behaviour of the function.

Syntax

Below given is the basic syntax of how the async function is used in C++ programs:

1. Without specifying the policy

```
template <class function (Fn), class... Args>
std::future<typename std::result_of<Fn(Args…)>::type>
async (Function && fn, Args&&... args);
```

In the above syntax, the launch policy is not specified in the function arguments. Launch policy is automatically selected, which is launch:: async | launch:: deferred.

2. Specifying the policy

```
template <class function(Fn), class... Args>
std::future<typename std::result_of<Fn(Args...)>::type>
async (launch policy, Function && fn, Args&&... args);
```

In the above syntax, launch policy is mentioned in the function arguments in order to specify before in which policy a function should execute.

Fn: It is the callable object or the function object. The return value of this function 'fn' is stored in the shared state, which is accessed by the object 'future'. In case of exceptions also, the value is set in the shared state, which the future object can also access.

args: It is the arguments or the parameters which are passed in the async function 'fn'.

One important point to note in the async function is that both the function 'Fn' and argument 'args' should be move constructible, and the function uses the decay copies of both the Fn and args.

Policy: Policy in C++ async plays an important role in the one which defines the asynchronous behaviour followed using the function async. There are basically 3 ways in which create async using different policies:

S.No	Policy name	Behavior
1.	launch::async	This launch policy assures the async behavior of the function, which means that the callable function will be executed in a new thread. It follows the easy evaluation policy in which calculation will be evaluated immediately in the work package in the new thread. In case of exception, it is stored in the shared state, which is accessed by std::future.
2.	launch::deferred	In this launch policy, a callable function is not executed in the new thread; instead, it follows the non-async behavior. It follows the lazy evaluation policy in which the call to the function is deferred (postponed) till the previous thread calls the get on the future object, which makes the shared state again accessible. The function then will no longer be deferred. In this case, the function will run its work package in the same thread. In case of exceptions, it is placed in the shared state of the future; then, it is made ready for the required function.
3.	launch::async l launch::deferred	This is an automatic launch policy. In this policy, the behavior is not defined. The system can choose either asynchronous or deferred depending on the implementation according to the optimized availability of the system. Programmers have no control over it on anything.

Return Value: The return value of async is the std:: future which is the shared state which is created by the function call of std::async. We can get the values from it using member **future::get** returned by the function.

```
// Example of checking the number is even or not using async
#include <iostream>       // library used for std::cout
#include <future>         // library used for std::async and std::futur
// function for checking the number is even or not
bool check_even (int num) {
std::cout << "Hello I am inside the function!! \n";
//checking the divisibility of number by 2 and returning a bool value
if(num%2==0) {
return true;
}
return false;
}
int main () {
// calling the above function check_even asynchronously and storing the result in future object
std::future<bool> ft = std::async (check_even,10);
std::cout << "Checking whether the number 10 is even or not.\n";
// retrieving the exact value from future object and waiting for check_even to return
bool rs = ft.get();
if (rs) {
std::cout << "Number mentioned by you is even \n";
}
else {
std::cout << "Sorry the number is odd \n";
}
return 0;
}
```

Quiz for part I & II

1. What does std::async provide in C++?
   A) A mechanism for creating threads.
   B) Asynchronous execution of a function.
   C) Synchronous execution of a function.
   D) A tool for memory management.

2. How is std::async different from std::thread?
   A) std::async can only execute functions asynchronously.
   B) std::async returns a future to obtain the result of the function execution.
   C) std::async cannot execute member functions of classes.
   D) std::async is more lightweight than std::thread.

3. Which of the following is true about std::async?
   A) The function is executed immediately upon calling std::async.
   B) std::async returns a std::thread object.
   C) The default launch policy is std::launch::async.
   D) std::async blocks the calling thread until the function completes.

4. How can you specify the launch policy for std::async?
   A) By passing a launch policy parameter to std::async.
   B) By setting a global launch policy variable.
   C) Launch policy cannot be specified for std::async.
   D) Launch policy is always std::launch::async.

5. What happens if an exception is thrown in the function executed by std::async?
   A) The exception is propagated to the caller when calling .get() on the future.
   B) The exception is silently ignored.
   C) The program crashes.
   D) The exception is handled internally by std::async.

6. Which of the following launch policies for std::async ensures deferred execution?

A) std::launch::async
B) std::launch::deferred
C) std::launch::sync
D) std::launch::parallel

7. What does deferred execution mean in the context of std::async?
A) The function is executed immediately upon calling std::async.
B) The function is executed asynchronously.
C) The function is executed synchronously.
D) The function is executed only when .get() is called on the future.

8. Which of the following is true about the return value of std::async?
A) It always returns a value.
B) It returns a std::future to obtain the result of the function execution.
C) It returns a std::shared_ptr to the result of the function execution.
D) It returns void.

Part III: C++ thread and multi-threading

std::thread

Multithreading is a feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU. Each part of such a program is called a thread. So, threads are lightweight processes within a process. Multithreading support was introduced in C++11. Prior to C++11, we had to use POSIX threads or <pthreads> library. While this library did the job the lack of any standard language-provided feature set caused serious portability issues. C++ 11 did away with all that and gave us std::thread. The thread classes and related functions are defined in the <thread> header file.

Syntax: std::thread thread_object (callable);

std::thread is the thread class that represents a single thread in C++. To start a thread we simply need to create a new thread object and pass the executing code to be called (i.e, a callable object) into the constructor of the object. Once the object is created a new thread is launched which will execute the code specified in callable. A callable can be any of the five:

- A Function Pointer
- A Lambda Expression
- A Function Object
- Non-Static Member Function
- Static Member Function

Note : A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared. Suppose a static member function f() is a member of class X . The static member function f() cannot access the nonstatic members X or the nonstatic members of a base class of X .

1. Example of launching thread on function pointer

```
void foo(param) {
   Statements;
}
// The parameters to the function are put after the comma
std::thread thread_obj(foo, params);
```

2. Example of launching thread on lambda expression

```
// Define a lambda expression
auto f = [] (params) {
   Statements;
};
// Pass f and its parameters to thread object constructor as
std::thread thread_object(f, params);
```

3. Example of launching thread on callable object

```
// Define the class of function object
class fn_object_class {
        // Overload () operator
        void operator() (params) {
        Statements;
        }
}
// Create thread object
std::thread thread_object(fn_object_class(), params)
```

4. Example of launching thread on class object function

```
// defining class
class Base {
public:
        // non-static member function
        void foo(param) { Statements; }
}
```

```
// object of Base Class
Base b;

// first parameter is the reference to the function and second paramter is reference of the object at last we have arguments
std::thread thread_obj(&Base::foo, &b, params);
```

Wait for thread to finish

Once a thread has started we may need to wait for the thread to finish before we can take some action. For instance, if we allocate the task of initializing the GUI of an application to a thread, we need to wait for the thread to finish to **ensure** that the GUI has loaded properly. To wait for a thread, use the std::thread::join() function. This function makes the current thread wait until the thread identified by *this has finished executing.

```
int main()
{
// Start thread t1
        std::thread t1(callable);

// Wait for t1 to finish
        t1.join();
}
// t1 has finished do other stuff;
return 0;
}
```

```
// C++ program to demonstrate multithreading using three different callables.
#include <iostream>
#include <thread>
using namespace std;

// A dummy function
void foo(int Z) {
        for (int i = 0; i < Z; i++) {
                cout << "Thread using function pointer as callable\n";
        }
}

// A callable object
class thread_obj {
public:
        void operator()(int x) {
                for (int i = 0; i < x; i++)
                        cout << "Thread using function object as callable\n";
        }
};

// class definition
class Base {
public:
  // non-static member function
  void foo(){
    cout << "Thread using non-static member function "
        "as callable"
      << endl;
  }
  // static member function
  static void foo1() {
    cout << "Thread using static member function as callable"  << endl;
  }
};
```

```
// Driver code
int main()
{
   cout << "Threads 1 and 2 and 3  operating independently"   << endl;

   // This thread is launched by using function pointer as callable
   thread th1(foo, 3);

   // This thread is launched by using function object as callable
   thread th2(thread_obj(), 3);

   // Define a Lambda Expression
   auto f = [](int x) {
       for (int i = 0; i < x; i++)
           cout << "Thread using lambda expression as callable\n";
   };

   // This thread is launched by using lambda expression as callable
   thread th3(f, 3);

   // object of Base Class
   Base b;
   thread th4(&Base::foo, &b);
   thread th5(&Base::foo1);

   // Wait for thread t1 to finish
   th1.join();
   // Wait for thread t2 to finish
   th2.join();
   // Wait for thread t3 to finish
   th3.join();
   // Wait for thread t4 to finish
   th4.join();
   // Wait for thread t5 to finish
   th5.join();

   return 0;
}
```

std::mutex revisit

A C++ mutex, short for "**mutual exclusion**", is a synchronisation primitive used in multithreading to protect shared resources from being accessed concurrently by multiple threads. Mutexes ensure that only one thread can access the protected resource at a time, thus preventing race conditions and ensuring data consistency.
In C++, mutexes are typically implemented using the std::mutex class provided by the C++ standard library (<mutex> header). Here's a brief overview of how mutexes work in C++:

- **Locking**: A thread that wants to access a shared resource locks the mutex associated with that resource using the lock() method of the std::mutex object. If the mutex is already locked by another thread, the calling thread will be blocked until the mutex becomes available.
- **Unlocking**: After completing its work with the shared resource, the thread unlocks the mutex using the unlock() method. This allows other threads waiting to access the resource to proceed.
- **Scoped Locking**: To simplify mutex usage and ensure proper unlocking even in case of exceptions, C++ provides the **std::lock_guard** class. std::lock_guard is a RAII (Resource Acquisition Is Initialisation) wrapper around a mutex. It locks the mutex upon construction and unlocks it automatically when it goes out of scope.

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx; // Declare a mutex

void sharedResourceAccess() {
    std::lock_guard<std::mutex> lock(mtx); // Lock the mutex
    // Access the shared resource here
    std::cout << "Thread " << std::this_thread::get_id() << " is accessing the shared resource.\n";
    // The mutex will be automatically unlocked when 'lock' goes out of scope
}

int main() {
    std::thread t1(sharedResourceAccess);
    std::thread t2(sharedResourceAccess);
    t1.join();
    t2.join();
    return 0;
}
```

C++ Thread Pool

The Thread Pool in C++ is used to manage and efficiently resort to a group (or pool) of threads. Instead of creating threads again and again for each task and then later destroying them, what a thread pool does is it maintains a set of pre-created threads now these threads can be reused again to do many tasks concurrently. By using this approach we can minimize the overhead that costs us due to the creation and destruction of threads. This makes our application more efficient. There is no in-built library in C++ that provides the thread pool, so we need to create the thread pool manually according to our needs.

What is Thread Pool?

A group of worker threads that are established at the program start and stored in a pool to be used at a later time are called thread pools. The Thread Pool effectively maintains and allocates existing threads to do several tasks concurrently, saving time compared to starting a new thread for each activity.

Need of Thread Pool in C++

- When several activities must be completed simultaneously, like in server applications, parallel processing, and parallelising loops, thread pools are frequently utilised.
- Thread Pools enhance overall performance by lowering the overhead of thread generation and destruction through thread reuse.

Key components of thread pool implementation

- there is an array of threads
- there is a FIFO queue of tasks (a Task should be a wrapper for a Callable object).
- when a client creates a task he enqueu() it into the task queue
- all threads are started up front in the pool and they never quit running; if there's no task in the queue they wait
- each incoming task notifies() a single thread to handle it
- if a Task returns a result the client can send a query for its value later
- the thread pool has a switch On/Off flag
- Thread synchronisation: a condition variable along with a mutex variable perform thread synchronisation
- mutex is used to synchronise access to shared data (the containers mostly)
- condition notifies threads upon task arrival and puts them to sleep when there are no tasks
- Try to reduce the usage of standard raw pointers as much as possible in your programs.

```
// C++ Program to demonstrate thread pooling

#include <condition_variable>
#include <functional>
#include <iostream>
#include <mutex>
#include <queue>
#include <thread>
using namespace std;

// Class that represents a simple thread pool
class ThreadPool {
public:
    // Constructor to creates a thread pool with given number of threads
    ThreadPool(size_t num_threads= thread::hardware_concurrency()) {
        // Creating worker threads
        for (size_t i = 0; i < num_threads; ++i) {
            threads_.emplace_back([this] {
                while (true) {
                    function<void()> task;
                    // The reason for putting the below  here is to unlock the queue before executing the task so that other
                    // threads can perform enqueue tasks
                    {
                        // Locking the queue so that data can be shared safely
                        unique_lock<mutex> lock(queue_mutex_);
                        // Waiting until there is a task to execute or the pool is stopped
                        cv_.wait(lock, [this] {
                            return !tasks_.empty() || stop_;
                        });
                        // exit the thread in case the pool is stopped and there are no tasks
                        if (stop_ && tasks_.empty()) {
                            return;
                        }
                        // Get the next task from the queue
                        task = move(tasks_.front());
                        tasks_.pop();
                    }
                    task();
                }
            });
        }
    }
// Destructor to stop the thread pool
  ~ThreadPool() {
    {
      // Lock the queue to update the stop flag safely
      unique_lock<mutex> lock(queue_mutex_);
      stop_ = true;
    }

    // Notify all threads
    cv_.notify_all();

    // Joining all worker threads to ensure they have completed their tasks
    for (auto& thread : threads_) {
      thread.join();
    }
  }

  // Enqueue task for execution by the thread pool
  void enqueue(function<void()> task)  {
    {
      unique_lock<std::mutex> lock(queue_mutex_);
      tasks_.emplace(move(task));
    }
    cv_.notify_one();
  }

private:
  // Vector to store worker threads
  vector<thread> threads_;
  // Queue of tasks
  queue<function<void()> > tasks_;
  // Mutex to synchronize access to shared data
  mutex queue_mutex_;
  // Condition variable to signal changes in the state of the tasks queue
  condition_variable cv_;

  // Flag to indicate whether the thread pool should stop or not
  bool stop_ = false;
};
```

Example

```
int main()
{
    // Create a thread pool with 4 threads
    ThreadPool pool(4);

    // Enqueue tasks for execution
    for (int i = 0; i < 5; ++i) {
        pool.enqueue([i] {
            cout << "Task " << i << " is running on thread " << this_thread::get_id() << endl;
            // Simulate some work
            this_thread::sleep_for(
                chrono::milliseconds(100));
        });
    }

    return 0;
}
```

Output

```
Task 0 is running on thread 140178994148928
Task 1 is running on thread 140178985756224
Task 2 is running on thread 140179010934336
Task 3 is running on thread 140179002541632
Task 4 is running on thread 140178994148928
```

Explanation

1. In the above code, we have used the following C++ features for the implementation of the thread pool:

2. A vector of worker threads, a task queue, a mutex for synchronization, a condition variable for signaling, and a boolean flag to indicate whether the pool should stop are all managed by the ThreadPool class.
3. The worker threads are initialized by the constructor, who then puts them in an endless loop while they wait for jobs to be enqueued. We use a wrapper class std::function over the given tasks.
4. A job is added to the queue and one of the worker threads is notified to begin executing it using the enqueue method.
5. To guarantee a clean shutdown, the destructor joins the worker threads, sets the stop flag, and informs all threads.
6. A ThreadPool with four threads is formed in the main function. Ten jobs are queued up, each of which prints a message including the task number and the thread ID that is currently carrying it out.
7. It should be noted that in a real-world situation, you would usually connect the threads or use some other kind of synchronization to make sure that all jobs are finished before the program ends.

Advantages of Thread Pooling in C++

The following are some main advantages of thread pooling in C++:

- Resource Management: Thread pools effectively manage resources by preventing resource depletion by restricting the number of threads that are executing concurrently.
- Enhanced Performance: By lowering the cost involved in establishing and terminating threads, reusing threads enhances performance.
- Scalability: Thread Pools are scalable in a variety of settings because they may dynamically modify the number of worker threads according to the capabilities of the system.

Disadvantages of Thread Pooling in C++

Thread pooling also have some limitations which are mentioned below:

- Thread pool adds complexity to the code hence managing threads and task queues might impact the performance of other lightweight tasks.
- The thread pool works on the assumption that each task is independent. So, Handling the dependencies between tasks is challenging when one task depends on the result of another task.
- The behavior of thread pools is platform-dependent. Hence, behavior may vary in different operating systems and C++ compilers.

An Example of Monte-Carlo Simulation using multi-threading

In this example, a multi-threading program using Monte-Carlo simulation to price a European call option is presented. It can be seen that using this idea properly will improve the code running efficiency. However, inappropriate usage will create large overhead in run-time, and leads to much poorer performance.

```
auto start1 = std::chrono::high_resolution_clock::now();
double pv = 0;
vector<std::future<double>> _futures;
for (size_t i =0 ; i < list.size(); i++ ){
_futures.push_back(std::async(std::launch::async, path_pv, list[i]));
}
for (auto && fut: _futures) {
pv += fut.get();
}
pv = pv / path;
auto stop1 = std::chrono::high_resolution_clock::now();
auto duration1 = std::chrono::duration_cast<std::chrono::milliseconds>(stop1 - start1);

cout << "Pv of the option is: " << pv <<endl;
cout << "time taken for aysnc function: " << duration1.count() << "ms." << endl;
```

```cpp
const auto processor_count = std::thread::hardware_concurrency();
cout << "cpu count: " << processor_count << endl;
auto start2 = std::chrono::high_resolution_clock::now();
double pv = 0;
vector<std::future<double>> _futures;
int chunk_size = processor_count-2;
auto iter = list.begin();
int step = int(list.size() / chunk_size);
while (iter <= list.end()) {
auto end = iter + step;
if (end >=list.end())
end = list.end();
vector<double>chunk(iter, end);
_futures.push_back(std::async(std::launch::async, path_pv2, chunk));
iter+=step;
}
for (auto && fut: _futures) {
pv += fut.get();
}
pv = pv / path;
auto stop2 = std::chrono::high_resolution_clock::now();
auto duration2 = std::chrono::duration_cast<std::chrono::milliseconds>(stop2 - start2);
```

Quiz for part II

1. What does multithreading refer to in C++?
   A) Execution of multiple processes simultaneously
   B) Execution of multiple threads within a single process concurrently
   C) Execution of multiple functions sequentially
   D) Execution of multiple classes simultaneously
2. How can you create a thread in C++ using the standard library?
   A) Using std::create_thread()
   B) Using std::thread()
   C) Using std::start_thread()
   D) Using std::run_thread()
3. What is the purpose of synchronization in multithreading?
   A) To increase the number of threads.
   B) To decrease the number of threads.
   C) To coordinate access to shared resources among multiple threads.
   D) To terminate threads gracefully.
4. What is a mutex in C++ multithreading?
   A) A tool for memory management.
   B) A tool for thread scheduling.
   C) A mutual exclusion primitive used to protect shared resources.
   D) A function to create threads.
5. What is a race condition in multithreading?
   A) A situation where two threads are waiting for each other to complete.
   B) A situation where the execution of threads is sequential.
   C) A situation where multiple threads access shared data concurrently, leading to unpredictable results.
   D) A situation where a thread is terminated unexpectedly.
6. How can a deadlock be avoided in multithreading?
   A) By using sleep statements.
   B) By avoiding the use of synchronization primitives.

C) By ensuring that locks are always acquired in the same order.
D) By terminating all threads when a deadlock occurs.

7. What is a condition variable in C++ multithreading?
   A) A variable used to store thread IDs.
   B) A variable used to check the condition of a thread.
   C) A synchronization primitive used to signal changes in shared data.
   D) A variable used to store thread priorities.
8. What is a benefit of multithreading in C++?
   A) Increased memory usage.
   B) Reduced CPU utilization.
   C) Improved responsiveness and performance in applications.
   D) Simplified debugging process.