Synchronization

Task Programming in C++

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

License Information

Introduction

- Multi-threading in C++ has two main limitations
  1. The number of software threads may be higher than the number of hardware threads
     - **Over-subscription** occurs every time the number of software threads ready to start is higher than the number of hardware threads available in the system
     - Over-subscription implies system overhead and some performance penalty
     - With threads, the global load must be managed manually

- To solve this problem, C++11 introduced **task-based** parallel programming

Introduction

- Multi-threading in C++ has two main limitations

2. Threads (the std::thread library) do not offer any direct way to return a value to the caller
   - In POSIX
     - A simple strategy is return a value with pthread_exit
     - More general strategies must be user-implemented (through global or local objects)
   - In native C++ only the general strategy is available (you must manipulate objects explicitly)

Returning Values from Threads:
In traditional multi-threading (using std::thread), there is no built-in mechanism to return a value from a thread directly. This means you have to use workarounds like global or local objects to share data between threads.
2. POSIX Threads:
POSIX threads (pthreads) offer a function pthread_exit to return a value from a thread. However, for more complex scenarios, you need to implement your own strategies to share data between threads.
3. Native C++ Threads:
In native C++ (using std::thread), you don't have a direct way to return values. You need to manually manage shared objects or use other synchronization mechanisms to get data from threads.
4. Futures and Promises:
C++11 introduced futures and promises to address this limitation. These constructs provide a way to return values from threads in a more straightforward and manageable manner.

Promise: An object that allows you to set a value that will be available in the future.
Future: An object that retrieves the value set by a promise once it becomes available.

- To solve this problem, C++11 introduced **futures** and **promises**

A promise object is created to hold the value that will be computed by the thread.
A future object is obtained from the promise, which will be used to retrieve the value.
A thread is started, and it sets the value in the promise.
The main thread waits for the future to get the value and then prints it.

Task processing

A task is an entity that runs asynchronously, producing output data that will become available (and useful) at a later time.

- A task is an entity that runs asynchronously producing output data that will become available (and useful) at a later time
  - The operating system associate a thread to a task in an automatic way
  - Balancing tasks is automatic, through work-stealing features
  - Tasks have the possibility of handling return values

Task processing

- In C++
  - Thread-based parallel programming relies on **std::thread** objects

std::thread is used to create a new thread that runs thread_function. This is a low-level approach where you manually manage threads.

```
std::thread t(thread_function);
```

  - Task-based parallel programming relies on **std::asynch** objects

Automatically manages task execution.
Returns a std::future object to retrieve the result.
Simplifies asynchronous programming by abstracting thread management

```
auto fut = std::asynch(thread_function);
```

std::async is a higher-level abstraction that automatically manages the creation and execution of tasks. It returns a std::future object, which can be used to retrieve the result of the asynchronous operation.
Comparison:

Thread-based programming with std::thread gives you more control but requires manual management of threads.
Task-based programming with std::async simplifies asynchronous execution and automatically handles thread management, making it easier to work with return values and manage tasks.
By using std::async, you can focus more on the logic of your tasks rather than the intricacies of thread management, which can lead to cleaner and more maintainable code.

Task processing

```
#include <future>
```

#include <future>: This header file is necessary to use std::async, std::future, and related functionalities.
future<T>: This is a template class that represents a future value of type T. It is used to retrieve the result of an asynchronous operation.
async(policy, function, args...): This function runs function asynchronously, with args as its arguments, and returns a future<T>.

Parameters for the thread function

```
future<T> async(policy, function, args...);
```

<T> is the type of the future

Asynchronous policy

“Thread” function

<T>: The type of the future. This indicates the type of value that the asynchronous function will return.
2. policy: The asynchronous policy that dictates how the function should be executed.
3. function: The function to be executed asynchronously.
4. args...: The arguments to be passed to the function.

- Function async (namespace std)
  - Is an alternative to std::thread to execute functions in parallel
  - Has and extra parameter, i.e., the policy
  - Returns a future of type T

Alternative to std::thread: std::async is an easier and higher-level alternative to std::thread for executing functions in parallel.
Extra Parameter (Policy): std::async takes an extra parameter called the policy, which determines how the function will be executed.
Returns a Future: The function returns a future<T>, which can be used to retrieve the result of the asynchronous operation.

For now, ignore it

Task processing

- The user may decide the running policy
  - There are three different types of policies

Policy	Description
This policy ensures that the function is run in a new thread immediately. It is useful for tasks that need to start right away and run concurrently with other tasks. launch::async	Asynchronous launch, i.e., a new thread is generated to run the new function.
launch::deferred This policy defers the execution of the function until the result is explicitly needed. This means the function will not run until you call get() on the future.	The call to the new function is deferred. The OS may never runt it.The new function will be run when we **wait** it or **get** its future.
This is the default policy where the system decides whether to run the function asynchronously or defer its execution based on the current state of the system. launch::async | launch::deferred	The policy to run the new thread is selected by the system accordingly to the availability of concurrency in the system. It is implementation dependent.

```
auto fut = std::async(std::launch::async, []() {
    // Your code here
    return 42;
});
int result = fut.get(); // Waits for the function to complete and gets the result
```

Lazy threading

```
auto fut = std::async(std::launch::deferred, []() {
    // Your code here
    return 42;
});
// The function runs when fut.get() is called
int result = fut.get();
```

Default policy

```
auto fut = std::async([]() {
    // Your code here
    return 42;
});
int result = fut.get(); // The system decides when to run the function
```

Examples

Running async tasks

```
auto f1 = std::async(std::launch::async, my_f, 10);
// Thread function my_f is run in a new thread

auto f2 = std::async(
  std::launch::deferred, my_f, 20);
// Thread function my_f is not run until we get
// its results or wait for it

auto f3 = std::async(
   std::launch::async | std::launch::deferred,
   my_f, 30);
// The system decides when running my_f.
// Possibly, it never runs my_f.


f2.wait()
// Invoke deferred function f2 (i.e., run it)
```

For now, we do not know what a future is

Force task f2 to be associated and run within a thread is

Example

Running policy

```
auto f = std::async (my_f);

//    f.wait_for(0s): This checks the status of the future without waiting (0 seconds). It returns immediately with the status of the future.

if (f.wait_for(0s)==std::future_status::deferred) {

   f.wait();

} else {

   while (f.wait_for(100ms)!=
     std::future_status::ready) {

     ... do something ...

   }
   ...
}
```

If it is async or deferred; it may never run

Wait for 0 seconds, i.e., do not wait, check the status

It is deferred: Use wait to force the execution.
It is asynch, it is already running.

Check status every 100 msecons

If it is not ready, do something in parallel

Here the future f is ready

To wait for 0s or 100ms use std::literals

f.wait_for(100ms): This waits for 100 milliseconds and checks the status of the future.
std::future_status::ready: This status indicates that the task is complete, and the result is ready.
The while loop continues to check the status every 100 milliseconds.

The slide demonstrates how to use std::async to create an asynchronous task and handle its completion status.
It shows how to check if the task is deferred and force its execution if necessary.
It also illustrates how to poll the status of an asynchronous task and perform other operations while waiting for it to complete.

Futures

- An async object will eventually hold the return value of the thread function in a future
  - A future is an object that can represent a value generated by some provider
  - Function **<future>::get** applied to a valid future
    - Blocks the thread until the object is ready
    - Returns the object (return with "return") once it is ready

<T> is the type of the future

```
future<T> async(policy, function, args...);
```

Example

```
#include <future>
...

bool is_prime (int n) {
  if (num <= 1) return false;
  if (num <= 3) return true;
  ...
  return false;
}

int main () {
  std::future<bool> fut = std::async(
    std::launch::async, is_prime, 117);

  // ... do other work ...

  bool ret = fut.get();
  cout << ret;

  return 0;
}
```

Check if num is prime

Run a new function in a thread

Wait for function is_prime to return and make the Boolean value available

Example

```
#include <future>
#include <iostream>

...
auto fut = std::async (
  std::launch::async,
  [](){
    std::vector<int> v;
    for (int i=0; i<100; i++)
      v.push_back(i);
    return v;
  }
);
...
auto ret = fut.get();
for (auto e: ret)
  std::cout << e << std::endl;
```

Run a new function thread

Work with **lambda** expressions

Wait for the future to be ready and **get** the return value

Shared futures

Unique Future (std::future<T>):
Only one instance referring to the event.
Cannot be copied.
Used when only one thread needs to retrieve the result.

- In C++ there are two types of future
  - Unique future, i.e., std::future<T>
    - There is only one instance referring to the event
  - Shared future, i.e., std::shared_future<T>
    - A shared_future object behaves like a future object, except that it can be copied
    - Multiple instances may refer to the same event
    - All intances will become ready at the same time and can be retrieved
    - May be used to signal multiple threads simultaneously, similarly to std::condition_variable::notify_all

Single Instance: There is only one instance of a std::future referring to the event. This means that once a std::future object is created, it cannot be copied or shared with other std::future objects.
Ownership: The std::future object has unique ownership of the result. Once the result is retrieved using get(), the future becomes invalid.

Usage: Typically used when only one thread needs to wait for and retrieve the result of an asynchronous operation.

```
std::future<int> fut = std::async(std::launch::async, []() { return 42; });
  int result = fut.get(); // Retrieve the result, fut is now invalid
```

Shared Future (std::shared_future<T>):
Definition: A shared future is similar to a unique future but can be copied and shared among multiple instances.
Characteristics:
Copyable: A std::shared_future object can be copied, allowing multiple instances to refer to the same event.
Multiple Instances: Multiple std::shared_future objects can refer to the same asynchronous result.
Simultaneous Readiness: All instances of std::shared_future will become ready at the same time, and the result can be retrieved from any of the instances.
Signaling Multiple Threads: std::shared_future can be used to signal multiple threads simultaneously, similar to how std::condition_variable::notify_all works.

```
std::shared_future<int> shared_fut = std::async(std::launch::async, []() { return 42; }).share();

// Multiple threads can now use shared_fut to get the result
std::thread t1([shared_fut]() {
    int result = shared_fut.get();
    std::cout << "Thread 1: " << result << std::endl;
});

std::thread t2([shared_fut]() {
    int result = shared_fut.get();
    std::cout << "Thread 2: " << result << std::endl;
});

t1.join();
t2.join();
```

Shared Future (std::shared_future<T>):
Can be copied and shared among multiple instances.
Multiple instances can refer to the same event.
All instances become ready at the same time.
Useful for signaling multiple threads simultaneously.

Example

Unique future

```
int sum(int a, int b) {
  std::this_thread::sleep_for(std::chrono::seconds(2));
  return a + b;
}

int main() {
  std::future<int> fut =
    std::async(std::launch::async, sum, 10, 20);
  ...

  int result1 = fut.get();
  std::cout << "Result: " << result1 << endl;

  int result2 = fut.get();
  std::cout << "Result: " << result2 << endl;

  return 0;
}
```

How is it different for deferred? Deferred Execution: The function sum runs only when fut.get() is called.
Blocking: fut.get() will block until the function completes and the result is available.

Wait and then get the future

Result: 30

terminate called after throwing an instance of 'std::future_error'
what(): std::future_error: No associated state
Aborted (core dumped)

Example

Shared future

```
int sum(int a, int b) {
  std::this_thread::sleep_for(std::chrono::seconds(2));
  return a + b;
}

int main() {
  std::shared_future<int> fut =
    std::async(std::launch::async, sum, 10, 20);
  ...

  int result1 = fut.get();
  std::cout << "Result: " << result1 << endl;

  int result2 = fut.get();
  std::cout << "Result: " << result2 << endl;

  return 0;
}
```

Wait and then get the future

Result: 30

Result: 30

Promises

- The most common situation where you encounter a future is with a call to an async
  - An async returns a future
  - A future represents a value that you do not yet have but will have eventually
- At the lowest level, a future comes from an associated promise
  - A promise is an object that can store a value to be retrieved by a future object
    - A promise is an object that you will eventually set
  - When the value is set, it will be made available through its corresponding future

1. Common Use Case:
"The most common situation where you encounter a future is with a call to an async": When you use std::async, it returns a std::future object.
"An async returns a future": std::async creates an asynchronous task and returns a future that will hold the result of that task.
"A future represents a value that you do not yet have but will have eventually": A future is a placeholder for a result that will be available at some point in the future.
2. Lowest Level: Future from Promise:
"At the lowest level, a future comes from an associated promise": A future is often created from a promise.
"A promise is an object that can store a value to be retrieved by a future object": A promise is used to set a value that a future will eventually hold.
"When the value is set, it will be made available through its corresponding future": Once the promise sets the value, the future can retrieve it.

Promises

Think of promise and future as creating a single-use channel for data": A promise and future pair can be thought of as a one-time communication channel.

- Think of promise and future as creating a single-use channel for data

"Creates the channel": The promise creates the channel.
"Writes data in the channel": The promise sets the value that will be communicated through the channel.

  - A promise
    - Creates the channel
    - Writes data in the channel

Promise Name

```
std::promise<type> pn;

pn.set_value(...);
```

  - A future
    - Connects to the other end of the channel
    - Waits and reads the data once it has been written

"Connects to the other end of the channel": The future is connected to the promise and will receive the value set by the promise.
"Waits and reads the data once it has been written": The future waits for the promise to set the value and then retrieves it.

Future Name

```
auto fn = pn.get_future();

fn.get();
```

Promise: Creates a channel and sets a value.
Future: Connects to the promise and retrieves the value once it is set.
Common Use Case: std::async returns a future, which represents a value that will be available in the future.
Channel Analogy: Think of promise and future as a single-use data channel where the promise writes data and the future reads it.

Promises

- The principal steps are
  - The main thread
    - Defines a promise
    - Associate a future to the promise

```
std::promise<type> pn;
auto fn = pn.get_future();
```

  - The working thread
    - Receive the promise
    - Executes the function and fulfills the promise

```
pn.set_value(...);
```

  - The main thread retrieves the result

```
fn.get();
```

Promises

- Further considerations

If a promise is destroyed without setting a value (using set_value or set_exception), the associated future will store an exception.

  - If we destroy the promise without setting a value, an exception is stored in the object

When get is called on the future, it will throw an exception if the promise was destroyed without setting a value.

    - Function **get** will return

  - The object associated to a promise is usually stored in the heap as it cannot be stored

The object associated to a promise is usually stored in the heap as it cannot be stored":
The value or exception associated with a promise is typically stored on the heap because the promise and future may outlive the scope in which they were created.

    - In the setter of the promise, as the setter can die
    - In the getter of the future, as we futures can be shares among several getters

Future: Represents the value that will be available in the future.
Storage: The value or exception is stored on the heap to ensure it remains accessible.
Promise: Sets the value or exception that the future will retrieve.

Example

Promise and future.
One thread (lambda)

Thread function

```
int f(int x) { return x + 1; }
```

Define the promise
This is the producer-write end

object is created. This is the producer end, which will set a value in the future.

Define the future from the promise
This is the consumer-read end

```
std::promise<int> promise;
auto future = promise.get_future();
```

object is obtained from the promise. This is the consumer end, which will retrieve the value set by the promise.

Get the promise in the capture list

```
// Launch f asynchronously
std::thread thread([&promise] (int x) {
    promise.set_value(f(x));
  }, 5
);
```

The thread function is a lambda that captures the promise by reference (&promise) and takes an integer x as an argument. Inside the lambda, promise.set_value(f(x)); is called. This sets the value of the promise to the result of f(x). In this case, f(5) is called, which returns 6, and this value is set in the promise.

Set the value (to be communicated) into the promise

: This call blocks until the value is set in the promise. Once the value is set, it retrieves the value (which is 6 in this case) and prints it.

Get the value

```
std::cout << future.get() << std::endl;
```

Example

Promise and future.
Two threads (lambdas)

This example demonstrates how to use std::promise and std::future with two separate threads: one for setting the promise and another for getting the future. Here's a step-by-step explanation:

Define the promise and the future from the promise

```
auto promise = std::promise<std::string>();
auto future = promise.get_future();
```

A std::promise object is created to hold a std::string. This is the producer end, which will set a value in the future.

A std::future object is obtained from the promise. This is the consumer end, which will retrieve the value set by the promise.

A new thread is launched. The thread function is a lambda that captures the promise by reference (&).

```
auto producer = std::thread([&] {
  promise.set_value("Hello World");
});
```

Run the thread setting the promise

Inside the lambda, promise.set_value("Hello World"); is called. This sets the value of the promise to the string "Hello World"

Another thread is launched. The thread function is a lambda that captures the future by reference (&).

```
auto consumer = std::thread([&] {
  std::cout << future.get();
});
```

Run the thread getting the future

Inside the lambda, std::cout << future.get() << std::endl; is called. This call blocks until the value is set in the promise. Once the value is set, it retrieves the value (which is "Hello World") and prints it.

```
producer.join();
consumer.join();
```

producer.join();: This ensures that the producer thread completes execution before the program continues.
consumer.join();: This ensures that the consumer thread completes execution before the program continues.

ummary
std::promise and std::future are used to communicate between threads.
The promise sets a value that the future can retrieve.
One thread (producer) sets the value in the promise.
Another thread (consumer) waits for the value using the future and then prints it.
Both threads are joined to ensure they complete execution before the program exits.

Example

Promise and future.
One thread (function)

One-way communication:
The thread set the promise and get the future

```
#include <future>
using namespace std;

void factorial (const int &N, promise<int>& pr) {
  int res = 1;
  for (int i=N; i> 1; i--)
  res *=i;
  pr.set_value(res);
}

int main () {
  promise<int> p;
  future<int> f = p.get_future();
  thread t = thread(factorial, 4, ref(p));
  // here we have the data
  int x = f.get();
  t.join();
}
```

Define the promise and the future from the promise

the reason we use std::move to pass the promise to the thread function is because std::promise objects are not copyable, only movable. This means you cannot make a copy of a promise object, but you can transfer ownership of it to another object or function. In the context of multithreading, this is useful because it ensures that only one specific thread has ownership and therefore can fulfill the promise. This helps to avoid data races and undefined behavior.

std::ref is used to create a reference_wrapper, which is a copyable and assignable object that emulates a reference. It's typically used when you need to pass references to functions that expect copies, such as std::thread. However, in the context of std::promise, using std::ref would not be appropriate. This is because std::promise is meant to be moved, not copied or passed by reference.

ref generates an object of type promise<int> to hold a reference to p

Example

One async task.
Two way sync

Two-ways communication:
The caller set the promise and get the future

The future must be passed by reference, since it doesn't support copy semantics

Define the promise and the future from the promise

```
#include <future>
using namespace std;
int factorial (std::future<int>& f ) {
  int res = 1;
  int N = f.get();
  for ( int i=N; i> 1; i-- )
  res *=i;
  return res;
}
int main () {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::future<int> fu =
    async(std::launch::async,factorial,std::ref(f));
  p.set_value(4);
  int x = fu.get();
}
```

Example

Two async tasks.
Two way sync

This function takes a std::promise<int> as an argument, sets the value of the promise to 18, and then exits.

```
void func1 (promise<int> p) {
  int res = 18;
  p.set_value(res);
}
```

This function takes a std::future<int> as an argument, retrieves the value from the future using f.get(), and returns it.

```
int func2 (future<int> f) {
  int res=f.get();
  return res;
}

int main () {
  promise<int> p;
  future<int> f = p.get_future();
  future<void> fu1 = async(func1, move(p) );
  future<int> fu2 = async(func2, move(f) );
  int x = fu2.get();
  return 0;
}
```

The move semantics is achieved by std::move

Example

Shared future

```
usign namespace std;
int factorial (shared_future<int> f) {
  int res = 1;
  int N = f.get();
  for (int i=N; i> 1; i--) res *=i;
  return res;
}

int main () {
  promise<int> p;
  future<int> f = p.get_future();
  shared_future<int> sf = f.share();

  future<int> fu1 = async(std::launch::async, factorial, sf);
  future<int> fu2 = async(std::launch::async,factorial, sf);
  future<int> fu3 = async(std::launch::async, factorial, sf);


  p.set_value(4);
  int r1=fu1.get(); int r2=fu2.get(); int r3=fu3.get();
  return 0;
}
```

we can also write the shared future directly

```
std::promise<int> p;
    std::shared_future<int> sf = p.get_future(); // Directly get the shared future
```

Conclusions

- The task-based approach
  - Makes the OS in charge of the parallelism
  - Makes the return value of a thread/task accessible
  - Run threads with a smart policy
    - CPU load balancing
      - The C++ library can run the function without spawning a thread
    - Avoid the raising of **std::system_error** in case of thread number reached the system limit
  - Allows futures to catch exceptions thrown by the function
    - With **std::thread()** the program terminates

Conclusions

- Thread-based approach
  - Is used to execute tasks that do not terminate till the end of the application
    - A thread entry point function is like a second, concurrent **main**
  - It is a more general concurrency model
    - Can be used for thread-based design patterns
  - Allows us to access to the pthread native handle
    - Makes the programmer in charge of the parallelism
    - Useful for advanced management (priority, affinity, scheduling policies, etc.)

Exercise

- Resorting only to asynchrnous tasks, write a C++ program to perform the matrix multiplication
  - C = A x B
- Constraints
  - The number of columns in A must equal the number of rows in B
  - Use C++ containers
    - Implement the matrices as vector of vectors
      - std::vector<std::vector<int>> a, b;
  - To compute a sum of products, it is possible to use the function **std::inner_product**

Solution

```
#include ...

void generateRandomMatrix (
  vector<vector<int>>& m, int nRow, int nCol){
  for (int i = 0; i < nRow; i++){
    vector<int> row;
    for (int j = 0; j < nCol; j++)
      row.push_back(rand()%3);
    m.push_back(row);
  }
}

void printMatrix(const std::vector<std::vector<int>>& m){
  for (auto & row : m){
    for (int element : row)
      cout << element << " ";
    cout << endl;
  }
}
```

Utility function:
Generate a random matrix

Utility function:
Display a matrix

Solution

Function threads

```
int computeSumOfProducts (
  const std::vector<int>& v1, const std::vector<int>& v2){

  return std::inner_product (
    v1.begin(), v1.end(), v2.begin(), 0
  );
}
```

Return result

Until C++11
Computes the sum of products on the range [v1.begin, v1.end] and the range beginning at v2.begin, initializing the accumulator at 0

Solution

Main thread

```
int main() {
  int nRowA = 2, nColA = 3,  nRowB = 3, nColB = 2;
  vector<std::vector<int>> a, b;
  vector<std::vector<std::future<int>>> futures;
  ofstream outputFile;

  cout << "Insert size of matrix A" << endl;
  cin >> nRowA >> nColA;
  cout << "Insert size of matrix B" << endl;
  cin >> nRowB >> nColB;

  if(nColA != nRowB) {
    cout << "Wrong size colA != rowB" << endl;
    return -1;
  }
```

Matrices

Matrix of futures

Read the size of A and B

Solution

```
generateRandomMatrix(a, nRowA, nColA);
// generate the transpose of B (columns -> rows)
// so that we can easily access the columns of B
// for the multiplication
generateRandomMatrix(b, nColB, nRowB);

// Generate the futures to compute the products
for (int i = 0; i < nRowA; i++){
  vector<std::future<int>> futureRow;
  for (int j = 0; j < nColB; j++) {
    future<int> f = std::async(
      std::launch::async | std::launch::deferred,
      computeSumOfProducts, a[i], b[j]);
    // Futures are not copyable so we have to use move
    // to store them in a vector
    futureRow.push_back(std::move(f));
  }
  futures.push_back(std::move(futureRow));
}
```

Generate matrix A

Generate matrix B

Create a future for each element

Insert futures in the matrix of futures

Solution

```
  cout << "A" << endl;
  printMatrix(a);
  cout << "B" << std::endl;
  printMatrix(b);

  outputFile.open("./output-matrix.txt");
  for (auto & row : futures){
    for (std::future<int> & f : row) {
        outputFile << f.get() << " ";
    }
    outputFile << std::endl;
  }
  outputFile.close();

  return 0;
}
```

Display input matrices

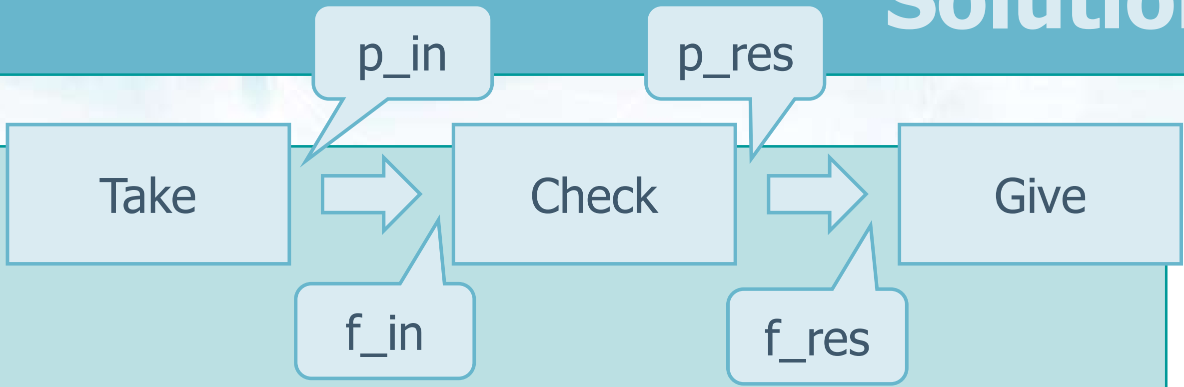
Wait futures to be ready and display result

Exercise

Exam 5 July 2021

- Write a C++ program with three tasks
  - Thread **take** reads a number from command line
  - Thread **check** checks whether the number is prime
  - Thread **give** displays the answer to standard output
- Thread communication should be made using promises and futures
  - All functions are acyclic

Solution

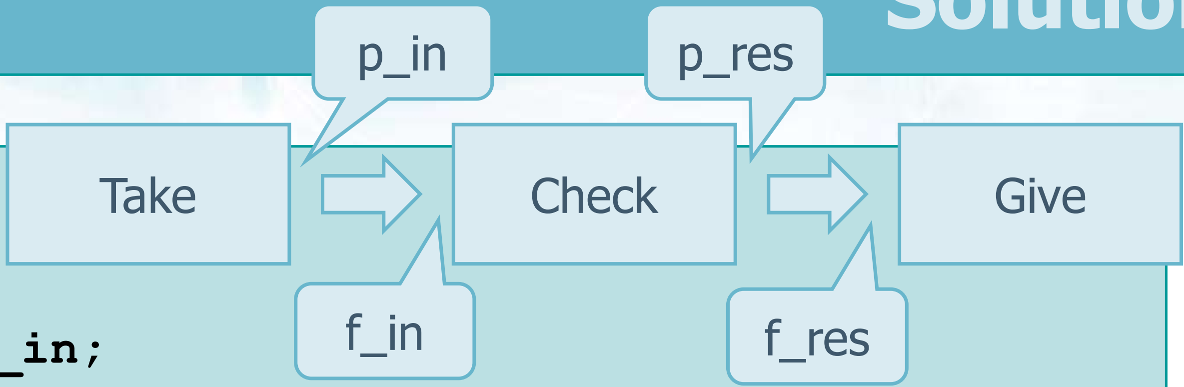



```
#include <iostream>
#include <thread>
#include <vector>
#include <future>

void take (std::promise<int>&);
void check (std::future<int>&, std::promise<bool>&);
void give (std::future<bool>&);
```

Thread functions

Solution




```
int main(){
  std::promise<int> p_in;
  std::future<int> f_in = p_in.get_future();

  std::promise<bool> p_res;
  std::future<bool> f_res = p_res.get_future();

  std::thread t1(take, std::ref(p_in));
  std::thread t2(check, std::ref(f_in), std::ref(p_res));
  std::thread t3(give, std::ref(f_res));

  t1.join();
  t2.join();
  t3.join();
  return 0;
}
```

Solution

Reading thread

```
void take (std::promise<int> &p_in) {
  int in;
  std::cout << "Insert a number" << std::endl;
  std::cin >> inp;
  p_in.set_value (in);
}
```

Set promise "in"

Writing thread

```
void give (std::future<bool>& f_res) {
  bool answer = f_res.get();
  std::string s0 (" ");
  if(!answer)
    s0=" NOT";
  std::cout << "Number is" << s0 << " prime";
}
```

Get future "ref"

Solution

Computation thread

```
void check (
  std::future<int> &f_in, std::promise<bool>& p_res)
{
    int n = f_in.get();
    bool prime=true;
    if (n <= 1){
      prime = false;
    }
    // Check from 2 to n-1
    for (int j=2; j<n; j++) {
      if (n % j == 0) {
        prime = false;
        break;
      }
    }
    p_res.set_value(prime);
}
```

Get future "in"

Set promise "res"