Managing threads

1 Basic thread management

- Each C++ program has at least one thread started by the C++ runtime, it is the thread running `main()`
- Your program can launch additional threads that has another function as an entry point. These thread run concurrently to each other and the main (initial) thread
- An thread exits when the function associated with it finishes. For example when `main()` finishes the initial thread exits

1.1 Launching a thread

Starting a thread always boils down to constructing a `std::thread` object for example like this:

```
void do_some_work();
std::thread my_thread(do_some_work);
```

Remark 1.1.1

Make sure that you include the `<thread>` header so the complier notices the definition of the `std::thread` class. **In the following code snippets includes will be omitted.**

As with much of the C++ Standard Library, `std::thread` works with any callable type for example:

```
class background_task
{
    public:
    void operator() () const
    {
        do_something();
        do_something_else();
    }
};

background_task f;
std::thread my_thread(f);
```

The function object is copied to the storage belonging to the thread where it is executed and invoked from there. Therefore it is essential that the copy behaves exactly like the original otherwise results might be unexpected.

Remark 1.1.2

When passing a temporary rather than a named variable the syntax can be the same as a function declaration and the compiler will interpret it as such:

```
    std::thread my_thread(background_task()); // function declaration
```

One avoids that by using one of the following syntaxes

```
    std::thread my_thread((background_task())); // extra pair of
parentheses
    std::thread my_thread{background_task()}; // uniform initialization
syntax
```

Another way achieving the correct interpretation would be to use lambda expressions (local/anonymous function) in the following way

```
    std::thread my_thread([]{
        do_something();
        do_something_else();
    });
```

1.2 Issues arising from using threads

When a thread is started one needs to explicitly state whether to wait for it to finish (by joining it - see section 1.3) or leave it to run on it's own (by detaching it - see section 1.4). When one does not decide before the `std::thread` object is destroyed the program is terminated by calling `std::terminate`. Note that you only have to decide before the `std::thread` object is destroyed. When not waiting for the thread to finish one has to insure that the data the thread is working on is still available.

Here is one example of a program where the thread function holds a local variable that is destroyed before the thread finishes:

```
    struct func {
        int& i;
        func(int& i_):i(i_){}
        void operator()()
        {
            for(unsigned j=0;j<1000000;++j)
            {
                do_something(i);
            }
        }
    }

    void oops() {
        int some_local_state=0;
```

```
        func my_func(some_local_state);
        std::thread my_thread(my_func);
        my_thread.detach();
    }
```

Here the new thread associated with `my_thread` will most likely still be running when `oops` exits, since one calls `detach()` on the object. Like this in some call of `do_something(i)` will result in accessing an already destroyed variable.

In the following the two main ways to avoid this are presented:

1. Make the thread self-contained and copy the data into the thread rather than sharing it
2. By *joining* with the thread

1.3 Joining with threads

If one needs to wait for a thread to complete, this is done by calling `join()` on the associated `std::thread` object. When looking into the code example from before it is sufficient to replace the call of `my_thread.detach()` with `my_thread.join()` to insure that the thread was finished before `oops()` was exited and the local variables were destroyed.

When you need a more fine grained control over waiting for a thread one can use mechanisms such as condition variables and futures but these will be elaborated in another presentation.

The act of *joining* a `thread` cleans up any storage associated to the thread, so the joined thread object is no longer associated with any other running thread. From this it follows that one can only call `join()`on a given thread once you have called it on it the method `joinable()` will return false.

1.3.1 Waiting in exceptional circumstances

- When one plans to detach a thread this can usually be done right after the thread has been started
- But when one plans to join the thread it is not that easy, one has to carefully pick the place where to call `join()` on the thread

Joining threads when exceptions are raised:

```
struct func;
void f()
{
    int some_local_state=0;
        func my_func(some_local_state);
        std::thread t(my_func);
        try
        {
            do_something_in_current_thread();
        }
    catch(...) {
        t.join();
        throw;
    }
```

```
    t.join();
}
```

The `try/catch`block insures that the thread is joined correctly even if a exception is raised during execution.

But this technique is error-prone, though using the standard **Resource Acquisition Is Initialization (RAII)** idiom provides a class and thus a simple mechanism to do exactly that.

A rewritten version using RAII of the listing above is given in the following:

```
class thread_guard
{
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_): t(t_){}
    ~thread_guard()
    {
        if(t.joinable())
        {
            t.join();
        }
    }
    thread_guard(thread_guard const&)=delete;
    thread_guard& operator=(thread_guard const&)=delete;
};

struct func;
void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    thread_guard g(t);
    do_something_in_current_thread();
}
```

- Execution of current thread reaches the end of `f` -> local objects are destroyed in reversed order of construction
- So the `thread_guard`object is destroyed first and the thread is joined with in the destructor
- This even happens if the function exits because `do_something_in_current_thread` throws an exception
- Also note the precondition in the destructor of ```thread_guard````
- Copy-constructor and copy-assignment are marked as `=delete`to avoid the object outliving the scope of the thread it was joining
- *Detaching* the thread would not raise these issues and would thus not require such an exception safe handling of the thread

1.4 Detaching threads

- *Detaching* a thread by calling `detach()` on the associated `std::thread` object on it leaves the thread to run in the background
- For *detached* threads there are no means of direct communication with it anymore
    - not possible to join with anymore
    - not possible to obtain a `std::thread` object that references to it
    - truly run in the background
    - ownership and control are passed to the C++ Runtime Library
- These threads are often called *daemon threads* that run in the background with no user interface
- When a thread is detached it is no longer joinable

2 Passing arguments to a thread function

- As simple as passing additional arguments to the `std::thread`constructor
- By default the arguments are copied into internal storage as rvalues
- This is even done when expecting a reference

```
    void f(int i,std::string const& s);
    std::thread t(f,3,”hello”);
```

Note that `hello` is passed in as `const char*` and implicitly converted to ```std::string````

This is particularly important when the argument supplied is a pointer to an automatic variable:

```
    void f(int i,std::string const& s);
    void oops(int some_param)
    {
        char buffer[1024];
        sprintf(buffer, "%i",some_param);
            std::thread t(f,3,buffer);
        t.detach();
    }
```

The problem here is that there is a significant chance that the function `oops` exits before the variable `buffer`has been converted to a `std::string` leading to undefined behavior. The solution is a type cast as follows:

```
    void f(int i,std::string const& s);
    void oops(int some_param)
    {
        char buffer[1024];
        sprintf(buffer, "%i",some_param);
            std::thread t(f,3,std::string(buffer));
        t.detach();
    }
```

The reverse scenario: an object is copied and you wanted a non-const reference won't compile. When you still want to do such things you need to wrap the parameter in `std::ref`.

Syntactical structures for calling member functions in a new thread are provided as follows:

```
class X
{
public:
    void do_lengthy_work();
};
X my_x;
std::thread t(&X::do_lengthy_work, &my_x);
```

When you want to pass an argument to the method, you have to pass it as the third argument of the thread constructor. So on and so forth for additional arguments.

2.1 Passing arguments where the arguments can only be *moved*

- An example of that would be the passing of a ```std::unique_ptr````

```
void process_big_object(std::unique_ptr<big_object>);
std::unique_ptr<big_object> p(new big_object);
p->prepare_data(42);
std::thread t(process_big_object,std::move(p));
```

Note that when the object is temporary this move is automatic for named values like `p` the move has to be made explicit as shown above

The `std::thread`class has the same ownership behaviors as `std::unique_ptr` so transferring the ownership of an `std::thread` object is also achieved by *moving* it.

This structure allows for only one object being associated with the thread of executing while still providing a mechanism for transferring this ownership as developer.

3 Transferring ownership of a thread

Motivation:

- wanting to write a function that creates a thread to run in the background, but passes ownership of the new thread to the calling function
- wanting to create a thread and pass the ownership in to some function
- Ownership: When a function 'owns' a thread it means that this function only exits/ returns when the thread is complete

Example:

```
    void some_function();
    void some_other_function();
    std::thread t1(some_function);
    std::thread t2=std::move(t1);
    t1=std::thread(some_other_function);
    std::thread t3;
    t3=std::move(t2);
    t1=std::move(t3); // <- Terminates since t1 already has a thread
associated with it
```

With this knowledge it is possible to transfer ownership of threads in and out from functions in the following ways:

Transfer out:

```
    std::thread f()
    {
        void some_function();
        return std::thread(some_function);
    }
    std::thread g()
    {
        void some_other_function(int);
        std::thread t(some_other_function,42);
        return t;
    }
```

Transfer in:

```
    void f(std::thread t);
    void g()
    {
        void some_function();
        f(std::thread(some_function));
        std::thread t(some_function);
        f(std::move(t));
    }
```

With this one can build on the `thread_guard` class from before and have it take care of the ownership of the thread. The benefits are the following:

- Avoids unpleasant consequences when the object outlive the thread it was referencing
- Provides safety that no one else can join or detach the thread once the ownership belongs to the object. In the following code snippet the class `scoped_thread` that implements this behavior along with a simple usage example is presented:

```
class scoped_thread
{
    std::thread t;
public:
    explicit scoped_thread(std::thread t_): t(std::move(t_))
    {
        if(!t.joinable())
            throw std::logic_error("No thread");
    }
    ~scoped_thread()
    {
        t.join();
    }
    scoped_thread(scoped_thread const&)=delete;
    scoped_thread& operator=(scoped_thread const&)=delete;
};

// Usage
struct func; See listing 2.1 void f()
{
    int some_local_state;
    scoped_thread t{std::thread(func(some_local_state))};
    do_something_in_current_thread();
}
```

3.1 A `joining_thread` class

One of the proposals for the C++17 standard was the `joining_thread` class, that would implement a extensive, robust and safe thread wrapper much like `scoped_thread`, but it didn't make it to consensus. With the knowledge obtained so far one can relatively easy implement such a class, hence it is presented in the following:

```
class joining_thread
{
    std::thread t;
public:
    joining_thread() noexcept=default;
    template<typename Callable,typename ... Args>
    explicit joining_thread(Callable&& func,Args&& ... args):
        t(std::forward<Callable>(func),std::forward<Args>(args)...)
    {}
    explicit joining_thread(std::thread t_) noexcept:
        t(std::move(t_))
    {}
    joining_thread(joining_thread&& other) noexcept:
        t(std::move(other.t))
    {}
    joining_thread& operator=(joining_thread&& other) noexcept
    {
        if(joinable())
```

```
                join();
            t=std::move(other.t);
            return *this;
        }
        joining_thread& operator=(std::thread other) noexcept
        {
            if(joinable())
                join();
            t=std::move(other);
            return *this;
        }
        ~joining_thread() noexcept
        {
            if(joinable())
                join();
        }
        void swap(joining_thread& other) noexcept
        {
            t.swap(other.t);
        }
        std::thread::id get_id() const noexcept{
            return t.get_id();
        }
        bool joinable() const noexcept
        {
            return t.joinable();
        }
        void join()
        {
            t.join();
        }
        void detach()
        {
            t.detach();
        }
        std::thread& as_thread() noexcept
        {
            return t;
        }
        const std::thread& as_thread() const noexcept
        {
        return t;
        }
    };
```

The move support in `std::thread` allows for containers of `std::thread`objects if those containers are move-aware (like the updated `std::vector<>`). This makes code like the following possible:

```
    void do_work(unsigned id);
    void f()
    {
```

```
            std::vector<std::thread> threads;
            for(unsigned i=0;i<20;++i)
            {
                threads.emplace_back(do_work,i);
            }
            for(auto& entry: threads)
                entry.join();
    }
```

This enables subdividing the work of an algorithm to multiple threads. Nevertheless the structure of the snippet above implies that: - the work done by the individual threads is self-contained - the result of their operations is purely the side effects on shared data

This is also a step toward automating the management of those threads rather than creating separate variables for those threads and joining with them directly. It is possible to treat them as a group.

4. Choosing the number of threads at runtime

- The C++ Standard Library provides a function that helps here namely `std::thread::hardware_concurrency`
    - It returns the number thread that can truly run concurrently on a system - for a multi-core system that might be the number of CPUs
        - Note that this is only a hint and the function can return 0 although there might be more threads available, this can be due to lack of information
- This information than can be used to identify the number of threads to use at runtime, liken this you can avoid oversubscription and undersubscription (using not enough or to many threads for a given task)

5. Identifying threads

- When developing concurrent programs one might come across a point where he wants to identify a thread
- Threads have an id of the type `std::thread::id`that can be retrieved in two ways
    - calling `get_id()`on a thread object, "not any thread" when no thread of execution is associated
        - the current thread id can be obtained by calling ```std::thread::get_id()````
- these id's can be copied and compared
    - provide a total ordering
- they are used for:
    - checking whether a thread should perform a certain action
        - storing data to be shared among threads without losing the associativity between thread and data