01.ThreadManagement.md

Thread management

Start with reading the "01a" document. It will give you explanations about concepts used with threads, especially rvalues and move semantics.

Concurrency and parallelism

Usually:

- Concurrency is used when you want to separate concerns or handle responsiveness
- Parallelism is used when the focus is taking advantage of the hardware to increase performance

Both will use multithreading.

Thread basics

You can launch a thread with a function:

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

using std::cout;
using std::endl;

void sayHello() {
	cout << "Hello world" << endl;
}

int main() {
	std::thread t(sayHello);

	t.join();                       // Wait for the thread
	return 0;
}
```

But also with a class that implements operator() or a lambda:

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

using std::cout;
using std::endl;

class Task {
public:
	void operator()() const {
		doSomething();
		doSomethingElse();
	}

private:
	void doSomething() const {
	cout << "Something" << endl;
    }

    void doSomethingElse() const {
	cout << "Something else" << endl;
    }
};

void doFunction() { cout << "Function" << endl; }

int main() {
	// Thread with a class that implements () operator (uses rvalue)
	std::thread t0{ Task() };

	// Thread with lambda
	std::thread t1([] {
		doFunction();
	});

	// Launch several times to remark the order is not deterministic

	t0.join();
	t1.join();
	return 0;
}
```

The thread are working. While the object is in scope, you should decide if you want to join them (wait for the end) or detach them (let them live until the function returns). You can join or detach them long after they are executed.

Note that a detached thread is no longer reachable.

A classic problem with threads is when you capture a local variable for them, then the thread still runs while the local variable get out of scope. Note that Visual Studio prevents this eventuality.

```
class RepeatQuickly {
	int& param;
	RepeatQuickly(int& paramP) : param(paramP) {}

	void doSomething(int& p) {
		cout << p << endl;
	}

	void operator()() {
		for (int j = 0; j < 100000; ++j) {
			doSomething(param);
		}
	}
};

int main() {
	// Error with variable that get out of scope while the thread uses it
	int localParam = 0;
	RepeatQuickly repeatQuickly(localParam);
	std::thread t2(repeatQuickly);
	t2.detach();

	return 0;
}
```

Take care to call join() in exceptions. One way to avoid that is to use the Resource Acquisition Is Initialisation (RAII) idiom and use join() in the destructor of the class.

```
class ThreadGuard {
	std::thread& t;

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

Passing argument to threads

You can pass additional parameters to the thread constructor to pass arguments to the function. By default those arguments are copied or used as rvalues (temporary copies) :

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

Here the string is converted to an std::string in the context of the thead.

You may have problems if you uses a function that needs a reference:

```
void updateDataForWidget(WidgetId w, WidgetData& data);
void oops(WidgetId w)
{
	WidgetData data;
	std::thread t(updateDataForWidget, w, data);
	displayStatus();
	t.join();
	processWidgetData(data);
}
```

In this case, updateDataForWidget will be called with a rvalue as a secod argument, which won't work, since it is impossible to pass a rvalue to a function that expects a non-const reference. To solve this problem, you can force the use of the reference:

```
std::thread t(updateDataForWidget, w, std::ref(data));
```

Because arguments can be passed as rvalues, you can std::move() an argument - like a unique pointer.

Transfer ownership

The ownership of a thread can be transfered with std::move. If you transfer ownership on an already associated thread, it will call std::terminate on the preexisting thread.

```
void someFunction();
void someOtherFunction();
std::thread t1(someFunction);
std::thread t2 = std::move(t1);
t1 = std::thread(someOtherFunction);
std::thread t3;
t3 = std::move(t2);
t1 = std::move(t3);    // Terminates t1
```

The move support in std::thread also allows for containers of std::thread objects, if those containers are move-aware.

```
void doWork(unsigned id);
void f() {
	std::vector<std::thread> threads;
	for(unsigned i=0; i<20; ++i) {
		threads.emplace_back(doWork, i);
	}
	for(auto& entry: threads)
		entry.join();                  // Wait for the threads to end
}
```

Thread ID

Thread identifiers are of type std::␣id and can be retrieved in two ways.

First, the identifier for a thread can be obtained from its associated std::thread object by calling the get_id() member function. If the std::thread object doesn't have an associated thread of execution, the call to get_id() returns a default-constructed std::␣id object, which indicates "not any thread."

Second, the identifier for the current thread can be obtained by calling std::this_thread:: get_id(), which is also defined in the header.

Instances of std::␣id are often used to check whether a thread needs to perform some operation. For example, if threads are used to divide work the initial thread that launched the others might need to perform its work slightly differently in the middle of the algorithm. In this case it could store the result of std::this_thread::get_id() before launching the other threads, and then the core part of the algorithm (which is common to all threads) could check its own thread ID against the stored value:

```
std::thread::id mainThread;
void corePartOfAlgorithm()
{
	if(std::this_thread::get_id() == mainThread)
	{
		doMainThreadWork();
	}
	doCommonWork();
}
```

Alternatively, the std::␣id of the current thread could be stored in a data structure as part of an operation. Later operations on that same data structure could then check the stored ID against the ID of the thread performing the operation to determine what operations are permitted/required.