C++11 Thread 1. Creating Threads

Let's look at the sample code (t1.cpp).

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

void thread_function()
{
    std::cout << "thread function\n";
}

int main()
{
    std::thread t(&thread_function);   // t starts running
    std::cout << "main thread\n";
    t.join();   // main thread waits for the thread t to finish
    return 0;
}
```

This code will print out (on linux system):

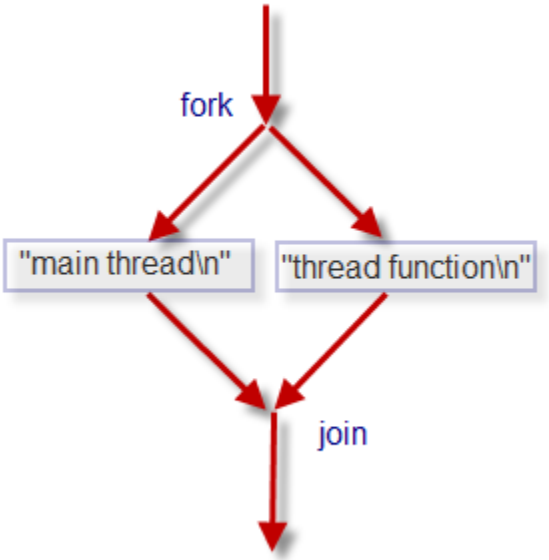
```
$ g++ t1.cpp -o t1 -std=c++11 -pthread
$ ./t2
thread function
main thread
```

First thing we want to do is creating a **thread** object (worker thread) and give it a work to do in a form of a function.

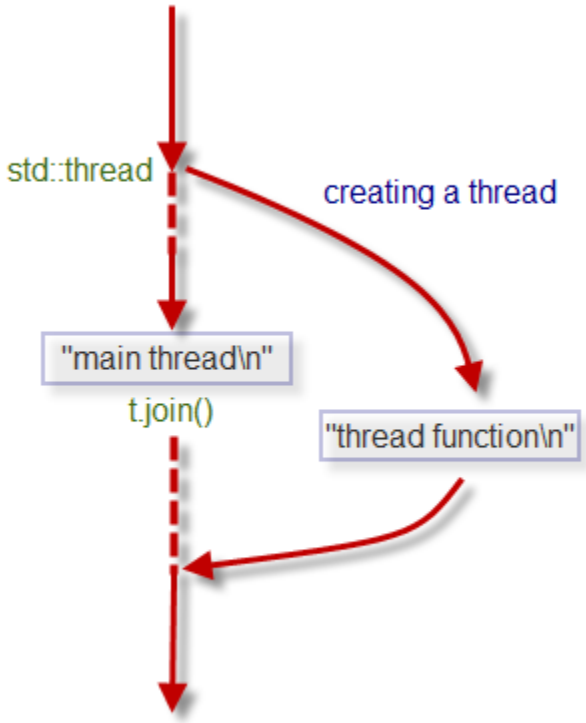
The main thread wants to wait for a thread to finish successfully. So, we used **join()**. If the initial main thread didn't wait for the new thread to finish, it would continue to the end of main() and end the program, possibly before the new thread have had a chance to run.

While the main thread is waiting, the main thread is idling. Actually, the OS may take the CPU away from the main thread.

Note that we have a new Standard C++ Library header **#include <thread>** in which the functions and classes for threads are declared.
Below is the diagram how the flow looks like.



However, in the real world, things are not that ideal and more likely to be asymmetric. Probably, it may look more like the next picture.



While the worker thread is starting via constructor **std::thread t**, there might be overhead of creating a thread (this overhead can be reduced by using thread pool). The dotted line indicates a possible blocked state.

Detaching Threads

We can make a new thread to run free to become a daemon process.

```
// t2.cpp
int main()
{
    std::thread t(&thread_function);
    std::cout << "main thread\n";
    // t.join();
    t.detach();
    return 0;
}
```

The detached child thread is now free, and runs on its own. It becomes a daemon process.

```
$ g++ t2.cpp -o t2 -std=c++11 -pthread
$ ./t2
main thread
```

Note that the detached thread didn't have a change to print its output to stdout because the main thread already finished and exited. This is one of the characteristics of multithreaded programming: we cannot be sure which thread runs first (not deterministic unless we use synchronization mechanism). In our case, because the time it takes to create a new thread, the main thread is most likely to finish ahead of our child thread.

One more thing we should note here is that even in this simple code we're sharing a common resource: std::cout. So, to make the code work properly, the main thread should allow our child thread to access the resource.

Once a thread detached, we cannot force it to join with the main thread again. So, the following line of the code is an error and the program will crash.

```
int main()
{
    std::thread t(&thread_function);
    std::cout << "main thread\n";
    // t.join();
    t.detach();
    t.join();   // Error
    return 0;
}
```

Once detached, the thread should live that way forever.
We can keep the code from crashing by checking using **joinable()**. Because it's not joinable, the join() function won't be called, and the program runs without crash.

```
int main()
{
    std::thread t(&thread_function);
    std::cout << "main thread\n";
    // t.join();
    if(t.joinable())
        t.join();
    return 0;
}
```

Passing Parameters to a thread

Here is an example of passing parameter to a thread. In this case, we're just passing a string:

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

```
#include <string>

void thread_function(std::string s)
{
    std::cout << "thread function ";
    std::cout << "message is = " << s << std::endl;
}

int main()
{
    std::string s = "Kathy Perry";
    std::thread t(&thread_function, s);
    std::cout << "main thread message = " << s << std::endl;
    t.join();
    return 0;
}
```

From the following output, we know the string has been passed to the thread function successfully.

```
thread function message is = Kathy Perry
main thread message = Kathy Perry
```

If we want to pass the string as a ref, we may want to try this:

```
void thread_function(std::string &s)
{
    std::cout << "thread function ";
    std::cout << "message is = " << s << std::endl;
    s = "Justin Beaver";
}
```

To make it sure that the string is really passed by reference, we modified the message at the end of the thread function. However, the output hasn't been changed.

```
thread function message is = Kathy Perry
main thread message = Kathy Perry
```

In fact, the message was passed by value not by reference. To pass the message by reference, we should modify the code a little bit like this using **ref**:

```
std::thread t(&thread_function, std::ref(s));
```

Then, we get modified output:

```
thread function message is = Kathy Perry
main thread message = Justin Beaver
```

There is another way of passing the parameter without copying and not sharing memory between the threads. We can use **move()**:

```
std::thread t(&thread_function, std::move(s));
```

Since the string moved from main() to thread function, the output from main does not have it any more:

```
thread function message is = Kathy Perry
main thread message =
```

Thread id

We can get id information using **this_thread::get_id()**:

```
int main()
{
    std::string s = "Kathy Perry";
    std::thread t(&thread_function, std::move(s));
    std::cout << "main thread message = " << s << std::endl;

    std::cout << "main thread id = " << std::this_thread::get_id()
<< std::endl;
```

```
    std::cout << "child thread id = " << t.get_id() << std::endl;

    t.join();
    return 0;
}
```

Output:

```
thread function message is = Kathy Perry
main thread message =
main thread id = 1208
child thread id = 5224
```

How many threads?

The thread library provides the suggestion for the number of threads:

```
int main()
{
    std::cout << "Number of threads = "
              <<  std::thread::hardware_concurrency() << std::endl;
    return 0;
}
```

Output:

```
Number of threads = 2
```