#include<iostream>

#include<thread>

using namespace std;

void m1(){

cout<<"i m thread";

}

int main(){

thread t(m1);

t.join();

return 0;

}

How to create a thread?

First, you have to include thread header in your program:

#include <thread>

When you want to create a thread, you have to create an object of a **thread** class.

//this thread does not represent any thread of execution

thread t\_empty;

As you can see, when default constructor of thread class is used, we do not pass any information to the thread. This means, that nothing is executed in this thread. We have to initialize a thread. It can be done in different ways.

Initializing thread with a function

When you create a thread, you can pass a pointer of a function to its constructor. Once thread is created, this function starts its work in a separate thread. Look on an example:

#include <iostream>

#include <thread>

using namespace std;

void threadFunc()

{

cout << "Welcome to Multithreading" << endl;

}

int main()

{

//pass a function to thread

thread funcTest1(threadFunc);

return 0;

}

Try to compile and run this program. It compiles without any errors but you will get a runtime error:

As you can see, main thread creates new thread **funcTest1** with a parameter **threadFunc**. Main thread does not wait for **funcTest1** thread termination. It continues its work. The main thread finishes execution, but **funcTest1** is still running. This causes error. All the threads must be terminated before main thread is terminated.

Join threads

Thread joining is done by using **join()** member function of a thread class:

void join();

This function returns only after all the threads are terminated. It means that the main thread will wait until child thread does not finish its execution:

Call join() for the thread, created in the previous example and run the program again:

#include <iostream>

#include <thread>

using namespace std;

void threadFunc()

{

cout << "Welcome to Multithreading" << endl;

}

int main()

{

//pass a function to thread

thread funcTest1(threadFunc);

//main is blocked until funcTest1 is not finished

funcTest1.join();

return 0;

}

As you can see, now program is executed successfully.

Joinable and not Joinable threads

After join() returns, thread becomes **not joinable.** A joinable thread is a thread that represents a thread of execution which has not yet been joined.

A thread is not joinable when it is default constructed or is moved/assigned to another thread or join() or detach() member function is called.

Not joinable thread can be destroyed safely.

You can check if a thread is joinable by using joinable() member function:

bool joinable()

This function returns true if the thread is joinable and false otherwise. It’s better to check if the thread is joinable before join() function is called:

#include <iostream>

#include <thread>

using namespace std;

void threadFunc()

{

cout << "Welcome to Multithreading" << endl;

}

int main()

{

//pass a function to thread

thread funcTest1(threadFunc);

//check if thread is joinable

if (funcTest1.joinable())

{

//main is blocked until funcTest1 is not finished

funcTest1.join();

}

return 0;

}

Detaching thread:

As we mentioned above, thread becomes not joinable after **detach()** member function is called

void detach();

This function detaches a thread from the parent thread. It allows parent and child threads to be executed independently from each other. After the call of detach() function, the threads are not synchronized in any way:

#include <iostream>

#include <thread>

using namespace std;

void threadFunc()

{

cout << "Welcome to Multithreading" << endl;

}

int main()

{

//pass a function to thread

thread funcTest1(threadFunc);

//check if thread is joinable

funcTest1.detach();

if (funcTest1.joinable())

{

//main is blocked until funcTest1 is not finished

funcTest1.join();

}

else

{

cout << "functTest1 is detached" << endl;

}

return 0;

}

You will notice that main thread is not waiting for the termination of its child thread.

Initializing thread with an object

You can initialize a thread not only with a function. You can use for this purpose function object (functor) or a member function of a class.

A functor is an object of a class that overloads operator **() -**function call operator.

If you want to initialize a thread with an object of a class, this class should overload operator(). It can be done in the following way:

class myFunctor

{

public:

void operator()()

{

cout << "This is my function object" << endl;

}

};

Now you can initialize a thread by passing an object of the class **myFunctor** to the constructor of a thread:

#include <iostream>

#include <thread>

using namespace std;

class myFunctor

{

public:

void operator()()

{

cout << "This is my function object" << endl;

}

};

int main()

{

myFunctor myFunc;

thread functorTest(myFunc);

if (functorTest.joinable())

functorTest.join();

return 0;

}

----------------------------------------------------------------

#include <iostream>

#include<thread>

using namespace std;

class Test{

public:

void operator()(){

cout<<"my thread...";

}

void m1(){

cout<<"My thread m1";

}

};

int main()

{

Test t;

thread t3(&Test::m1,t);

t3.join();

return 0;

}

-----------------------------------------------------------------------------

passing argumnet to function:

void m1(int a, int b){

cout<<"sum is :"<<a+b;

}

int main()

{

thread t3(m1,2,3);

t3.join();

return 0;

}

Thread ID

Every thread has its unique identifier. Class thread has public member function that returns the ID of the thread:

id get\_id();

The returned value is of type id that is specified in thread class.

#include <iostream>

#include <thread>

using namespace std;

void showMessage(){

cout<<"My thread"<<endl;

}

int main()

{

//create 3 different threads

thread t1(showMessage);

thread t2(showMessage);

thread t3(showMessage);

//get id of all the threads

thread::id id1 = t1.get\_id();

thread::id id2 = t2.get\_id();

thread::id id3 = t3.get\_id();

//join all the threads

if (t1.joinable())

{

t1.join();

cout << "Thread with id " << id1 << " is terminated" << endl;

}

if (t2.joinable())

{

t2.join();

cout << "Thread with id " << id2 << " is terminated" << endl;

}

if (t3.joinable())

{

t3.join();

cout << "Thread with id " << id3 << " is terminated" << endl;

}

return 0;

}

Sleeping a thread:

#include <iostream>

#include <thread>

#include <chrono>

using namespace std;

void showMessage(){

for(int i=0;i<=5;i++){

cout<<"My thread"<<endl;

std::this\_thread::sleep\_for(std::chrono::milliseconds(2000));

}

}

int main()

{

//create 3 different threads

thread t1(showMessage);

thread t2(showMessage);

thread t3(showMessage);

//get id of all the threads

thread::id id1 = t1.get\_id();

thread::id id2 = t2.get\_id();

thread::id id3 = t3.get\_id();

//join all the threads

if (t1.joinable())

{

t1.join();

cout << "Thread with id " << id1 << " is terminated" << endl;

}

if (t2.joinable())

{

t2.join();

cout << "Thread with id " << id2 << " is terminated" << endl;

}

if (t3.joinable())

{

t3.join();

cout << "Thread with id " << id3 << " is terminated" << endl;

}

return 0;

}