Start thread with member function

I am trying to construct a `std::thread` with a member function that takes no arguments and returns `void`. I can't figure out any syntax that works - the compiler complains no matter what. What is the correct way to implement `spawn()` so that it returns a `std::thread` that executes `test()`?

```
#include <thread>
class blub {
  void test() {
  }
public:
  std::thread spawn() {
    return { test };
  }
};
```

c++ multithreading c++11

edited Apr 20 '15 at 17:58 — Barry, 126k 13 177 355

asked May 20 '12 at 12:55 — abergmeier, 3,675 5 23 51

1 Do u mean the function returns void, called void or it just doesn't have any parameters. Can u add the code for what you are trying to do? – Zaid Amir May 20 '12 at 13:02

Have you tested? (I haven't yet.) Your code seems to rely on the RVO (return-value-optimzation), but I don't think you are supposed to do so. I think using `std::move( std::thread(func) );` is better, for `std::thread` doesn't have a copy-constructor. – RnMss Oct 10 '13 at 11:28

2 @RnMss: you can rely on RVO, using `std::move` is redundant in this case - were this not true, and there was no copy constructor, the compiler would give an error anyway. – Qualia Oct 22 '15 at 13:19

4 Answers

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

class bar {
public:
  void foo() {
    std::cout << "hello from member function" << std::endl;
  }
};

int main()
{
  std::thread t(&bar::foo, bar());
  t.join();
}
```

EDIT: Accounting your edit, you have to do it like this:

```
  std::thread spawn() {
    return std::thread(&blub::test, this);
  }
```

UPDATE: I want to explain some more points, some of them have also been discussed in the comments.

The syntax described above is defined in terms of the INVOKE definition (§20.8.2.1):

> Define INVOKE (f, t1, t2, ..., tN) as follows:
>
> - (t1.*f)(t2, ..., tN) when f is a pointer to a member function of a class T and t1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;
> - ((*t1).*f)(t2, ..., tN) when f is a pointer to a member function of a class T and t1 is not one of the types described in the previous item;
> - t1.*f when N == 1 and f is a pointer to member data of a class T and t 1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;
> - (*t1).*f when N == 1 and f is a pointer to member data of a class T and t 1 is not one of the types described in the previous item;
> - f(t1, t2, ..., tN) in all other cases.

Another general fact which I want to point out is that by default the thread constructor will copy all arguments passed to it. The reason for this is that the arguments may need to outlive the calling thread, copying the arguments guarantees that. Instead, if you want to really pass a reference, you can use a `std::reference_wrapper` created by `std::ref`.

```
std::thread (foo, std::ref(arg1));
```

By doing this, you are promising that you will take care of guaranteeing that the arguments will still exist when the thread operates on them.

Note that all the things mentioned above can also be applied to `std::async` and `std::bind`.

edited Nov 5 '13 at 10:26

answered May 20 '12 at 13:07

inf
14.3k 7 45 85

1 At least this way it compiles. Though I have no idea why you are passing the instance as the second argument. – abergmeier May 20 '12 at 13:37

10 @LCID: The multi-argument version of the `std::thread` constructor works as if the arguments were passed to `std::bind`. To call a member function, the first argument to `std::bind` must be a pointer, reference, or shared pointer to an object of the appropriate type. – Dave S May 20 '12 at 13:49

Where do you take it from that the constructor acts like an implicit `bind`? I can't find that anywhere. – Kerrek SB May 20 '12 at 13:58

3 @KerrekSB, compare [thread.thread.constr]p4 with [func.bind.bind]p3, the semantics are quite similar, defined in terms of the INVOKE pseudocode, which defines how member functions are called – Jonathan Wakely May 20 '12 at 14:39

2 remember that not static member functions as first parameter take instance of class (it's not visible for programmer), so when passing this method as raw function you will always encounter a problem during compilation and declaration mismatch. – zoska Oct 10 '13 at 11:56

Since you are using C++11, lambda-expression is a nice&clean solution.

```
class blub {
    void test() {}
  public:
    std::thread spawn() {
      return std::thread( [this] { this->test(); } );
    }
};
```

since `this->` can be omitted, it could be shorten to:

```
std::thread( [this] { test(); } )
```

or just

```
std::thread( [=] { test(); } )
```

edited Aug 20 '15 at 2:26

answered Oct 10 '13 at 11:37

RnMss
1,374 1 13 30

3 In general, you shouldn't use `std::move` when returning a local variable by value. This actually inhibits RVO. If you just return by value (without the move) the compiler may use RVO, and if it doesn't the standard

says it has to invoke move semantics. – zmb Oct 10 '13 at 11:53

@zmb, with the exception that you want code to compile on VC10, you have to move if the return type is not CopyConstructable. – abergmeier Oct 10 '13 at 13:20

@zmb `This actually inhibits RVO.` Yes it does. But that is a hack, and RVO is a hack, too. It was invented for the pre-C++11 days, where people hadn't got a better idea. And then RValue-Ref came to rescue. So, I'd suggest keep using move-constructor and wait for someday that compilers use RVO for move-constructors only, instead of the copy ones. :) – RnMss Oct 10 '13 at 18:13

3 RVO still generates better code than move semantics, and is not going away. – Jonathan Wakely Oct 9 '14 at 10:41

1 Be careful with `[=]` . With that you can inadvertently copy a huge object. In general, it's a *code smell* to use `[&]` or `[=]` . – RustyX Sep 9 '16 at 8:37

Here is a complete example

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

class Wrapper {
   public:
      void member1() {
          std::cout << "i am member1" << std::endl;
      }
      void member2(const char *arg1, unsigned arg2) {
          std::cout << "i am member2 and my first arg is (" << arg1 << ") and
second arg is (" << arg2 << ")" << std::endl;
      }
      std::thread member1Thread() {
          return std::thread([=] { member1(); });
      }
      std::thread member2Thread(const char *arg1, unsigned arg2) {
          return std::thread([=] { member2(arg1, arg2); });
      }
};
int main(int argc, char **argv) {
   Wrapper *w = new Wrapper();
   std::thread tw1 = w->member1Thread();
   std::thread tw2 = w->member2Thread("hello", 100);
   tw1.join();
   tw2.join();
   return 0;
}
```

Compiling with g++ produces the following result

```
g++ -Wall -std=c++11 hello.cc -o hello -pthread

i am member1
i am member2 and my first arg is (hello) and second arg is (100)
```

answered Aug 24 '15 at 7:31

hop5
147 1 2

---

not really relevant to the OP question, but why do you allocate Wrapper on the heap (and not deallocate it)? do you have java/c# background? – Alessandro Teruzzi Oct 13 '16 at 13:34

Some users have already given their answer and explained it very well.

I would like to add few more things related to thread.

1. How to work with functor and thread. Please refer to below example.
2. The thread will make its own copy of the object while passing the object.

```
#include<thread>
#include<Windows.h>
#include<iostream>

using namespace std;

class CB
{

public:
    CB()
    {
        cout << "this=" << this << endl;
    }
    void operator()();
};

void CB::operator()()
{
    cout << "this=" << this << endl;
    for (int i = 0; i < 5; i++)
    {
        cout << "CB()=" << i << endl;
        Sleep(1000);
    }
}
```

```
void main()
{
    CB obj;     // please note the address of obj.

    thread t(obj); // here obj will be passed by value
                   //i.e. thread will make it own local copy of it.
                    // we can confirm it by matching the address of
                    //object printed in the constructor
                    // and address of the obj printed in the function

    t.join();
}
```

Another way of achieving the same thing is like:

```
void main()
{
    thread t((CB()));

    t.join();
}
```

But if you want to pass the object by reference then use the below syntax:

```
void main()
{
    CB obj;
    //thread t(obj);
    thread t(std::ref(obj));
    t.join();
}
```

answered Feb 20 at 18:43

Mohit

321 2 4