Callbacks in C++

Vitali Tkachenka

About me

Vitali Tkachenka

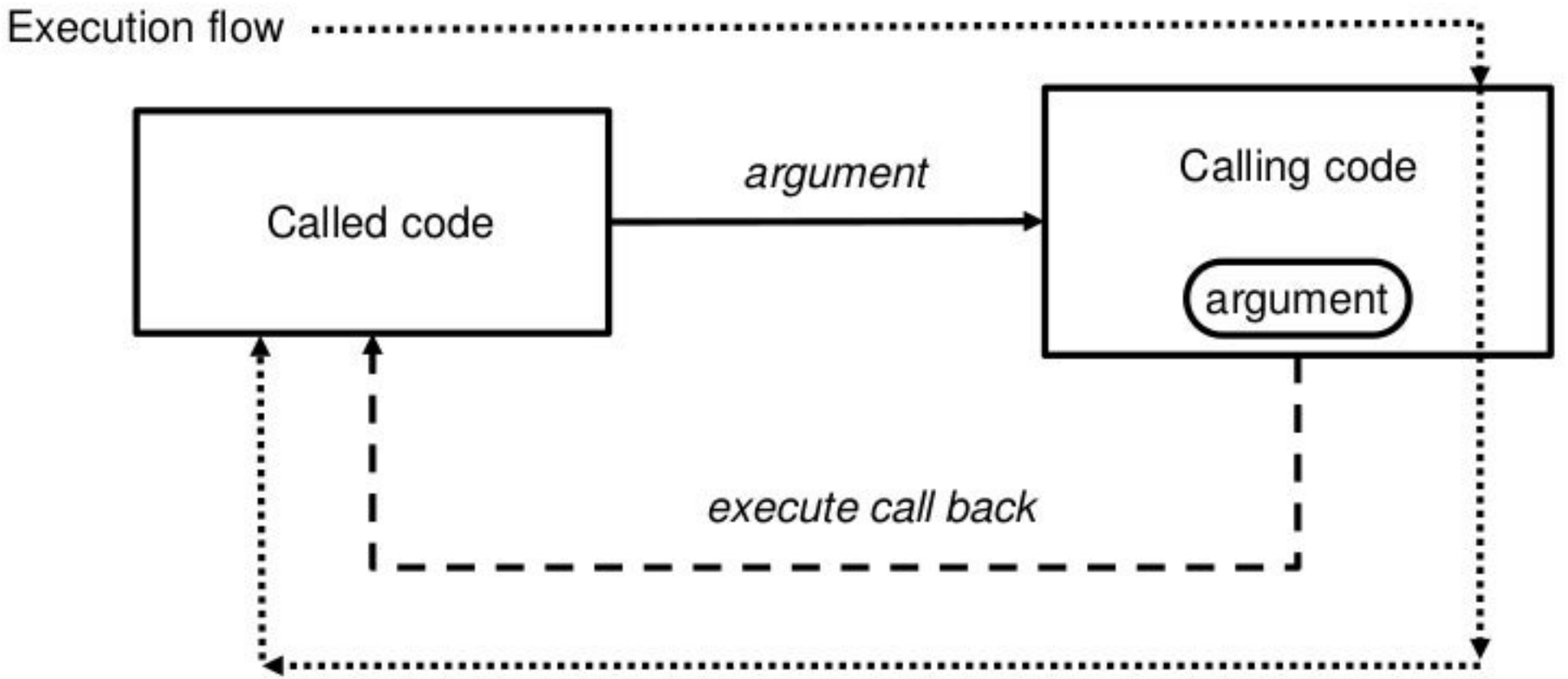
Lead software engineer

Minsk, Belarus

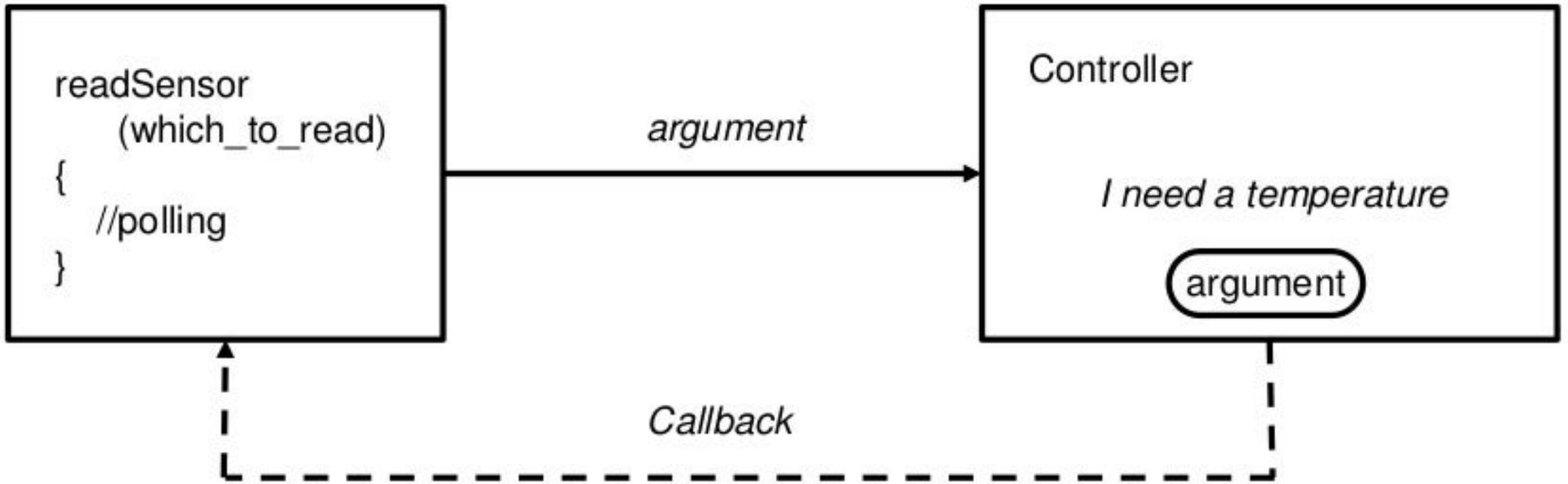
corehard.by

Introduction

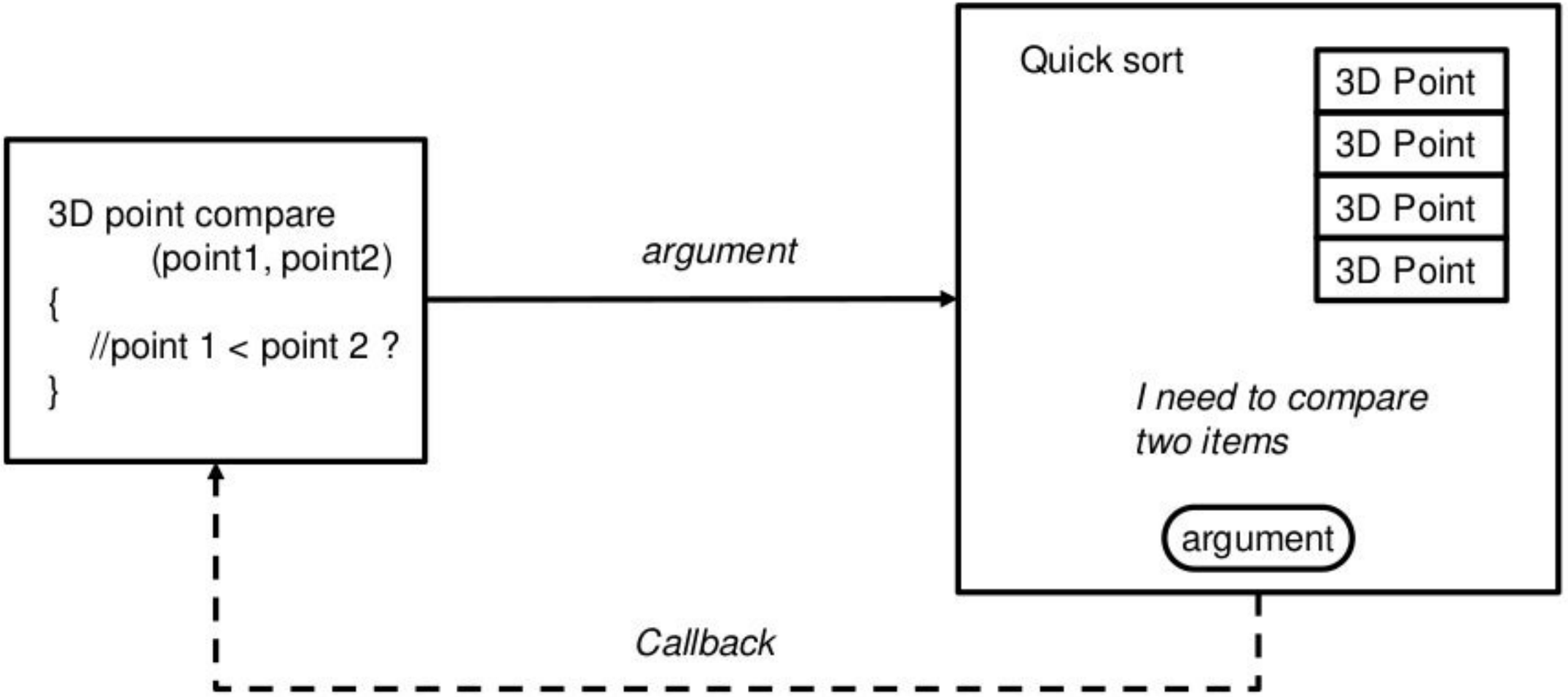
Callback is any executable code that is passed as an argument to other code that is expected to call back (execute) the argument at a given time.



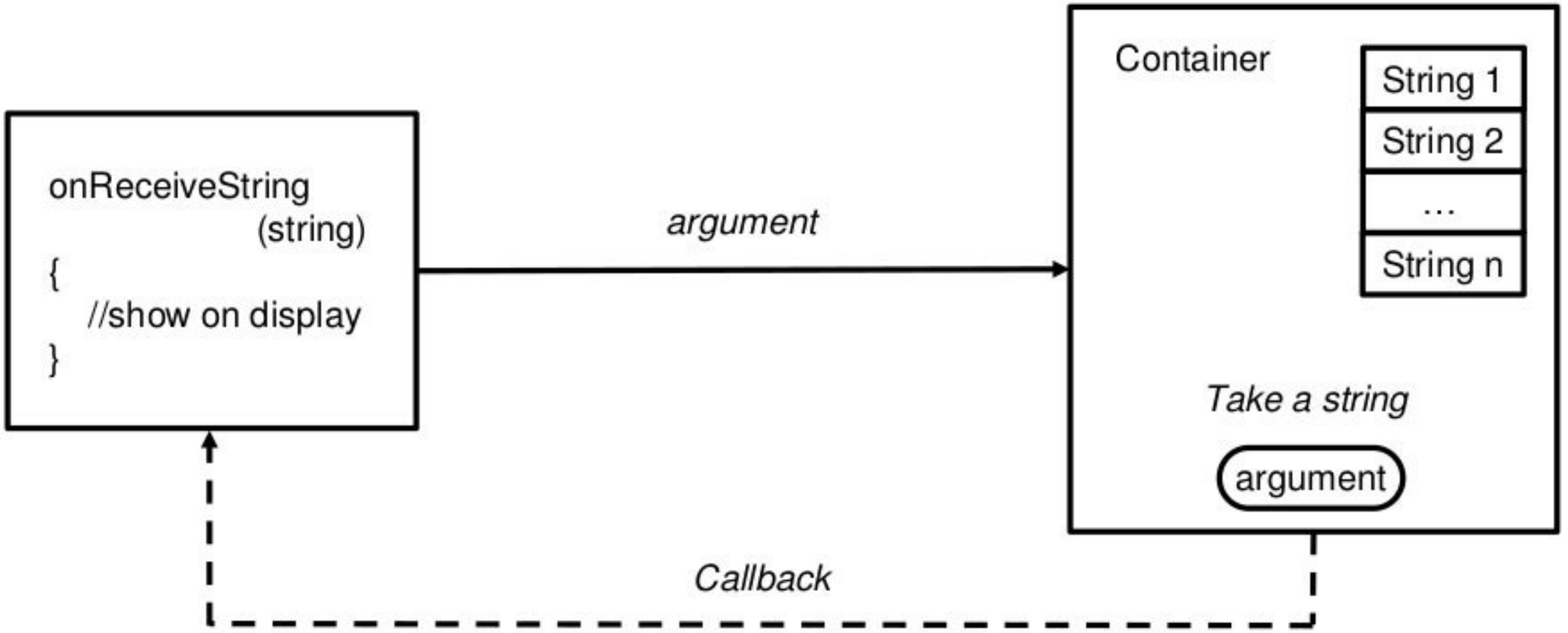
Use case: value inquiry



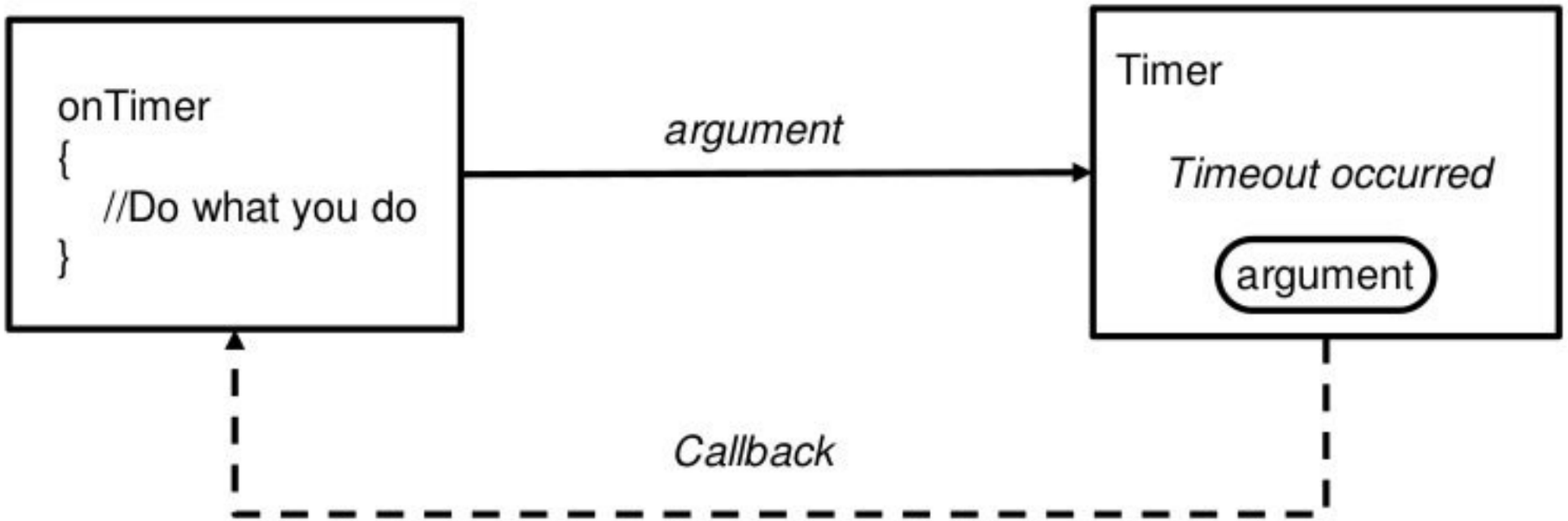
Use case: calculation inquiry



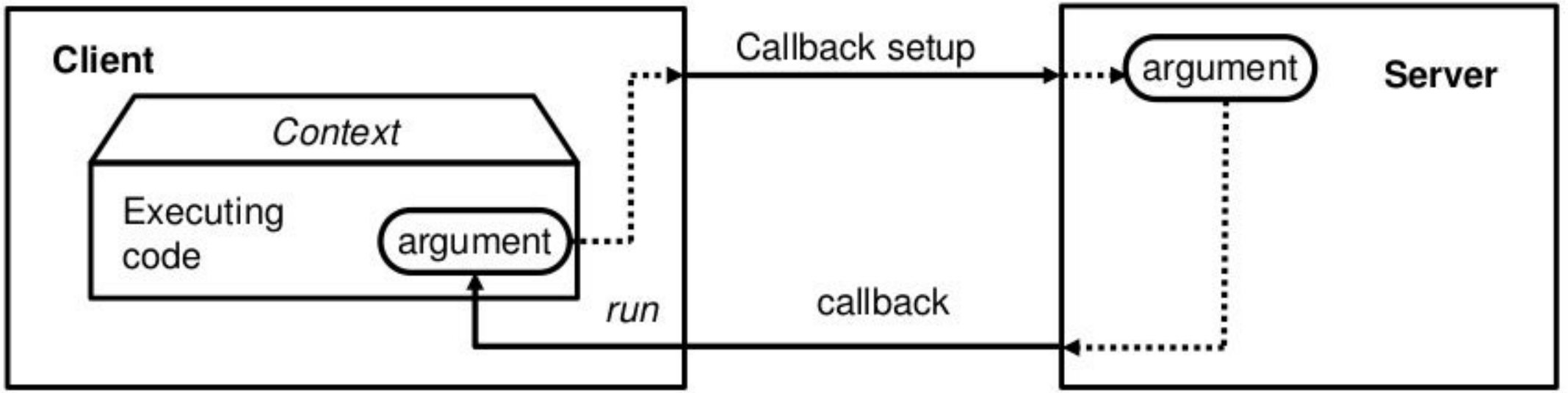
Use case: enumeration of items



Use case: event notification



Callback model



1. How to wrap the executing code?
2. How to keep the code as an argument?
3. How to pass the context?

Terms and definitions: entities

- **Client** – a module that implements executable code.
- **Server** – a module that implements a call of the executable code (callback).
- **Argument** – a stored entry to the callback code.
- **Callback setup** – a procedure of the argument saving.
- **Context** – a set of variable values and states that influences callback function behavior.

Terms and definitions: call types

- **Blocking (synchronous) callback:** if client calls some server function, the callback is invoked before the end of function.

- **Deferred (asynchronous) callback:** can be invoked at any time.

```
void DoCallback()
{
    //Callback implementation
}
void Execute()
{
    //...
    DoCallback(); //Synchronous
    //...
}
```

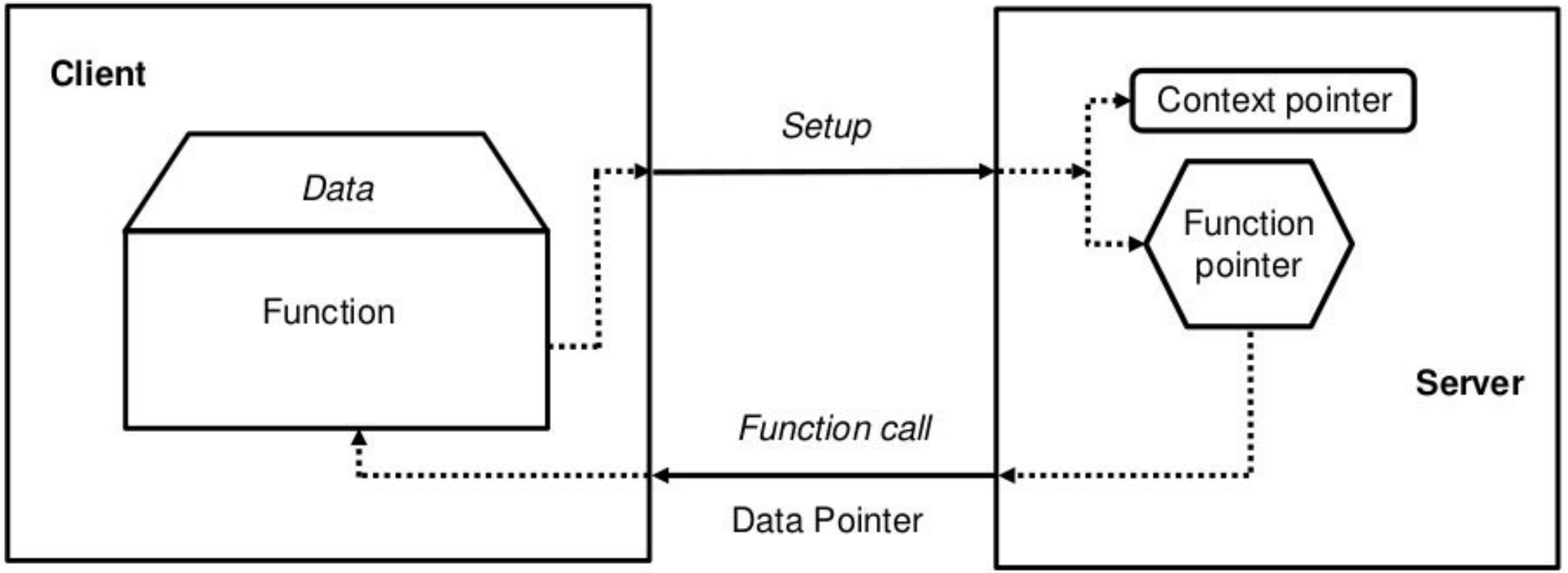
```
void DoCallback()
{
    //Callback implementation
}
void Execute()
{
    //...
    new std::thread(DoCallback); //Asynchronous
    //...
}
```

Terms and definitions: server as API

- **System API:** the server declares an interface as a set of functions that support standard call protocol.

- **C++ API:** the server declares interface as C++ definitions (classes, functions, enumerations…).

Way 1: pointer to function



Pointer to function: server implementation

```
using ptr_callback_t = void(*)(int eventID, void* clientContext);//type is pointer to the function

ptr_callback_t g_clientCallback = nullptr; //declaration of pointer variable

void* g_clientContext = nullptr; //pointer to the client context

void SetCallback(ptr_callback_t clientCallback, void* clientContext) //Callback setup
{
    g_clientCallback = clientCallback; g_clientContext = clientContext;
}

void ServerStart()
{
    int eventID = 0;
    //Server does some work
    if (g_clientCallback) g_clientCallback(eventID, g_clientContext); //callback invoke
}

void WaitServerFinish()
{ //Wait some event...}
```

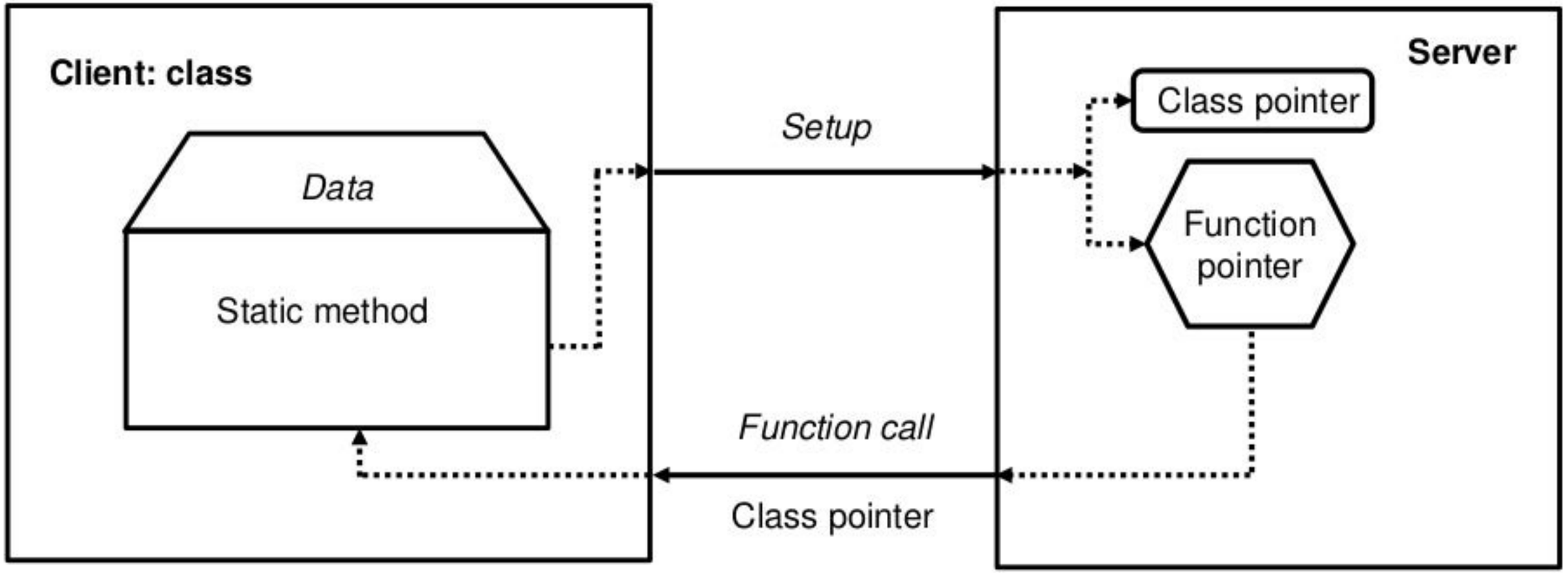
Pointer to function: client implementation

```
struct ClientData
{
    //some context data
};

void CallbackHandler(int eventID, void* someData) //Declare function for callback handler
{
    //It will be called by server
    ClientData* clientData = (ClientData*)someData;   //cast to client data
}

int main()
{
    ClientData clientData;

    SetCallback(CallbackHandler, &clientData); //callback setup

    ServerStart();
    WaitServertFinish();
}
```

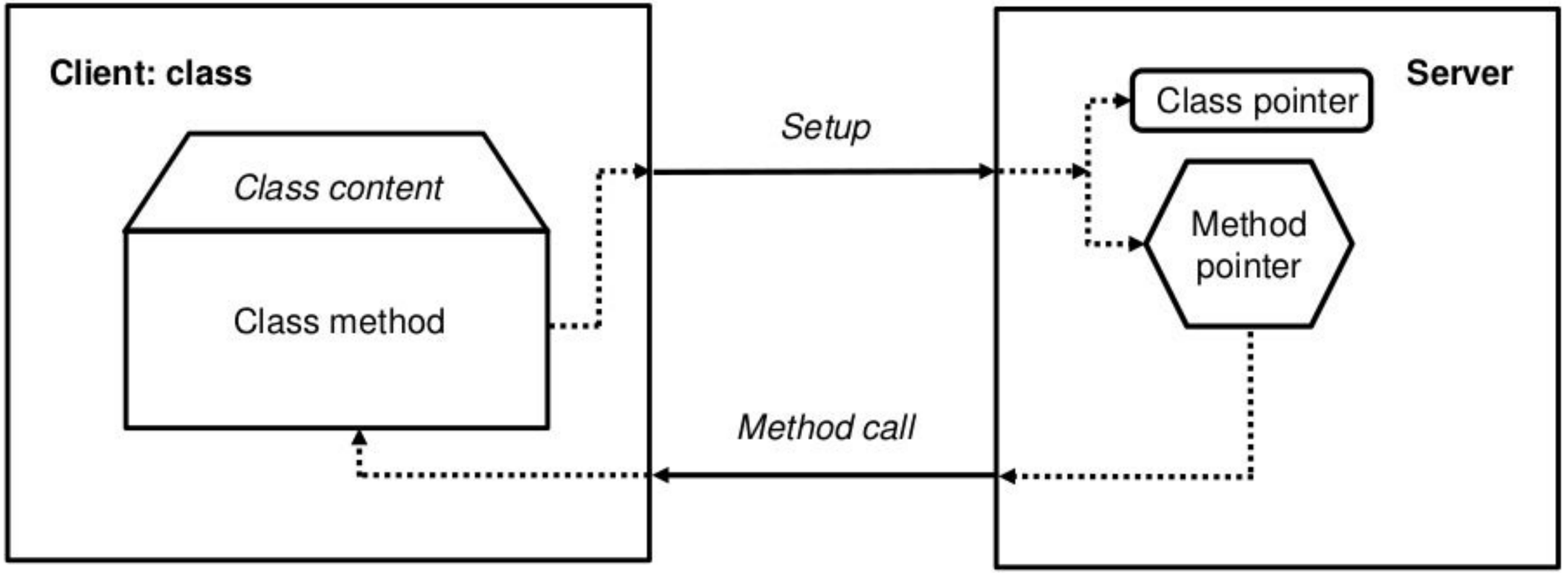
Pointer to function: pros and cons

+	-
Simple implementation	Server has to keep a client context
Server and client are independent	Clumsy way of the context translating
Can be used for system API	Does not follow object-oriented paradigm
C code compatibility	

Way 2: pointer to the class static method



Way 3: pointer to the class method



Pointer to class method: server implementation

```
class Client;

class Server
{
public:
    using ptr_method_callback_t = void(Client::*)(int); //pointer to the method of class-client

    void setCallback(Client* ptrCallbackClass, ptr_method_callback_t ptrCallbackMethod) {
        ptrClientCallbackClass = ptrCallbackClass; ptrClientCallbackMethod = ptrCallbackMethod;
    }

    void start() {
        int eventID = 0;
        //Some actions
        if (ptrClientCallbackClass) {
            (ptrClientCallbackClass->*ptrClientCallbackMethod)(eventID);
        }
    }
private:
    Client* ptrClientCallbackClass = nullptr;
    ptr_method_callback_t ptrClientCallbackMethod = nullptr;
};
```

Pointer to class method: client implementation

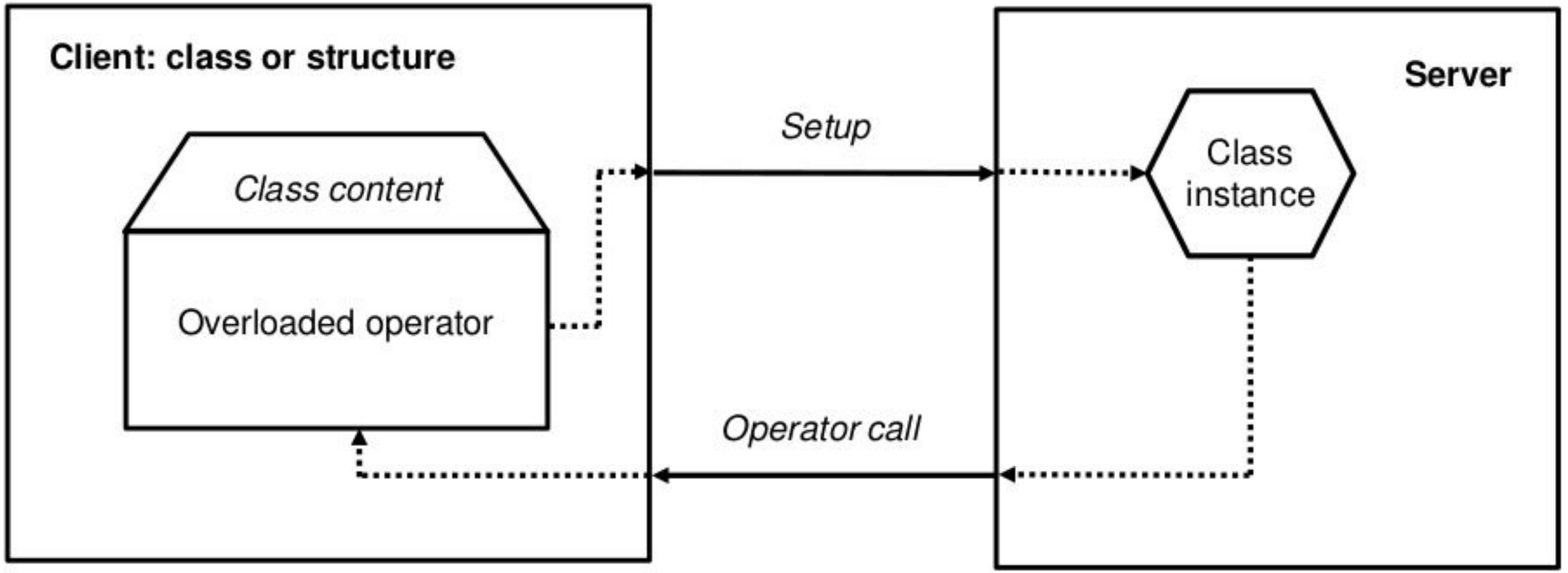
```
class Client
{
public:
    virtual void callbackHandler1(int eventID);
    virtual void callbackHandler2(int eventID);
};

class Client1 : public Client
{
public:
    void callbackHandler1(int eventID) override;
};

class Client2 : public Client
{
public:
    void callbackHandler2(int eventID) override;
};
```

```
int main()
{

Server server;

Client  client;
Client1 client1;
Client2 client2;

server.setCallback(&client,  &Client::callbackHandler1);

server.setCallback(&client,  &Client::callbackHandler2);

server.setCallback(&client1, &Client::callbackHandler1);

server.setCallback(&client1, &Client::callbackHandler2);

server.setCallback(&client2, &Client::callbackHandler1);

server.setCallback(&client2, &Client::callbackHandler2);

}
```

Pointer to class method: pros and cons

+	-
No need to translate the context	Complex implementation
Flexibility	Name of base client class must be declared in server
	Client have to inherit base class
	Server must store both method pointer and client instance

Way 4: function object (functor)



Function object: implementation

```
class Client
{
public:
    void operator() (int eventID)
    {
    //It will be called by server
    };
};
```

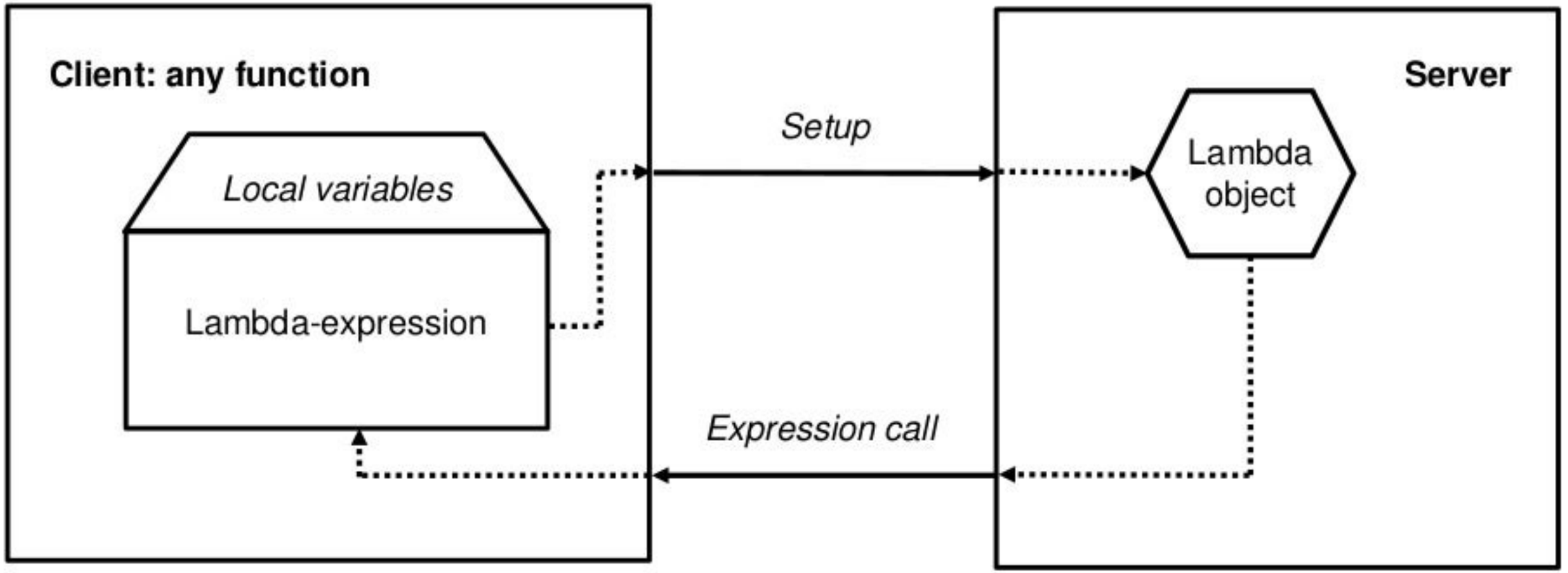
```
int main()
{
    Server server;
    Client client;
    server.setCallback(client);
    server.start();
}
```

```
class Server
{
public:
    void setCallback(const Client& callback)
    {
        callbackObject = callback;
    }

    void start()
    {
        int eventID = 0;
        //Some actions
        callbackObject(eventID);

    }
private:
    Client callbackObject;
};
```

Function object: pros and cons

+	-
Simple and clear implementation	Server and client are bound via common functional object
No need to translate the context	Cannot be implemented in API
Convenient for using in templates	
Low latency	

Way 5: lambda expression



Lambda expression: server implementation

```
class Server
{
public:
    using ptr_callback_t = void(*) (int);

    void setCallback(ptr_callback_t clientCallback)
    {
        storedCallback = clientCallback;
    }

    void start()
    {
        int eventID = 0;
        //Some actions
        if (storedCallback) storedCallback(eventID);
    }

private:
    ptr_callback_t storedCallback = nullptr;
};
```

C++ standard allows conversion from lambda to the function pointer only in case lambda does not capture variables.

Therefore, server implementation is the same as the implementation for pointer to any other function.

Lambda expression: client implementation

```
int main()
{
Server server;

auto callbackHandler = [](int eventID)
{
//It will be called by the server
};

server.setCallback(callbackHandler);

server.start();
}
```

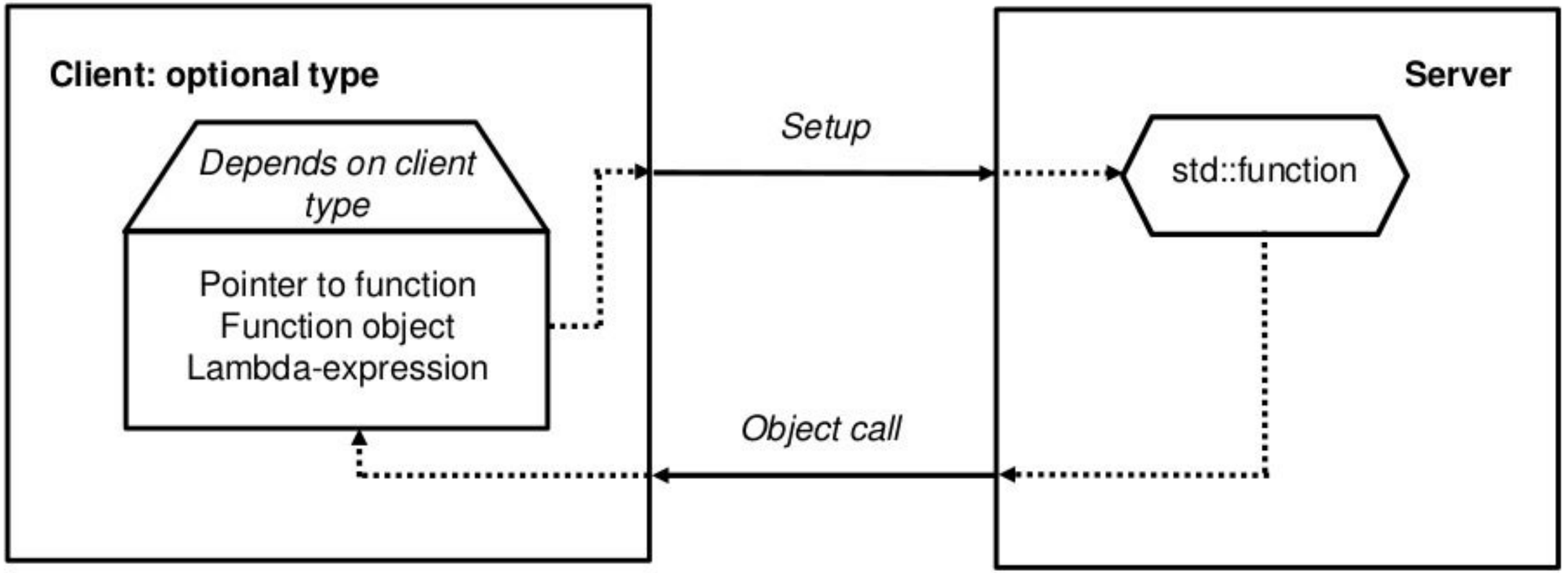
If lambda tries to capture something, this code will not be compiled because C++ standard allows conversion lambda to the pointer only if it does not capture variables.

To keep lambda with capture, the server have to use std::function

Lambda-expression: pros and cons

+	-
Simple implementation	To call lambda with variable capture, the server must use *std::function*
Smart way of context management	

Way 6: std::function



std::function: server implementation

```
#include <functional>

class Server
{
public:
    using callback_t = std::function<void(int)>;

    void setCallback(callback_t callback) { callbackHandler = callback; }

    void start()
    {
        int eventID = 0;
        //Some actions
        if (callbackHandler)
            callbackHandler(eventID);
    }
private:
    callback_t callbackHandler;
};
```

std::function: client implementation

```
class Client {
public:
    void operator()(int eventID);
    static void staticCallbackHandler(int eventID);
    void setLambdaCallback(Server* server) {

        server->setCallback([this](int eventID) {
            processedEvent = eventID; methodCallbackHandler(eventID);
        });
    }
private:
    int processedEvent;
    void methodCallbackHandler(int eventID);
};

int main()
{
    Server server;   Client client;

    server.setCallback(client.staticCallbackHandler); //The handler is a pointer to function.
    server.setCallback(client); //The handler is a function object.
    server.setCallback([](int eventID) {}); //The handler is a lambda function.
    client.setLambdaCallback(&server); //The handler is a lambda function inside a client.
}
```

As 'this' is captured, we have the access to the all class variables and methods.
We don't need to use qualificator.

std::function: pros and cons

+	-
Polymorphism: the server supports callback for different argument types	A comparatively high latency
	Template implementation restrictions

Comparison

	Function pointer	Function object	Class method	Lambda	std:: function
Simple implementation	●	●		●	○
Component independence	●		○	○	
No context translation		●	●	●	●
Safety		●	○	●	●
Flexibility	○		●	●	●
Polymorphism			○		●
Low latency	○	●	○	○	
System API	●			○	
C++ API	●		●	●	

Legend:

● *Complete support*

○ *Partial support*

▭ *No support*

What to chose?

Most important requirement	Solution
System API implementation	Function pointer
Low latency	Function object
Flexibility	Class method
Polymorphism	std::function

Integral estimation

Legend: *2 - Complete support; 1 – partially support; 0 – no support* ***IF*** *– importance factor*

	IF	Function pointer	Function object	Class method	Lambda	std:: function
Simple implementation	1	2	2	0	2	1
Safety	1	0	2	1	2	2
Flexibility	1	1	0	2	2	2
Summary		**3**	**4**	**3**	**6**	**5**

Now

inversion: *2 --> 0; 1 --> -1; 0 --> -2*

Summary		-1	-2	-3	-2	-4
Low latency	1	-1	0	-1	-1	-2
System API	1	0	-2	-2	-1	-2

May be

Many thanks!

Questions welcome :)

Vitali Tkachenka

Tkachenko_vitaliy@hotmail.com

+375-29-360-19-95