بسم الله الرحمن الرحیم

مهسان

تکـــیه‌گـاه شمـــا
در دنیای هوشمند

Some needed Libraries

An extremely brief to introduction of Libraries

mahsan.co

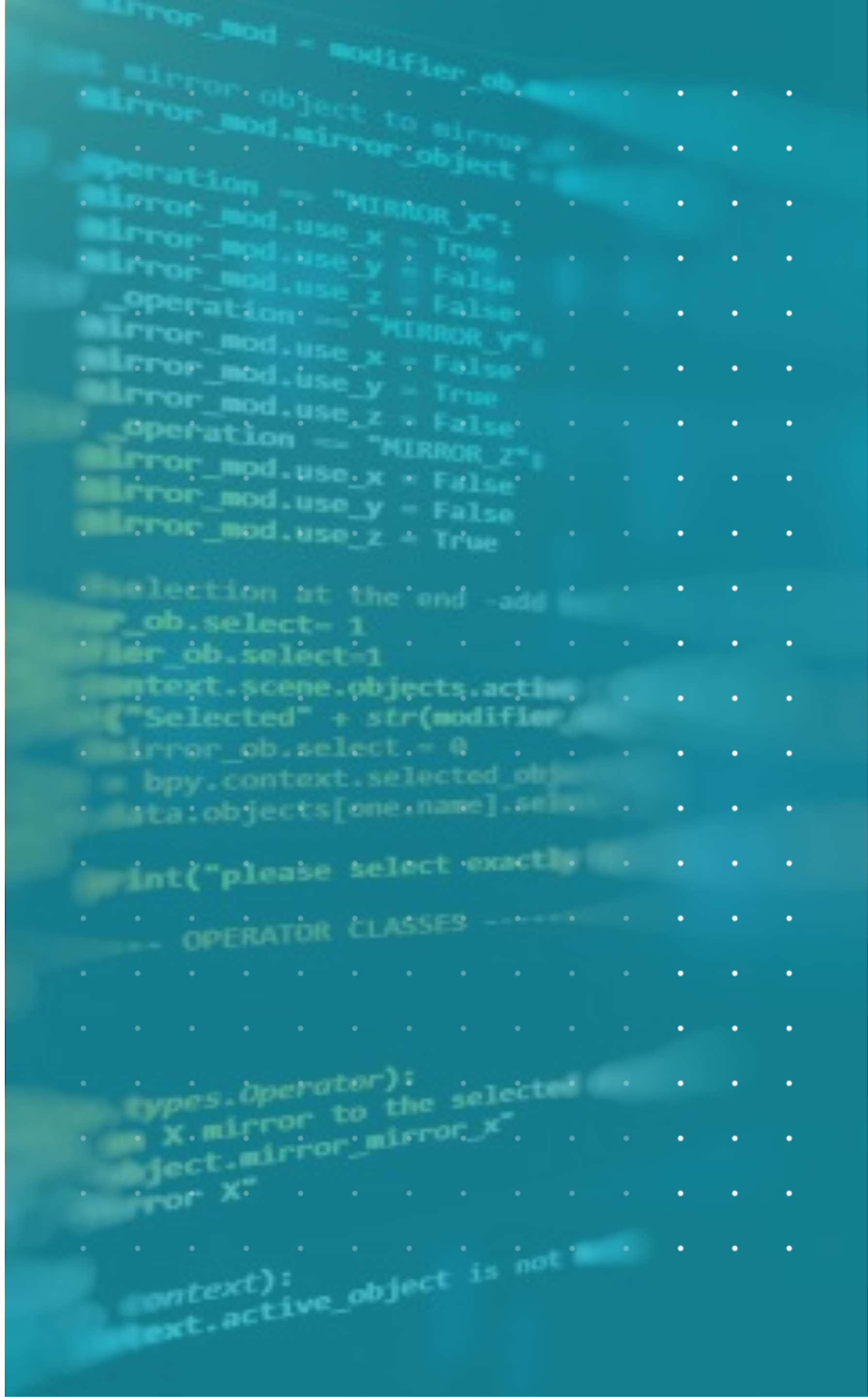
سید سروش
حسین‌علی‌پور

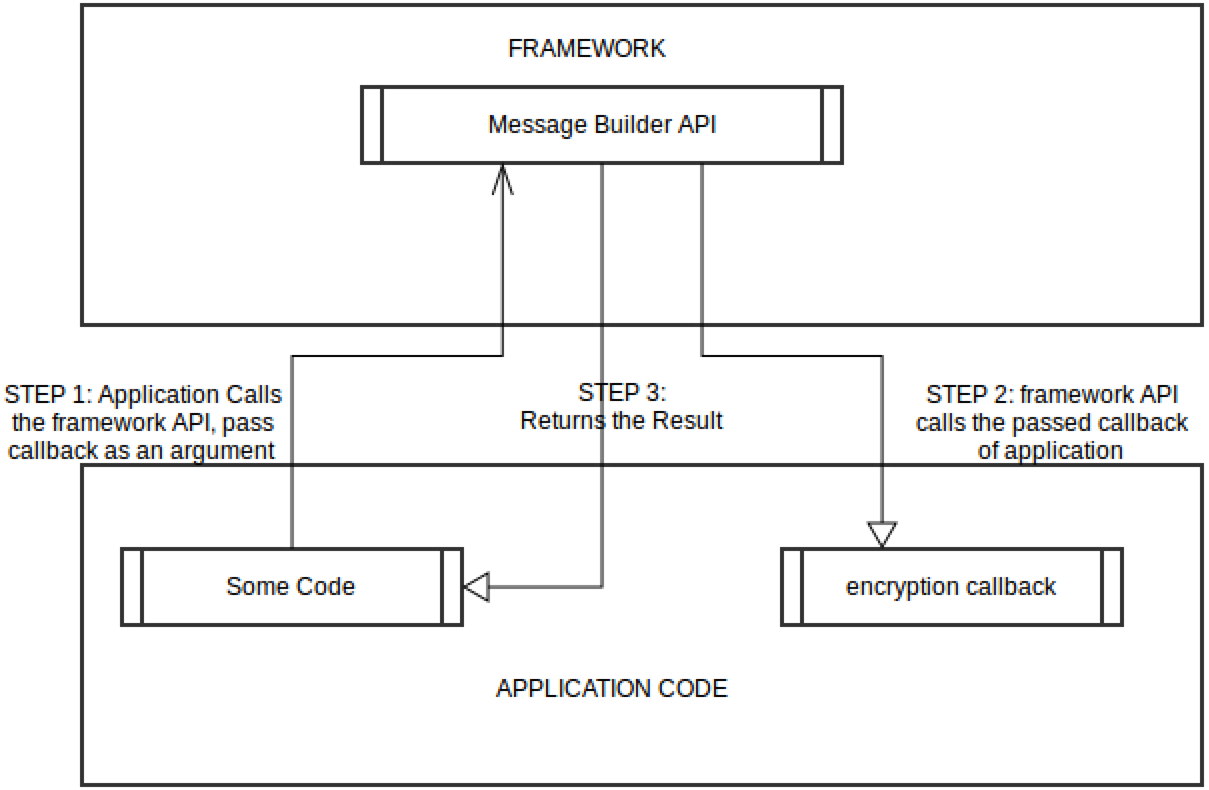
Function Pointers

Designing Callbacks in C++

انگیزه

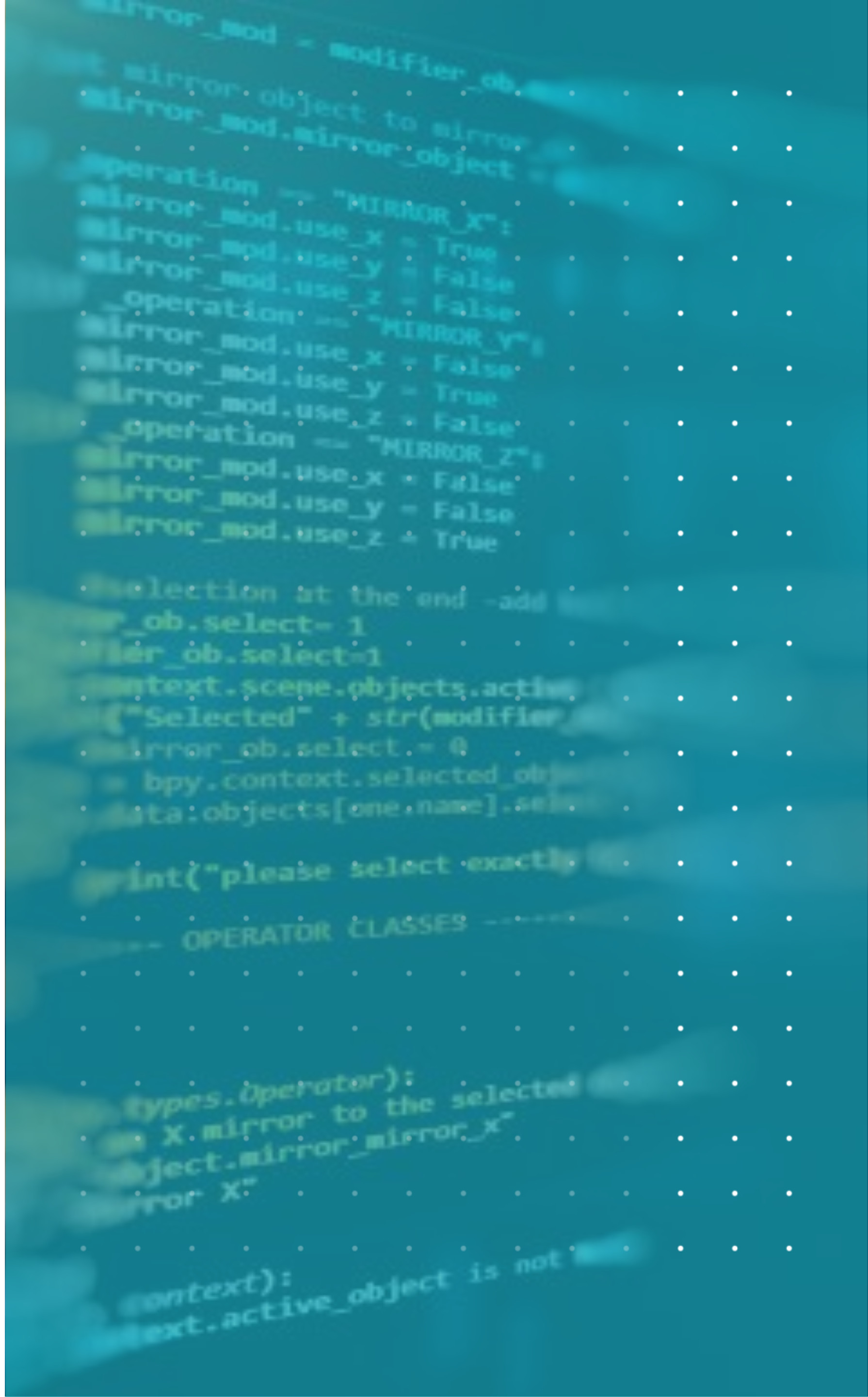
- Callback is a function that we pass to another APIs as argument while calling them. Now these APIs are expected to use our provided callback at some point.



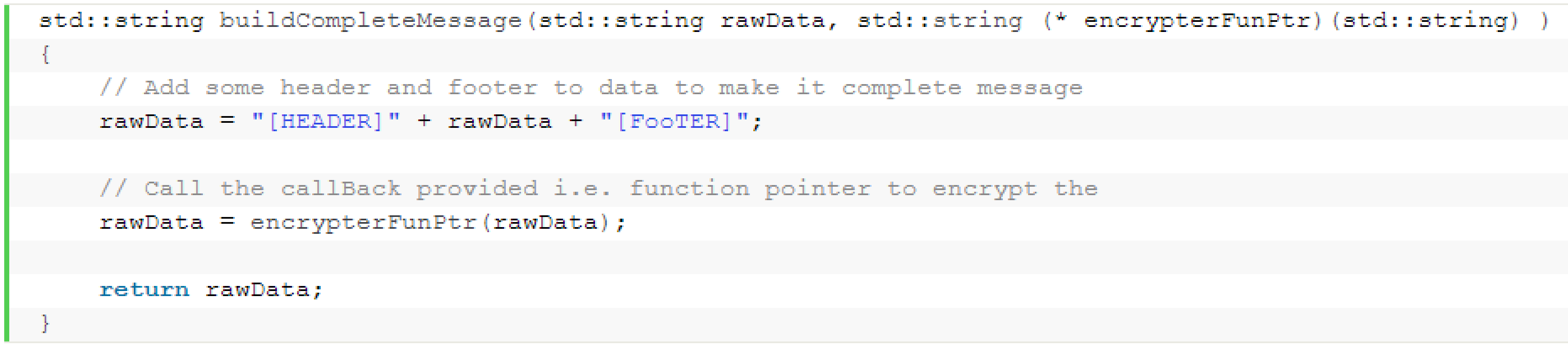

FRAMEWORK
Message Builder API
STEP 1: Application Calls the framework API, pass callback as an argument
STEP 3: Returns the Result
STEP 2: framework API calls the passed callback of application
Some Code
encryption callback
APPLICATION CODE

One Example

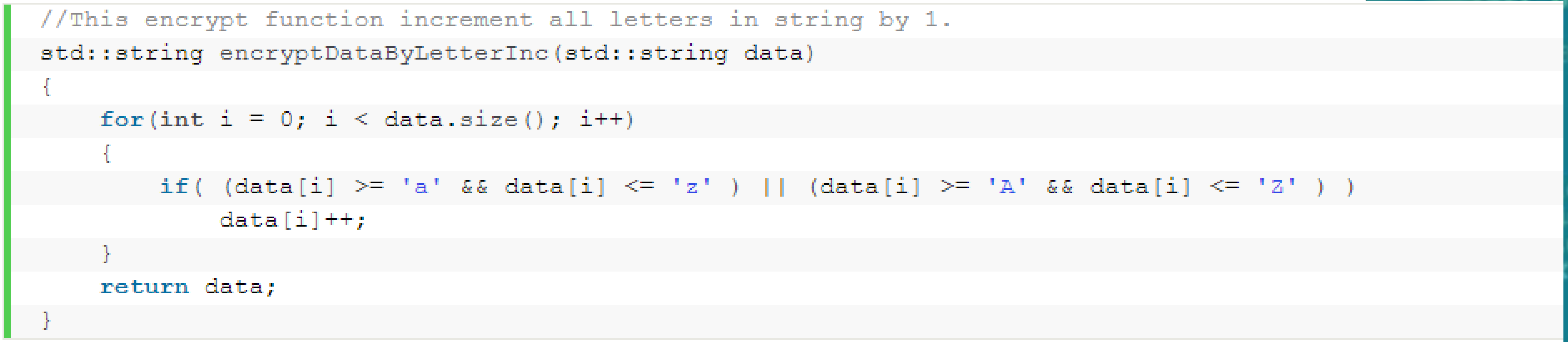
- From a framework we got an API that can build complete message from provided raw data. This API will perform following steps
    - Add header and footer in raw Data to make the message.
    - Encrypt the complete message.
    - Return the message



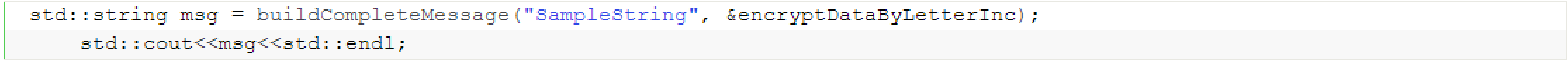
ادامه مثال



```cpp
std::string buildCompleteMessage(std::string rawData, std::string (* encrypterFunPtr)(std::string) )
{
    // Add some header and footer to data to make it complete message
    rawData = "[HEADER]" + rawData + "[FooTER]";

    // Call the callBack provided i.e. function pointer to encrypt the
    rawData = encrypterFunPtr(rawData);

    return rawData;
}
```



```cpp
//This encrypt function increment all letters in string by 1.
std::string encryptDataByLetterInc(std::string data)
{
    for(int i = 0; i < data.size(); i++)
    {
        if( (data[i] >= 'a' && data[i] <= 'z' ) || (data[i] >= 'A' && data[i] <= 'Z' ) )
            data[i]++;
    }
    return data;
}
```

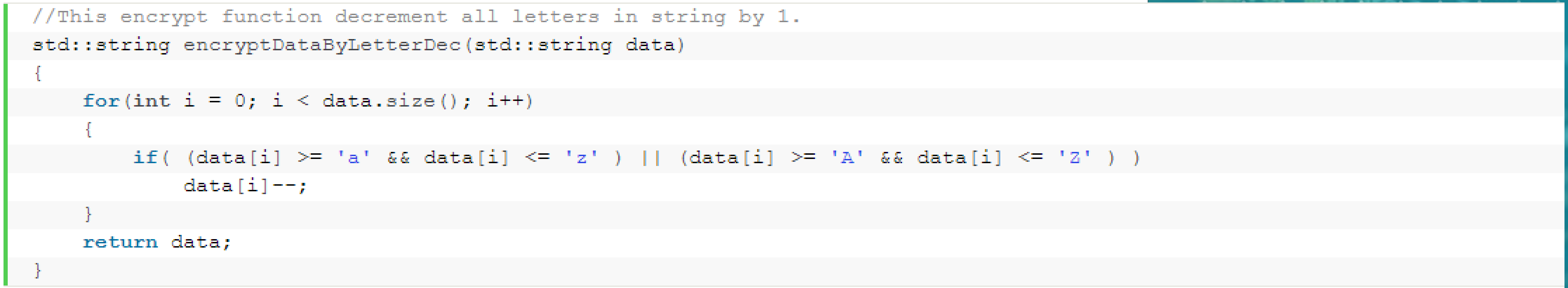


```
std::string msg = buildCompleteMessage("SampleString", &encryptDataByLetterInc);
     std::cout<<msg<<std::endl;
```

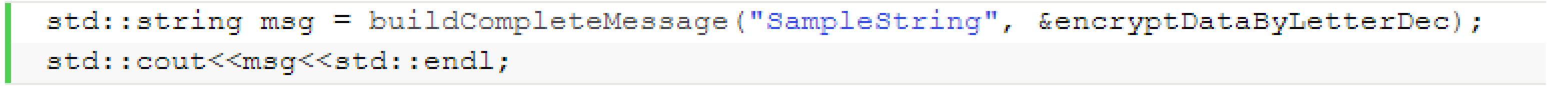
Output:

```
[IFBEFS]TbnqmfTusjoh[GppUFS]
```

Another Encryptor



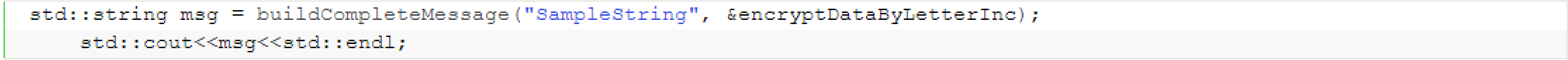
```
//This encrypt function decrement all letters in string by 1.
std::string encryptDataByLetterDec(std::string data)
{
    for(int i = 0; i < data.size(); i++)
    {
        if( (data[i] >= 'a' && data[i] <= 'z' ) || (data[i] >= 'A' && data[i] <= 'Z' ) )
            data[i]--;
    }
    return data;
}
```



```
std::string msg = buildCompleteMessage("SampleString", &encryptDataByLetterDec);
std::cout<<msg<<std::endl;
```

Output:

[GD@CDQ]R`lokdRsqhmf[EnnSDQ]



```
std::string msg = buildCompleteMessage("SampleString", &encryptDataByLetterInc);
    std::cout<<msg<<std::endl;
```

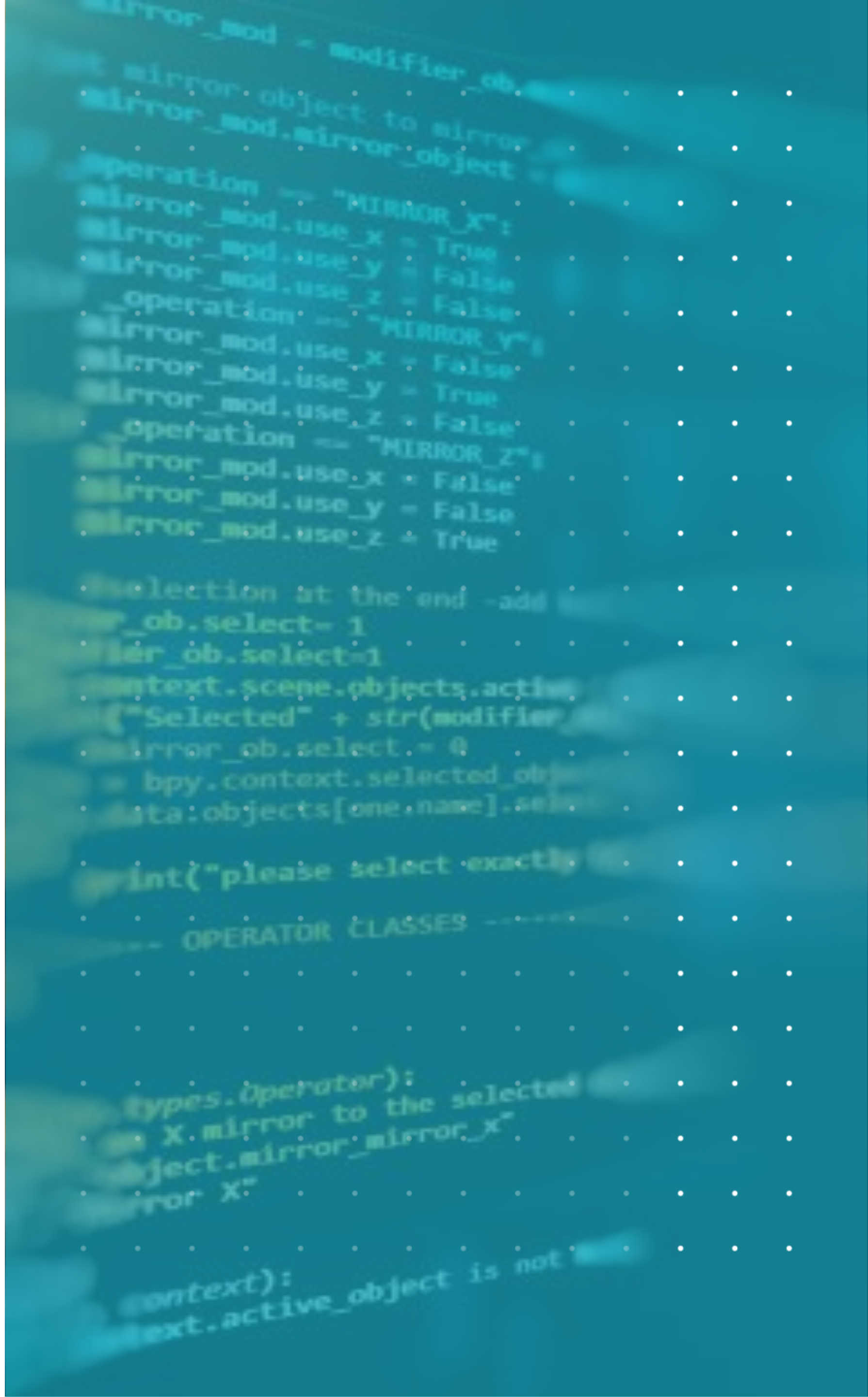
Designing Callbacks

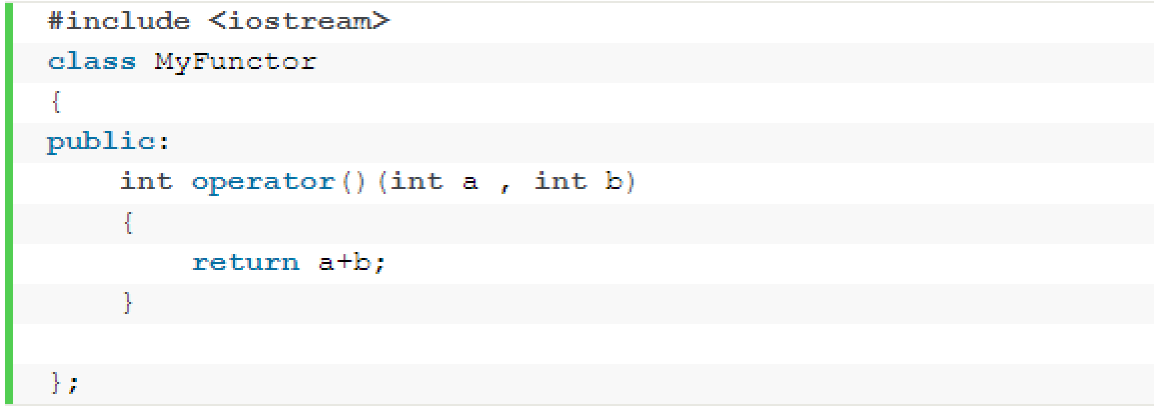
Function Objects & Functors

Functor

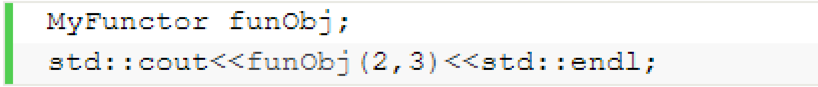
- A Function Object / Functor is a kind of Callback with State.

- Object of a class which has overloaded operator() is called Function Object or Functor i.e. a class with overloaded operator() function is as follows

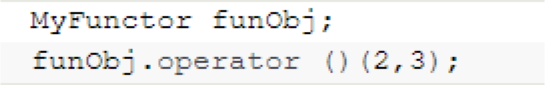




```
#include <iostream>
class MyFunctor
{
public:
    int operator()(int a , int b)
    {
        return a+b;
    }

};
```

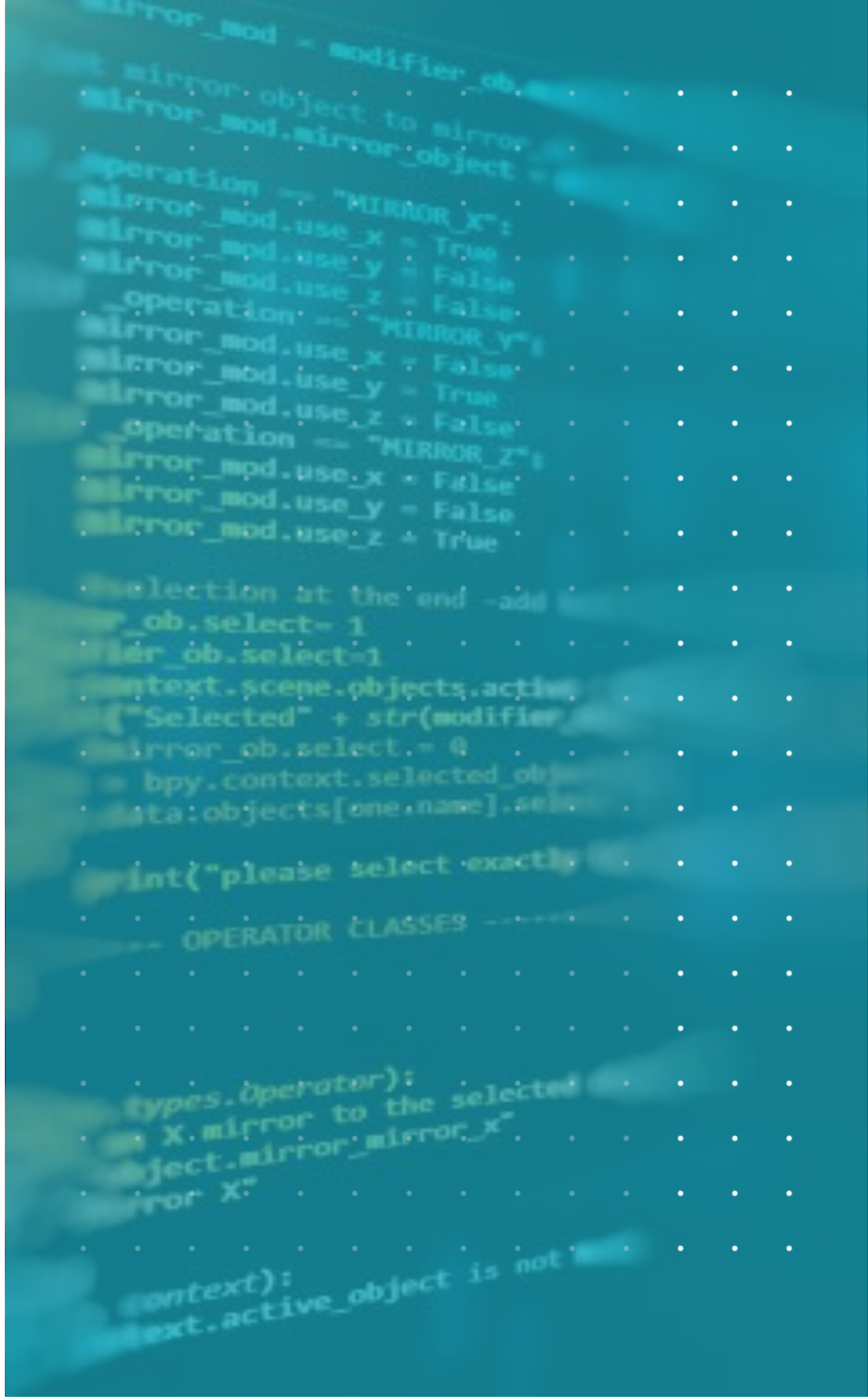


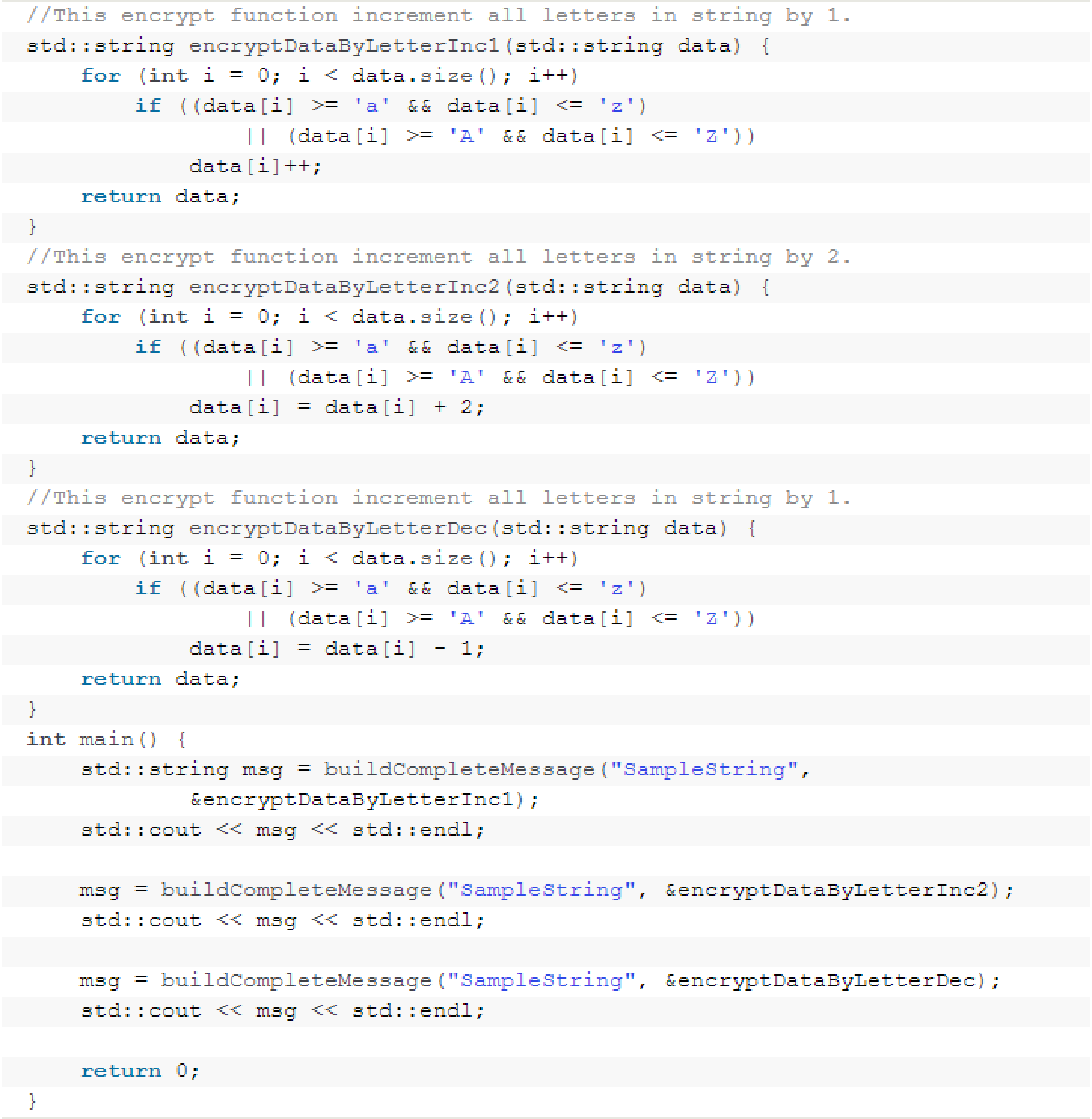
```
MyFunctor funObj;
std::cout<<funObj(2,3)<<std::endl;
```



```
MyFunctor funObj;
funObj.operator ()(2,3);
```

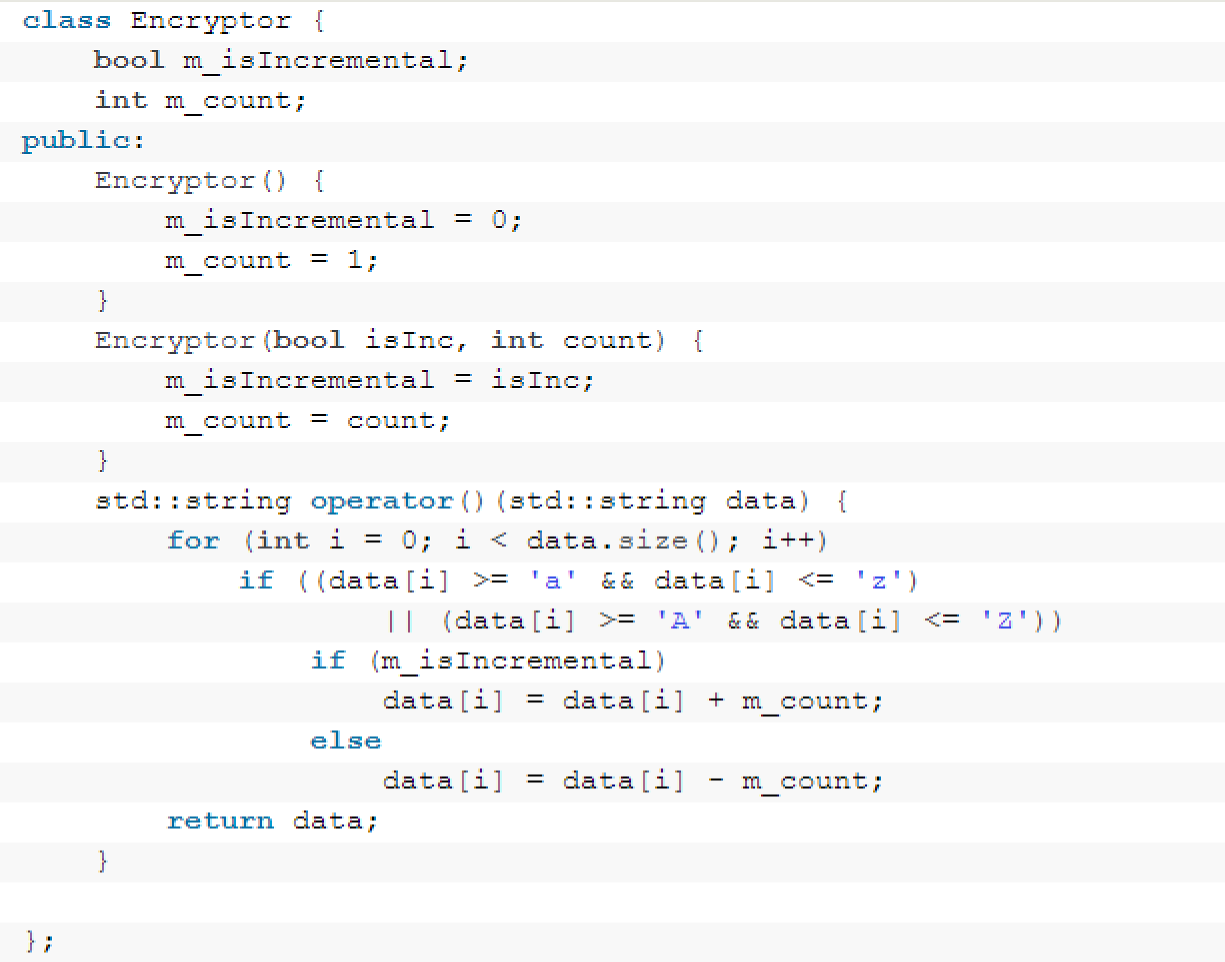
we want to call this framework API three times with three different types of encryption logics



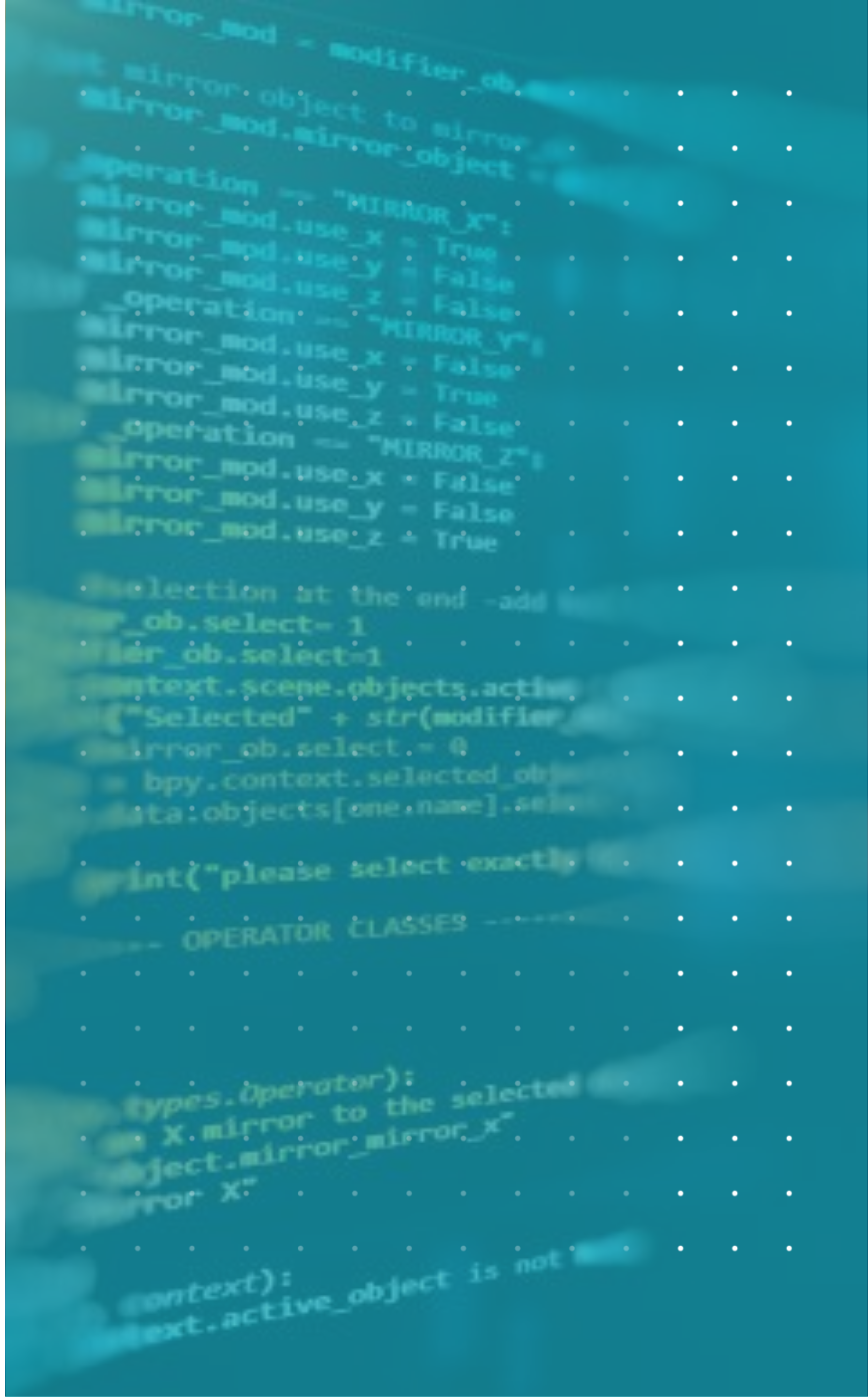


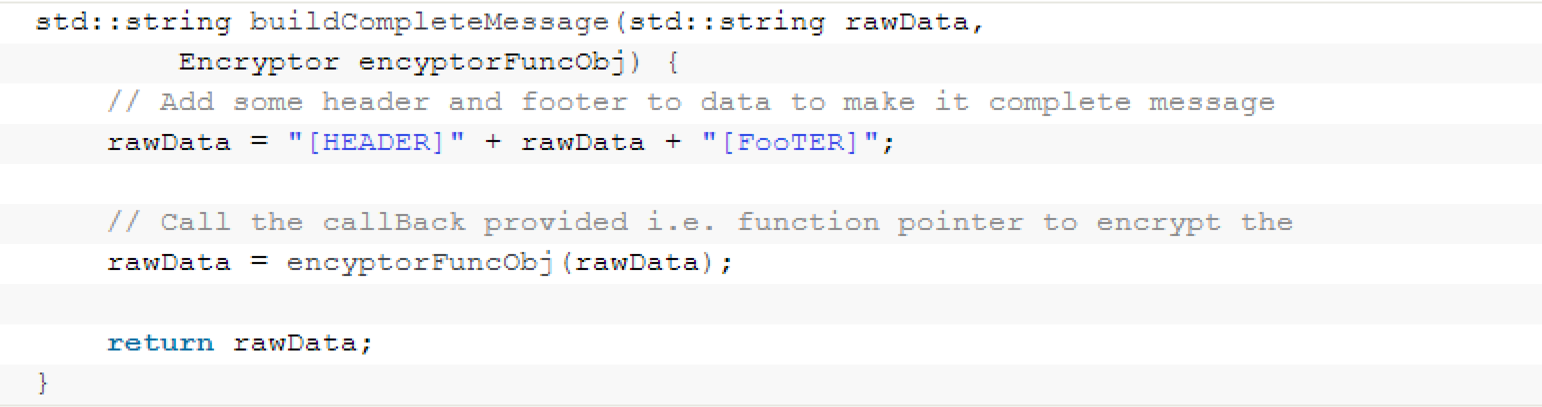
```cpp
//This encrypt function increment all letters in string by 1.
std::string encryptDataByLetterInc1(std::string data) {
    for (int i = 0; i < data.size(); i++)
        if ((data[i] >= 'a' && data[i] <= 'z')
                || (data[i] >= 'A' && data[i] <= 'Z'))
            data[i]++;
    return data;
}
//This encrypt function increment all letters in string by 2.
std::string encryptDataByLetterInc2(std::string data) {
    for (int i = 0; i < data.size(); i++)
        if ((data[i] >= 'a' && data[i] <= 'z')
                || (data[i] >= 'A' && data[i] <= 'Z'))
            data[i] = data[i] + 2;
    return data;
}
//This encrypt function increment all letters in string by 1.
std::string encryptDataByLetterDec(std::string data) {
    for (int i = 0; i < data.size(); i++)
        if ((data[i] >= 'a' && data[i] <= 'z')
                || (data[i] >= 'A' && data[i] <= 'Z'))
            data[i] = data[i] - 1;
    return data;
}
int main() {
    std::string msg = buildCompleteMessage("SampleString",
            &encryptDataByLetterInc1);
    std::cout << msg << std::endl;

    msg = buildCompleteMessage("SampleString", &encryptDataByLetterInc2);
    std::cout << msg << std::endl;

    msg = buildCompleteMessage("SampleString", &encryptDataByLetterDec);
    std::cout << msg << std::endl;

    return 0;
}
```

bind state with function pointers

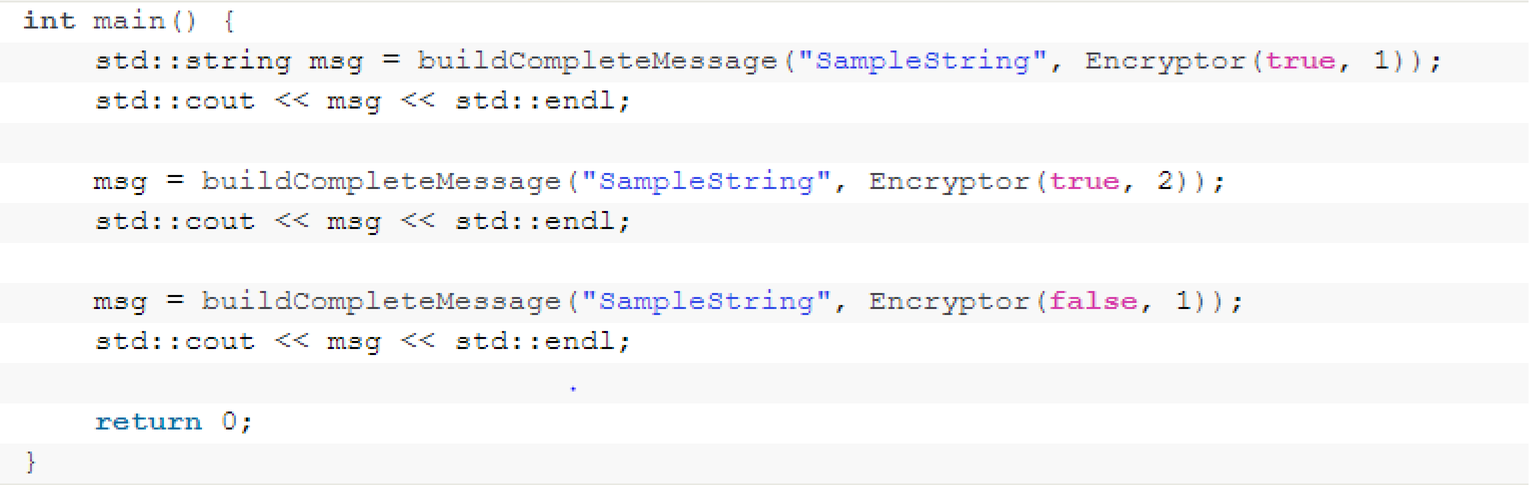


```
class Encryptor {
    bool m_isIncremental;
    int m_count;
public:
    Encryptor() {
        m_isIncremental = 0;
        m_count = 1;
    }
    Encryptor(bool isInc, int count) {
        m_isIncremental = isInc;
        m_count = count;
    }
    std::string operator()(std::string data) {
        for (int i = 0; i < data.size(); i++)
            if ((data[i] >= 'a' && data[i] <= 'z')
                    || (data[i] >= 'A' && data[i] <= 'Z'))
                if (m_isIncremental)
                    data[i] = data[i] + m_count;
                else
                    data[i] = data[i] - m_count;
        return data;
    }

};
```





```
std::string buildCompleteMessage(std::string rawData,
        Encryptor encyptorFuncObj) {
    // Add some header and footer to data to make it complete message
    rawData = "[HEADER]" + rawData + "[FooTER]";

    // Call the callBack provided i.e. function pointer to encrypt the
    rawData = encyptorFuncObj(rawData);

    return rawData;
}
```



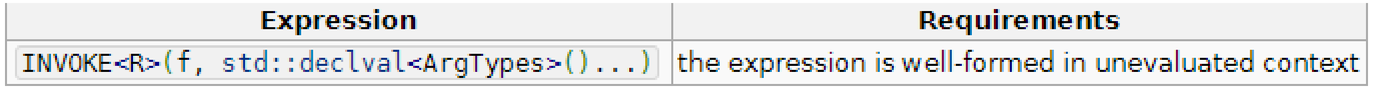
```
int main() {
    std::string msg = buildCompleteMessage("SampleString", Encryptor(true, 1));
    std::cout << msg << std::endl;

    msg = buildCompleteMessage("SampleString", Encryptor(true, 2));
    std::cout << msg << std::endl;

    msg = buildCompleteMessage("SampleString", Encryptor(false, 1));
    std::cout << msg << std::endl;

    return 0;
}
```

std::function

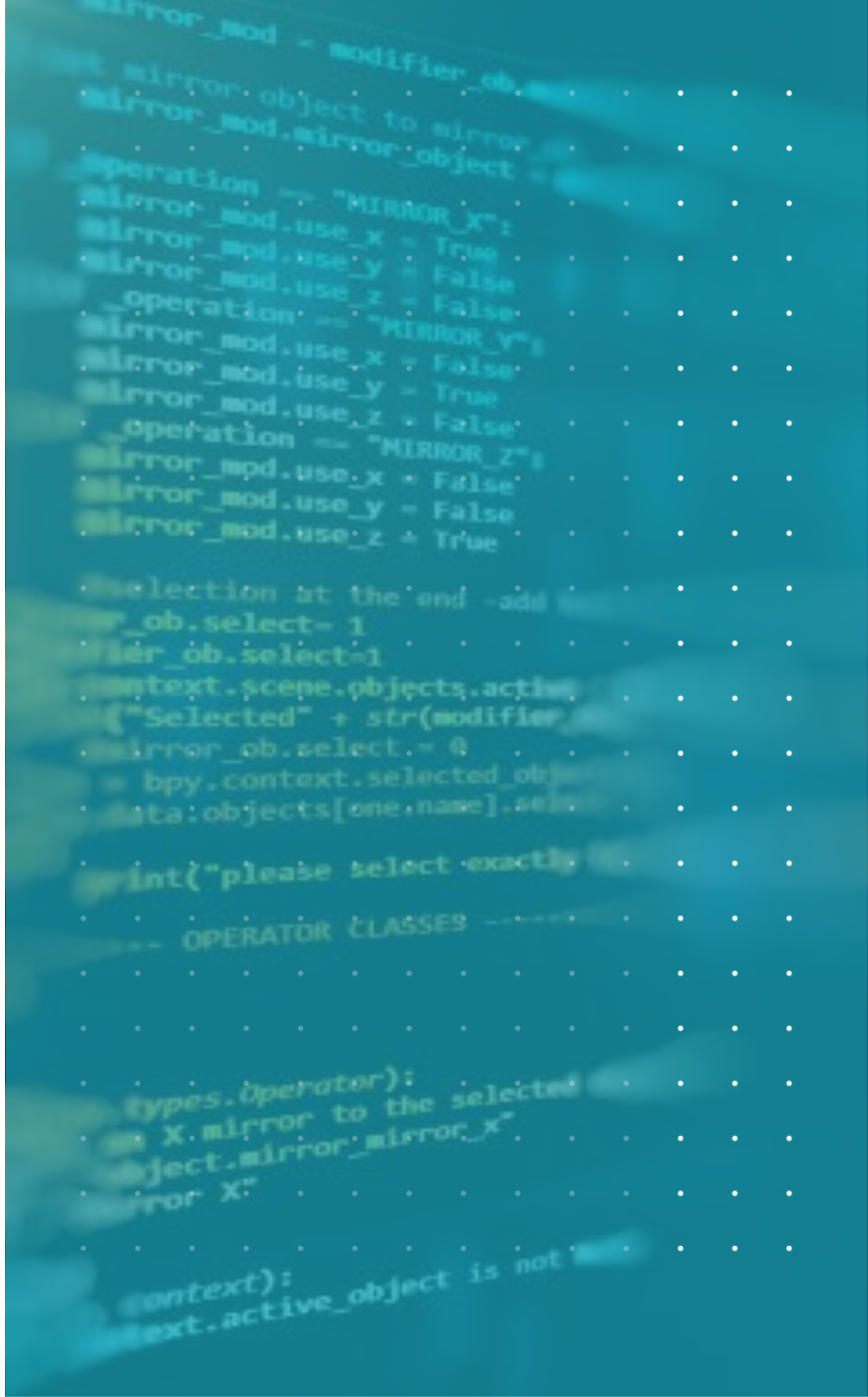
C++۱۱: std::function and std::bind

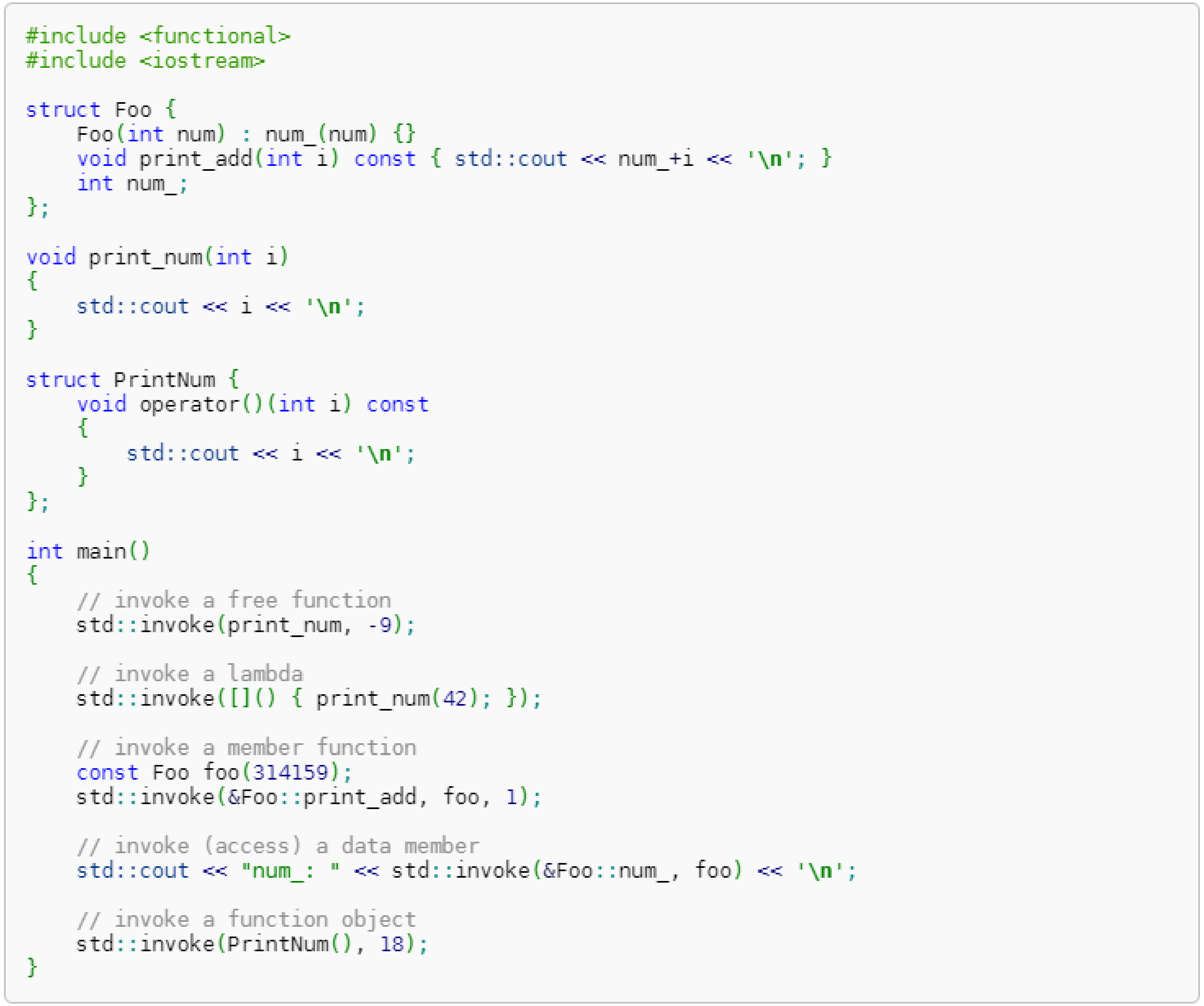
Callable Target

- A Callable type is a type for which the INVOKE operation is applicable. This operation may be performed explicitly using the library function std::invoke. (since C++۱۷)
- The type T satisfies Callable if
- Given
  - f, an object of type T
  - ArgTypes, suitable list of argument types
  - R, suitable return type
- The following expressions must be valid:



Expression	Requirements
`INVOKE<R>(f, std::declval<ArgTypes>()...)`	the expression is well-formed in unevaluated context

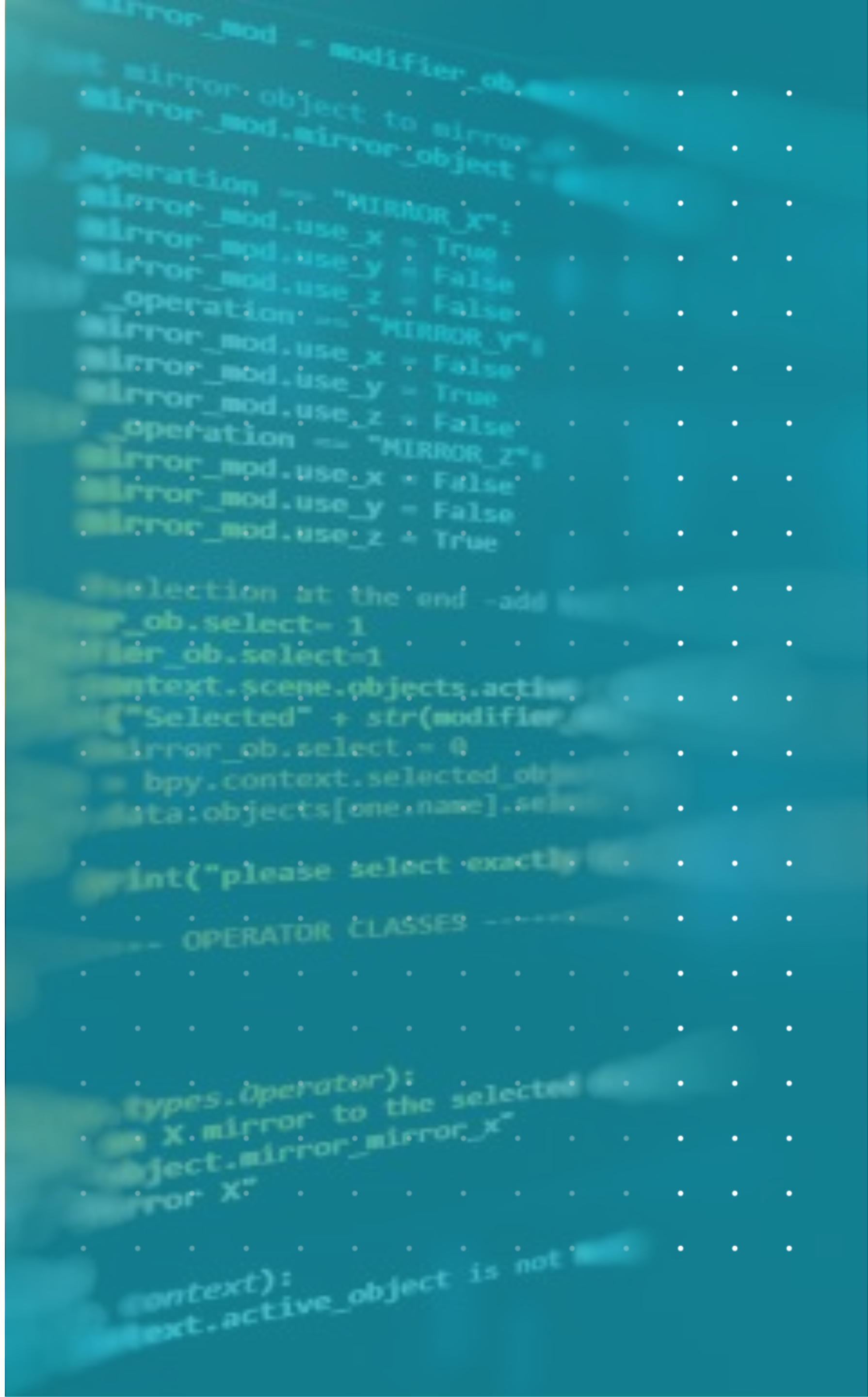




```cpp
#include <functional>
#include <iostream>

struct Foo {
    Foo(int num) : num_(num) {}
    void print_add(int i) const { std::cout << num_+i << '\n'; }
    int num_;
};

void print_num(int i)
{
    std::cout << i << '\n';
}

struct PrintNum {
    void operator()(int i) const
    {
        std::cout << i << '\n';
    }
};

int main()
{
    // invoke a free function
    std::invoke(print_num, -9);

    // invoke a lambda
    std::invoke([]() { print_num(42); });

    // invoke a member function
    const Foo foo(314159);
    std::invoke(&Foo::print_add, foo, 1);

    // invoke (access) a data member
    std::cout << "num_: " << std::invoke(&Foo::num_, foo) << '\n';

    // invoke a function object
    std::invoke(PrintNum(), 18);
}
```

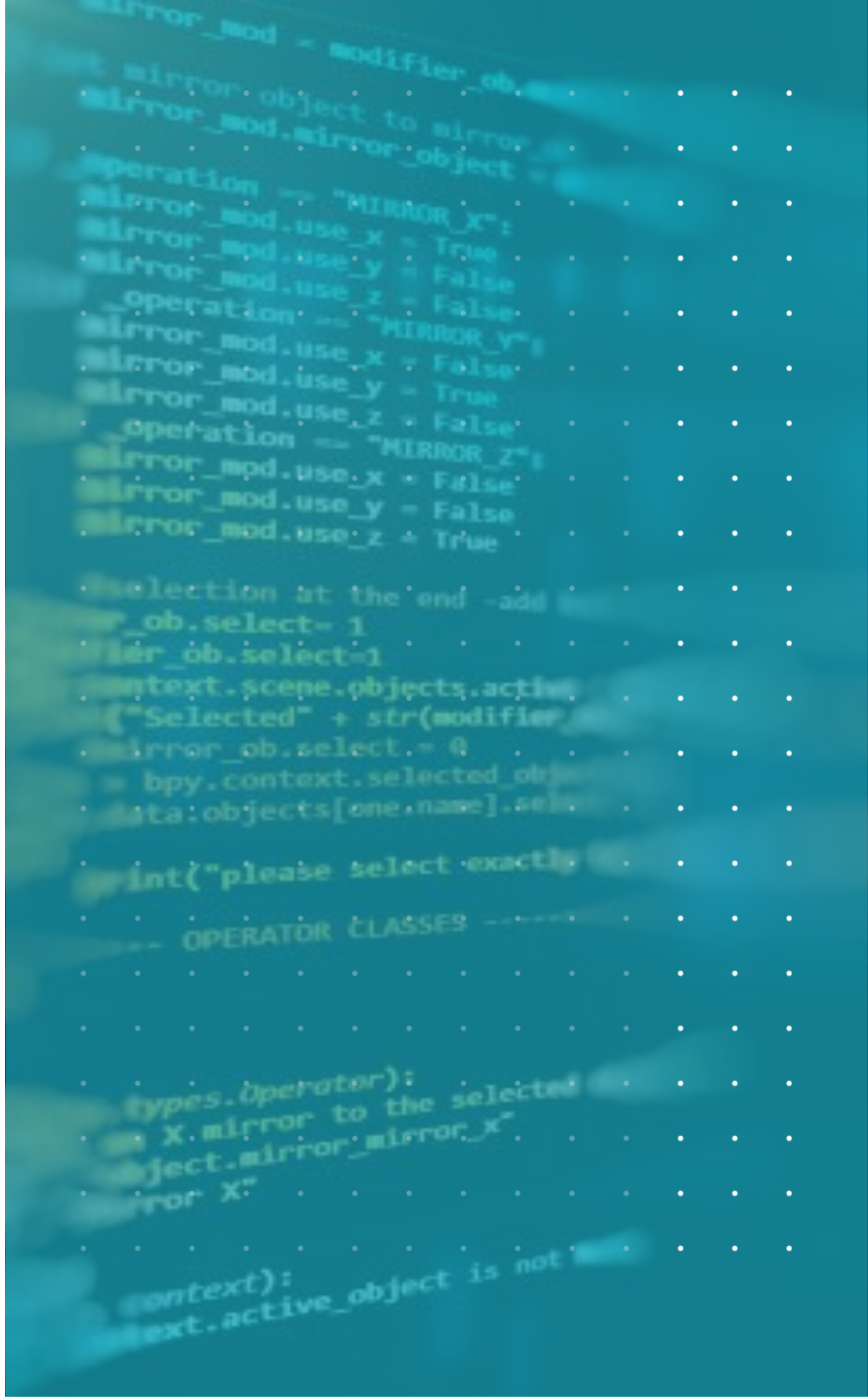
- std::function and std::bind were born inside the Boost C++ Library, but they were incorporated into the new C++۱۱ standard.

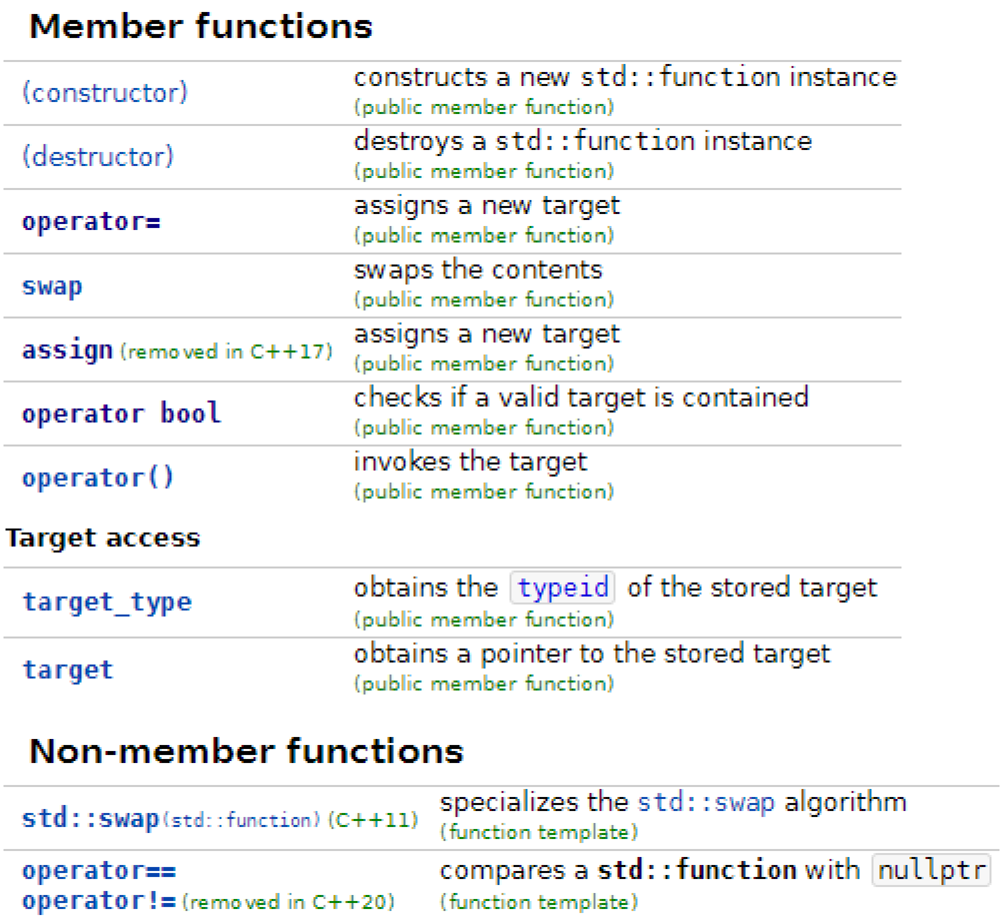
- std::function is a STL template class that provides a very convenient wrapper to a simple function, to a functor or to a lambda expression.



Continue

- Class template std::function is a general-purpose polymorphic function wrapper. Instances of std::function can store, copy, and invoke any Callable target
- The stored callable object is called the target of std::function.
- If a std::function contains no target, it is called empty. Invoking the target of an empty std::function results in std::bad_function_call exception being thrown.





Member functions

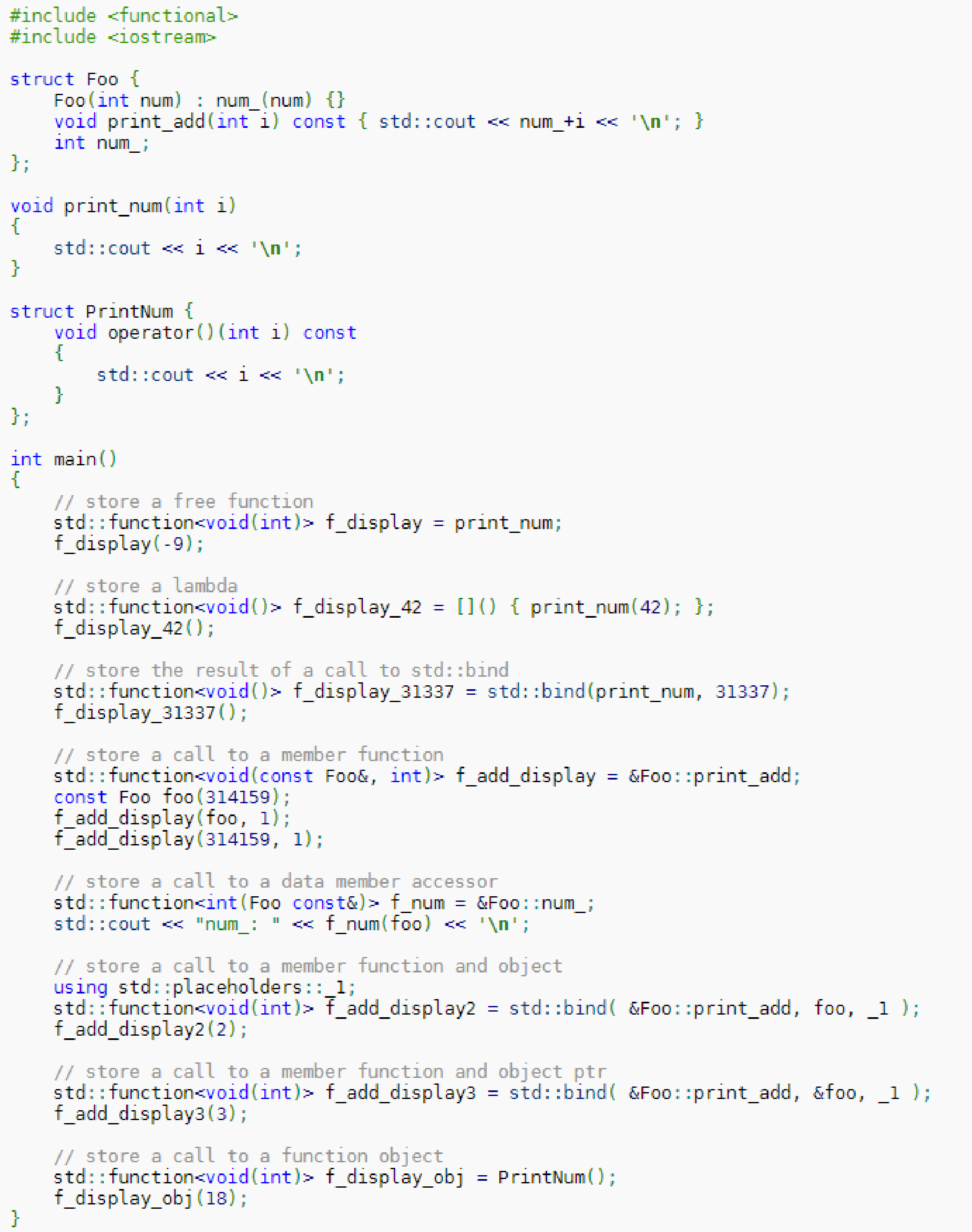
(constructor)	constructs a new std::function instance (public member function)
(destructor)	destroys a std::function instance (public member function)
operator=	assigns a new target (public member function)
swap	swaps the contents (public member function)
assign (removed in C++17)	assigns a new target (public member function)
operator bool	checks if a valid target is contained (public member function)
operator()	invokes the target (public member function)

Target access

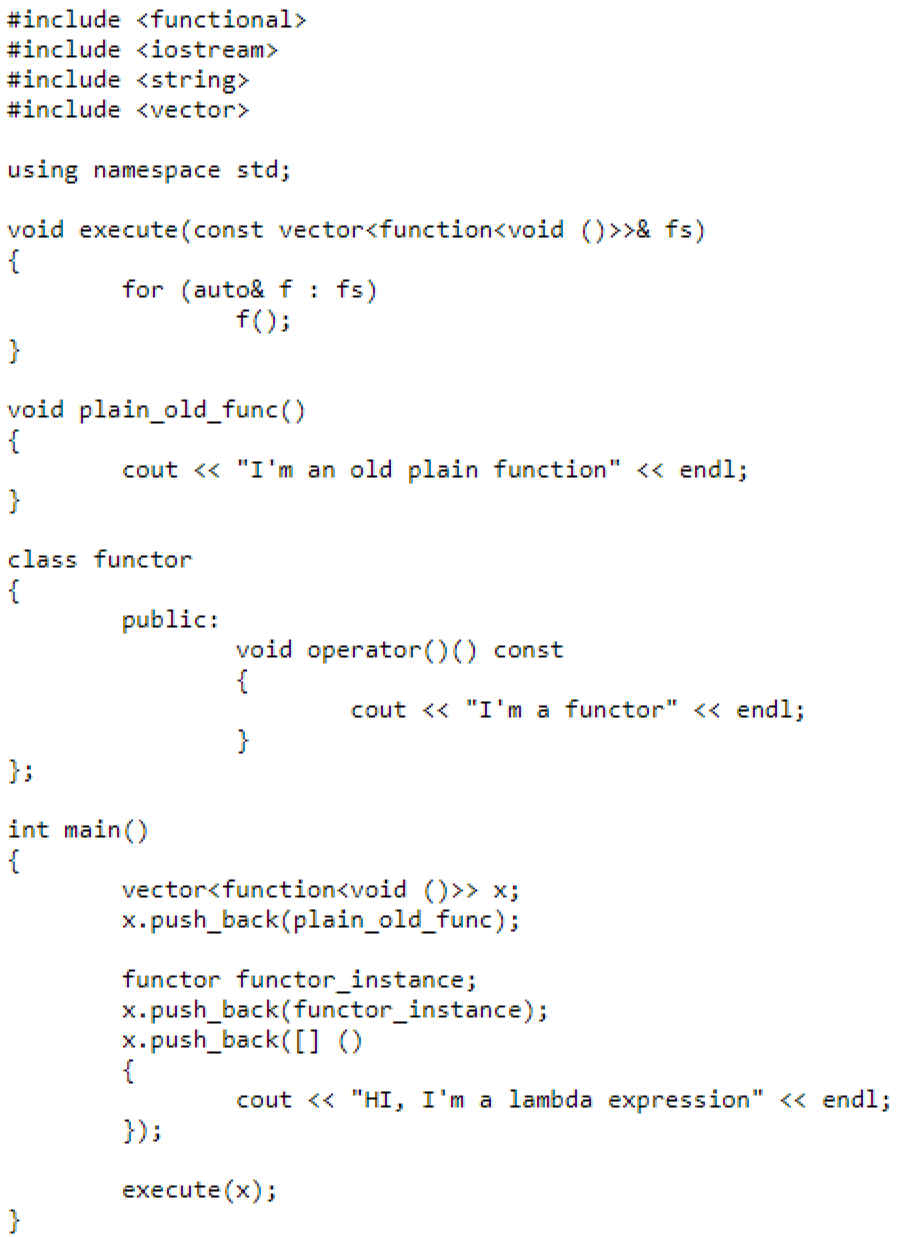
target_type	obtains the typeid of the stored target (public member function)
target	obtains a pointer to the stored target (public member function)

Non-member functions

std::swap(std::function) (C++11)	specializes the std::swap algorithm (function template)
operator== **operator!=** (removed in C++20)	compares a **std::function** with nullptr (function template)

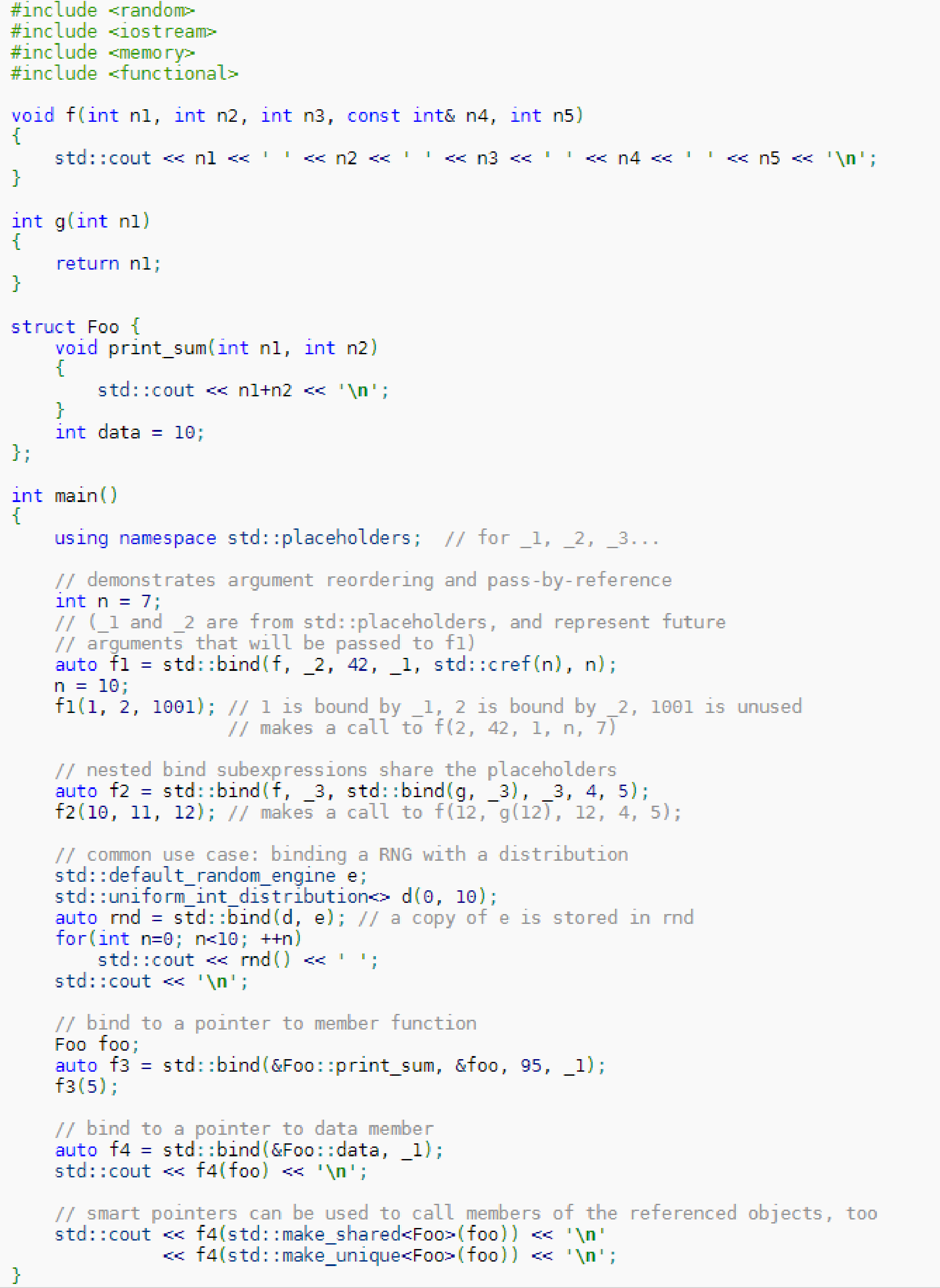


```cpp
#include <functional>
#include <iostream>

struct Foo {
    Foo(int num) : num_(num) {}
    void print_add(int i) const { std::cout << num_+i << '\n'; }
    int num_;
};

void print_num(int i)
{
    std::cout << i << '\n';
}

struct PrintNum {
    void operator()(int i) const
    {
        std::cout << i << '\n';
    }
};

int main()
{
    // store a free function
    std::function<void(int)> f_display = print_num;
    f_display(-9);

    // store a lambda
    std::function<void()> f_display_42 = []() { print_num(42); };
    f_display_42();

    // store the result of a call to std::bind
    std::function<void()> f_display_31337 = std::bind(print_num, 31337);
    f_display_31337();

    // store a call to a member function
    std::function<void(const Foo&, int)> f_add_display = &Foo::print_add;
    const Foo foo(314159);
    f_add_display(foo, 1);
    f_add_display(314159, 1);

    // store a call to a data member accessor
    std::function<int(Foo const&)> f_num = &Foo::num_;
    std::cout << "num_: " << f_num(foo) << '\n';

    // store a call to a member function and object
    using std::placeholders::_1;
    std::function<void(int)> f_add_display2 = std::bind( &Foo::print_add, foo, _1 );
    f_add_display2(2);

    // store a call to a member function and object ptr
    std::function<void(int)> f_add_display3 = std::bind( &Foo::print_add, &foo, _1 );
    f_add_display3(3);

    // store a call to a function object
    std::function<void(int)> f_display_obj = PrintNum();
    f_display_obj(18);
}
```



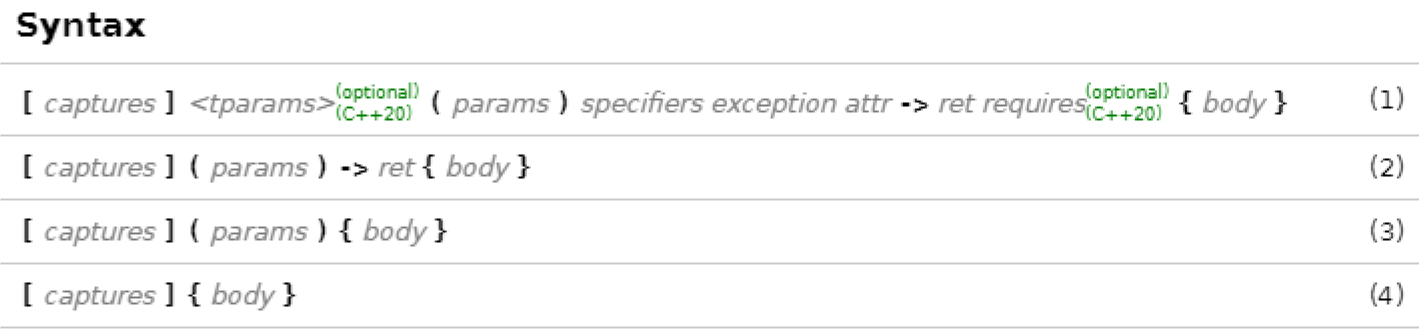
```cpp
#include <functional>
#include <iostream>
#include <string>
#include <vector>

using namespace std;

void execute(const vector<function<void ()>>& fs)
{
        for (auto& f : fs)
                f();
}

void plain_old_func()
{
        cout << "I'm an old plain function" << endl;
}

class functor
{
        public:
                void operator()() const
                {
                        cout << "I'm a functor" << endl;
                }
};

int main()
{
        vector<function<void ()>> x;
        x.push_back(plain_old_func);

        functor functor_instance;
        x.push_back(functor_instance);
        x.push_back([] ()
        {
                cout << "HI, I'm a lambda expression" << endl;
        });

        execute(x);
}
```

std::bind



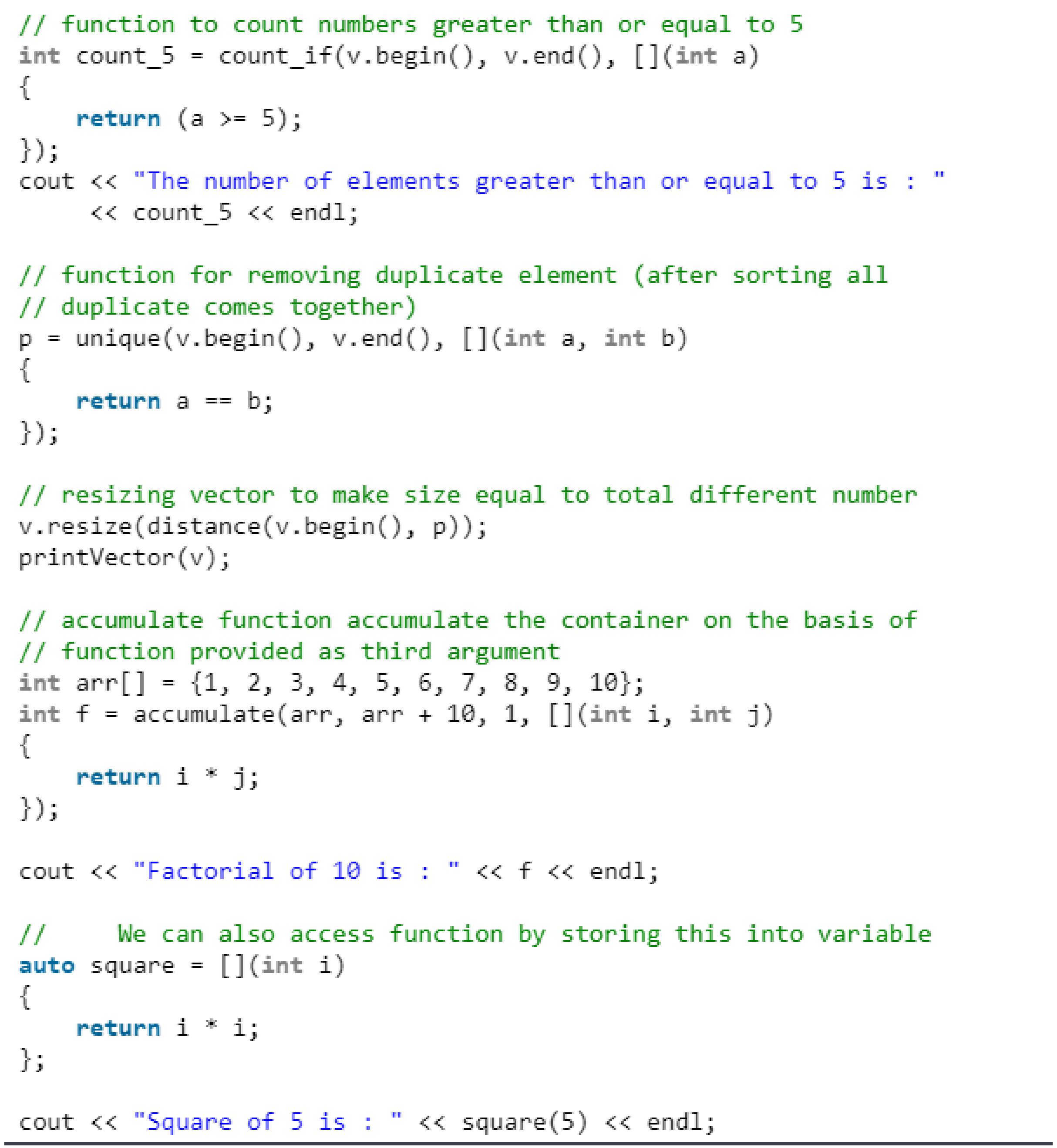
```
#include <random>
#include <iostream>
#include <memory>
#include <functional>

void f(int n1, int n2, int n3, const int& n4, int n5)
{
    std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4 << ' ' << n5 << '\n';
}

int g(int n1)
{
    return n1;
}

struct Foo {
    void print_sum(int n1, int n2)
    {
        std::cout << n1+n2 << '\n';
    }
    int data = 10;
};

int main()
{
    using namespace std::placeholders;  // for _1, _2, _3...

    // demonstrates argument reordering and pass-by-reference
    int n = 7;
    // (_1 and _2 are from std::placeholders, and represent future
    // arguments that will be passed to f1)
    auto f1 = std::bind(f, _2, 42, _1, std::cref(n), n);
    n = 10;
    f1(1, 2, 1001); // 1 is bound by _1, 2 is bound by _2, 1001 is unused
                    // makes a call to f(2, 42, 1, n, 7)

    // nested bind subexpressions share the placeholders
    auto f2 = std::bind(f, _3, std::bind(g, _3), _3, 4, 5);
    f2(10, 11, 12); // makes a call to f(12, g(12), 12, 4, 5);

    // common use case: binding a RNG with a distribution
    std::default_random_engine e;
    std::uniform_int_distribution<> d(0, 10);
    auto rnd = std::bind(d, e); // a copy of e is stored in rnd
    for(int n=0; n<10; ++n)
        std::cout << rnd() << ' ';
    std::cout << '\n';

    // bind to a pointer to member function
    Foo foo;
    auto f3 = std::bind(&Foo::print_sum, &foo, 95, _1);
    f3(5);

    // bind to a pointer to data member
    auto f4 = std::bind(&Foo::data, _1);
    std::cout << f4(foo) << '\n';

    // smart pointers can be used to call members of the referenced objects, too
    std::cout << f4(std::make_shared<Foo>(foo)) << '\n'
              << f4(std::make_unique<Foo>(foo)) << '\n';
}
```

Lambda Expression



Syntax

[*captures*] *<tparams>*(optional)(C++20) (*params*) *specifiers exception attr* -> *ret requires*(optional)(C++20) { *body* }	(1)
[*captures*] (*params*) -> *ret* { *body* }	(2)
[*captures*] (*params*) { *body* }	(3)
[*captures*] { *body* }	(4)

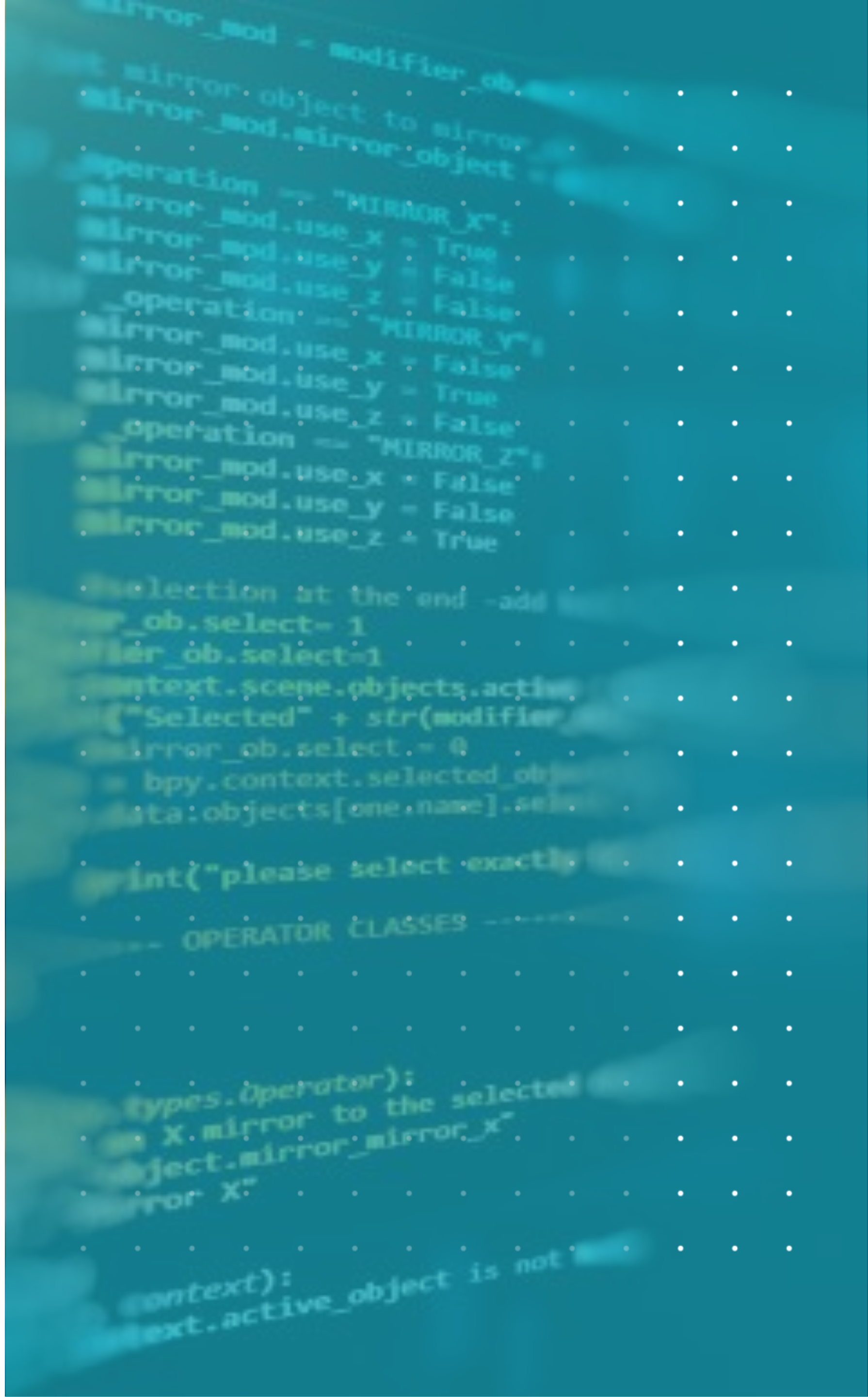


```cpp
// function to count numbers greater than or equal to 5
int count_5 = count_if(v.begin(), v.end(), [](int a)
{
    return (a >= 5);
});
cout << "The number of elements greater than or equal to 5 is : "
     << count_5 << endl;

// function for removing duplicate element (after sorting all
// duplicate comes together)
p = unique(v.begin(), v.end(), [](int a, int b)
{
    return a == b;
});

// resizing vector to make size equal to total different number
v.resize(distance(v.begin(), p));
printVector(v);

// accumulate function accumulate the container on the basis of
// function provided as third argument
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int f = accumulate(arr, arr + 10, 1, [](int i, int j)
{
    return i * j;
});

cout << "Factorial of 10 is : " << f << endl;

//     We can also access function by storing this into variable
auto square = [](int i)
{
    return i * i;
};

cout << "Square of 5 is : " << square(5) << endl;
```

Capture

- can capture external variables from enclosing scope by three ways :
  - Capture by reference
  - Capture by value
  - Capture by both (mixed capture)

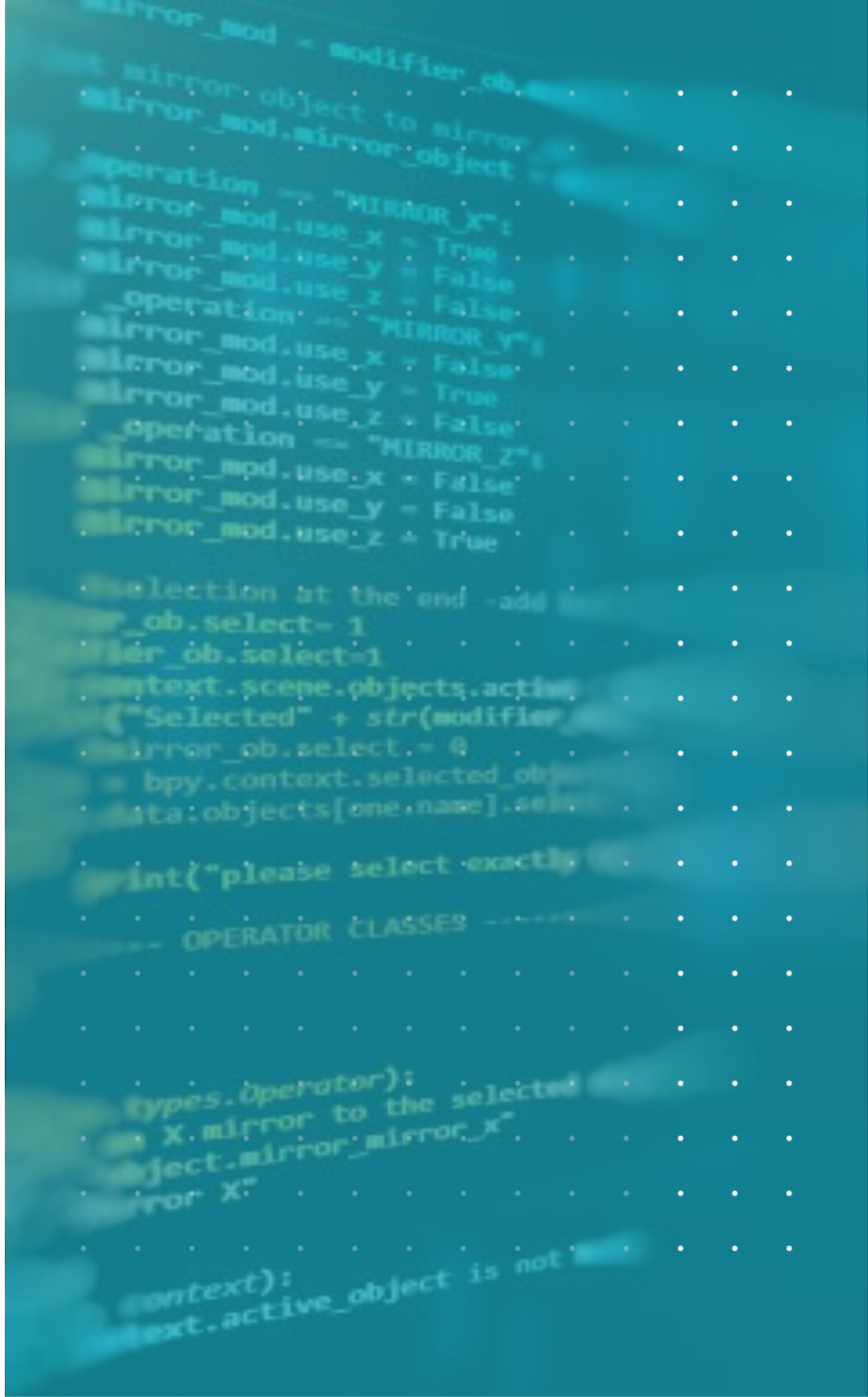
- Syntax used for capturing variables :
  - [&] : capture all external variable by reference
  - [=] : capture all external variable by value
  - [a, &b] : capture a by value and b by reference

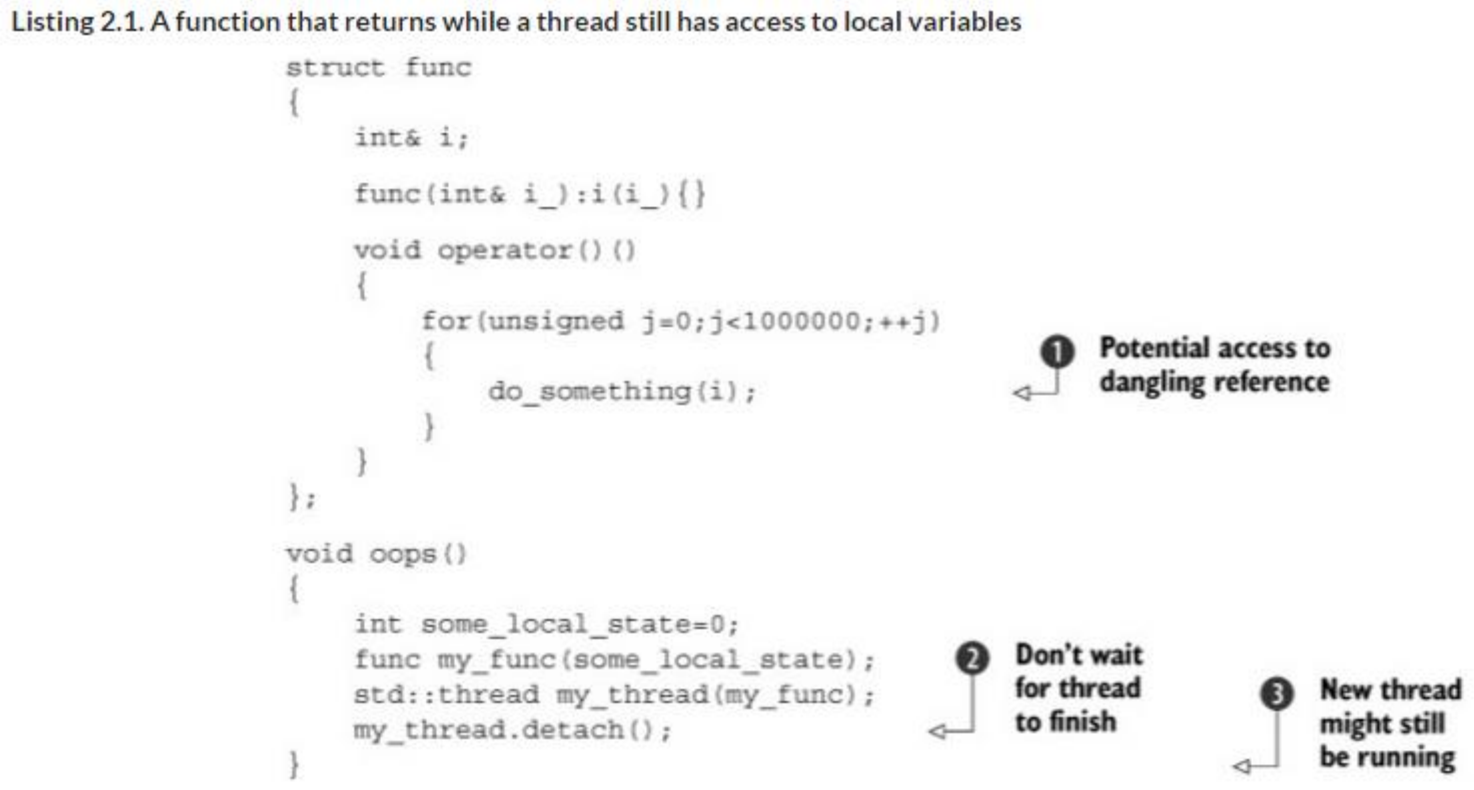


std::thread

- Each of the std::thread object has an associated ID and we can fetch using,

- std::this_thread::get_id()





Listing 2.1. A function that returns while a thread still has access to local variables

```
struct func
{
    int& i;

    func(int& i_):i(i_){}

    void operator()()
    {
        for(unsigned j=0;j<1000000;++j)
        {
            do_something(i);
        }
    }
};

void oops()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
}
```

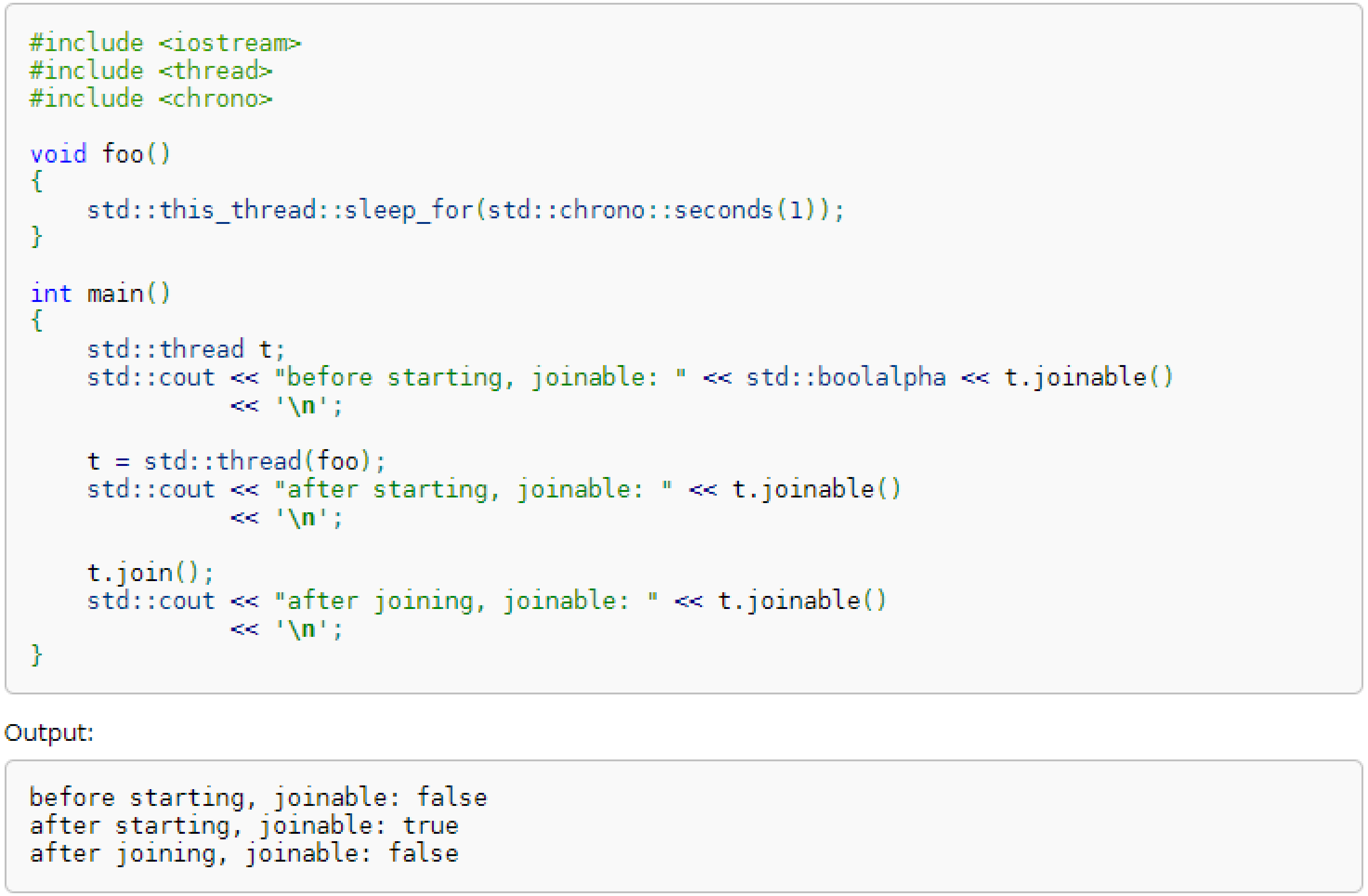
1 Potential access to dangling reference

2 Don't wait for thread to finish

3 New thread might still be running

joinable

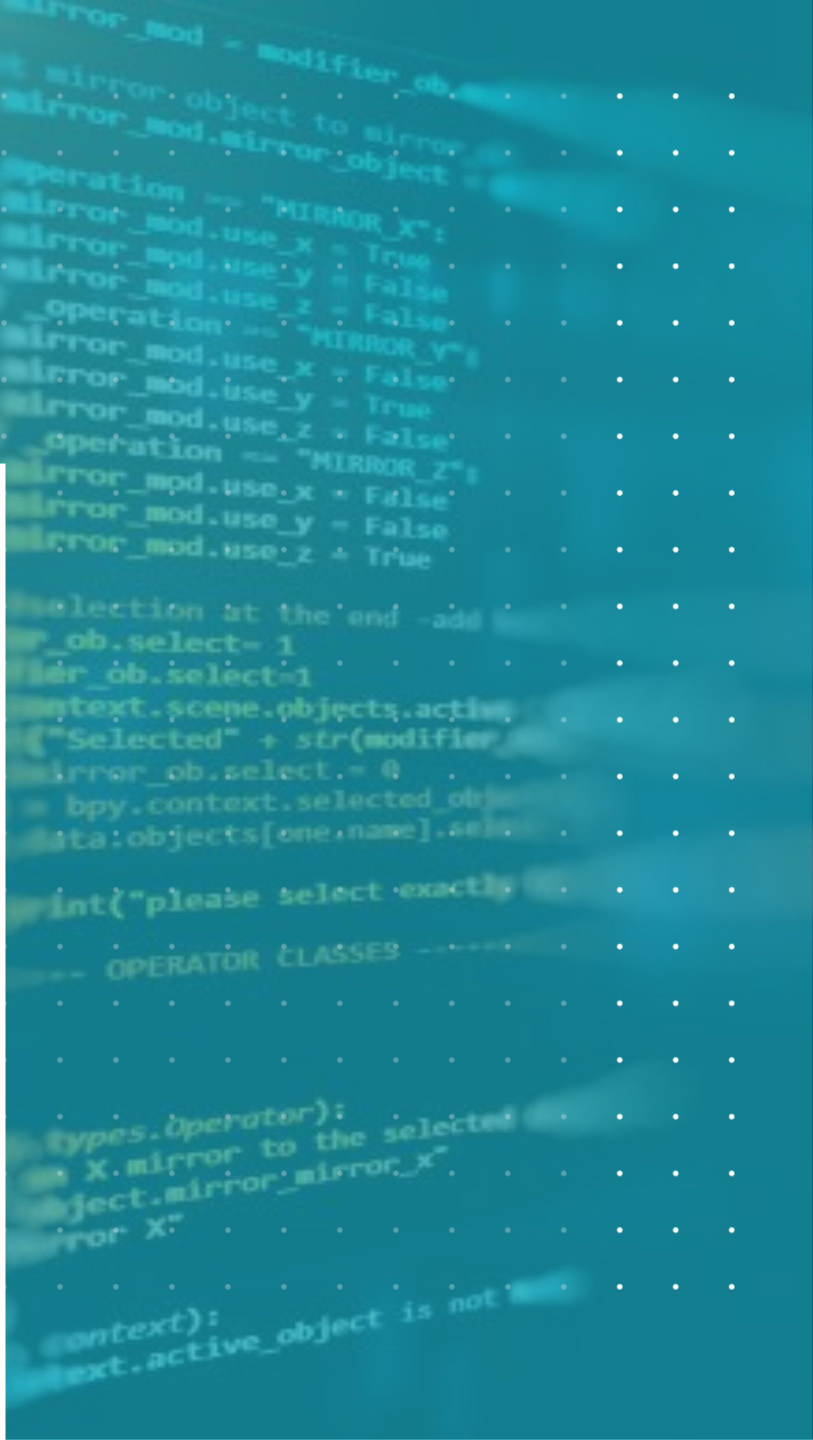
- Checks if the std::thread object identifies an active thread of execution. Specifically, returns true if get_id() != std::thread::id()



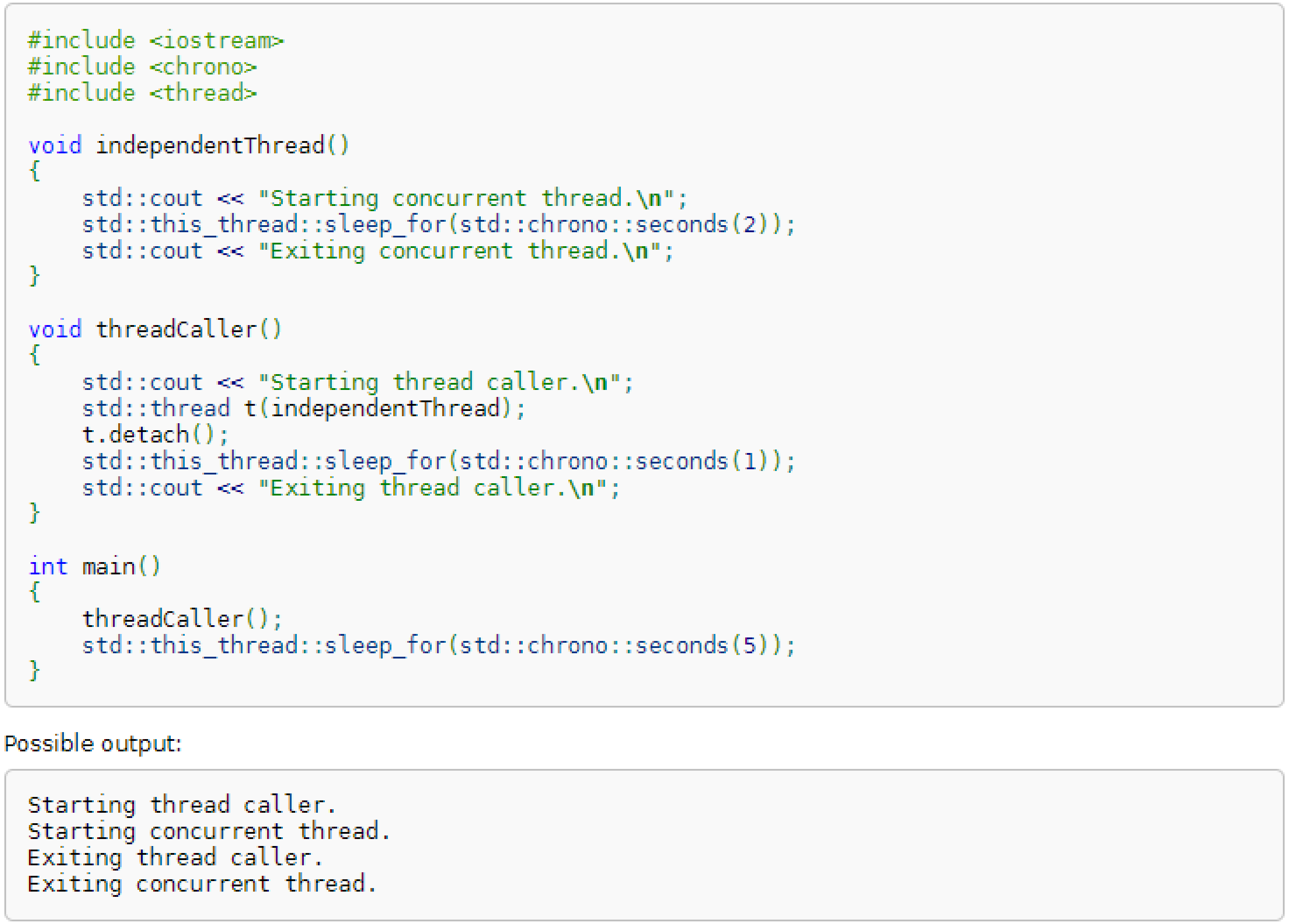
```
#include <iostream>
#include <thread>
#include <chrono>

void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
    std::thread t;
    std::cout << "before starting, joinable: " << std::boolalpha << t.joinable()
              << '\n';

    t = std::thread(foo);
    std::cout << "after starting, joinable: " << t.joinable()
              << '\n';

    t.join();
    std::cout << "after joining, joinable: " << t.joinable()
              << '\n';
}
```

Output:

```
before starting, joinable: false
after starting, joinable: true
after joining, joinable: false
```



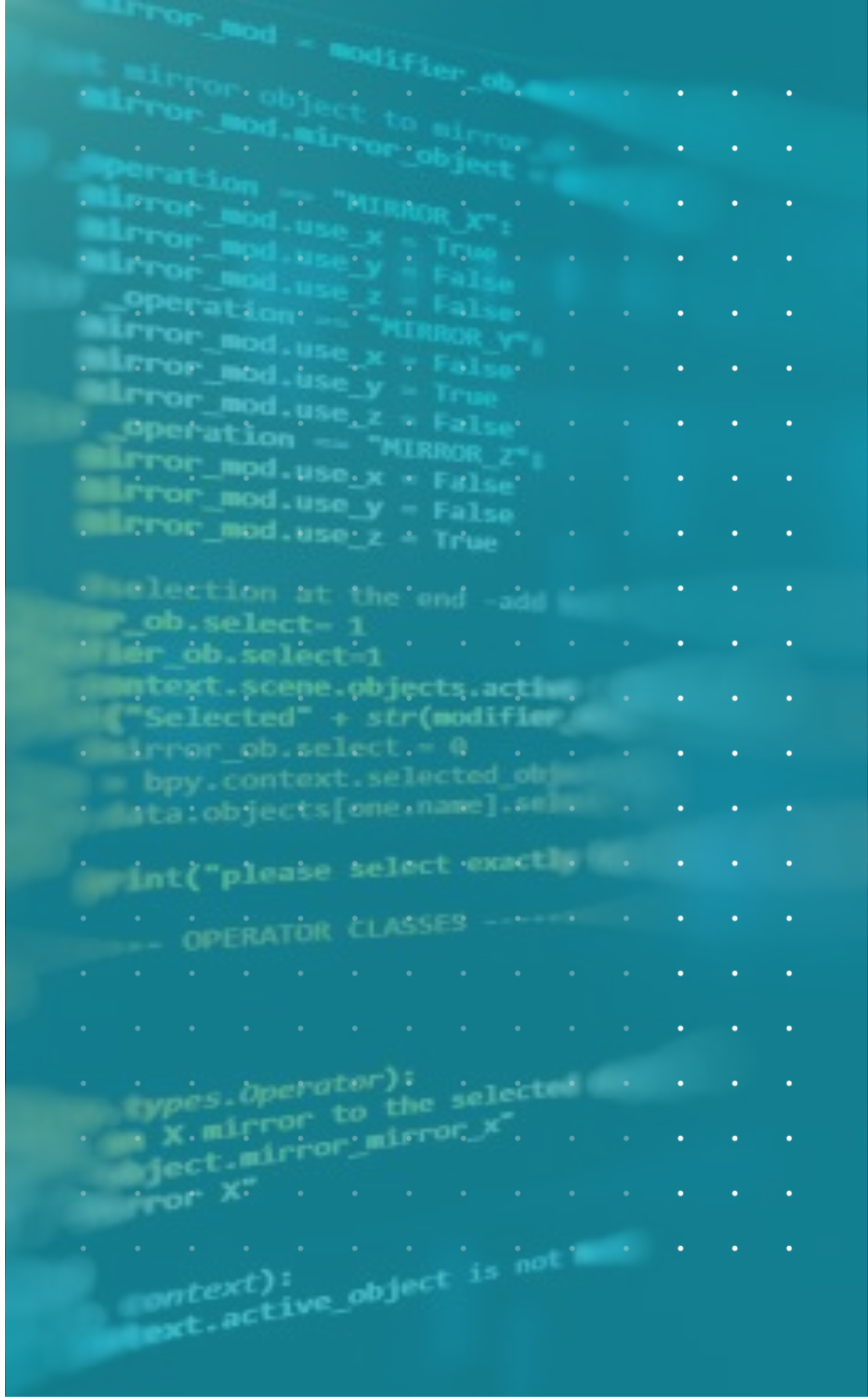
Detach



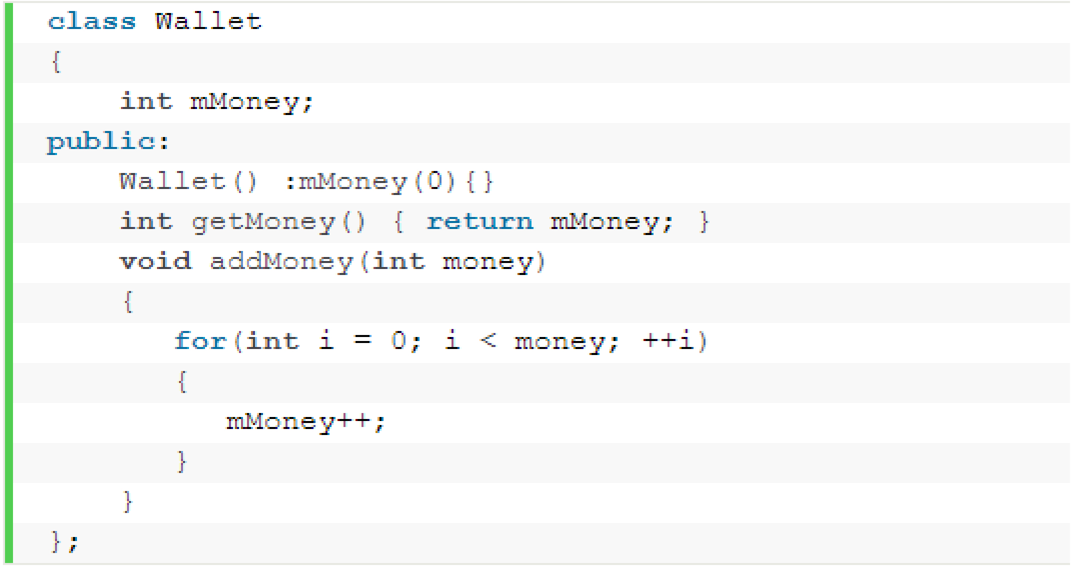
```cpp
#include <iostream>
#include <chrono>
#include <thread>

void independentThread()
{
    std::cout << "Starting concurrent thread.\n";
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "Exiting concurrent thread.\n";
}

void threadCaller()
{
    std::cout << "Starting thread caller.\n";
    std::thread t(independentThread);
    t.detach();
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Exiting thread caller.\n";
}

int main()
{
    threadCaller();
    std::this_thread::sleep_for(std::chrono::seconds(5));
}
```

Possible output:

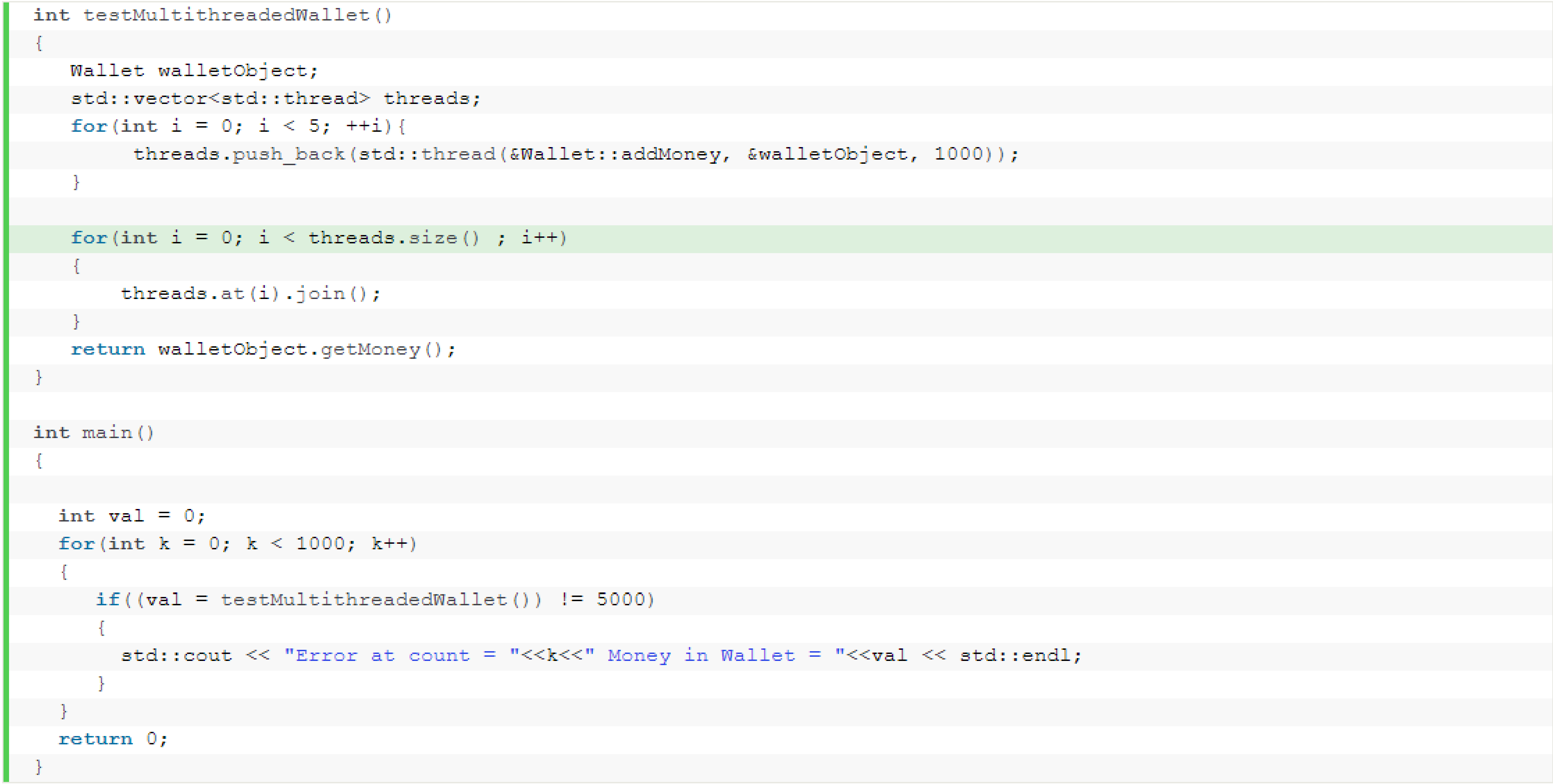
```
Starting thread caller.
Starting concurrent thread.
Exiting thread caller.
Exiting concurrent thread.
```



Race Condition



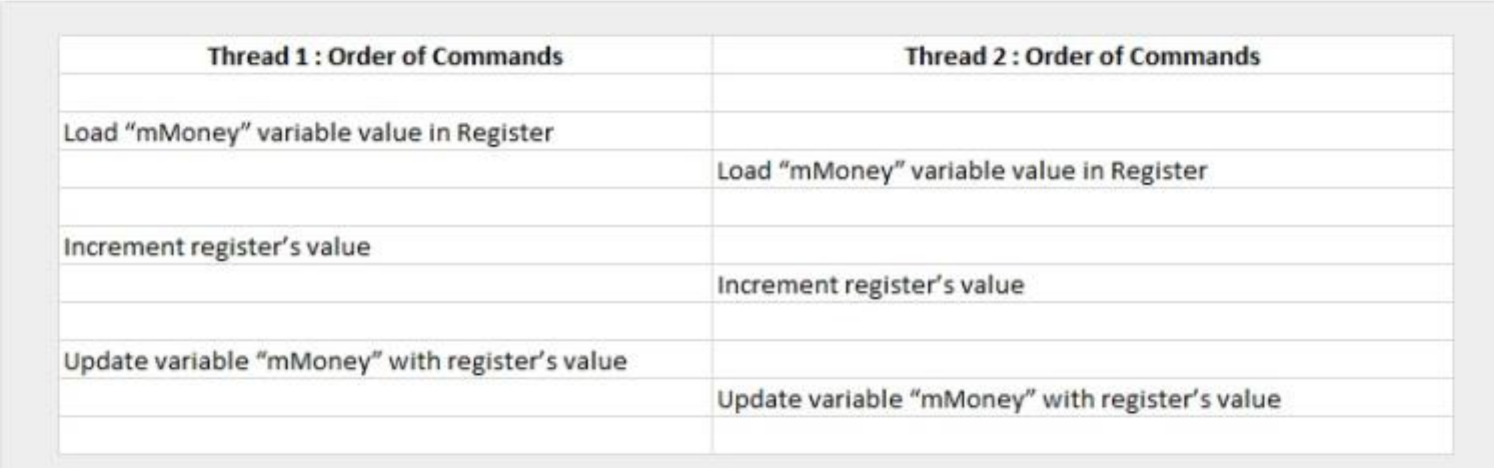
```
class Wallet
{
    int mMoney;
public:
    Wallet() :mMoney(0){}
    int getMoney() { return mMoney; }
    void addMoney(int money)
    {
       for(int i = 0; i < money; ++i)
       {
          mMoney++;
       }
    }
};
```



```
int testMultithreadedWallet()
{
   Wallet walletObject;
   std::vector<std::thread> threads;
   for(int i = 0; i < 5; ++i){
        threads.push_back(std::thread(&Wallet::addMoney, &walletObject, 1000));
   }

   for(int i = 0; i < threads.size() ; i++)
   {
       threads.at(i).join();
   }
   return walletObject.getMoney();
}

int main()
{

  int val = 0;
  for(int k = 0; k < 1000; k++)
  {
     if((val = testMultithreadedWallet()) != 5000)
     {
       std::cout << "Error at count = "<<k<<" Money in Wallet = "<<val << std::endl;
     }
  }
  return 0;
}
```

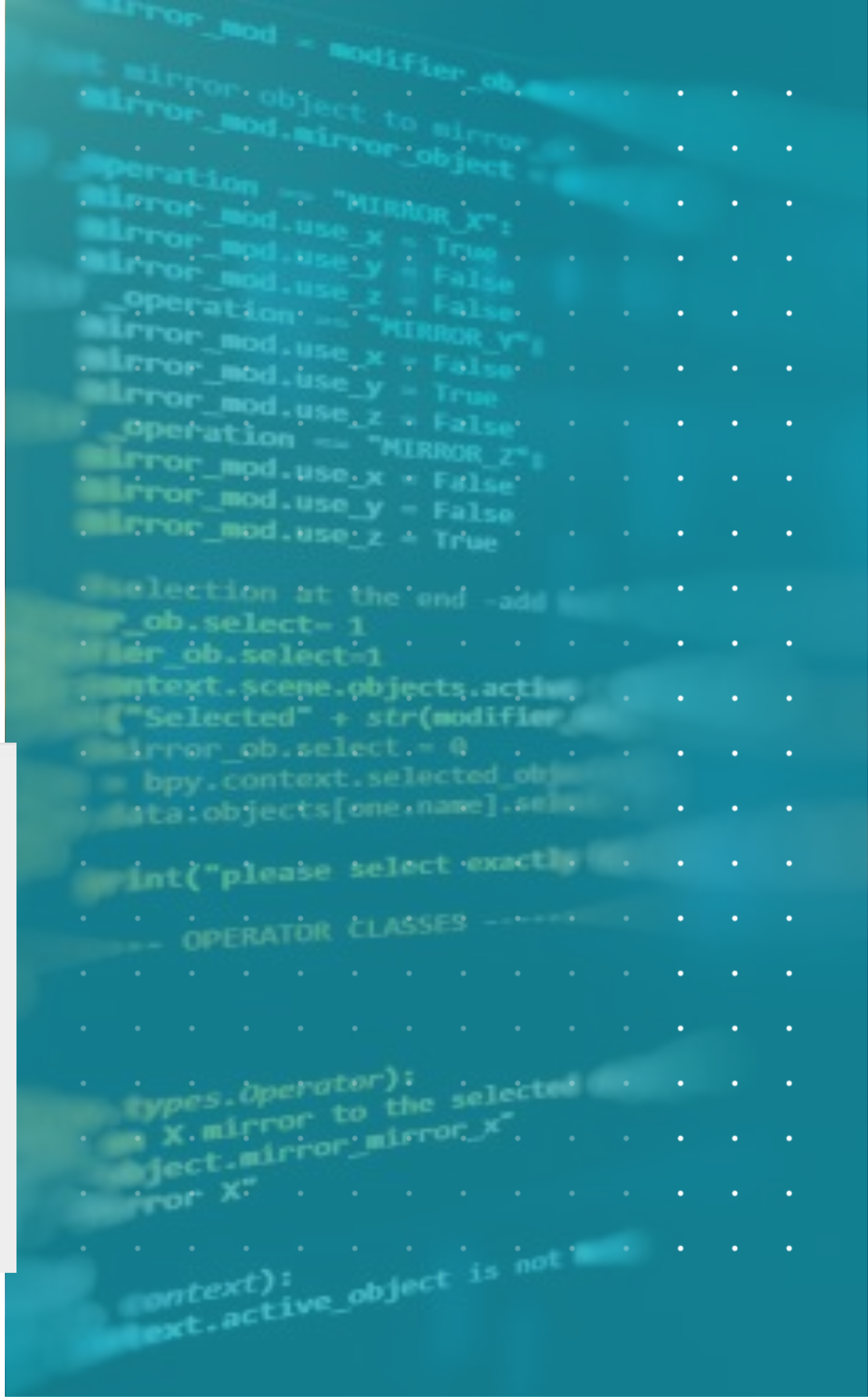
Why this happened

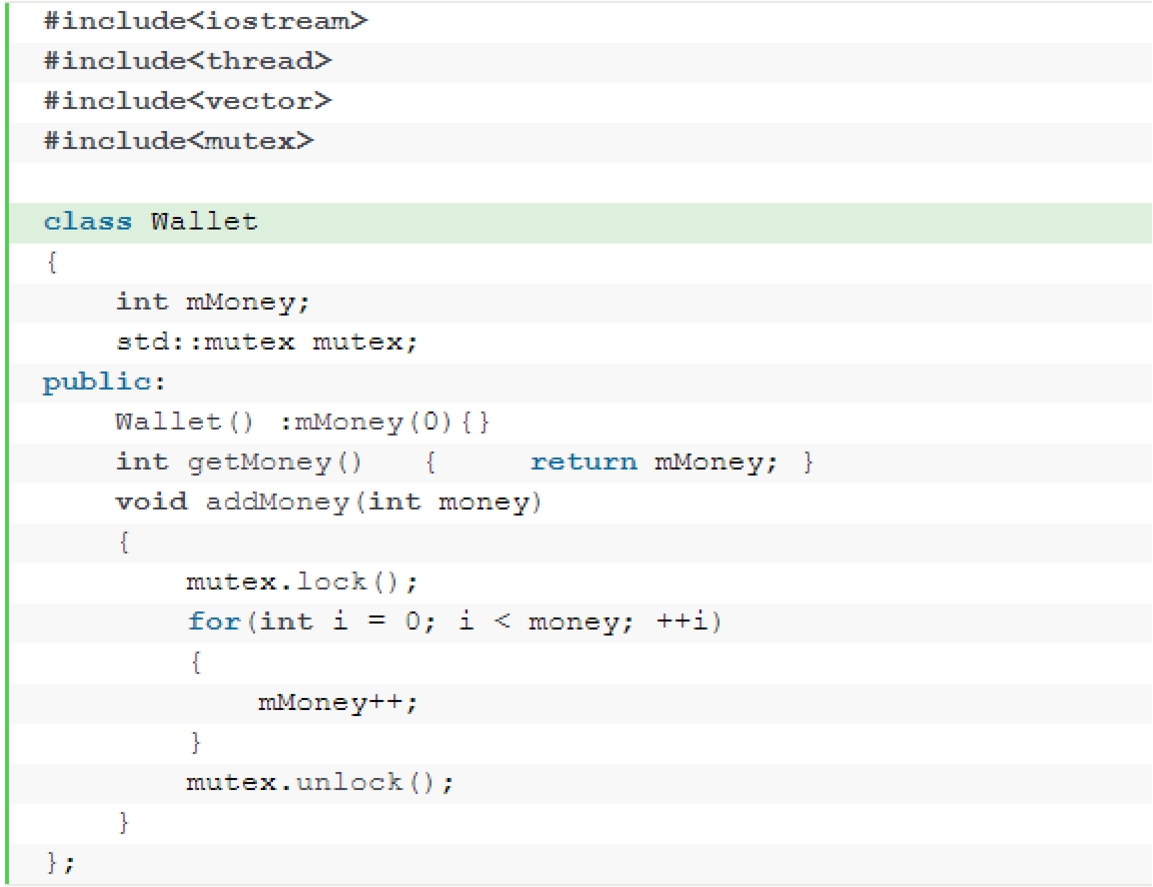
- Load “mMoney” variable value in Register
- Increment register’s value
- Update variable “mMoney” with register’s value



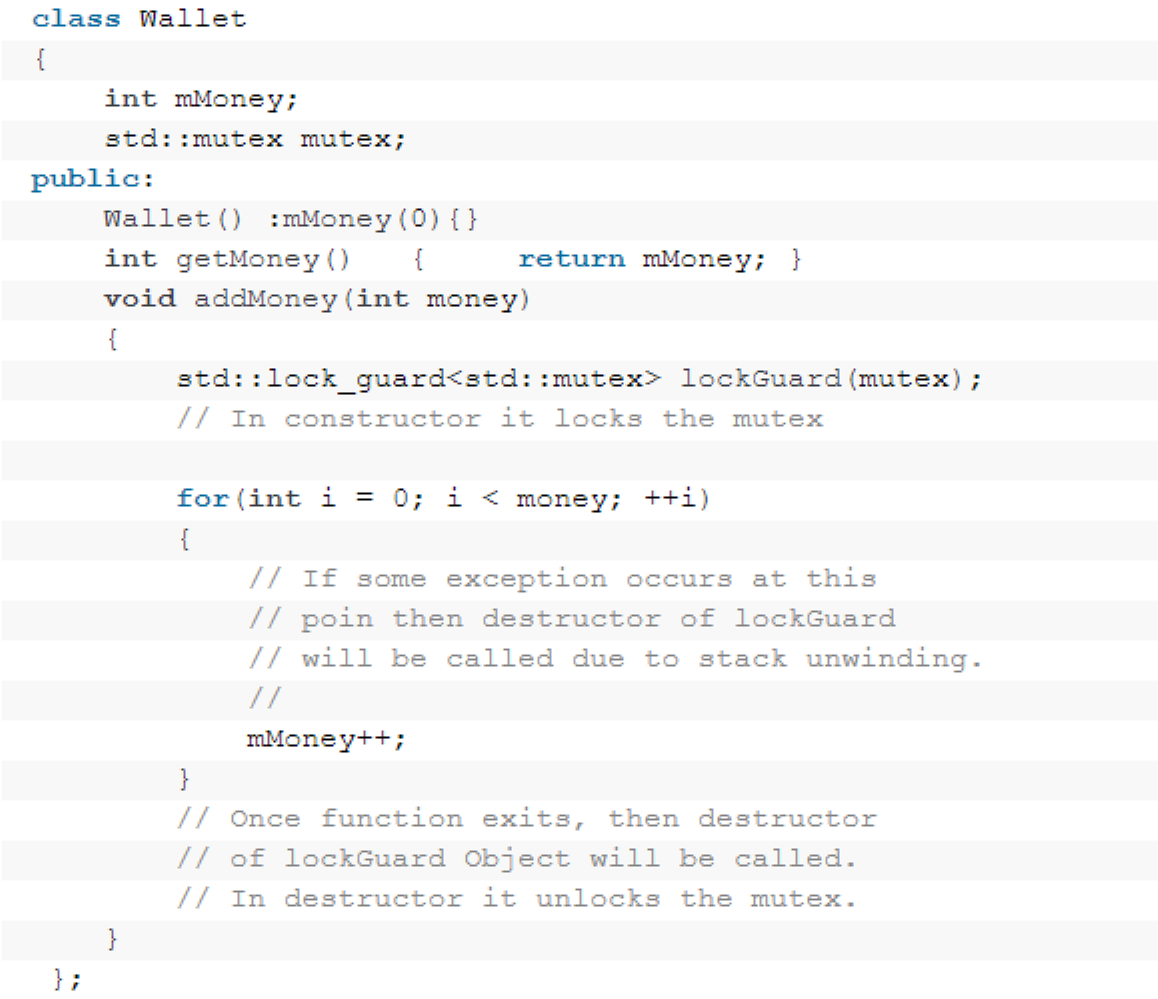
Thread 1 : Order of Commands	Thread 2 : Order of Commands
Load “mMoney” variable value in Register	
	Load “mMoney” variable value in Register
Increment register’s value	
	Increment register’s value
Update variable “mMoney” with register’s value	
	Update variable “mMoney” with register’s value

— Order of Executions Of Commands



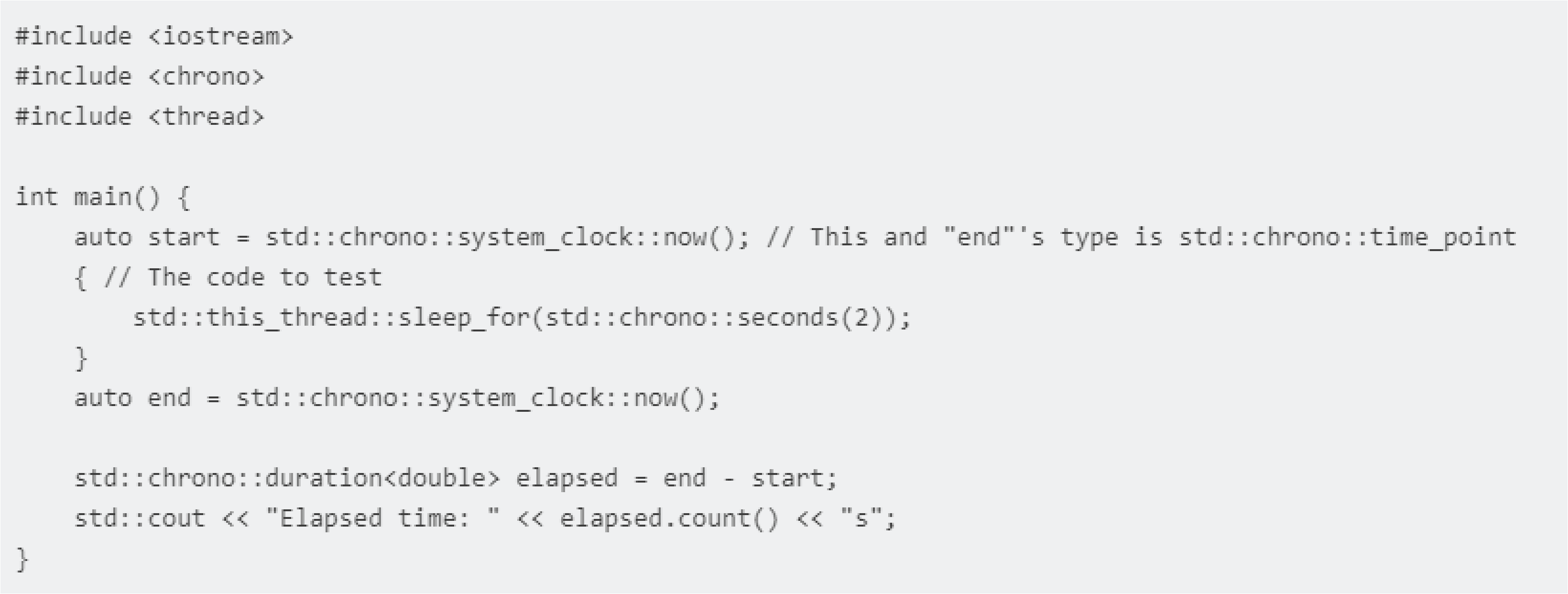


```
#include<iostream>
#include<thread>
#include<vector>
#include<mutex>

class Wallet
{
    int mMoney;
    std::mutex mutex;
public:
    Wallet() :mMoney(0){}
    int getMoney()   {     return mMoney; }
    void addMoney(int money)
    {
        mutex.lock();
        for(int i = 0; i < money; ++i)
        {
            mMoney++;
        }
        mutex.unlock();
    }
};
```



```
class Wallet
{
    int mMoney;
    std::mutex mutex;
public:
    Wallet()  :mMoney(0){}
    int getMoney()    {      return mMoney; }
    void addMoney(int money)
    {
        std::lock_guard<std::mutex> lockGuard(mutex);
        // In constructor it locks the mutex

        for(int i = 0; i < money; ++i)
        {
            // If some exception occurs at this
            // poin then destructor of lockGuard
            // will be called due to stack unwinding.
            //
            mMoney++;
        }
        // Once function exits, then destructor
        // of lockGuard Object will be called.
        // In destructor it unlocks the mutex.
    }
 };
```

std::chrono



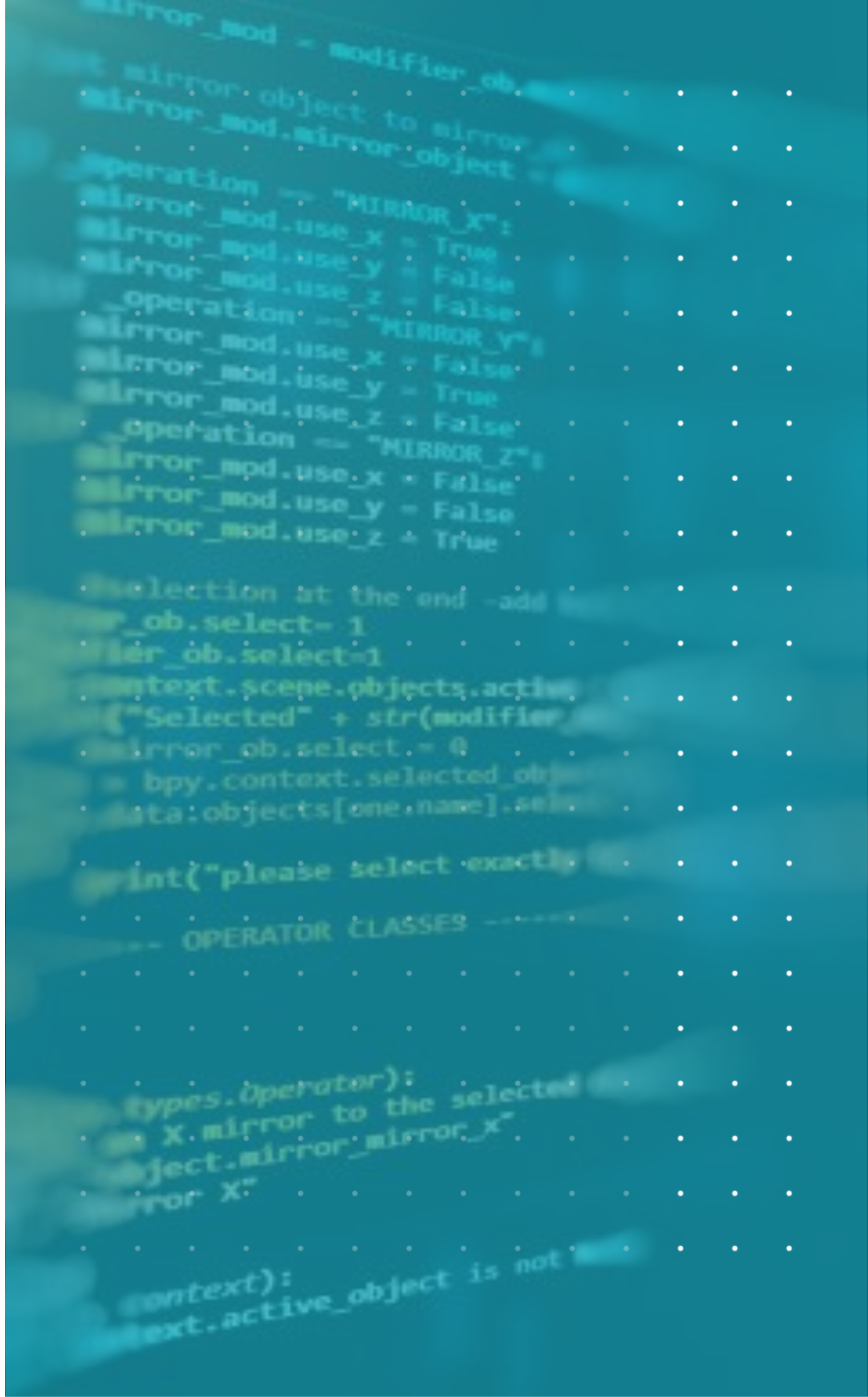
```cpp
#include <iostream>
#include <chrono>
#include <thread>

int main() {
    auto start = std::chrono::system_clock::now(); // This and "end"'s type is std::chrono::time_point
    { // The code to test
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << "s";
}
```

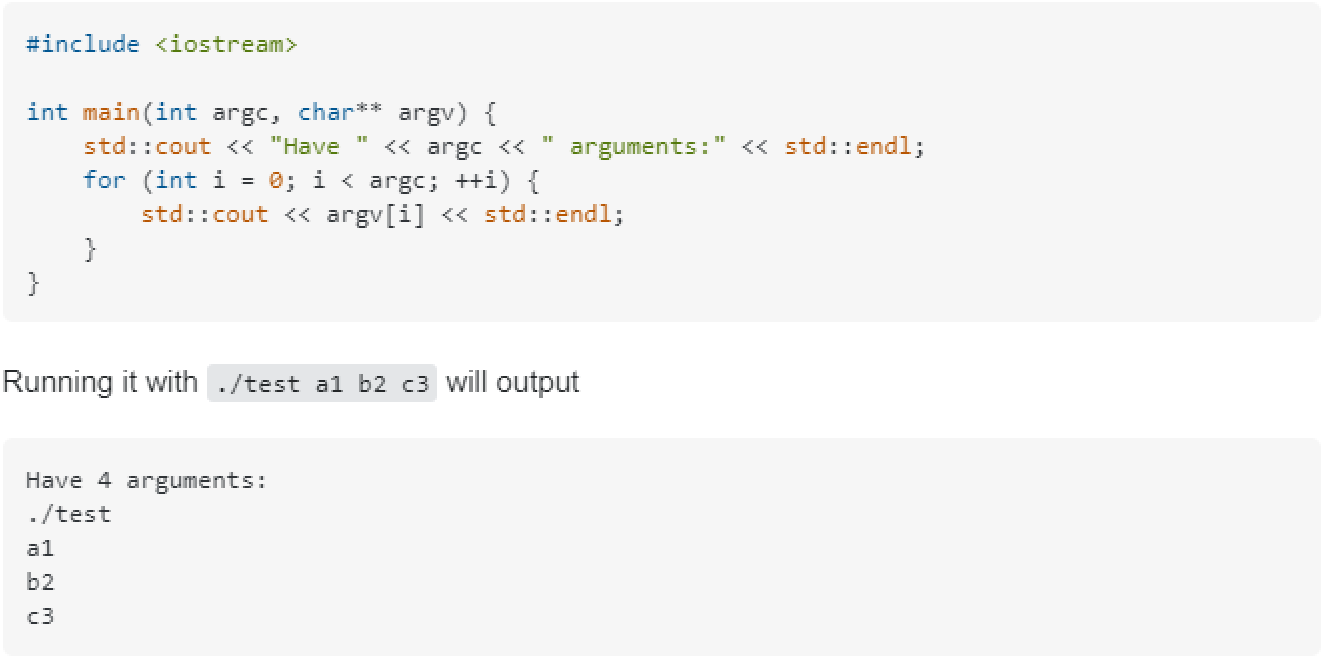
OPTARG

https://linux.die.net
/man/۳/optarg

What does int argc, char *argv[] mean?

- argc will be the number of strings pointed to by argv
- This will (in practice) be ۱ plus the number of arguments, as virtually all implementations will prepend the name of the program to the array.



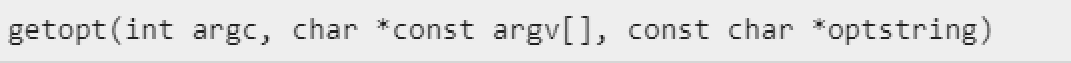


```cpp
#include <iostream>

int main(int argc, char** argv) {
    std::cout << "Have " << argc << " arguments:" << std::endl;
    for (int i = 0; i < argc; ++i) {
        std::cout << argv[i] << std::endl;
    }
}
```

Running it with `./test a1 b2 c3` will output

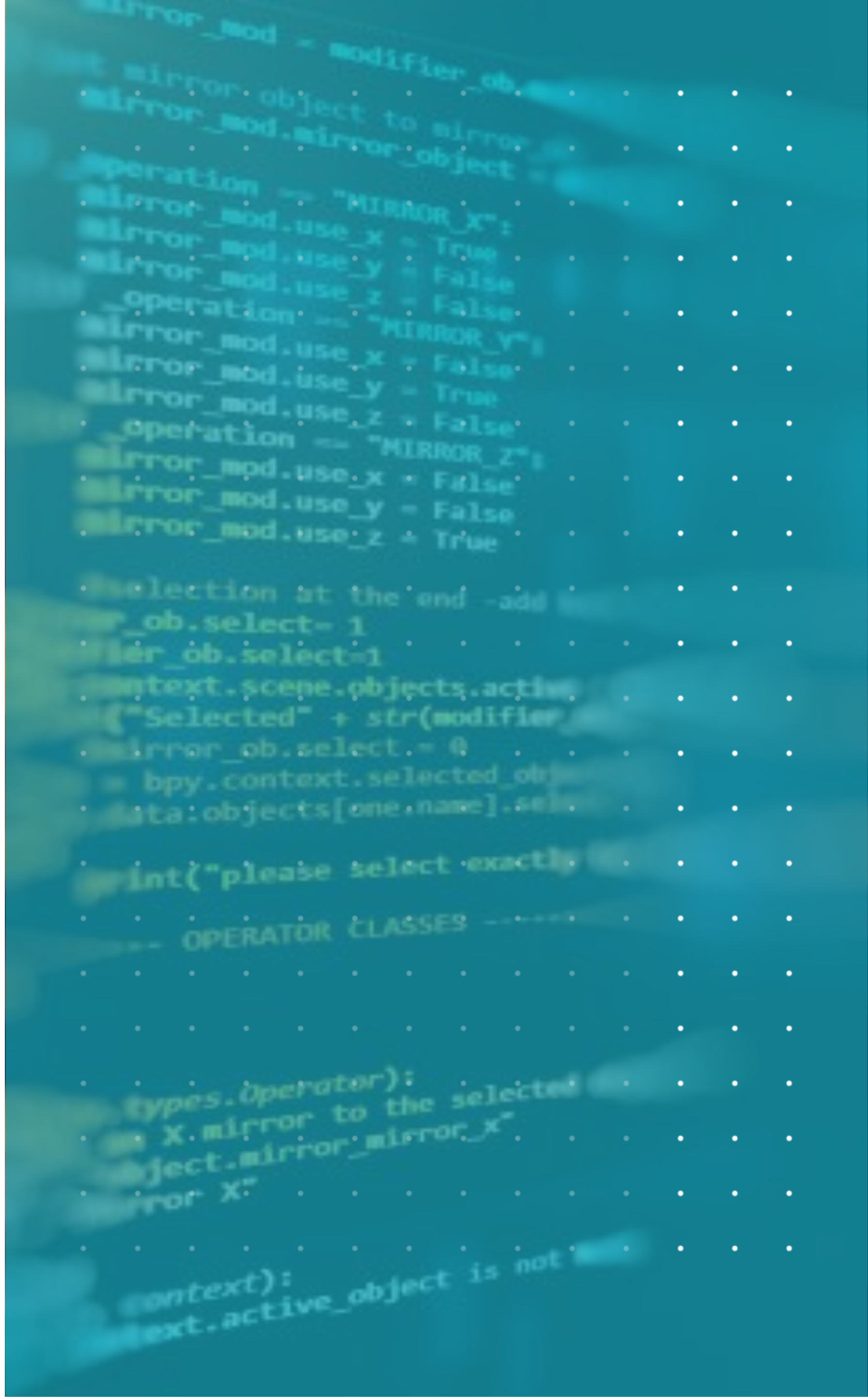
```
Have 4 arguments:
./test
a1
b2
c3
```

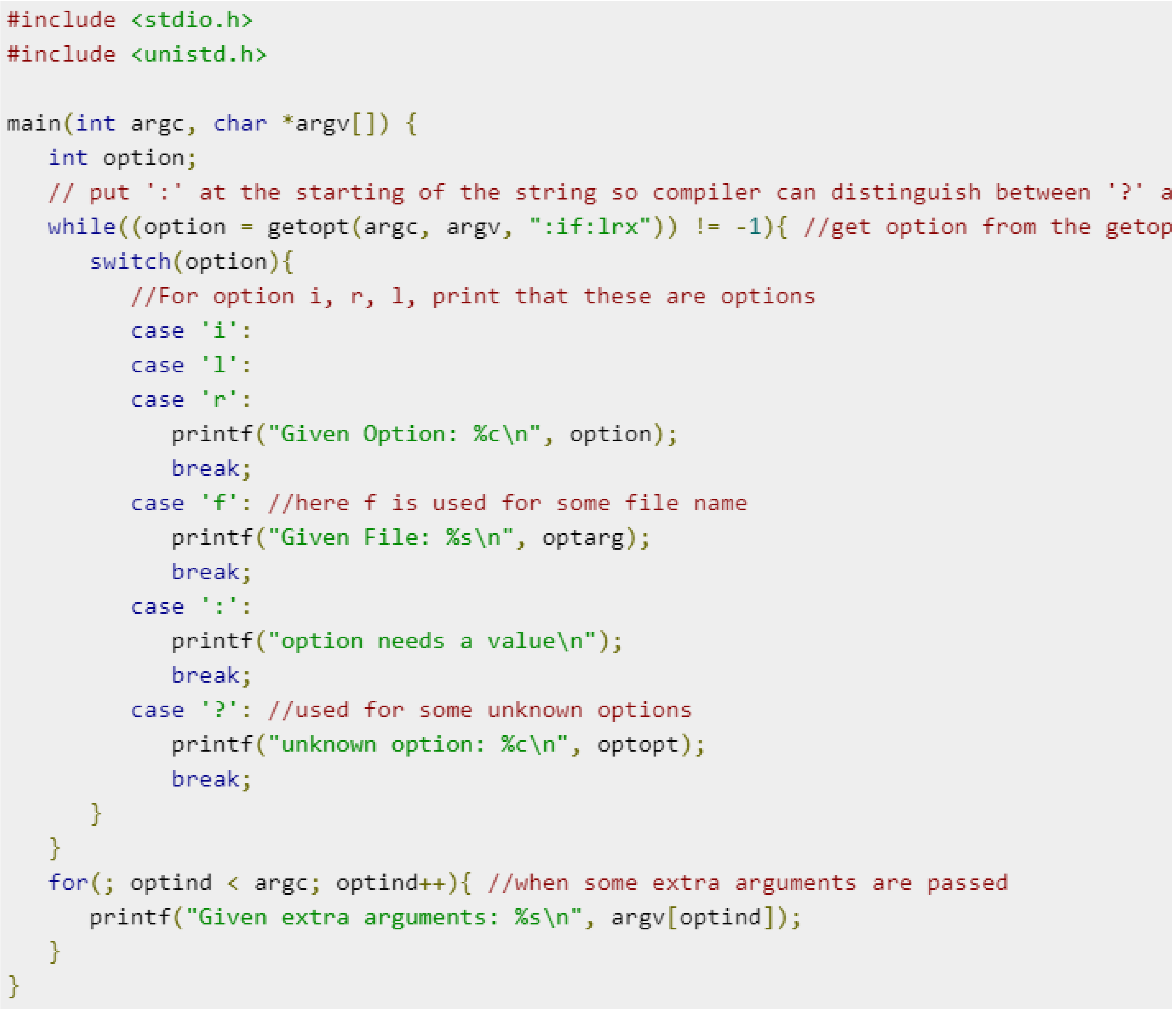
Syntax



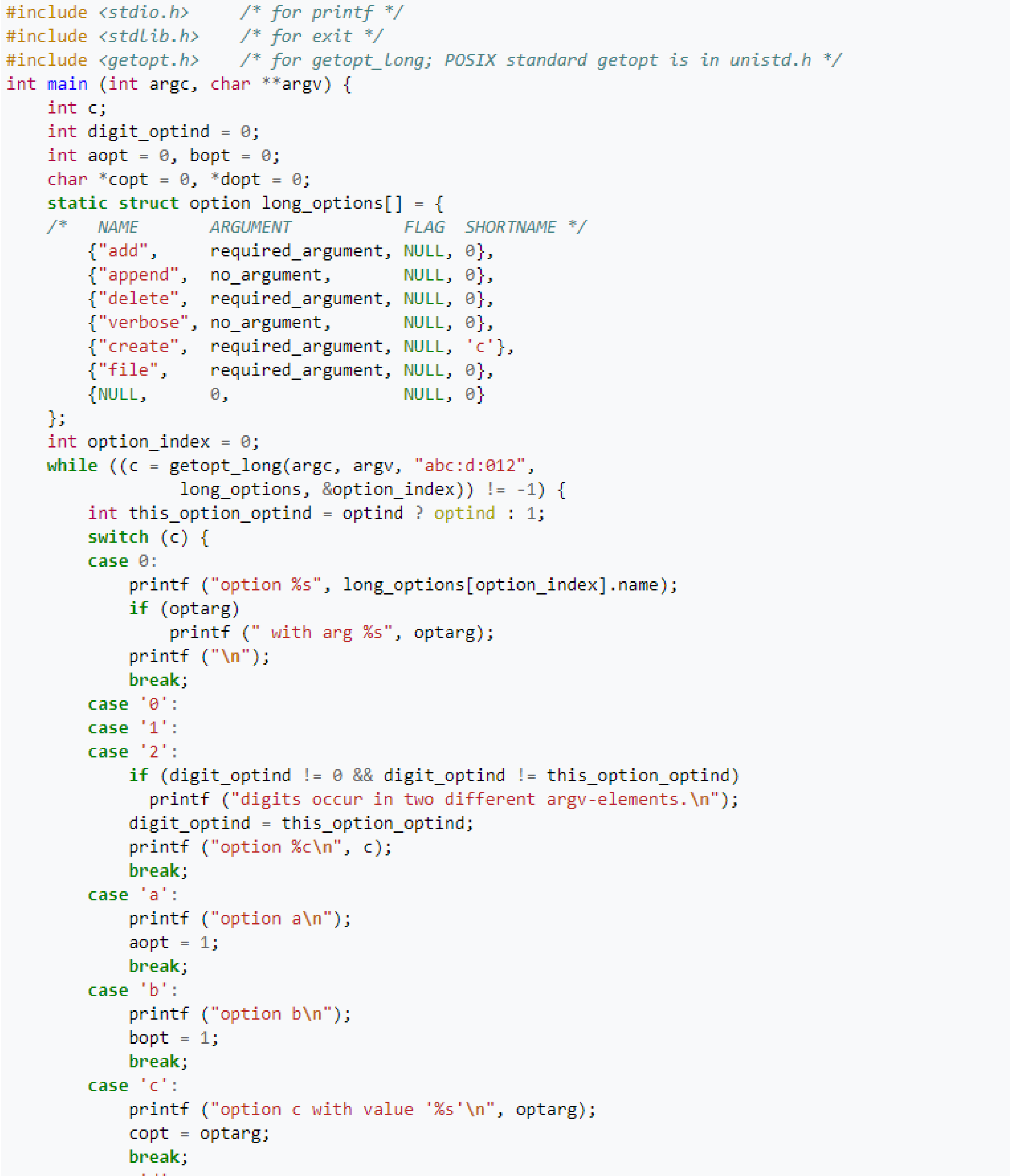
```
getopt(int argc, char *const argv[], const char *optstring)
```

- If the option takes a value, then that value will be pointed by optarg.
- It will return -۱, when no more options to process
- Returns ‘?’ to show that this is an unrecognized option, it stores it to optopt.
- Sometimes some options need some value, If the option is present but the values are not there, then also it will return ‘?’. We can use ‘:’ as the first character of the optstring, so in that time, it will return ‘:’ instead of ‘?’ if no value is given.





```
#include <stdio.h>
#include <unistd.h>

main(int argc, char *argv[]) {
   int option;
   // put ':' at the starting of the string so compiler can distinguish between '?' a
   while((option = getopt(argc, argv, ":if:lrx")) != -1){ //get option from the getop
      switch(option){
         //For option i, r, l, print that these are options
         case 'i':
         case 'l':
         case 'r':
            printf("Given Option: %c\n", option);
            break;
         case 'f': //here f is used for some file name
            printf("Given File: %s\n", optarg);
            break;
         case ':':
            printf("option needs a value\n");
            break;
         case '?': //used for some unknown options
            printf("unknown option: %c\n", optopt);
            break;
      }
   }
   for(; optind < argc; optind++){ //when some extra arguments are passed
      printf("Given extra arguments: %s\n", argv[optind]);
   }
}
```



```c
#include <stdio.h>     /* for printf */
#include <stdlib.h>    /* for exit */
#include <getopt.h>    /* for getopt_long; POSIX standard getopt is in unistd.h */
int main (int argc, char **argv) {
    int c;
    int digit_optind = 0;
    int aopt = 0, bopt = 0;
    char *copt = 0, *dopt = 0;
    static struct option long_options[] = {
    /*   NAME       ARGUMENT           FLAG  SHORTNAME */
        {"add",     required_argument, NULL, 0},
        {"append",  no_argument,       NULL, 0},
        {"delete",  required_argument, NULL, 0},
        {"verbose", no_argument,       NULL, 0},
        {"create",  required_argument, NULL, 'c'},
        {"file",    required_argument, NULL, 0},
        {NULL,      0,                 NULL, 0}
    };
    int option_index = 0;
    while ((c = getopt_long(argc, argv, "abc:d:012",
                 long_options, &option_index)) != -1) {
        int this_option_optind = optind ? optind : 1;
        switch (c) {
        case 0:
            printf ("option %s", long_options[option_index].name);
            if (optarg)
                printf (" with arg %s", optarg);
            printf ("\n");
            break;
        case '0':
        case '1':
        case '2':
            if (digit_optind != 0 && digit_optind != this_option_optind)
              printf ("digits occur in two different argv-elements.\n");
            digit_optind = this_option_optind;
            printf ("option %c\n", c);
            break;
        case 'a':
            printf ("option a\n");
            aopt = 1;
            break;
        case 'b':
            printf ("option b\n");
            bopt = 1;
            break;
        case 'c':
            printf ("option c with value '%s'\n", optarg);
            copt = optarg;
            break;
```

از توجه شما سپاسگزاریم

مهسان

تکیه‌گاه شما
در دنیای هوشمند