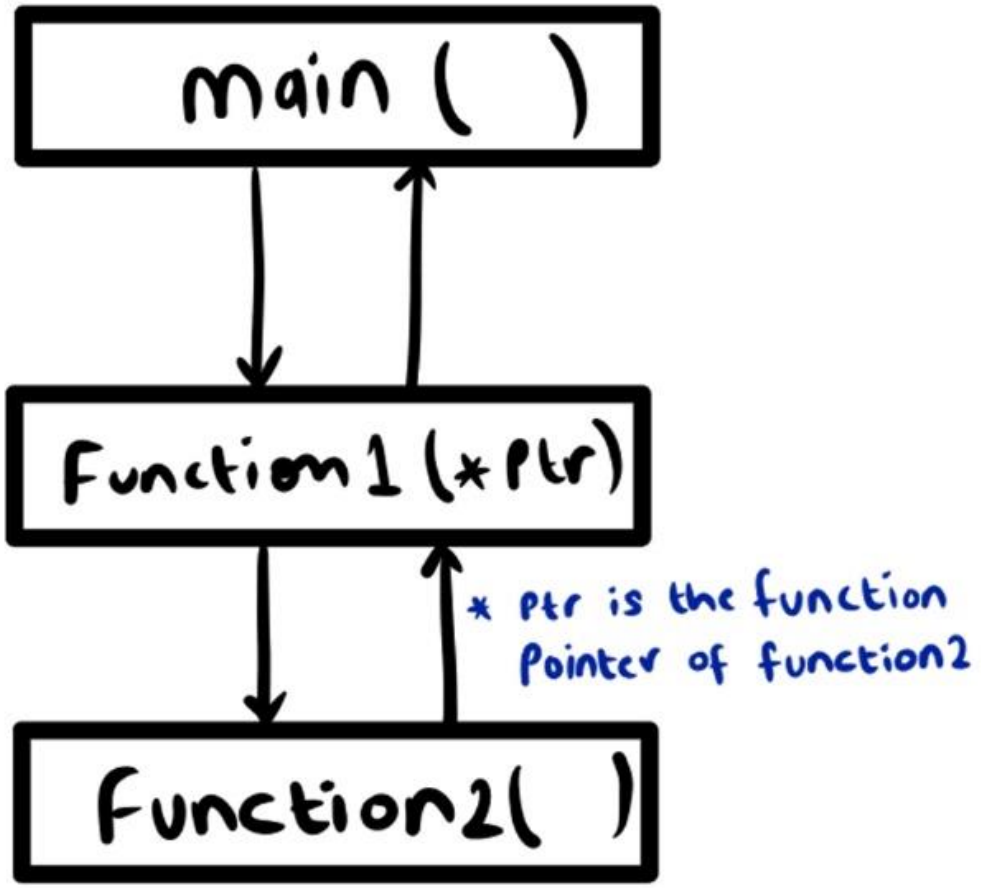
Lambda Expression

Mohamed Saied

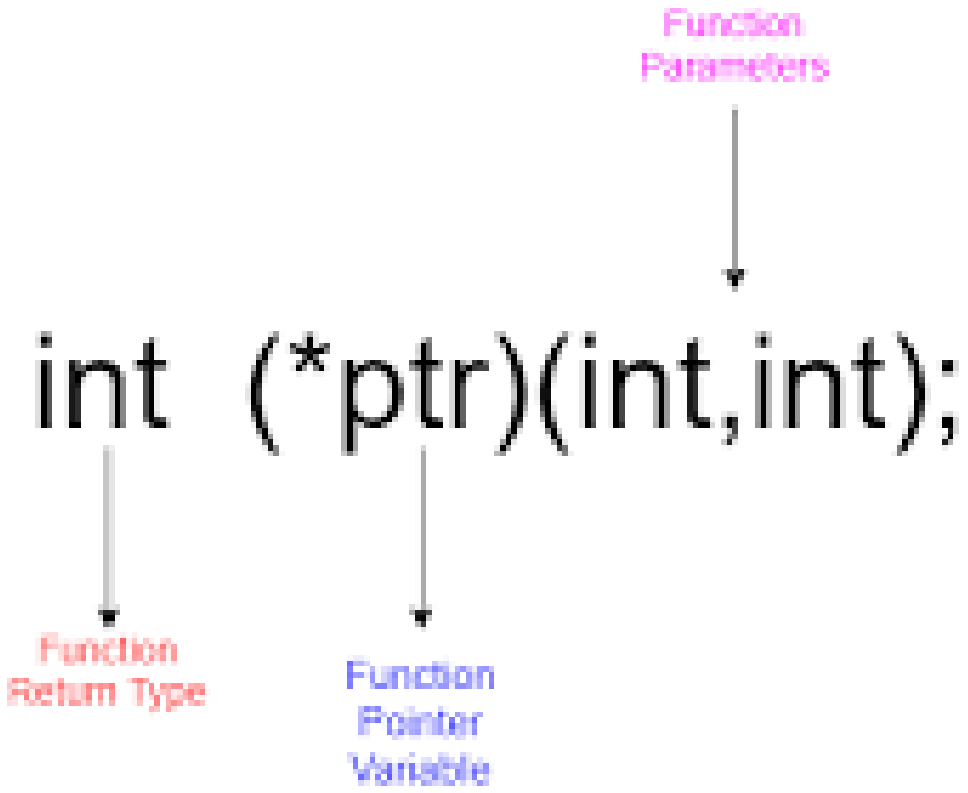
Pointer to function

- we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.



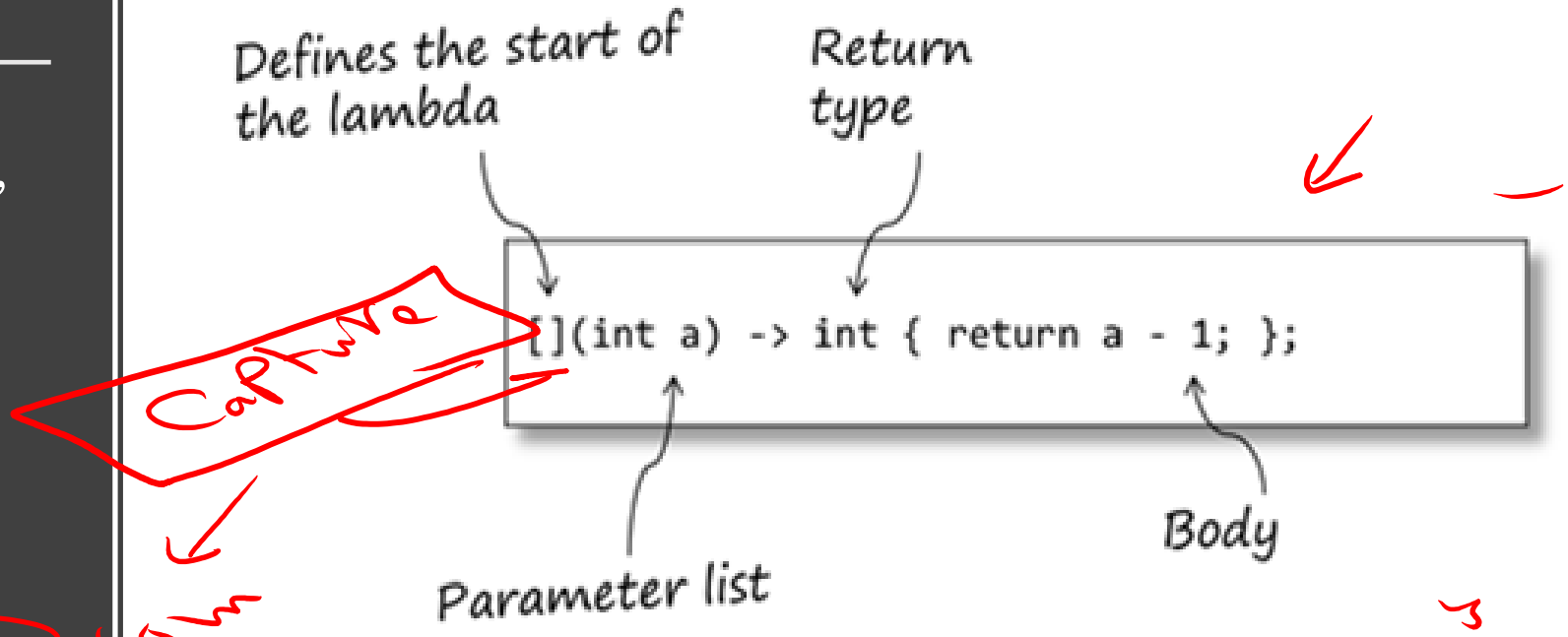
Pointer to function

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
        printf("Value of a is %d\n", a);
}

int main()
{
        // fun_ptr is a pointer to function fun()
        void (*fun_ptr)(int) = &fun;

        /* The above line is equivalent of following two
        void (*fun_ptr)(int);
        fun_ptr = &fun;
        */

        // Invoking fun() using fun_ptr
        (*fun_ptr)(10);

        return 0;
}
```


Function
Parameters
int (*ptr)(int,int);
Function
Return Type
Function
Pointer
Variable

Lambda Expression

- A lambda is an ad-hoc, locally-scoped function (well, more strictly, a functor). Basically, lambdas are syntactic sugar, designed to reduce a lot of the work required in creating ad-hoc functor classes



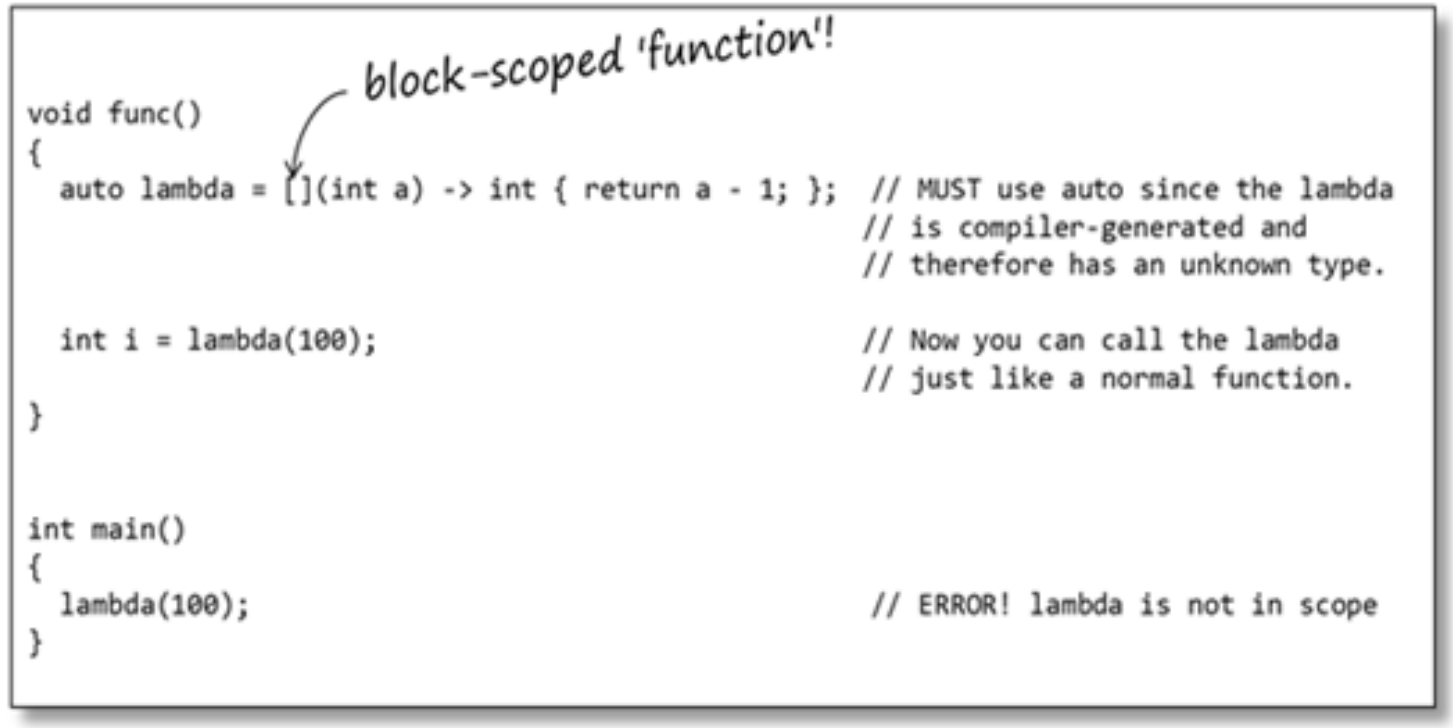
Lambda basics

The brackets ([]) mark the declaration of the lambda; it can have parameters, and it should be followed by its body (the same as any other function).

When the lambda is executed. the parameters are passed using the standard ABI mechanisms. One difference between lambdas and functions: lambda parameters can’t have defaults.

The body of the lambda is just a normal function body, and can be arbitrarily complex (although, as we’ll see, it’s generally good practice to keep lambda bodies relatively simple)

Block scoped function



Use lambda with Algorithm

```
int main()
{
  vector<X> v;
  v.push_back(X());
  v.push_back(X());

  auto lambda = [] (X& elem) -> void { elem.op(); };

  for_each(v.begin(), v.end(), lambda);
}
```

Use lambda in algorithm

Lambda scoped to the for_each algorithm

```
class X
{
public:
  void op();
  int getVal();
};


int main()
{
  vector<X> v;
  v.push_back(X());
  v.push_back(X());

  for_each(v.begin(), v.end(), [](X& elem) { elem.op(); });

  auto i =
    find_if(v.begin(), v.end(), [](X& elem)->bool {return (elem.getVal() != 0);});
}
```

Lambda is scoped to the for_each algorithm

Lambda is scoped to the find_if algorithm

Lambda expression

User code

```
[](X& elem) { elem.op(); }
```

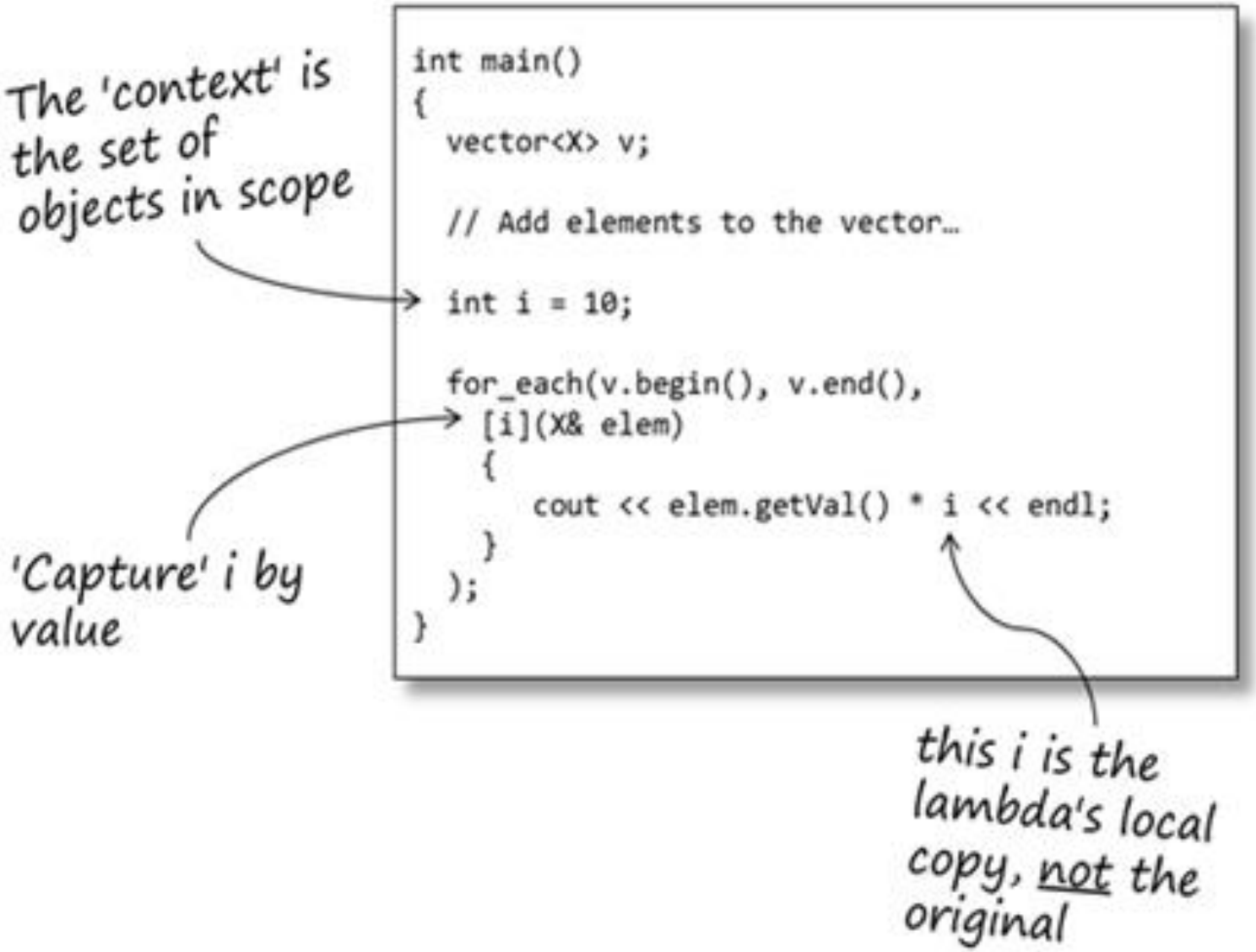
Compiler generated (conceptual)

```
class _SomeCompilerGeneratedName_
{
public:
  void operator() (X& elem) const
  {
    elem.op();
  }
};
```

```
int main()
{
  vector<X> v;
  v.push_back(X());
  v.push_back(X());

  for_each(v.begin(), v.end(), [](X& elem) { elem.op(); });
}
```

Compiler generated (conceptual)

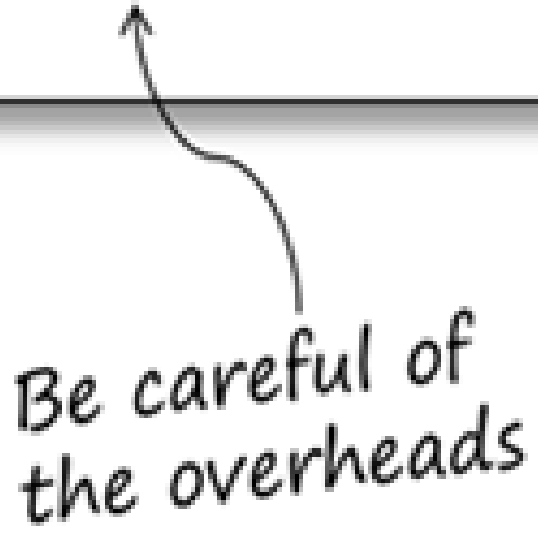
```
class _SomeCompilerGeneratedName_
{
  // As previous…
};

int main()
{
  vector<X> v;
  v.push_back(X());
  v.push_back(X());

  for_each(v.begin(), v.end(), _SomeCompilerGeneratedName_{});
}
```


The 'context' is the set of objects in scope
'Capture' i by value
this i is the lambda's local copy, not the original
int main()
{
vector<X> v;
// Add elements to the vector…
int i = 10;
for_each(v.begin(), v.end(),
[i](X& elem)
{
cout << elem.getVal() * i << endl;
}
);
}

Capture total by reference

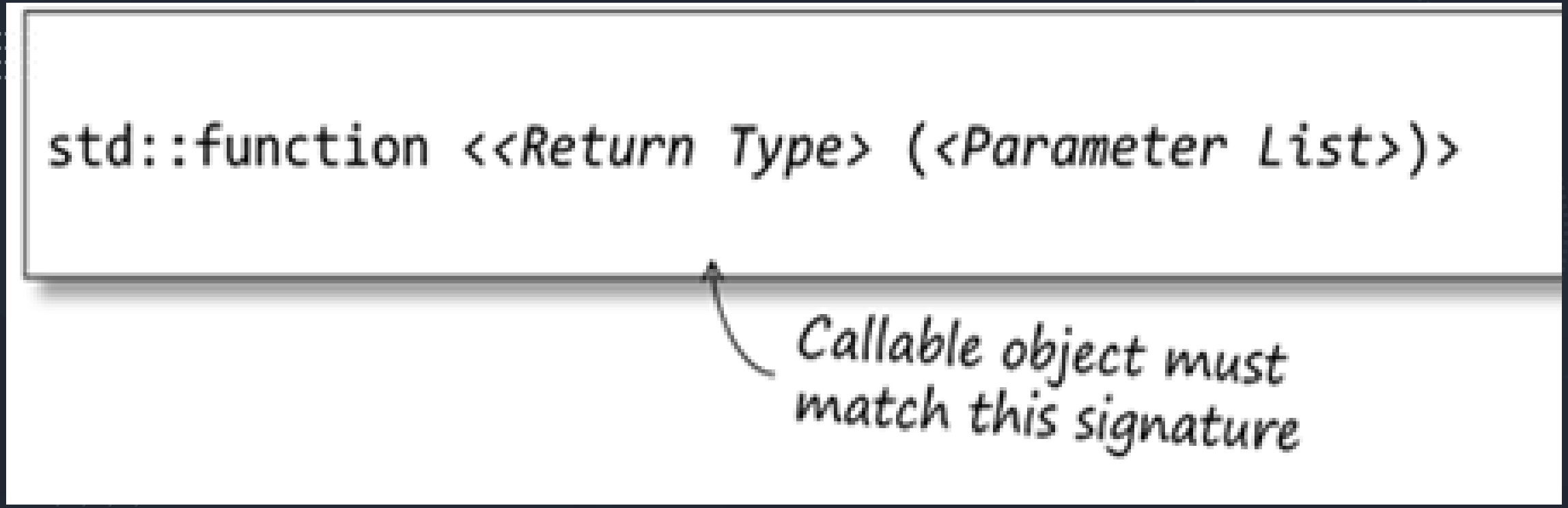
```
int main()
{
  vector<X> v;

  // Add elements to the vector…

  int total = 0;

  for_each(v.begin(), v.end(),
    [&total](X& elem) { total += elem.getVal(); });

  cout << total << endl;
}
```

```
int i;
double d;
X theX;
std::vector<double> v(1000);

auto lam1 = [&]() { /* code… */ };        // Capture everything by reference
auto lam2 = [=]() { /* code… */ };        // Capture everything by value
```



Std::function

- `std::function` is a template class that can hold any callable object that matches its signature. `std::function` provides a consistent mechanism for storing, passing and accessing these objects.



Std::function

- `std::function` can be thought of as a generic pointer-to-function that can point at any callable object, provided the callable object matches the signature of the `std::function`. And, unlike C's pointer-to-function, the C++ compiler provides strong type-checking on the parameters of the callable object (including the return type).

```
#include <functional>

class SimpleCallback
{
public:
  SimpleCallback (std::function<void(void)> f) : callback(f) {}
  void execute();

private:
  std::function<void(void)> callback; // void (*callback)(void)
};


void SimpleCallback::execute()
{
  if (callback != nullptr)           // Is the function valid?
  {
    callback();                      // Call like a normal function
  }
}
```

With functors...

```
class Functor
{
public:
  void operator()() { cout << "Functor" << endl; }
};

int main()
{
  Functor functor;
  SimpleCallback callback(functor);
  callback.execute();
}
```

With functions...

```
void func()
{
  cout << "Free function" << endl;
}

int main()
{
  SimpleCallback callback(func);
  callback.execute();
}
```

...or with lambdas

```
int main()
{
  SimpleCallback callback([]() { cout << "Lambda" << endl; });
  callback.execute();
}
```

Callback with std::function and Lambdas