Conception Objets Avancée

Functional programming in C++

Giuseppe Lipari

CRIStAL - Université de Lille

Outline

Outline

Function objects

Lambda

Functional programming in C++

A functional List

Functions or objects?

- A function object is any class that implements `operator()`
- A function object can be *called* as it were a simple function
- However, it is also an object, so it can contain state
- It is sometimes called functor

```
class Fn {
public:
    double operator()(double x);
   ...
};

Fn obj; // construction

double result = obj(3.14); // calls Fn::operator(3.14)
```

Calling functions and functors

- The syntax has been conceived so that it is impossible to distinguish to call to a function object from the call to a normal function.
  - In this way, you can call a functor wherever you can call a function
  - This is particularly useful with templates
  - Many std library functions take a function object, so that they can be easily generalised
- The difference between a function and a functor is that the latter can store state in the internal variables
  - the behaviour is different from one call to another

Example: summing numbers

- Write a function that sums all element of a vector
- Can we write it in a generic way ?
  - see `examples/sum.cpp`

Outline

Lambda functions

- Sometimes the amount of code to be written in a functor is very small;
- However, a functor requires, as a minimum, to write a class and overload the `operator()`
- With *lambda-functions* it is possible to write the code in-line
- lambda-function: a function with no name
  - The name of a function is only a convention to be able to call the function later on
  - If the function is in-line, we can omit its name, since it can be called immediately, or its address passed to another function

Lambda syntax

- Basic syntax

```
auto func = [] () -> int {
    cout << "Hello" << endl;
    return 0;
};
```

This declares a functor object that

1. takes no argument
2. returns an integer
3. the body just prints "hello" on the terminal before returning 0

Lambda syntax II

```
[<capture>] (<param_list>) -> <return> { <body> };
```

1. `[]` is the *capture specification*, and contains the list of variables of outer scopes that can be used inside the lambda function
2. `()` contains the parameter list
3. `->` precedes the return type
   - The return value specification can be omitted if it can automatically be deduced by the compiler
4. `{}` contains the code and is a regular function code

Example

```
int main()
{
    vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int sum = 0;

    for_each(v.begin(), v.end(),
             [&sum](int x) {
                 sum += x;
            });

    cout << "Sum (with functor) : " << sum << endl;

    double d = 0;
    int n = 0;
    for_each(v.begin(), v.end(),
             [&d, &n](int x) {
                 d += x; ++n;
            });

    cout << "Avg (with functor) : " << d/n << endl;
}
```

Capturing

- Closure: a lambda function + the variables in the scope of the function
- The capture specification is used to delimit the variables in the outer scope that can be used inside the body of the lambda function
- Syntax:
  - `[]` captures nothing
  - `[&]` captures any variable by reference
  - `[=]` captures any variable by copy
  - `[=,&foo]` captures any variable by copy, except foo that is captured by reference
  - `[bar]` captures bar by copy
  - `[this]` captures the *this* pointer of the enclosing class

Example of closure

- examples/closure.cpp

```
#include <vector>
#include <iostream>

template<typename Iter, typename Fun>
void apply(Iter b, Iter e, Fun f)
{
        for (Iter i = b; i != e; i++) f(*i);
}


int main()
{
    std::vector<int> v = {1, 2, 3, 4, 5};
    int sum = 0;

    apply(std::begin(v), std::end(v), [&sum](int x) { sum += x; });

    std::cout << "sum = " << sum << std::endl;
}
```

Another example of closure

- examples/closure2.cpp

```
#include <iostream>

using namespace std;

auto create_fun(int x)
{
    return [x] (int y) { return x + y; };
}

int main()
{
    auto f1 = create_fun(5);
    auto f2 = create_fun(7);

    cout << f1(5) << endl;
    cout << f2(5) << endl;

    return 0;
}
```

- What happens if you capture x by reference?

Closure on mutable variables

- examples/closure3.cpp

```
#include <iostream>

int global = 0;

auto create_mutable(int x)
{
    return [x, &global](int y) { global = x + y; };
}

int main()
{
    auto f3 = create_mutable(5);
    auto f4 = create_mutable(7);

    std::cout << "global = " << global << std::endl;
    f3(5);
    std::cout << "global = " << global << std::endl;
    f4(5);
    std::cout << "global = " << global << std::endl;

    return 0;
}
```

Lambda inside a class

- If you capture `this`, then you can use all local variables with automatic indirection

```
class Foo
{
public:
    Foo () : _x( 3 ) {}
    void func () {
        [this] () { cout << _x; } ();
    }

private:
    int _x;
};

int main()
{
    Foo f;
    f.func();
}
```

Type of lambda function

- What type is a lambda function (or a functor?)
  - Knowing the type is useful if we want to write a non-template function that takes a functor as argument

```
std::function<int (int, double)> f =
    [] (int x, double y) -> int {
        // code
    };
```

- `std::function` is a template that takes as arguments the return type, followed by the list of arguments within parenthesis
- `std::function` object can “contain”
  - a function pointer
  - a function object
  - a lambda function

Exercise

- Given a vector (or list) of doubles, write a function computes the sum of the squares
  - Solution: `examples/sumsquares.cpp`

Transform and reduce

From C++17 on:

- `std::transform` corresponds to the classical map operation of functional languages
  - transforms a sequence of objects by calling a function and storing the results in a different sequence of objects
- `std::reduce` corresponds to the classical reduce operation of functional languages
  - combines the elements in a sequence to obtain a single result
  - example: `accumulate` is a specialization of reduce

Exercise

- Exercise:
  - write a moving average function over a set of double numbers

```
template<class ItIn, ItOut>
double moving_avg(ItIn a, ItIt b, int n, ItOut c);
```

  - hint: use the `std::transform()` and `std::reduce` library functions

Outline

Pure functions

- In functional programming the emphasis is on pure functions
  - a function is pure if it has no side effects
  - a pure function always returns the same value when called with the same parameters
- Examples:
  - mathematical functions like *abs(x)*, *sin(x)*, *cos(x)*
  - (stable) sorting of a vector always produces the same result
  - determinant of a matrix,
  - ecc.
- Pure functions are desirable because
  - they have no internal state $\rightarrow$ maybe easier to understand and to code
  - they can be executed concurrently because they do not modify shared memory

The functional way of programming

- Basic ideas:
  1. Functions are first class objects
     - declare functions
     - compose functions
     - partially evaluate a function
     - lazy evaluation (computation is performed only if needed)
  2. Avoid state as much as you can
     - you should avoid variables
     - substitute loops with recursion
- Essential tools:
  1. Functors
  2. Lambdas

Example

- Suppose you want to sum all elements of a container in C/C++
- Straightforward (non functional) solution:

```
// only works when non-empty
vector<int> elems;
...
int s = 0;
for (auto it = elems.begin(); it != elems.end(); ++it)
   s += *it;

cout << "Sum: " << s << endl;
```

Example II

- Now the functional (recursive) version:

```
vector<int> elems;
...
cout << "Sum: " << compute_sum(elems.begin(), elems.end()) << endl;
...

template<typename It>
int compute_sum(It b, It e ) {
    if (b == e) return 0;
    else return *b + compute_sum(++b, e);
}
```

Functions are first class

In function programming, the emphasis is on functions: so we need to manipulat functions in several ways:

- threat them as objects
  - passing them to functions, and returning from functions
- Compose function:
  - given f and g, obtain $h = f \cdot g$
- currying
  - bind a parameter: given $f(a, b, c)$, compute $f'(a, b) = f(a, b, \overline{c})$

Funtional header

- The std library provides many ways for manipulating functions

```
template< class >
class function; /* undefined */

template< class R, class... Args >
class function<R(Args...)>
```

- Instances of `std::function` can store, copy, and invoke any callable target
  - functions, lambda expressions, bind expressions, or other function objects

Example II

- Typical functional code:

```
int main()
{
    std::vector<std::string> words = {"This", "is", "a", "test"};

    std::vector<std::size_t> lengths;
    std::transform(words.begin(),
                   words.end(),
                   std::back_inserter(lengths), [](const string &n) { return
    std::cout << "The string lengths are ";
    for(auto n : lengths) std::cout << n << ' ';
}
```

Binding

- Binding arguments

```
template< class F, class... Args >
/*unspecified*/ bind( F&& f, Args&&... args );

template< class R, class F, class... Args >
/*unspecified*/ bind( F&& f, Args&&... args );
```

  - The function template `bind` generates a forwarding call wrapper for `f`
  - Calling this wrapper is equivalent to invoking `f` with some of its arguments bound to `args`.
  - The arguments to `bind` are copied or moved, and are never passed by reference

Examples

```cpp
void f(int a, double b);

int x = 5;
double y = 7;
f(x, y);

// Reordering
using namespace std::placeholders; // for _1, _2, etc.
auto f_ord = std::bind(f, _2, _1);
f_ord(y, x);
```

Examples (cont.)

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

Foo foo;
auto f3 = std::bind(&Foo::print_sum, &foo, 95, _1);
f3(5);

auto f4 = std::bind(&Foo::data, _1);
std::cout << f4(foo) << '\n';
```

Composition and currying

- Instead of the complex `std::bind` we can use the simpler lambda functions:
  - To compose two functions f and g:

```
int f(int x);
int g(int y);
// we want to create h(x) = g(f(x));
auto h = [](int x) { return g(f(x)); };
```

  - To bind a parameter:

```
int f(x,y);
const int k = 1234;
// we want to create h(x) = f(x, k)
auto h = [k](int x) { return f(x, k); };
```

Generic composition

- See examples/compose.cpp
  - taken from here

Outline

Immutable data structures

- One important aspect of functional programming are immutable data structures
  - An immutable data structure cannot be modified in place
  - If we want to modify it, we have to create a new object that contains the modifications
- In this way, functions acting on the data structure are pure by design
  - they cannot produce global effects
- As an exercise, we will see how to code an immutable List in C++
  - The code is derived from Bartosz Milewski's course on Functional Programming in C++ https://bartoszmilewski.com/2013/11/13/functional-data-structures-in-c-lists/

Basic structure

```
template<class T>
class List
{
    struct Item {...};
public:
    List() : _head(nullptr) {}
    List(T v, List tail) : _head(new Item(v, tail._head)) {}
    bool isEmpty() const { return !_head; }
private:
    // may be null
    Item const * _head;
};
```

- Since all lists are immutable, we do not need to deep copy all elements
  - when adding an element to the head, we can just copy the pointer to the rest of the list
  - A list that has been created empty will always remain empty!

Item

- The list item contains elements of type T, plus a pointer to the next element

```
struct Item {
    Item(T v, Item const * tail) : _val(v), _next(tail) {}
    T _val;
    Item const * _next;
};
```

- How to obtain the first element ?

```
T front() const {
    assert(!isEmpty());
    return _head->_val;
}
```

Operations

- How to "remove" the first element ?

```
template<class T>
class List {
private:
    explicit List (Item const * items) : _head(items) {}

public:
    List pop_front() const {
        assert(!isEmpty());
        return List(_head->_next);
    }
};
```

- Notice that the `pop_front()` is declared const, because it does not actually remove anything
  - it just returns the rest of the list

Releasing resources

- There is no garbage collector in C++, so we have to take care of de-allocation.
  - Since the memory of an Item is shared across multiple lists, it is not clear who owns the memory (who should release it)
  - This it a job for `shared_ptr` !!

```
std::shared_ptr<const Item> _head;
```

  - Construction:

```
List() {}
List(T v, List const & tail)
    : _head(std::make_shared<Item>(v, tail._head)) {}
```

Releasing resources

- Also, Item needs a shared pointer :

```
struct Item
{
    Item(T v, std::shared_ptr<const Item> const & tail)
        : _val(v), _next(tail) {}
    T _val;
    std::shared_ptr<const Item> _next;
};
```

- All done!
  - the resources are automatically released when the reference counter goes to zero
  - Pay attention to circular references ...

Operations on Lists

- Suppose we want to call a function on every element of the list

```
template<class U, class T, class F>
List<U> fmap(F f, List<T> lst)
{
    static_assert(std::is_convertible<F, std::function<U(T)>>::value,
                  "fmap requires a function type U(T)");
    if (lst.isEmpty())
        return List<U>();
    else
        return List<U>(f(lst.front()), fmap<U>(f, lst.pop_front()));
}
```

- This is more or less equivalent to the `std::transform()` function, except that `transform` uses iterators and loops, while `fmap` uses recursion
- Suppose you have a list of characters, that you want to trasform to upperCase:

```
auto charLst2 = fmap<char>(toupper, charLst);
```

Example

- How to compute the list of the prime numbers between $[1; N]$
  - two algorithms: eratostene and primes2.
- See `examples/eratostene.cpp`
- See `examples/primes2.cpp`
- Look at the run-time ...
  - Which one has better performance? Why?