Lecture 14

C++ as a multiparadigm programming language

TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
9th October 2017

Ahmed Rezine, IDA, Linköping University

14.1

Content

Contents

14.2

1 Introduction

Multiparadigm language

C++ is a multiparadigm language and let the programmer choose and combine between the various characteristics of the language:

- structured
- procedural
- object-oriented
- generic
- functional

The functional aspects of C++ have improved with C++11

- lambda expression
- variadic templates
- STL-function bind and function

14.3

2 Functional programming

2.1 Functional style

Programming in functional style

- Automatic inference with
  - **`auto`** and `decltype`
- Support for lambda expression
  - closures
  - functions as data
- Partial function application
  - `std::function` and `std::bind`

```
std::function<double(double)> f = std::bind(std::divide,
                                            std::placeholrder::_1, 2.0);
```

  - lambda expression and **`auto`**
- High order functions of the algorithms in STL
- List manipulation with variadic templates

- Pattern matching with full and partial template specialization
- Lazy evaluation with `std::async`

```
auto value = std::async(std::launch::deferred, []{ ... });
```

14.4

2.2 Why functional programming?

Why functional programming?

- STL
  - effective use of lambda expression

```
accumulate(vec.begin(), vec.end(), 0
           [](int a, int b){return a+b;});
```

- Template programming
  - recognition of functional patterns

```
template <int N>
struct Fac{ static int const val= N * Fac<N-1>::val; };
template <>
struct Fac<0>{ static int const val= 1; };
```

- Better programming style
  - discuss side effect
  - can be more concise or clearer

```
for (auto v: vec) cout << v << "␣" << endl;
copy(v.begin(), v.end(), ostream_iterator<T>(cout, "␣"));
```

14.5

2.3 What is functional programming?

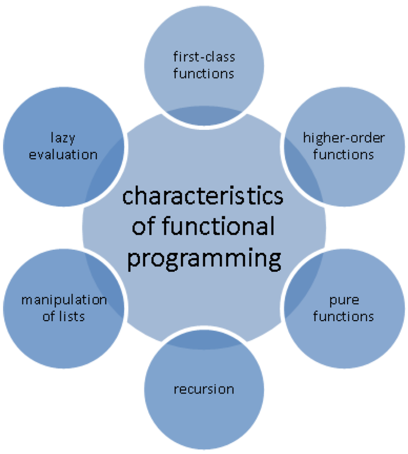
What is functional programming?

- **Functional programming** is similar to programming with mathematical functions
- **Mathematical functions** are functions that given the same arguments returns the same answer
  - functions must not have any side effect
  - call to the function can be replaced by its results
  - an optimiser can change the order of function calls or perform calls in different threads
  - application flow is driven by dependencies and not by the sequence of instructions

14.6

2.4 Characteristics

Characteristics of functional programming



14.7

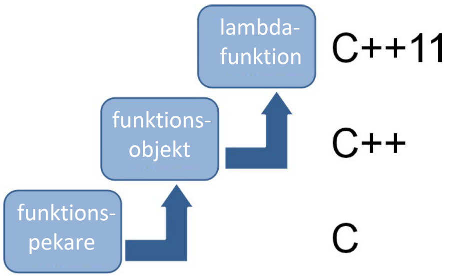
First-class citizens

- In a programming language, a **first-class citizen** is an entity that:
  - can be passed as an argument
  - can be returned from a function
  - can be assigned to a variable
- In most programming language:
  - scalar types are first-class citizens
  - array and strings are not generally (ie in C)
- what about functions?

14.8

First-class functions

- First class functions are first-class citizens:
  - Functions are treated as data
  - Name of the function is not important
- Functions
  - can be passed as arguments to other functions
  - functions can be returned by other functions
  - can be assigned to variables or stored in data structures



14.9

First class function: call table

```
map<const char,function<double(double,double)>> tab;

tab.insert(make_pair('+',[](double a,double b){return a + b;}));
tab.insert(make_pair('-',[](double a,double b){return a - b;}));
tab.insert(make_pair('*',[](double a,double b){return a * b;}));
tab.insert(make_pair('/',[](double a,double b){return a / b;}));

cout << "3.5+4.5= " << tab['+'](3.5,4.5) << endl; // 8
cout << "3.5*4.5= " << tab['*'](3.5,4.5) << endl; // 15.75

tab.insert(make_pair('^', [](double a,double b){return pow(a,b);}));
cout << "3.5^4.5= " << tab['^'](3.5,4.5) << endl; // 280.741
```

14.10

High order functions

Higher order functions are functions which accept other function as arguments and/or return them as a result.

- The three classics:
  - **map**: apply a function to each element in a list (`std::transform` in C++)
  - **filter**: remove element from a list (`std::remove_if` in C++)
  - **fold**: reduce a list to a single element by successive application of a binary operation (`std::accumulate` in C++)

14.11

High order functions

All programming language which support functionnal style offers **map**, **filter** and **fold**

Haskell	Python	C++
map	map	std::transform
filter	filter	std::remove_if
fold	reduce	std::accumulate

- **map**, **filter** and **fold** are three powerfull function that are applicable in many cases
  - map + reduce = MapReduce

14.12

Higher order functions

- List and vector:
  - Haskell:

```
vec= [1 . . 9]
str= ["Programming","in","a","functional","style."]
```

  - Python:

```
vec=range(1,10)
str=["Programming","in","a","functional","style."]
```

  - C++:

```
vector<int> vec{1,2,3,4,5,6,7,8,9}
vector<string>str{"Programming","in","a","functional","style."}
```

14.13

Higher order functions

- map
  - Haskell:

```
map(\a -> a^2) vec
map(\a -> length a) str
```

  - Python:

```
map(lambda x : x*x, vec)
map(lambda x : len(x), str)
```

  - C++:

```
transform(vec.begin(),vec.end(),vec.begin(),
          [](int i){ return i*i; });
transform(str.begin(),str.end(),back_inserter(vec2),
          [](string s){ return s.length(); });
```

- Result: [1,4,9,16,25,36,49,64,81]
  [11,2,1,10,6]

14.14

Higher order functions

- filter
  - Haskell:

```
filter(\x -> x<3 || x>8) vec
filter(\x -> isUpper(head x)) str
```

  - Python:

```
filter(lambda x: x<3 or x>8 , vec)
filter(lambda x: x[0].isupper(), str)
```

  - C++:

```
auto it= remove_if(vec.begin(),vec.end(),
                   [](int i){ return !((i < 3) or (i > 8)) });
auto it2= remove_if(str.begin(),str.end(),
                   [](string s){ return !(isupper(s[0])); });
```

- Result: [1,2,9]
  ["Programming"]

14.15

Higher order functions

- fold
  - Haskell:

```
foldl (\a b -> a * b) 1 vec
foldl (\a b -> a ++ ":" ++ b ) "" str
```

  - Python:

```
reduce(lambda a , b: a * b, vec, 1)
reduce(lambda a, b: a + b, str, "")
```

  - C++:

```
accumulate(vec.begin(),vec.end(),1,
           [](int a, int b){ return a*b; });
accumulate(str.begin(),str.end(),string(""),
           [](string a,string b){ return a+":"+b; });
```

- Result: 362800 and ":Programming:in:a:functional:style."

14.16

"Pure" function

"Pure" vs "impure" function (from the book of The Real World Haskell)

Pure function	Impure function
Allways produces the same results given the same arguments	Can produce different result given the same argument.
Never has a side effect	Can have side effect
Never change state	Can change a global state in the program, system or world

- Pure functions are isolated. The program will be easier to
  - reason about
  - refactor and test
- Very good for optimisation
  - Save the result of a call
  - Rearrange pure function call or share it between threads

14.17

“Pure” function

- Monad is Haskell solution to handle impure world
- A monad
  - encapsulate the impure world in pure Haskell
  - it is an imperative subsystem in Haskell
  - it is a structure representing calculation
  - must define the composition of calculations
- Example:
  - I/O-monad for handling I/O operations
  - Maybe-monad for computations that can fail
  - List-monad for calculations of zero or more valid responses
  - Authorisation-monad for computation that require permission

14.18

Recursion

```
Fac<5>::value =
              = 5 * Fac<4>::value
              = 5 * 4  * Fac<3>::value
              = 5 * 4 * 3 * Fac<2>::value
              = 5 * 4 * 3 * 2 * Fac<1>::value
              = 5 * 4 * 3 * 2 * 1 *  Fac<0>::value
              = 120
```

- Loop:
  - Recursion are controlstructure
  - A loop `for( int i <= 10; ++i)` needs a variable `i`
    - Mutable variable are not valid in language such as Haskell
- Recursion combined with list manipulation is a powerfull pattern in functionnal languages

14.19

Recursion

- Recursion
  - Haskell:

    ```
    fac 0 = 1
    fac n = n * fac (n-1)
    ```

  - C++:

    ```
    template<int N>
    struct Fac{
        static int const value = N * Fac<N-1>::value;
    };

    template <>
    struct Fac<0>{
        static int const value = 1;
    };
    ```

- Resultat: fac(5) == Fac<5>::value == 120

14.20

List manipulation

- List manipulation (LISt Processing) is important in functional programming:
  - transform a list into an other list
  - reduce a list to a single value
- The functional pattern for list manipulation:
  1) Handle first element `x`
  2) Handle rest of the list (`xs`) recursively => Go to step 1)
  - Example:

```
mySum [] = 0
mySum (x:xs) = x + mySum xs
mySum [1,2,3,4,5]                // 15
myMap f [] = []
myMap f (x:xs)= f x: myMap f xs
myMap (\x -> x*x)[1,2,3]         // [1,4,9]
```

14.21

List manipulation

```
template<int ...> struct mySum;

template<>struct
mySum<>{
    static const int value= 0;
};

template<int i, int ... tail> struct
mySum<i,tail...>{
    static const int value= i + mySum<tail...>::value;
};

int sum= mySum<1,2,3,4,5>::value; // sum == 15
```

- Implementing myMap with variadic template is not going to be fun. . .

14.22

Lazy evaluation

- Lazy evaluation only evaluate an expression if it is needed
  - Haskell is lazy because the following works

```
    length [2+1, 3*2, 1/0, 5-4]
```

  - C++ is eager but the following works

```
template <typename... Args>
void mySize(Args... args) {
    cout << sizeof...(args) << endl;
}
mySize("Wohoo", 1/0);
```

- Advantages:
  - Save time and memory
  - Work with infinite data structures

14.23

Lazy evaluation

- Example:

```
successor i = i: (successor (i+1))
take 5 ( successor 10 ) // [10,11,12,13,14]

odds = takeWhile (< 1000) . filter odd . map (^2)
[1..] = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 ... Control -C
odds [1..]             // [1,9,25, ... , 841,961]
```

- Special case: short circuit evaluation

```
if ( true || (1/0) ) cout << "short␣circuit␣evaluation␣in␣C++\n";

if ( pointer && pointer->can_run() ) cout << "Check␣if␣pointer␣is␣non␣null!\n";
```

14.24

What is missing?

- List comprehensions:
  - Syntactic sugar on map and filter
  - Example:

```
[(s,len(s)) for s in ["Only","for"]] # [('Only', 4), ('for', 3)]
[i*i for i in range(11) if i%2 == 0] # [0,4,16,36,64,100]
```

- Funciton composition:
  - Programing with lego pieces
  - Example:

```
(reverse . sort)[10,2,8,1,9,5,3,6,4,7] - - [10,9,8,7,6,5,4,3,2,1]
theLongestTitle = head . reverse . sortBy(comparing length) .
    filter isTitle
theLongestTitle words("A␣Sentence␣Full␣Of␣Titles.")
```

    - Result: “Sentence”

14.25