Program specialization and metaprogramming

Laboratory Report

Michał Wojtaszek  
MS Inf. Sem VI 2020

Laboratory work No. 1

*Introduction into C++*

*template metaprogramming*

**1. Aims**

To get acquainted with the basics of template metaprogramming in C++.

**2. Tasks**

Learn how to do develop generic functions and classes using C++ template metaprogramming.

Learn how to produce efficient C++ code using template specialization.

**3. Work**

**A. Compile-time programs**

#include <iostream>

using namespace std;

template<int N>

class Factorial {

public:

enum { value = N \* Factorial<N - 1>::value };

};

template<>

class Factorial<1> {

public:

enum { value = 1 };

};

int main()

{

std::cout << Factorial<5>::value;

}

**What is the result of this program?**

Output: 120 (value of 5!)

The program calculate the recursive value during building the program, so at run-time we get only effect of previous work.

**B. Compile-time functions**

inline void swap(int& a, int& b)

{

int temp = a;

a = b;

b = temp;

}

template<int I, int J>

class IntSwap {

public:

static inline void compareAndSwap(int\* data)

{

if (data[I] > data[J])

swap(data[I], data[J]);

}

};

template<int I, int J>

class IntBubbleSortLoop {

private:

enum { go = (J <= I - 2) };

public:

static inline void loop(int\* data)

{

IntSwap<J, J + 1>::compareAndSwap(data);

IntBubbleSortLoop<go ? I : 0, go ? (J + 1) : 0>::loop(data);

}

};

template<>

class IntBubbleSortLoop<0, 0> {

public:

static inline void loop(int\*)

{ }

};

template<int N>

class IntBubbleSort {

public:

static inline void sort(int\* data)

{

IntBubbleSortLoop<N - 1, 0>::loop(data);

IntBubbleSort<N - 1>::sort(data);

}

};

template<>

class IntBubbleSort<1> {

public:

static inline void sort(int\* data)

{ }

};

int main()

{

int array[4] = { 9,2,7,6 };

IntBubbleSort<4>::sort(array);

for (int i = 0; i < 4; i++)

std::cout << array[i] << '\n';

}

**What is the result of this program?**

The result is sorted list, like 2, 6, 7, 9.

//generic metafunction calls (inherits from) it self

template<int IIn, int ISum = 1>

struct Factorial : Factorial<IIn - 1, IIn \* ISum> {

};

//specialized metafunction has a value and does not inherit

template<int ISum> //take ISum as a wild card

struct Factorial<1, ISum> {

enum { value = ISum };

};

int main()

{

int i = Factorial<4>::value;

cout << i;

}

The output is the result of 4!, 24.

**Temporary variables**

template<int N>

class countBits {

enum {

bit3 = (N & 0x08) ? 1 : 0,

bit2 = (N & 0x04) ? 1 : 0,

bit1 = (N & 0x02) ? 1 : 0,

bit0 = (N & 0x01) ? 1 : 0

};

public:

enum { nbits = bit0 + bit1 + bit2 + bit3 };

};

int main()

{

int i = countBits<13>::nbits;

cout << i;

}

The output is 3.

5.

template <size\_t N, class T>

class DotProduct {

public:

static T eval(T\* a, T\* b)

{

return DotProduct<1, T>::eval(a, b)

+ DotProduct<N - 1, T>::eval(a + 1, b + 1);

}

};

template <class T>

class DotProduct<1, T> {

public:

static T eval(T\* a, T\* b)

{

return (\*a) \* (\*b);

}

};

template <size\_t N, class T>

inline T dot(T\* a, T\* b)

{

return DotProduct<N, T>::eval(a, b);

}

int main()

{

int a[4] = { 1,100,0,-1 };

int b[4] = { 2,2,2,2 };

cout << dot<4>(a, b);

}

**What is the result of this program?**

The output is 200.

**Challenge**

#include <iostream>

#include <chrono>

using namespace std;

template<int N>

class Factorial {

public:

enum { value = N \* Factorial<N - 1>::value };

};

template<>

class Factorial<1> {

public:

enum { value = 1 };

};

int main()

{

std::chrono::steady\_clock::time\_point begin = std::chrono::steady\_clock::now();

std::cout << Factorial<10>::value;

std::chrono::steady\_clock::time\_point end = std::chrono::steady\_clock::now();

std::cout << "Time difference = " << std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count() << "[µs]" << std::endl;

return 0;

}

I chose to challenge the same meta-function with different parameters.

for 10 is 3628800Time difference = 938 micro seconds 3256

For 100 Time difference = 4379 micro seconds

The difference is significant and depends on value of calculated number.

**What have you learned?**

During these labs I learned basic concepts of programming compile-time code. I have never heard about it before. I revised also some C++ basics (the success for me is that I didn’t need to google how to print something on the screen :D).

Laboratory work No. 2

Expression templates

**1. Aims**

To get acquainted with the expression templates technique in C++.

**2. Tasks**

1. To learn how to build linguistic structures representing a computation at compile time.

2. To learn how to use expression templates to achieve C++ code optimization.

**3. Work**

#include <iostream>

template<class A>

class DExpr {

private:

A a\_;

public:

DExpr()

{ }

DExpr(const A& x)

: a\_(x)

{ }

double operator()(double x) const

{

return a\_(x);

}

};

class DExprIdentity {

public:

DExprIdentity()

{ }

double operator()(double x) const

{

return x;

}

};

template<class A, class B, class Op>

class DBinExprOp {

A a\_;

B b\_;

public:

DBinExprOp(const A& a, const B& b)

: a\_(a), b\_(b)

{ }

double operator()(double x) const

{

return Op::apply(a\_(x), b\_(x));

}

};

class DExprLiteral {

private:

double value\_;

public:

DExprLiteral(double value)

{

value\_ = value;

}

double operator()(double x) const

{

return value\_;

}

};

class DApAdd {

public:

DApAdd() { }

static inline double apply(double a, double b)

{

return a + b;

}

};

class DApDivide {

public:

DApDivide() { }

static inline double apply(double a, double b)

{

return a / b;

}

};

//operator+(double, DExpr)

template<class A>

DExpr<DBinExprOp<DExprLiteral, DExpr<A>, DApAdd> >

operator+(double x, const DExpr<A>& a)

{

typedef DBinExprOp<DExprLiteral, DExpr<A>, DApAdd> ExprT;

return DExpr<ExprT>(ExprT(DExprLiteral(x), a));

}

// operator/(DExpr, DExpr)

template<class A, class B>

DExpr<DBinExprOp<DExpr<A>, DExpr<B>, DApDivide> >

operator/(const DExpr<A>& a, const DExpr<B>& b)

{

typedef DBinExprOp<DExpr<A>, DExpr<B>, DApDivide> ExprT;

return DExpr<ExprT>(ExprT(a, b));

}

template<class Expr>

void evaluate(DExpr<Expr> expr, double start, double end)

{

const double step = 1.0;

for (double i = start; i < end; i += step)

std::cout << expr(i) << std::endl;

}

int main()

{

DExpr<DExprIdentity> x; // Placeholder

evaluate(x / (1.0 + x), 0.0, 10.0);

}

**What is the result of program execution?**

The result of the program:

0

0.5

0.666667

0.75

0.8

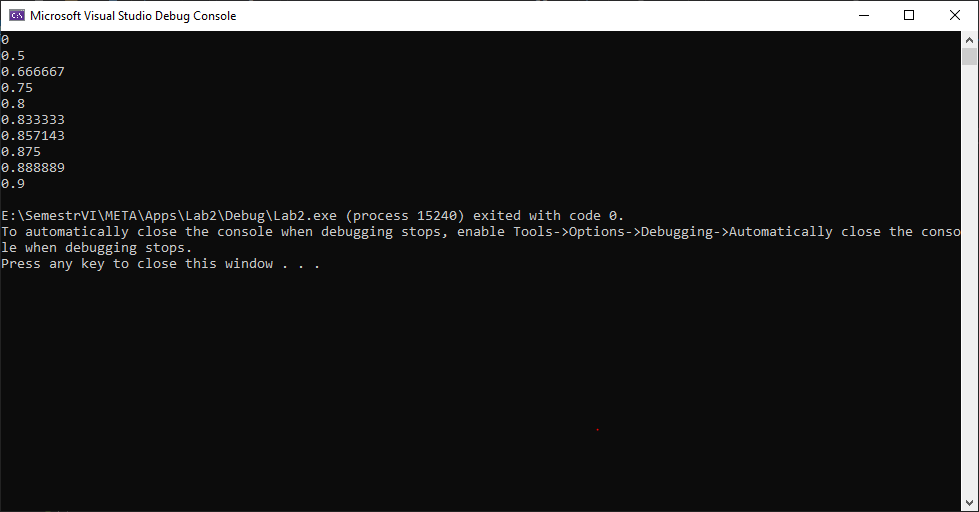
0.833333

0.857143

0.875

0.888889

0.9



1. **Challenge**

1. Modify the program to incorporate more functions such as sqrt(), exp() and log() into

expressions, by defining appropriate functions and applicative templates. Use these functions for implementing an expression object representing a normal distribution:

double mean = 5.0, sigma = 2.0; // math constants

DExpr<DExprIdentity> x;

evaluate(1.0 / (sqrt(2 \* M\_PI) \* sigma) \* exp(sqr(x - mean) /

(-2 \* sigma \* sigma)), 0.0, 10.0);

2. It is possible to generalize the classes presented here to expressions involving arbitrary types

(instead of just doubles).

1. **What have you learned?**

Complete this.