ИПМ им.М.В.Келдыша РАН

Абрау-2017 • Труды конференции

М.М. Краснов

Применение метапрограммирования шаблонов C++ для решения вычислительных задач

Рекомендуемая форма библиографической ссылки

Краснов М.М. Применение метапрограммирования шаблонов C++ для решения вычислительных задач // Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции (18-23 сентября 2017 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2017. — С. 290-305. — URL: **http://keldysh.ru/abrau/2017/33.pdf** doi:**10.20948/abrau-2017-33**

Размещена также презентация к докладу

Применение метапрограммирования шаблонов C++ для решения вычислительных задач

М.М. Краснов

ИПМ им. М.В. Келдыша РАН

Аннотация. Рассматривается применение метапрограммирования шаблонов языка C++ для упрощения записи алгоритмов и для переноса части вычислений (различных целочисленных констант, например, биномиальных коэффициентов) на стадию компиляции, что теоретически позволяет ускорить выполнение программ и может быть весьма актуальным для численного решения задач математической физики. Метапрограммирование шаблонов позволяет автоматизировать некоторые сложные вычисления, такие, например, как раскрытие скобок в сложных выражениях и вычисление символьных производных от формул. Это помогает избавиться от трудно отлавливаемых ошибок, возникающих при проведении подобных вычислений вручную (на бумаге) из-за невнимательности.

Ключевые слова: язык C++, метапрограммирование шаблонов, шаблоны выражений, символьное дифференцирование, операторный подход к программированию, CUDA.

1. Введение

Под метапрограммированием в языке C++ подразумеваются вычисления на стадии компиляции. Основным инструментом для этого являются шаблоны классов, называемые в этом случае (в случае их использования для вычислений на стадии компиляции) метафункциями. Параметрами таких метафункций служат параметры шаблонов классов, а значения метафункций (их может быть несколько) хранятся в статических (static) переменных или определениях типов (typedef-ах) класса. Язык C++ накладывает определённые ограничения на то, что может быть передано в качестве параметров шаблонов, и на типы статических переменных, значения которых могут быть вычислены на стадии компиляции. Это могут быть константы целочисленных типов и произвольные типы (как встроенные в язык простые типы, так и классы). В частности, нельзя передавать в качестве параметров шаблонов и вычислять на стадии компиляции константы вещественных типов, а также шаблоны классов.

При подстановке значений параметров в шаблон класса получается класс. При этом для разных наборов значений параметров компилятор создаёт разные классы. Кроме того, для некоторых конкретных значений параметров можно создавать «особые» классы, имеющие другую реализацию, чем общий шаблон

класса. Такие «особые» классы называются специализациями классов. Специализации классов могут быть частичными (когда конкретные значения указываются для части параметров) или полными (когда конкретные значения указываются для всех параметров). Под полным шаблоном класса подразумевается совокупность основного определения шаблона класса и всех его специализаций. В частности, специализации классов используются для организации рекурсивных циклов. В рекурсивных функциях важно вовремя завершить рекурсию, в метапрограммировании для этого как раз и служат специализации классов для предельных значений параметров шаблонов.

Шаблоны (классов и функций) – один из самых сложных элементов языка C++. Они появились в языке не сразу и первоначально были задуманы для того, чтобы можно было писать универсальные коллекции (которые могут хранить значения произвольных типов) и алгоритмы (например, алгоритм сортировки набора объектов произвольного типа). Метапрограммирование – другое, не менее важное применение шаблонов. С его помощью можно вынести часть вычислений на стадию компиляции и, таким образом, ускорить (иногда существенно) выполнение программ, что бывает особенно важно при численном решении задач математической физики, которые могут считаться весьма долго. Кроме того, применения метапрограммирования шаблонов может упростить решение некоторых логически сложных задач и уменьшить вероятность ошибок в программах, связанных с невнимательностью. Такие ошибки часто бывает очень трудно найти. В качестве примеров таких задач можно привести автоматическое раскрытие скобок в сложных выражениях и символьное дифференцирование произвольных выражений.

Несколько слов по поводу графических ускорителей CUDA (см. [6]). Как уже говорилось выше, компилятор для CUDA поддерживает шаблоны, и никаких проблем с использованием метапрограммирования нет. Более того, начиная с версии 7.0 компилятор nvcc поддерживает стандарт языка C++11. Но это не сильно помогает, так как для переноса программы на CUDA её всё равно нужно сильно переписать. Так можно ли написать программу один раз так, чтобы компилировать и запускать её можно было в том числе и на CUDA? Существует несколько решений этой задачи, один из них – стандарт OpenACC. Этот подход аналогичен OpenMP, т.е. решение заключается в автоматическом распараллеливании витков цикла. Применительно к OpenACC это означает, что исходные данные будут скопированы на CUDA, там параллельно обработаны, а затем результаты будут скопированы обратно в главный (host) процессор. Такой подход не представляется оптимальным. Автором предлагается альтернативный, т.н. «сеточно-операторный» подход к программированию, в котором при работе с CUDA данные изначально расположены в памяти CUDA, и копирование «туда-сюда» не требуется. Исходные данные при необходимости копируются в память CUDA в начале работы программы, а в конце работы копируются обратно в главный процессор. Сеточно-операторный подход к программированию также основывается на шаблонном метапрограммировании.

2. Простейшие метафункции

В данном разделе мы покажем простейшие метафункции, которые в дальнейшем будут использоваться для более сложных вычислений. Как уже говорилось, метафункция с точки зрения языка C++ – это шаблон класса, параметры метафункции – это параметры этого шаблона класса, в качестве них могут передаваться или целочисленные константы, значения которых известны во время компиляции, или типы.

Факториал

Пусть число N известно во время компиляции. Тогда значение N! во время компиляции можно вычислить с помощью следующей метафункции:

```
template<unsigned N>
struct factorial {
  static const unsigned value = N * factorial<N - 1>::value;
};
template<>
struct factorial<0> {
  static const unsigned value = 1;
};
// Пример обращения:
const unsigned f5 = factorial<5>::value; // 120
```

Первый шаблон класса factorial является общим случаем и реализует рекурсивный вызов по известному правилу. Второй класс – это т.н. «полная специализация» шаблона класса и служит для завершения рекурсии. Результат метафункции хранится в static const переменной value типа unsigned. Если число N окажется слишком большим, то результат в переменную типа unsigned не поместится. В этом случае компилятор выдаст ошибку. Но в реальности в численных методах слишком большие числа для факториала не встречаются, и таких проблем не возникает.

Биномиальные коэффициенты

Пусть числа N и K известны во время компиляции. Тогда биномиальные коэффициенты в соответствие с известной формулой могут быть вычислены с помощью следующей метафункции:

```
template<unsigned N, unsigned K>
struct C {
  static const unsigned value =
    factorial<N>::value / factorial<K>::value / factorial<N-K>::value;
};
// Пример обращения:
const unsigned i = C<5, 2>::value; // 10
```

Интеграл от одночлена

Пусть требуется вычислить трёхмерный интеграл по стандартному трёхмерному симплексу от одночлена вида $x^\alpha y^\beta z^\gamma$, причём числа α, β и γ известны во время компиляции. Напомним, что стандартный трёхмерный симплекс – это тетраэдр с вершинами в точках (0, 0, 0), (1, 0, 0), (0, 1, 0) и (0, 0, 1). Вычисление таких интегралов нам понадобится в дальнейшем, когда мы будем вычислять интегралы от произвольных многочленов по произвольному тетраэдру. Такой интеграл вычисляется аналитически и равен

$$\int_0^1 x^\alpha dx \int_0^{1-x} y^\beta dy \int_0^{1-x-y} z^\gamma dz = \frac{\alpha! \cdot \beta! \cdot \gamma!}{(\alpha+\beta+\gamma+3)!}$$

Это число вещественное (меньше единицы) и на стадии компиляции вычислено быть не может. Но обратное к нему число целое (это ясно из комбинаторных соображений). Вот текст метафункции, вычисляющий на стадии компиляции число, обратное интегралу от одночлена:

```
template<unsigned alpha, unsigned beta, unsigned gamma>
struct monomial_3d {
  static const unsigned value =
    factorial<alpha + beta + gamma + 3>::value /
      (factorial<alpha>::value * factorial<beta>::value
        * factorial<gamma>::value);
};
```

Вычисление типа значения итератора

В стандартной библиотеке языка C++ есть функция accumulate, вычисляющая сумму элементов между двумя итераторами. Прототип этой функции следующий:

```
template<typename IT, typename T>
T accumulate(IT first, IT last, T init);
```

Эта функция в качестве параметров принимает два итератора (начала и конца) и начальное значение. Она имеет один недостаток: тип возвращаемого значения определяется типом начального значения, и поэтому нужно быть очень внимательным, чтобы не сделать досадную ошибку. Поясним это на примере следующего кода:

```
double a[3] = { 1.23, 2.34, 3.45 };
double sum = std::accumulate(a, a + 3, 0);
```

Теперь, внимание, вопрос: чему будет равно значение переменной sum? Правильный ответ – 6 (кто не верит, может проверить). Результат, очевидно, совсем не тот, который мы ожидаем (ожидаемый результат 7.02). Причина неправильного результата в том, что в качестве начального значения мы передали ноль целого типа. Соответственно, результат функции также целого типа. Более того, округление до целого делается на каждом шаге внутреннего

цикла. Правильный вызов функции accumulate в данном случае должен быть следующий:

```
double sum = std::accumulate(a, a + 3, 0.);
```

Всего лишь одна точка, а какая большая разница! Теперь поставим цель написать свою функцию accumulate (назовём её, например, my_accumulate), которая тип возвращаемого значения брала бы не из типа начального значения, а из типа значения, который хранится в коллекции, на которую указывают итераторы. При этом нужно, чтобы, как и в функцию accumulate, в нашу функцию можно было передавать как итераторы из стандартной библиотеки языка C++, так и указатели на массив (как в нашем примере). Итераторы из стандартной библиотеки хранят тип значения в определении типа value_type. Решить задачу можно с помощью следующей метафункции get_value_type:

```
template<typename IT>
struct get_value_type {
  typedef typename IT::value_type type;
};
template<typename T>
struct get_value_type<T*>{
  typedef T type;
};
template<typename T>
struct get_value_type<const T*>{
  typedef T type;
};
```

Основное определение метафункции рассчитано на итераторы из стандартной библиотеки, а две частичных специализации рассчитаны на указатели (простой и const). С помощью этой метафункции нашу функцию my_accumulate можно написать следующим образом:

```
template<typename IT, typename T>
typename get_value_type<IT>::type my_accumulate(
  IT first, IT last, T init)
{
  typename get_value_type<IT>::type result = init;
  for (; first != last; ++first)
    result += *first;
  return result;
}
```

Кстати, необходимости писать метафункцию get_value_type не было. В стандартной библиотеке для работы с итераторами есть метафункция std::iterator_traits, возвращающая в переменной value_type как раз то, что нам нужно. С её использованием прототип нашей функции выглядел бы так:

```
template<typename IT, typename T>
typename std::iterator_traits<IT>::value_type my_accumulate(
```

```
  IT first, IT last, T init);
```

3. На стыке компиляции и исполнения

Пусть нужно вычислить степенную функцию x^N. Здесь x – вещественное число, а N – целое (положительное, ноль или отрицательное). Обычная реализация такой функции на языке C могла бы выглядеть примерно так:

```
double pow(double x, int N){
      if(N < 0) return 1 / pow(x, -N);
      else if(N == 0) return 1;
      else if(N % 2 == 0){
            double p = pow(x, N / 2);
            return p * p;
      } else return pow(x, N – 1) * x;
}
```

Пусть теперь число N известно во время компиляции. Так как x – вещественное число, то и результат – также вещественное число и на стадии компиляции вычислен быть не может. Кроме того, значение x становится известным только во время исполнения. Значит, написать обычную метафункцию (как для факториала) нельзя. Но оказывается, что на стадию компиляции можно вынести все условные операторы (а они зависят только от N, которое во время компиляции известно). На стадии исполнения останутся только умножения. Кроме того, при включенном оптимизаторе рекурсивные вызовы будут подставлены (inline), и вычисление степенной функции будет очень быстрым. Покажем, как это можно сделать.

В начале обратим внимание на одну из важных идиом языка C++ – т.н. SFINAE, Substitution Failure Is Not An Error (неудача при подстановке не является ошибкой). Эта идиома используется в метафункции enable_if из стандартной библиотеки языка C++, которая принимает два параметра шаблона – булевскую константу и тип. Если булевская константа равна true, то enable_if в типе type возвращает переданный во втором параметре тип, иначе ничего не возвращает. Покажем реализацию этой метафункции:

```
template<bool B, typename T = void> struct enable_if;
template<typename T>
struct enable_if<true, T>{ typedef T type; };
template<typename T>
struct enable_if<false, T>{};
```

В случае, когда булевская константа равна false, использование возвращаемого типа type для задания типа возвращаемого функцией значения не приводит к ошибке компиляции, вместо этого функция просто становится невидимой. Тело функции в этом случае проверяется только на синтаксическую правильность. Второй параметр метафункции enable_if (тип) может быть

опущен, в этом случае подразумевается значение void. Поясним сказанное на примере функции pow:

```
template<int N, typename T>
typename enable_if<(N < 0), T>::type
pow(T x){ return 1 / pow<-N>(x); }

template<int N, typename T>
typename enable_if<(N == 0), T>::type
pow(T x){ return 1; }

template<int N, typename T>
typename enable_if<(N > 0) && (N % 2 == 0), T>::type
pow(T x){ T p = pow<N / 2>(x); return p * p; }

template<int N, typename T>
typename enable_if<(N > 0) && (N % 2 == 1), T>::type
pow(T x){ return pow<N - 1>(x) * x; }

y = pow<3>(x); // Пример вызова
```

Благодаря SFINAE при любом значении N видимой будет ровно одна реализация функции pow. Обратите внимание также на то, что второй шаблонный параметр функции pow при вызове можно не указывать. Он определяется компилятором автоматически по типу параметра x.

4. Более сложные примеры

Простые числа

На этапе компиляции несложно проверить целое число на простоту. Вот текст метафункции is_prime, принимающей в качестве параметра целое число и возвращающей значение true или false в зависимости от его простоты:

```
template<int n, int i>
struct is_prime0 : std::conditional<(n % i) == 0,
  std::false_type, is_prime0<n, i - 1>
>::type {};

template<int n>
struct is_prime0<n, 1> : std::true_type {};

template<int n>
struct is_prime : is_prime0<n, n / 2>{};
```

Метафункция is_prime использует вспомогательную функцию is_prime0 с двумя параметрами, которая рекурсивно (а это, как мы уже говорили, единственный способ организации циклов в метапрограммировании) проверяет, делится ли число n на число i и меньшие него. Если число i достигает единицы, то срабатывает частичная специализация метафункции

is_prime0, которая говорит, что число n простое (возвращает значение true). Если же n делится на i, большее единицы, то оно не простое. Метафункция is_prime0, в свою очередь, использует метафункцию conditional из стандартной библиотеки, реализующую условный оператор (if). Вот как устроена эта метафункция:

```
template<bool b, typename T1, typename T2>
struct conditional {
  typedef T1 type;
};
template<typename T1, typename T2>
struct conditional<false, T1, T2> {
  typedef T2 type;
};
```

Метафункция conditional возвращает тип T1, если первый параметр равен true и T2, если первый параметр равен false.

На следующем шаге определим метафункцию prime_at, принимающую в качестве параметра порядковый номер простого числа (начиная с нуля) и возвращающую значение простого числа с указанным порядковым номером:

```
template<unsigned n>
struct find_prime : std::conditional<is_prime<n>::value,
  std::integral_constant<unsigned, n>,
  find_prime<n + 2>
>::type {};

template<unsigned n>
struct prime_at : find_prime<prime_at<n - 1>::value + 2>{};

template<>
struct prime_at<0> : std::integral_constant<unsigned, 2>{};

template<>
struct prime_at<1> : std::integral_constant<unsigned, 3>{};
```

Метафункция prime_at использует вспомогательную метафункцию find_prime, осуществляющую поиск простого числа, начиная с указанного в качестве параметра числа. Метафункция prime_at передаёт ей в качестве параметра значение предыдущего простого числа плюс два. Два первых простых числа (с порядковыми номерами ноль и один) задаются явно с помощью полных специализаций класса prime_at.

5. Цикл времени компиляции

Продемонстрируем применение цикла времени компиляции на примере вычисления интеграла по стандартному двумерному симплексу от бинома – выражения вида $(ax+by)^N$, где a и b – вещественные числа, известные только во время исполнения программы, а N – неотрицательная целочисленная константа,

известная во время компиляции. Разложив бином по известной формуле, получим:

$$\iint (ax+by)^N dS = \sum_{K=0}^{N} C_N^K a^K b^{N-K} \iint x^K y^{N-K} dS$$

Так как a и b – это вещественные числа, известные только во время исполнения, то этот интеграл не может быть вычислен во время компиляции. Но мы уже умеем считать во время компиляции биномиальные коэффициенты и интегралы от одночленов по стандартному двумерному симплексу. Кроме того, мы умеем вычислять степенную функцию оптимальным образом, вынося все условные операторы на стадию компиляции. Единственное, чего нам не хватает, это организации цикла. При этом на каждом витке цикла текущий индекс цикла (число K) должен быть известен во время компиляции, иначе мы не сможем вычислить ни биномиальные коэффициенты, ни интегралы от одночленов, ни степенную функцию. То есть обычный цикл времени исполнения нам не подходит. Организовать цикл времени компиляции можно разными способами, как средствами препроцессора, так и с помощью метапрограммирования. С помощью препроцессора это можно сделать с использованием макроса BOOST_PP_REPEAT из библиотеки boost. При этом тело цикла просто будет повторено нужное число раз с разными значениями индекса цикла. Но это не наш путь, так как мы изучаем метапрограммирование. Итак, вот тело функции (обычной, времени исполнения) meta_loop, исполняющей цикл времени компиляции указанное число раз:

```
template<unsigned N, unsigned I, class Closure>
typename std::enable_if<(I == N)>::type
meta_loop0(Closure &closure){}

template<unsigned N, unsigned I, class Closure>
typename std::enable_if<(I < N)>::type
meta_loop0(Closure &closure){
     closure.template apply<I>();
     meta_loop_<N, I + 1>(closure);
}
template<unsigned N, class Closure>
void meta_loop(Closure &closure){
     meta_loop0<N, 0>(closure);
}
```

Функция meta_loop шаблонная, и в качестве первого параметра шаблона она принимает общее число итераций, которые надо сделать. В качестве обычного параметра (времени исполнения) она принимает объект (т.н. «замыкание», closure) произвольного класса, в котором на каждой итерации нужно вызвать шаблонный метод apply без параметров, которому в качестве параметра шаблона передаётся индекс текущей итерации. Функция meta_loop использует вспомогательную шаблонную функцию meta_loop0, принимающую

два шаблонных параметра – общее число итераций и номер текущей итерации (начиная, как принято в языках C/C++, от нуля). Естественно, в качестве начального индекса цикла передаётся значение ноль. Функция meta_loop0 в том случае, если индекс цикла меньше общего числа итераций, вызывает в объекте closure метод apply, передав ему в качестве параметра шаблона индекс текущей итерации, а затем рекурсивно вызывает саму себя с индексом цикла на единицу больше (как мы уже говорили, в метапрограммировании рекурсия – единственно возможный способ организации циклов). Когда индекс цикла достигает общего числа итераций, срабатывает первая версия функции meta_loop0, которая ничего не делает и, таким образом, завершает рекурсию.

Теперь нам надо организовать суммирование. Мы также будем использовать пользовательский объект «замыкание» (closure), но теперь в нём (точнее, в его классе) должен быть, во-первых, определён тип value_type, указывающий тип вычисляемого значения, и, во-вторых, шаблонный метод value без параметров, принимающий в качестве параметра шаблона индекс текущей итерации и возвращающий значение члена ряда с указанным индексом. Вот тело соответствующей функции abstract_sum, вычисляющий значение ряда указанной длины:

```
template<class Closure>
struct abstract_sum_closure
{
  typedef typename Closure::value_type value_type;
  abstract_sum_closure(Closure &closure) :
    closure(closure), result(value_type()){}

  template<unsigned I>
  void apply(){
    result += closure.template value<I>();
  }
  Closure &closure;
  value_type result;
};

template<unsigned N, class Closure>
typename Closure::value_type abstract_sum(Closure &closure)
{
  abstract_sum_closure<Closure> my_closure(closure);
  meta_loop<N>(my_closure);
  return my_closure.result;
}
```

Функция abstract_sum реализована через функцию meta_loop, поэтому она вначале создаёт своё замыкание (класса abstract_sum_closure), удовлетворяющее требованиям (концепции) этой функции. А именно, в классе abstract_sum_closure есть шаблонный метод apply без параметров, принимающий в качестве параметра шаблона текущий индекс итерации.

Ну и, наконец, для реализации нашей задачи (интегрирования бинома) нужно реализовать замыкание для бинома и саму функцию binom:

```
template<unsigned N, typename T>
struct binom_closure
{
  typedef T value_type;

  binom_closure(value_type a, value_type b) : a(a), b(b){}

  template<unsigned K>
  value_type value() const {
    return C<N, K>::value * pow<K>(a) * pow<N-K>(b) /
      monomial_2d<K, N-K>::value;
  }
private:
  const value_type a, b;
};

template<unsigned N, typename T>
T binom(T a, T b)
{
  binom_closure<N, T> closure;
  return abstract_sum<N + 1>(closure);
}
```

Так как в разложении бинома степени N будет N+1 слагаемых, то в функцию abstract_sum в качестве параметра шаблона передаётся число N+1. Значение параметра шаблона T можно не передавать, оно определяется компилятором автоматически из типов параметров a и b. Пример обращения к функции binom:

```
const double a = 1.2, b = 3.4;
const unsigned N = 5;
const double bi = binom<N>(a, b);
```

6. Шаблоны выражений

Зададимся вопросом: что значит «задать формулу»? При этом мы хотим по одним формулам получать другие формулы, например, по формуле получать её производную, которая также является формулой, и от неё, в свою очередь, также можно получить производную в виде формулы. Ясно, что, например, функции для задания формул не годятся.

В языке C++ есть достаточно распространённый способ задания выражений, основанный на шаблонах, под названием «шаблоны выражений» (expression templates). С помощью этого механизма можно, оставаясь в рамках языка C++, определить свой язык в соответствие с некоторой грамматикой. При этом выражение на этом языке будет оставаться корректным выражением и на

языке C++. Такой язык носит название «предметно-ориентированный язык» (domain specific language, DSL). Определим для формул такой язык и реализуем его с помощью механизма шаблонов выражений. Вначале определим грамматику нашего языка выражений:

«выражение»::=*«сумма»* | -*«терм»* | +*«терм»*;
«сумма»::=*«слагаемое»* |
«слагаемое»+*«выражение»* |
«слагаемое»-*«выражение»*;
«слагаемое»::=*«терм»* |
*«терм»***«слагаемое»* |
«терм»/*«слагаемое»*;
«терм»::=*«константа»* | *«переменная»* |
«имя-функции»(*«выражение»*) | (*«выражение»*);

Данная грамматика является корректной, но сложноватой. Для наших нужд больше подойдёт более общая следующая грамматика:
«выражение»::=*«константа»* | *«переменная»* |
«выражение»+*«выражение»* | *«выражение»*-*«выражение»* |
*«выражение»***«выражение»* | *«выражение»*/*«выражение»* |
«имя-функции»(*«выражение»*) | (*«выражение»*) |
-*«выражение»*;

Разбор выражения будет производить компилятор, в частности, он, в соответствии со своими правилами, будет обрабатывать скобки, старшинство операций и левую ассоциативность арифметических операций. Следующим шагом определим класс выражения:

```
template<class E>
struct expression : math_object_base<E>{};
```

Этот класс не несёт в себе никакого функционала и является чисто маркерным. Он принимает в качестве параметра шаблона конечный класс и передаёт его дальше в свой базовый класс. Все наши дальнейшие классы будут наследоваться непосредственно от этого класса expression. Предположим, что все наши конечные классы выражений вычисляют своё значение от одной переменной (назовём её x), причём тип значения выражения (для простоты) – double. Простейшие выражения (в соответствие с нашей грамматикой) – это константа и переменная. Вот определения этих простейших выражений:

```
struct constant : expression<constant>
{
  constant(double value) : value(value){}

  double operator()(double x) const {
```

```
    return value; // Возвращаемое значение не зависит от значения
переменной.
  }
private:
  double value;
};

struct variable : expression<variable>
{
  double operator()(double x) const {
    return x;
  }
};
```

Следующее по сложности выражение – это отрицание выражения (класс negate). Класс отрицаемого выражения передаётся как параметр шаблона класса. Кроме того, мы для произвольного выражения определяем оператор отрицания (operator-). Вот исходный текст отрицания выражения:

```
template<class E>
struct negate : expression<negate<E> >
{
  negate(const expression<E> &expr) : expr(expr.self()){}

  double operator()(double x) const {
    return -expr(x);
  }
private:
  const E expr;
};
template<class E>
negate<E> operator-(const expression<E> &expr){
  return negate<E>(expr);
}
```

Выражению отрицания нужно сохранить выражение, которое оно отрицает, для последующих вычислений. При этом мы не имеем права сохранять ссылку на это выражение, т.к. это выражение может быть временным и для него может не быть переменной, ссылку на которую можно сохранять. Поэтому приходится делать копию выражения. В нашем случае это не страшно, т.к. все выражения занимают мало памяти.

Следующее по сложности – это арифметические операции с выражениями. Они уже оперируют с двумя выражениями, кроме того, появляется арифметическая операция (сложение, вычитание, умножение или деление). Для всех арифметических операций мы сделаем единый класс, а саму операцию передадим как параметр. Вот текст класса для бинарного выражения:

```
template<class E1, class OP, class E2>
struct binary_expression :
```

```
  expression<binary_expression<E1, OP, E2> >
{
  binary_expression(const expression<E1> &expr1,
    const OP &op, const expression<E2> &expr2) :
    expr1(expr1.self()), op(op), expr2(expr2.self()){}

  double operator()(double x) const {
    return op(expr1(x), expr2(x));
  }
private:
  const E1 expr1;
  const OP op;
  const E2 expr2;
};

#define DEFINE_BIN_OP(oper, OP) \
  template<class E1, class E2> \
  binary_expression<E1, std::OP<double>, E2> operator oper \
  (const expression<E1> &expr1, const expression<E2> &expr2){ \
    return binary_expression<E1, std::OP<double>, E2> \
      (expr1, std::OP<double>(), expr2); \
  } \
template<class E> \
 binary_expression<E, std::OP<double>, constant> operator oper \
  (const expression<E> &expr, double value){ \
    return binary_expression<E, std::OP<double>, constant> \
      (expr, std::OP<double>(), constant(value)); \
  } \
template<class E> \
 binary_expression<constant, std::OP<double>, E> operator oper \
  (double value, const expression<E> &expr){ \
    return binary_expression<constant, std::OP<double>, E> \
      (constant(value), std::OP<double>(), expr); \
  }

DEFINE_BIN_OP(+, plus)
DEFINE_BIN_OP(-, minus)
DEFINE_BIN_OP(*, multiplies)
DEFINE_BIN_OP(/, divides)
```

Макрос DEFINE_BIN_OP определяет арифметическую операцию для двух произвольных выражений, а также для выражения и константы и константы и выражения. Не сложно реализовать и стандартные математические функции для выражений.

Теперь мы можем строить выражения любой сложности, передавать их в качестве параметров, многократно производить по ним вычисления и т.д.

```
template<class E>
void f(const expression<E> &expr0)
```

```
{
  const E &expr = expr0.self();
  std::cout << expr(3) << std::endl;
  std::cout << expr(1.5) << std::endl;
}
int main(int argc, char* argv[])
{
  variable x;
  f(x * x + M_PI);
  auto expr2 = 5 * (-x * (x + 1));
  f(expr2);
  return 0;
}
```

7. Заключение

Язык программирования C++ может широко применяться для численного решения задач математической физики, причём эффективно использоваться могут многие отличительные свойства языка, включая объектно-ориентированное программирование и метапрограммирование шаблонов. Опыт программистов, пытавшихся использовать язык C++ для решения численных задач и получавших замедление по сравнению с программами на «чистом» C, говорит лишь о том, что использование языка C++, возможно, было не совсем правильным. Разработчики языка C++ одной из главных своих целей всегда ставили эффективность, и язык C++ – действительно эффективный язык. Причём это относится не только к скорости исполнения пользовательского кода, но и к стандартной библиотеке. Применение стандартных функций стало ещё удобнее в стандарте языка C++11, в котором появились лямбда-выражения, которые можно передавать в стандартные функции и которые позволяют писать обработчики «на месте», а не в отдельных классах или функциях за пределами текущей исполняемой функции.

Особое место в языке занимают шаблоны, появившиеся в языке далеко не сразу, а только через полтора десятилетия после появления самого языка. Шаблоны оказались удобными не только для реализации коллекций и алгоритмов, но и позволили использовать метапрограммирование шаблонов, что позволяет вынести часть вычислений на стадию компиляции, и, применительно к задачам математического моделирования, позволяет писать ещё более эффективные программы. Программа, написанная с активным использованием метапрограммирования шаблонов, может работать существенно быстрее аналогичной программы, написанной на языке C.

В данной работе на различных примерах, как относительно простых, как и достаточно сложных показано, как можно писать метафункции (функции времени компиляции), выносящие часть вычислений со стадии исполнения на стадию компиляции, что позволяет, во-первых, уменьшить объём вычислений на стадии исполнения и, тем самым, ускорить программу, и, во-вторых,

автоматизировать часть рутинной работы, такой, как раскрытие скобок и символьное дифференцирование, и уменьшить число возможных ошибок в программе, связанных с человеческой невнимательностью.

Работа выполнена при поддержке Российского научного фонда, проект № 17-71-30014.

Литература

1. Bjarne Stroustrup. The C++ Programming Language, Fourth Edition. Addison-Wesley, 2013. ISBN 978-0-321-56384-2, 1368 с.
2. Бьерн Страуструп. Программирование: Принципы и практика с использованием C++, второе издание. пер. с англ., Вильямс, 2016, 1328 с. ISBN 978-5-8459-1949-6, 978-0-321-99278-9.
3. Бьерн Страуструп. Дизайн и эволюция языка C++. ДМК Пресс, 2016, 446 с. ISBN 978-5-97060-419-9, 978-0-201-54330-8.
4. David Abrahams, Aleksey Gurtovoy. C++ Template Metaprogramming. Addison-Wesley, 2004, 400 с. ISBN 978-0-321-22725-6.
5. T. Veldhuizen, Expression Templates. C++ Report, Vol. 7 No. 5 June 1995, pp. 26-31.
6. NVidia CUDA. URL: http://www.nvidia.com/object/cuda_home_new.html
7. Краснов М.М. Операторная библиотека для решения трёхмерных сеточных задач математической физики с использованием графических плат с архитектурой CUDA. // Математическое моделирование, 2015, т. 27, с. № 3, 109-120.
8. Краснов М.М. Сеточно-операторный подход к программированию задач математической физики. Диссертация на соискание ученой степени кандидата физико-математических наук. М., 2017.
   URL: http://keldysh.ru/council/1/2017-krasnov/diss.pdf
9. Краснов М.М., Ладонкина М.Е. Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Применение шаблонного метапрограммирования языка C++. // Программирование, 2017 г., № 3, с. 56-68.
10. Краснов М.М. Применение символьного дифференцирования для решения ряда вычислительных задач // Препринты ИПМ им. М.В. Келдыша. 2017. № 4. 24 с. doi: 10.20948/prepr-2017-4
    URL: http://library.keldysh.ru/preprint.asp?id=2017-4