14

Числовые вычисления

Цель вычислений — понимание, а не цифры…
— Р.У. Хэмминг

…но для студента цифры часто оказываются лучшей дорогой к пониманию.
— А. Ральстон

14.1. Введение

C++ не был разработан, в первую очередь, для числовых вычислений. Однако числовые вычисления обычно выполняются в контексте другой деятельности, такой как доступ к базам данных, сетевое взаимодействие, управление приборами, графика, моделирование и финансовый анализ, поэтому C++ становится привлекательным средством для вычислений, являющихся частью более крупной системы. Кроме того, численные методы прошли долгий путь от простых циклов по векторам чисел с плавающей точкой. Там, где как часть вычислений необходимы более сложные структуры данных, становятся актуальными сильные стороны C++. В итоге сегодня C++ широко используется для научных, инженерных, финансовых и других вычислений с помощью сложных численных методов. Как следствие появились средства и методы, поддерживающие такие вычисления. В этой главе описываются части стандартной библиотеки, которые поддерживают числовые вычисления.

14.2. Математические функции

В заголовочном файле <cmath> находятся *стандартные математические функции*, такие как sqrt(), log() или sin() для аргументов типов float, double и long double.

	Стандартные математические функции
abs(x)	Абсолютное значение
ceil(x)	Наименьшее целое, не меньшее x
floor(x)	Наибольшее целое, не большее x
sqrt(x)	Квадратный корень; значение x должно быть неотрицательным
cos(x)	Косинус
sin(x)	Синус
tan(x)	Тангенс
acos(x)	Арккосинус; результат неотрицателен
asin(x)	Арксинус; возвращается результат, ближайший к нулю
atan(x)	Арктангенс
cosh(x)	Гиперболический косинус
sinh(x)	Гиперболический синус
tanh(x)	Гиперболический тангенс
exp(x)	Экспонента (*e* в степени x)
log(x)	Натуральный логарифм (по основанию *e*); значение x должно быть положительным
log10(x)	Десятичный логарифм

Версии функций для типа complex (§14.4) находятся в заголовочном файле <complex>. Для каждой функции возвращаемый тип совпадает с типом аргумента.

Об ошибках сообщается путем установки errno из заголовочного файла <cerrno> в EDOM для ошибки области определения и ERANGE — для ошибки области значений. Например:

```
void f()
{
    errno = 0;      // Сброс старого состояния ошибки
    sqrt(-1);
    if (errno==EDOM)
        cerr << "sqrt() не определена для отрицательного аргумента";
    errno = 0;      // Сброс старого состояния ошибки
    pow(numeric_limits<double>::max(),2);
```

```
    if (errno == ERANGE)
        cerr << "Результат pow() слишком велик для double";
}
```

Еще несколько математических функций находятся в заголовочном файле <cstdlib>, а так называемые *специальные математические функции*, такие как beta(), rieman_zeta() и sph_bessel(), также находятся в заголовочном файле <cmath>.

14.3. Числовые алгоритмы

В заголовочном файле <numeric> находится небольшое множество обобщенных числовых алгоритмов, таких как accumulate().

Числовые алгоритмы	
x=accumulate(b,e,i)	x — сумма i и элементов последовательности [b:e)
x=accumulate(b,e,i,f)	accumulate с использованием f вместо +
x=inner_product(b,e, b2,i)	x — скалярное произведение [b:e) и [b2:b2+(e-b)), т.е. сумма i и (*p1)*(*p2) для каждого p1 из [b:e) и соответствующего p2 из [b2:b2+(e-b))
x=inner_product(b,e, b2,i,f,f2)	inner_product с использованием f и f2 вместо + и *
p=partial_sum(b,e,out)	i-й элемент в [out:p) является суммой элементов [b:b+i]
p=partial_sum(b,e,out,f)	partial_sum с использованием f вместо +
p=adjacent_difference (b,e,out)	i-й элемент в [out:p) равен *(b+i)-*(b+i-1) для i>0; если e-b>0, то *out равно *b
p=adjacent_difference (b,e,out,f)	adjacent_difference с использованием f вместо -
iota(b,e,v)	Каждому элементу в [b:e) присваивается значение ++v; таким образом, последовательность становится равной v+1, v+2, ...
x=gcd(n,m)	x — наибольший общий делитель целых чисел n и m
x=lcm(n,m)	x — наименьшее общее кратное целых чисел n и m

Эти алгоритмы обобщают распространенные операции, такие как вычисление суммы, позволяя применять их ко всем видам последовательностей. Они также делают операцию, применяемую к элементам этих последовательностей, параметром. Для каждого алгоритма общая версия дополняется версией, применяющей наиболее распространенный оператор для этого алгоритма. Например:

```
list<double> lst {1, 2, 3, 4, 5, 9999.99999};
auto s = accumulate(lst.begin(),lst.end(),0.0); // Сумма: 10014.9999
```

Описанные алгоритмы работают для любой последовательности стандартной библиотеки и могут иметь операции, предоставляемые в качестве аргументов (§14.3).

14.3.1. Параллельные алгоритмы

В заголовочном файле <numeric> числовые алгоритмы имеют немного различающиеся параллельные версии (§12.9).

Параллельные числовые алгоритмы	
`x=reduce(b,e,v)`	`x=accumulate(b,e,v)`, за исключением порядка вычислений
`x=reduce(b,e)`	`x=reduce(b,e,V{})`, где `V` — тип значения `b`
`x=reduce(pol,b,e,v)`	`x=reduce(b,e,v)` со стратегией выполнения `pol`
`x=reduce(pol,b,e)`	`x=reduce(pol,b,e,V{})`, где `V` — тип значения `b`
`p=exclusive_scan (pol,b,e,out)`	`p=partial_sum(b,e,out)` в соответствии со стратегией `pol`, исключая `i`-й элемент из `i`-й суммы
`p=inclusive_scan (pol,b,e,out)`	`p=partial_sum(b,e,out)` со стратегией выполнения `pol` и включением `i`-го элемента в `i`-ю сумму
`p=transform_reduce (pol,b,e,f,v)`	`f(x)` для каждого `x` из [`b`:`e`), затем `reduce`
`p=transform_exclusive_ scan(pol,b,e,out,f,v)`	`f(x)` для каждого `x` из [`b`:`e`), затем `exclusive_scan`
`p=transform_inclusive_ scan(pol,b,e,out,f,v)`	`f(x)` для каждого `x` из [`b`:`e`), затем `inclusive_scan`

Для простоты я не показал версии алгоритмов, которые принимают в качестве аргумента функтор, а не просто используют + и =. За исключением `reduce()`, я также не показал версии со стратегией выполнения по умолчанию (последовательное выполнение) и значением по умолчанию.

Так же, как и для параллельных алгоритмов в заголовочном файле <algorithm> (§12.9), мы можем определить стратегию выполнения:

```
vector<double> v {1, 2, 3, 4, 5, 9999.99999};
// Вычисление суммы с использованием double в качестве накопителя:
auto s = reduce(v.begin(),v.end());

vector<double> large;
// ... Заполнение large большим количеством значений ...
```

```
// Вычисление суммы с использованием доступного параллелизма:
auto s2 = reduce(par_unseq,large.begin(),large.end());
```

Параллельные алгоритмы (например, `reduce()`) отличаются от последовательных (например, `accumulate()`) тем, что допускают выполнение операций над элементами в неопределенном порядке.

14.4. Комплексные числа

Стандартная библиотека поддерживает семейство типов комплексных чисел по аналогии с классом `complex`, описанным в §4.2.1. Для поддержки комплексных чисел, в которых скаляры являются числами одинарной точности с плавающей запятой (`float`), двойной точности с плавающей запятой (`double`) и другими, тип `complex` стандартной библиотеки является шаблоном:

```
template<typename Scalar>
class complex
{
public:
    // Аргументы функции по умолчанию; см. §3.6.1:
    complex(const Scalar& re ={}, const Scalar& im ={});
    // ...
};
```

Для комплексных чисел поддерживаются обычные арифметические операции и наиболее распространенные математические функции. Например:

```
void f(complex<float> fl, complex<double> db)
{
    complex<long double> ld {fl+sqrt(db)};
    db += fl*3;
    fl = pow(1/fl,2);
    // ...
}
```

Функции `sqrt()` и `pow()` (возведение в степень) находятся среди обычных математических функций, определенных в заголовочном файле стандартной библиотеки <complex> (§14.2).

14.5. Случайные числа

Случайные числа полезны во многих контекстах, таких как тестирование, игры, моделирование и безопасность. Разнообразие областей применения отражается в широком выборе генераторов случайных чисел, предоставляемых стандартной библиотекой в заголовочном файле <random>. Генератор случайных чисел состоит из двух частей.

[1] Собственно *генератора* (engine), производящего последовательность случайных или псевдослучайных чисел.

[2] *Распределения* (distribution), которое отображает полученные значения в математическое распределение в некотором диапазоне.

Примерами распределений являются `uniform_int_distribution` (все целые числа получаются с одинаковой вероятностью), `normal_distribution` (нормальное (гауссово) распределение) и `exponential_distribution` (экспоненциальное распределение); каждое из них применяется для определенного диапазона. Например:

```
using my_engine = default_random_engine;            // Генератор
using my_distribution = uniform_int_distribution<>; // Распределение

my_engine re {};                  // Генератор по умолчанию
my_distribution one_to_six {1,6}; // Распределение в диапазоне 1..6

auto dice = [](){ return one_to_six(re); } // Создание ГСЧ
int x = dice();                            // Бросание кости: x∈[1:6]
```

Благодаря своему бескомпромиссному вниманию к общности и производительности случайные числа стандартной библиотеки были охарактеризованы одним экспертом как “то, чем хочет быть каждая библиотека случайных чисел, когда вырастет”. Однако эту часть стандартной библиотеки вряд ли можно считать дружелюбной по отношению к новичкам. Использование инструкций `using` и лямбда-выражений делает код немного более понятным.

Для новичков (с любыми базовыми знаниями) серьезным препятствием может стать очень общий интерфейс библиотеки случайных чисел. Однако для начала работы зачастую достаточно простого генератора случайных чисел с равномерным распределением. Например:

```
Rand_int rnd {1,10}; // Генератор случайных чисел в диапазоне [1:10]
int x = rnd();       // x — случайное число в диапазоне [1:10]
```

Итак, как мы можем получить такой генератор? Нам нужно что-то наподобие рассмотренного выше `dice()`, что объединяет генератор с распределением внутри класса `Rand_int`:

```
class Rand_int
{
public:
    Rand_int(int low, int high) :dist{low,high} { }
    int operator()() { return dist(re); } // Генерация int
    void seed(int s) { re.seed(s); }      // Инициализация ГСЧ
private:
    default_random_engine re;
    uniform_int_distribution<> dist;
};
```

Это определение по-прежнему находится на “уровне эксперта”, но применение `Rand_int()` возможно уже на первой неделе курса C++ для новичков. Например:

```
int main()
{
    constexpr int max = 9;
    Rand_int rnd {0,max};          // ГСЧ с равномерным распределением

    vector<int> histogram(max+1); // Вектор подходящего размера
    for (int i=0; i!=200; ++i)
        ++histogram[rnd()];        // Заполнение гистограммы частотами

    for (int i = 0; i!=histogram.size(); ++i)    // Вывод гистограммы
    {
        cout << i << '\t';
        for (int j=0; j!=histogram[i]; ++j) cout << '*';
        cout << endl;
    }
}
```

Результатом оказывается (обнадеживающе скучное) равномерное распределение (с разумными статистическими отклонениями):

```
0 ********************
1 ****************
2 ********************
3 ********************
4 ****************
5 ************************
6 **************************
7 ***********
8 ***********************
9 *************************
```

Стандартной графической библиотеки в C++ не существует, поэтому я использую “ASCII-графику”. Очевидно, что существует множество программ с открытым исходным кодом, коммерческой графики и графических библиотек для C++, но я в этой книге ограничусь стандартными средствами ISO.

14.6. Векторная арифметика

`vector`, описанный в §11.2, был разработан в качестве общего механизма для хранения значений, который был бы гибким и вписывался в архитектуру контейнеров, итераторов и алгоритмов. Однако он не поддерживает математические векторные операции. Добавление таких операций к `vector` было бы простым, но его универсальность и гибкость исключают возможности оптимизации, которые часто считаются необходимыми для серьезных числен-

ных методов. Поэтому стандартная библиотека предоставляет (в заголовочном файле <valarray>) векторный шаблон valarray, который является менее общим и более поддающимся оптимизации для численных вычислений:

```
template<typename T>
class valarray
{
    // ...
};
```

Для valarray поддерживаются обычные арифметические операции и наиболее распространенные математические функции. Например:

```
void f(valarray<double>& a1, valarray<double>& a2)
{
    // Числовые операторы *, +, / и = для массивов:
    valarray<double> a = a1*3.14+a2/a1;
    a2 += a1*3.14;
    a = abs(a);
    double d = a2[7];
    // ...
}
```

В дополнение к арифметическим операциям valarray предлагает быстрый доступ, облегчающий реализацию многомерных вычислений.

14.7. Границы числовых значений

В заголовочном файле <limit> стандартная библиотека предоставляет классы, которые описывают свойства встроенных типов — такие, как максимальный показатель степени для float или количество байтов в int. Например, мы можем проверить во время компиляции, знаковым ли типом является char:

```
static_assert(numeric_limits<char>::is_signed,"Символы беззнаковые!");
static_assert(100000<numeric_limits<int>::max(),"Слишком малый int!");
```

Обратите внимание, что вторая проверка работает только потому, что numeric_limits<int>::max() является constexpr-функцией (§1.6).

14.8. Советы

[1] Проблемы, связанные с числовыми вычислениями, зачастую довольно тонкие. Если вы не уверены на 100% в математических аспектах числовых вычислений, обратитесь за советом к специалисту или проведите эксперименты (или сделайте и то, и другое); §14.1.

[2] Не пытайтесь работать с серьезными числовыми вычислениями, ограничиваясь возможностями “голого” языка — используйте библиотеки; §14.1.

[3] Перед тем как писать цикл для вычисления значения из последовательности, рассмотрите возможность применения `accumulate()`, `inner_product()`, `partial_sum()` и `adjacent_difference()`; §14.3.

[4] Используйте `std::complex` для комплексной арифметики; §14.4.

[5] Для получения генератора случайных чисел свяжите генератор с распределением; §14.5.

[6] Следите, чтобы ваши случайные числа были достаточно случайными; §14.5.

[7] Не используйте `rand()` из стандартной библиотеки C; для серьезного применения этот генератор недостаточно случаен; §14.5.

[8] Используйте `valarray` для численных вычислений, когда эффективность времени выполнения важнее гибкости по отношению к операциям и типам элементов; §14.6.

[9] Свойства числовых типов доступны посредством `numeric_limits`; §14.7.

[10] Используйте `numeric_limits` для проверки пригодности числовых типов для предполагаемого применения; §14.7.