

Содержание

Предисловие	13
Благодарности	15
Ждем ваших отзывов!	16
ГЛАВА 1. Основы	17
1.1. Введение	17
1.2. Программы	18
1.2.1. Hello, World!	19
1.3. Функции	20
1.4. Типы, переменные и арифметика	22
1.4.1. Арифметика	24
1.4.2. Инициализация	25
1.5. Область видимости и время жизни	26
1.6. Константы	28
1.7. Указатели, массивы и ссылки	29
1.7.1. Нулевой указатель	31
1.8. Проверки	33
1.9. Отображение на аппаратные средства	35
1.9.1. Присваивание	36
1.9.2. Инициализация	37
1.10. Советы	38
ГЛАВА 2. Пользовательские типы	41
2.1. Введение	41
2.2. Структуры	42
2.3. Классы	43
2.4. Объединения	46
2.5. Перечисления	47
2.6. Советы	49
ГЛАВА 3. Модульность	51
3.1. Введение	51
3.2. Раздельная компиляция	53
3.3. Модули (C++20)	55
3.4. Пространства имен	57

3.5. Обработка ошибок	58
3.5.1. Исключения	59
3.5.2. Инварианты	61
3.5.3. Альтернативные варианты обработки ошибок	63
3.5.4. Контракты	65
3.5.5. Статические проверки	66
3.6. Аргументы и возвращаемые значения функций	67
3.6.1. Передача аргументов	68
3.6.2. Возврат значения	69
3.6.3. Структурное связывание	71
3.7. Советы	72
ГЛАВА 4. Классы	75
4.1. Введение	75
4.2. Конкретные типы	77
4.2.1. Арифметический тип	78
4.2.2. Контейнер	80
4.2.3. Инициализация контейнеров	82
4.3. Абстрактные типы	84
4.4. Виртуальные функции	88
4.5. Иерархии классов	89
4.5.1. Преимущества иерархий	92
4.5.2. Навигация по иерархии	94
4.5.3. Избежание утечки ресурсов	95
4.6. Советы	96
ГЛАВА 5. Основные операции	99
5.1. Введение	99
5.1.1. Основные операции	100
5.1.2. Преобразования типов	102
5.1.3. Инициализаторы членов	103
5.2. Копирование и перемещение	103
5.2.1. Копирование контейнеров	104
5.2.2. Перемещение контейнеров	106
5.3. Управление ресурсами	108
5.4. Обычные операции	110
5.4.1. Сравнения	111
5.4.2. Операции с контейнерами	111
5.4.3. Операции ввода-вывода	112

8	Содержание	
	5.4.4. Пользовательские литералы	112
	5.4.5. swap ()	114
	5.4.6. hash<>	114
	5.5. Советы	114
	ГЛАВА 6. Шаблоны	117
	6.1. Введение	117
	6.2. Параметризованные типы	118
	6.2.1. Ограниченные аргументы шаблона (C++20)	120
	6.2.2. Аргументы-значения шаблонов	121
	6.2.3. Вывод аргументов шаблона	121
	6.3. Параметризованные операции	123
	6.3.1. Шаблоны функций	124
	6.3.2. Функциональные объекты	125
	6.3.3. Лямбда-выражения	126
	6.4. Шаблонные механизмы	130
	6.4.1. Шаблоны переменных	130
	6.4.2. Псевдонимы	131
	6.4.3. if времени компиляции	132
	6.5. Советы	133
	ГЛАВА 7. Концепты и обобщенное программирование	135
	7.1. Введение	135
	7.2. Концепты (C++20)	136
	7.2.1. Применение концептов	137
	7.2.2. Перегрузка на основе концептов	138
	7.2.3. Корректный код	140
	7.2.4. Определение концептов	141
	7.3. Обобщенное программирование	142
	7.3.1. Использование концептов	143
	7.3.2. Абстракции с использованием шаблонов	143
	7.4. Вариативные шаблоны	145
	7.4.1. Выражения свертки	147
	7.4.2. Передача аргументов	148
	7.5. Модель компиляции шаблонов	149
	7.6. Советы	150
	ГЛАВА 8. Обзор библиотеки	153
	8.1. Введение	153
	8.2. Компоненты стандартной библиотеки	154

8.3. Заголовочные файлы и пространство имен стандартной библиотеки	155
8.4. Советы	157
ГЛАВА 9. Строки и регулярные выражения	159
9.1. Введение	159
9.2. Строки	160
9.2.1. Реализация <code>string</code>	162
9.3. Представления строк	163
9.4. Регулярные выражения	165
9.4.1. Поиск	166
9.4.2. Запись регулярных выражений	167
9.4.3. Итераторы	172
9.5. Советы	173
ГЛАВА 10. Ввод и вывод	175
10.1. Введение	175
10.2. Вывод	176
10.3. Ввод	177
10.4. Состояние ввода-вывода	179
10.5. Ввод-вывод пользовательских типов	180
10.6. Форматирование	182
10.7. Файловые потоки	183
10.8. Строковые потоки	184
10.9. Ввод-вывод в стиле C	185
10.10. Файловая система	185
10.11. Советы	190
ГЛАВА 11. Контейнеры	193
11.1. Введение	193
11.2. <code>vector</code>	194
11.2.1. Элементы	197
11.2.2. Проверка выхода за границы диапазона	197
11.3. <code>list</code>	199
11.4. <code>map</code>	202
11.5. <code>unordered_map</code>	203
11.6. Обзор контейнеров	205
11.7. Советы	207

ГЛАВА 12. Алгоритмы	211
12.1. Введение	211
12.2. Применение итераторов	213
12.3. Типы итераторов	215
12.4. Итераторы потоков	216
12.5. Предикаты	219
12.6. Обзор алгоритмов	220
12.7. Концепты (C++20)	221
12.8. Алгоритмы над контейнерами	226
12.9. Параллельные алгоритмы	227
12.10. Советы	228
ГЛАВА 13. Утилиты	229
13.1. Введение	230
13.2. Управление ресурсами	230
13.2.1. <code>unique_ptr</code> и <code>shared_ptr</code>	231
13.2.2. <code>move()</code> и <code>forward()</code>	234
13.3. Проверка выхода за границы диапазона: <code>gsl::span</code>	236
13.4. Специализированные контейнеры	238
13.4.1. <code>array</code>	240
13.4.2. <code>bitset</code>	242
13.4.3. <code>pair</code> и <code>tuple</code>	243
13.5. Альтернативы	245
13.5.1. <code>variant</code>	245
13.5.2. <code>optional</code>	247
13.5.3. <code>any</code>	248
13.6. Аллокаторы	249
13.7. Время	250
13.8. Адаптация функций	251
13.8.1. Лямбда-выражения в качестве адаптеров	252
13.8.2. <code>mem_fn()</code>	252
13.8.3. <code>function</code>	252
13.9. Функции типов	253
13.9.1. <code>iterator_traits</code>	253
13.9.2. Предикаты типов	256
13.9.3. <code>enable_if</code>	257
13.10. Советы	258

ГЛАВА 14. Числовые вычисления.....	261
14.1. Введение	261
14.2. Математические функции	262
14.3. Числовые алгоритмы	263
14.3.1. Параллельные алгоритмы	264
14.4. Комплексные числа	265
14.5. Случайные числа	265
14.6. Векторная арифметика	267
14.7. Границы числовых значений	268
14.8. Советы	268
ГЛАВА 15. Параллельные вычисления.....	271
15.1. Введение	271
15.2. Задания и потоки	272
15.3. Передача аргументов	274
15.4. Возврат результатов	275
15.5. Совместное использование данных	276
15.6. Ожидание событий	278
15.7. Обмен информацией с заданиями	280
15.7.1. <code>future</code> и <code>promise</code>	280
15.7.2. <code>packaged_task</code>	282
15.7.3. <code>async()</code>	283
15.8. Советы	284
ГЛАВА 16. История и совместимость.....	287
16.1. История	287
16.1.1. Временная диаграмма развития C++	288
16.1.2. Ранние годы	290
16.1.3. Стандарты ISO C++	294
16.1.4. Стандарты и стиль	297
16.1.5. Использование C++	298
16.2. Эволюция возможностей C++	298
16.2.1. Возможности языка C++11	298
16.2.2. Возможности языка C++14	300
16.2.3. Возможности языка C++17	301
16.2.4. Компоненты стандартной библиотеки C++11	301
16.2.5. Компоненты стандартной библиотеки C++14	302
16.2.6. Компоненты стандартной библиотеки C++17	303

12 Содержание

16.2.7. Удаленные и nereкомендуемые возможности	303
16.3. Совместимость C/C++	304
16.3.1. C и C++ — братья	304
16.3.2. Проблемы совместимости	306
16.4. Список литературы	309
16.5. Советы	313
Предметный указатель	315

Числовые вычисления

Цель вычислений — понимание, а не цифры...

— Р.У. Хэмминг

*...но для студента цифры часто оказываются
лучшей дорогой к пониманию.*

— А. Ральстон

- ◆ Введение
- ◆ Математические функции
- ◆ Числовые алгоритмы
 - Параллельные алгоритмы
- ◆ Комплексные числа
- ◆ Случайные числа
- ◆ Векторная арифметика
- ◆ Границы числовых значений
- ◆ Советы

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	
<code>abs(x)</code>	Абсолютное значение
<code>ceil(x)</code>	Наименьшее целое, не меньше x
<code>floor(x)</code>	Наибольшее целое, не больше x
<code>sqrt(x)</code>	Квадратный корень; значение x должно быть неотрицательным
<code>cos(x)</code>	Косинус
<code>sin(x)</code>	Синус
<code>tan(x)</code>	Тангенс
<code>acos(x)</code>	Арккосинус; результат неотрицателен
<code>asin(x)</code>	Арсинус; возвращается результат, ближайший к нулю
<code>atan(x)</code>	Арктангенс
<code>cosh(x)</code>	Гиперболический косинус
<code>sinh(x)</code>	Гиперболический синус
<code>tanh(x)</code>	Гиперболический тангенс
<code>exp(x)</code>	Экспонента (e в степени x)
<code>log(x)</code>	Натуральный логарифм (по основанию e); значение x должно быть положительным
<code>log10(x)</code>	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	
<code>x=accumulate(b, e, i)</code>	<code>x</code> — сумма <code>i</code> и элементов последовательности <code>[b:e)</code>
<code>x=accumulate(b, e, i, f)</code>	<code>accumulate</code> с использованием <code>f</code> вместо <code>+</code>
<code>x=inner_product(b, e, b2, i)</code>	<code>x</code> — скалярное произведение <code>[b:e)</code> и <code>[b2:b2+(e-b))</code> , т.е. сумма <code>i</code> и <code>(*p1) * (*p2)</code> для каждого <code>p1</code> из <code>[b:e)</code> и соответствующего <code>p2</code> из <code>[b2:b2+(e-b))</code>
<code>x=inner_product(b, e, b2, i, f, f2)</code>	<code>inner_product</code> с использованием <code>f</code> и <code>f2</code> вместо <code>+</code> и <code>*</code>
<code>p=partial_sum(b, e, out)</code>	<code>i</code> -й элемент в <code>[out:p)</code> является суммой элементов <code>[b:b+i)</code>
<code>p=partial_sum(b, e, out, f)</code>	<code>partial_sum</code> с использованием <code>f</code> вместо <code>+</code>
<code>p=adjacent_difference(b, e, out)</code>	<code>i</code> -й элемент в <code>[out:p)</code> равен <code>*(b+i) - *(b+i-1)</code> для <code>i > 0</code> ; если <code>e-b > 0</code> , то <code>*out</code> равно <code>*b</code>
<code>p=adjacent_difference(b, e, out, f)</code>	<code>adjacent_difference</code> с использованием <code>f</code> вместо <code>-</code>
<code>iota(b, e, v)</code>	Каждому элементу в <code>[b:e)</code> присваивается значение <code>++v</code> ; таким образом, последовательность становится равной <code>v+1, v+2, ...</code>
<code>x=gcd(n, m)</code>	<code>x</code> — наибольший общий делитель целых чисел <code>n</code> и <code>m</code>
<code>x=lcm(n, m)</code>	<code>x</code> — наименьшее общее кратное целых чисел <code>n</code> и <code>m</code>

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

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:  $x \in [1:6]$ 
```

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 встроженных 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.