ГЛАВА 3

Библиотечные типы данных

В этой главе…

3.1. Пространства имен и объявления `using` 102
3.2. Библиотечный тип `string` 104
3.3. Библиотечный тип `vector` 114
3.4. Знакомство с итераторами 120
3.5. Библиотечный тип `bitset` 125
Резюме 130
Термины 130

Кроме базовых типов, описанных в главе 2, “Переменные и базовые типы”, библиотека языка C++ предоставляет богатое разнообразие абстрактных типов данных. Из наиболее важных библиотечных типов следует упомянуть классы `string` и `vector`, которые определяют символьные строки переменного размера и коллекции соответственно. С классами `string` и `vector` связан сопутствующий тип, известный как *итератор* (iterator). Он используется для доступа к символам строк и элементам векторов. Эти библиотечные типы являются абстракциями более простых базовых типов, массивов и указателей, которые являются частью языка.

Еще один библиотечный тип, `bitset`, предоставляет абстрактный способ управления коллекцией битов. Этот класс предоставляет более удобный способ работы с битами данных, чем обеспечивают встроенные побитовые операторы для значений целочисленных типов.

В этой главе рассматриваются такие библиотечные типы, как `vector`, `string` и `bitset`. Следующая глава посвящена массивам и указателям, а глава 5, “Выражения”, встроенным побитовым операторам.

Все типы, описанные в главе 2, “Переменные и базовые типы”, относились к низкоуровневым типам данных, они представляют такие абстракции, как числа или символы, и определены в терминах их машинного представления.

Кроме типов, определенных в самом языке C++, стандартная библиотека предоставляет набор дополнительных высокоуровневых *абстрактных типов данных* (abstract data type). Высокоуровневыми эти библиотечные типы называют потому, что они отражают более сложные концепции, а абстрактными — потому что при их использовании можно не заботиться о том, как эти типы представлены. Достаточно лишь знать, какие операции они выполняют.

Двумя наиболее важными библиотечными типами являются `string` и `vector`. Класс `string` позволяет использовать символьные строки переменной длины, а класс `vector` — хранить наборы объектов одинакового типа. Эти классы очень важны, поскольку они обладают многими преимуществами перед базовыми типами, определенными в языке. В главе 4, "Массивы и указатели", описаны аналогичные базовые конструкции языка, которые хоть и подобны, но менее гибки и подвержены ошибкам, чем библиотечные типы `string` и `vector`.

Еще одним библиотечным типом, обеспечивающим удобство и эффективность работы, является класс `bitset`. Этот класс позволяет работать со значениями как с коллекцией битов, обеспечивая более простые способы выполнения побитовых операций, чем позволяют встроенные побитовые операторы, рассматриваемые в разделе 5.3 на стр. 179.

Однако прежде чем переходить к изучению библиотечных типов, имеет смысл рассмотреть механизм, который упрощает доступ к именам, определенным в библиотеке.

3.1. Пространства имен и объявления `using`

До сих пор имена из стандартной библиотеки упоминались в программах явно, т.е. перед каждым из них было указано имя пространства имен `std`. Например, при чтении со стандартного устройств ввода применялась форма записи `std::cin`. Здесь использован оператор области видимости `::` (раздел 1.2.2 стр. 30). Он означает, что имя, указанное в правом операнде оператора, следует искать в области видимости указанной в левом операнде. Таким образом, код `std::cin` означает, что используемое имя `cin` определено в пространстве имен `std`. При частом использовании библиотечных имен такая форма записи может оказаться чересчур громоздкой.

К счастью, существуют и более простые способы применения членов пространств имен. В этом разделе описан самый надежный механизм: *объявление* `using` (`using` declaration). Другие способы, позволяющие упростить использование имен из других пространств, рассматриваются в разделе 17.2 (стр. 744).

Объявление `using` позволяет получать доступ к именам из другого пространства имен без указания префикса *`имя_пространства_имен`*`::`. Объявление `using` имеет следующий формат.

```
using пространствоимен::имя;
```

После того как объявление `using` было сделано один раз, к указанному в нем имени можно обращаться без указания пространства имен.

```
#include <string>
#include <iostream>
// объявление using свидетельствует о намерении использовать
// указанные имена именно из пространства имен std
using std::cin;
using std::string;
int main()
{
    string s;  // ok: теперь string - это синоним std::string
    cin >> s;  // ok: теперь cin - это синоним std::cin
    cout << s; // ошибка: объявления using нет; здесь нужно указать
               // полное имя
```

```
    std::cout << s; // ok: явно указано применение cout из
                    // пространства имен std
}
```

Использование имени из пространства имен без уточнения версии в объявлении `using` является ошибкой, хотя некоторые компиляторы могут оказаться не в состоянии обнаружить ее.

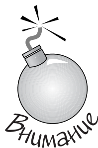
Для каждого имени необходимо индивидуальное объявление using

Объявление `using` применяется только к одному элементу пространства имен. Это позволяет жестко задавать имена, используемые в каждой программе. Поэтому если необходимо использовать несколько имен из пространства имен `std` (или любого другого), объявление `using` понадобится для каждого из них. Например, программу суммирования (стр. 28) можно переписать следующим образом.

```
#include <iostream>
// объявления using для имен из стандартной библиотеки
using std::cin;
using std::cout;
using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1
         << " and " << v2
         << " is "  << v1 + v2 << endl;
    return 0;
}
```

Объявления `using` для имен `cin`, `cout` и `endl` означают, что их можно теперь использовать без префикса `std::`, что делает код проще и читабельней.

Начиная с этого момента подразумевается, что код примеров снабжен объявлениями `using` для имен из стандартной библиотеки. Таким образом, в тексте примеров кода будет применяться форма записи `cin`, а не `std::cin`. Чтобы сократить код примеров, необходимые при компиляции объявления `using` здесь не приводятся. Аналогично, в примерах кода больше не будут отображаться необходимые при компиляции директивы `#include`. В табл. А.1 (стр. 841) приложения А, “Библиотека”, перечислены библиотечные имена и соответствующие им заголовки стандартной библиотеки, которые были использованы в этой книге.



Не забывайте, что перед компиляцией примеров этой книги в их код следует добавлять соответствующие директивы `#include` и объявления `using`.

Определения классов, использующих типы из стандартной библиотеки

Однако внутри файлов заголовка *всегда* необходимо использовать только полностью определенные имена библиотечных типов. Дело в том, что препроцессор просто копирует содержимое заголовка в текст программы. Таким образом, при подключении

файла заголовка, его содержимое без изменений войдет в состав файла программы. Следовательно, размещение объявления using внутри файла заголовка эквивалентно размещению того же объявления using в каждой использующей его программе независимо от того, *нужно это или нет*.

Общепринятой практикой является определение в заголовках только того, что абсолютно необходимо.

Упражнения раздела 3.1

Упражнение 3.1. Перепишите программу раздела 2.3 (стр. 65), вычисляющую результат возведения заданного числа в указанную степень так, чтобы используя соответствующие объявления using обеспечить доступ к библиотечным именам без префикса std::.

3.2. Библиотечный тип string

Тип string предназначен для символьных строк переменной длины. Библиотека отвечает за операции управления памятью, необходимые для хранения наборов символов, а также за разнообразные вспомогательные операции.

Подобно любым библиотечным типам, использующие класс string программы должны сначала подключить соответствующий заголовок. Такие программы будут более компактными, если в их начало включить также соответствующее объявление using.

```
#include <string>
using std::string;
```

3.2.1. Определение и инициализация строк

Таблица 3.1. Способы инициализации объекта класса string

`string s1;`	Стандартный конструктор; s1 — пустая строка
`string s2(s1);`	Инициализация строки s2 как копии строки s1
`string s3("value");`	Инициализация строки s3 как копии строкового литерала
`string s4(n, 'c');`	Инициализация строки s4 последовательностью из n символов c

Внимание! Библиотечный тип string и строковые литералы

По исторически сложившимся причинам, а также в целях совместимости с языком С, символьные строковые литералы имеют тип, *отличный* от типа string стандартной библиотеки. Этот факт может ввести в заблуждение, поэтому его следует учитывать при использовании строкового литерала и класса string.

Библиотечный тип string обладает несколькими конструкторами (раздел 2.3.3 стр. 71). Конструктор — это специальная функция-член, которая позволяет инициализировать объекты данного класса. Список наиболее часто используемых конст-

рукторов класса string приведен в табл. 3.1. Когда объект не инициализируется явно, по умолчанию используется стандартный конструктор (раздел 2.3.4 стр. 74).

Упражнения раздела 3.2.1

Упражнение 3.2. Что такое стандартный конструктор?

Упражнение 3.3. Назовите три способа инициализации объекта класса string.

Упражнение 3.4. Каковы значения переменных s и s2?

```
string s;
int main() {
    string s2;
}
```

3.2.2. Чтение и запись строк

Как было продемонстрировано в главе 1, "Первые шаги", для чтения и записи значений встроенных типов данных (таких как int, double и т.д.) используется библиотека iostream. Таким образом, используя библиотеки iostream и string можно организовать чтение и запись строк при помощи стандартных операторов ввода и вывода.

```
// Обратите внимание, перед компиляцией этот код следует дополнить
// директивами #include и объявлениями using
int main()
{
    string s;          // пустая строка
    cin >> s;          // чтение разделяемой пробелами строки в s
    cout << s << endl; // запись s на стандартное устройство вывода
    return 0;
}
```

Эта программа начинается с определения переменной s типа string.

```
cin >> s;          // чтение разделяемой пробелами строки в s
```

Следующая строка (см. выше) читает данные со стандартного устройства ввода и сохраняет их в переменной s. Оператор ввода строк осуществляет следующие действия.

- Читает и отбрасывает все предваряющие непечатаемые символы (например пробелы, символы новой строки и табуляции).
- Затем читает значащие символы, пока не встретится следующий непечатаемый символ.

Таким образом, если ввести "   **Hello World!**   " (обратите внимание на предваряющие и завершающие пробелы), фактически будет получено значение "**Hello**" без пробелов.

Операторы ввода и вывода строк ведут себя аналогично операторам встроенных типов. В частности, они возвращают как результат свой левый операнд. Таким образом, операторы чтения или записи можно объединять в цепочки.

```
string s1, s2;
    cin >> s1 >> s2; // сначала прочитать в переменную s1,
                     // а затем в переменную s2
    cout << s1 << s2 << endl; // отобразить обе строки
```

Если в этой версии программы осуществить предыдущий ввод, вывод будет следующим.

```
Hello World!
```

Чтобы откомпилировать эту программу, необходимо добавить соответствующие директивы `#include` (для библиотек `iostream` и `string`), а также объявления using для всех используемых библиотечных имен: `string`, `cin`, `cout` и `endl`.

Начиная с этого места подразумевается, что в код программ следует включить необходимые директивы `#include` и объявления `using`.

Чтение неопределенного количества строк

Подобно операторам ввода, читающим встроенные типы, оператор ввода класса `string` возвращает поток, из которого он осуществляет чтение. Следовательно, оператор ввода класса `string` можно использовать в условии продолжения, точно так же, как это было сделано при чтении целых чисел в программе на стр. 41. Приведенная ниже программа читает набор строк со стандартного устройства ввода и построчно записывает прочитанное на стандартное устройство вывода.

```
int main()
{
    string word;
    // читать до конца файла, писать каждое слово с новой строки
    while (cin >> word)
        cout << word << endl;
    return 0;
}
```

В данном случае для чтения строк используется оператор ввода. Этот оператор возвращает поток `istream`, из которого он читает данные, для проверки в условии оператора `while`, который в результате выполняет чтение до тех пор, пока поток допустим. Поток остается допустимым до тех пор, пока не будет введен символ конца файла или недопустимое значение. В теле цикла `while` прочитанное в условии значение выводится на стандартное устройство вывода. Выход из цикла (и из программы) происходит при вводе символа конца файла.

Применение функции `getline()` для чтения целой строки

Класс `string` обладает дополнительный вспомогательной функцией ввода-вывода `getline()`. Функции `getline()` передают поток ввода и строку. Она читает всю введенную строку из потока и сохраняет ее в переданной строковой переменной, *без символа новой строки*. В отличие от оператора ввода, функция `getline()` не игнорирует предваряющие пробелы. Однако каждый раз, когда функция `getline()` встречает символ новой строкой, даже если это первый введенный символ, она останавливает чтение из буфера ввода и завершает работу. В результате, если первым окажется символ новой строки, переданная строковая переменная будет пуста.

Функция `getline()` возвращает переданный ей аргумент типа `istream`, чтобы подобно оператору ввода использовать его при проверке условий. Например, предыдущую программу можно переписать так, чтобы она вместо одного слова читала и выводила целую строку.

```
int main()
{
    string line;
```

```
    // читать строку до конца файла
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

Поскольку переменная `line` не будет содержать символа новой строки, для построчной записи результата его придется ввести принудительно. Для этого, как обычно, используется манипулятор `endl`, который, кроме перевода строки, осуществляет сброс буфера вывода.

Символ новой строки, который вынуждает функцию `getline()` завершить работу, отбрасывается и в строковой переменной *не сохраняется*.

Упражнения раздела 3.2.2

Упражнение 3.5. Напишите программу, читающую со стандартного устройства ввода целые строки. Измените программу так, чтобы она читала отдельные слова.

Упражнение 3.6. Объясните, как поступают с символами предваряющих пробелов при чтении данных в строковую переменную, когда используется функция `getline()`.

3.2.3. Операции со строками

Список наиболее часто используемых операций со строками приведен в табл. 3.2.

Таблица 3.2. Строковые операции

`s.empty()`	Возвращает значение `true`, если строка `s` пуста. В противном случае возвращает значение `false`
`s.size()`	Возвращает количество символов в строке `s`
`s[n]`	Возвращает символ номер `n` в строке `s`; нумерация начинается с 0
`s1 + s2`	Возвращает строку, состоящую из содержимого строк `s1` и `s2`
`s1 = s2`	Заменяет символы строки `s1` копией содержимого `s2`
`v1 == v2`	Возвращает значение `true`, если строки `v1` и `v2` совпадают. В противном случае возвращает значение `false`
`!=`, `<`, `<=`, `>` и `>=`	Имеют обычное назначение

Строковые операции `size()` и `empty()`

Длина строки равна количеству ее символов. Именно это значение и возвращает функция `size()`.

```
int main()
{
    string st("The expense of spirit\n");
    cout << "The size of " << st << "is " << st.size()
         << " characters, including the newline" << endl;
    return 0;
}
```

Если откомпилировать и запустить на выполнение эту программу, результат будет следующим.

```
The size of The expense of spirit
is 22 characters, including the newline
```

Зачастую необходимо выяснить, не является ли строка пустой. Для этого можно сравнить ее размер с нулем.

```
if (st.size() == 0)
    // ok: пуста
```

В данном случае фактическое количество символов в строке не важно, главное знать, не равно ли оно нулю. Ответ на тот же вопрос можно получить и проще — используя функцию-член `empty()`.

```
if (st.empty())
    // ok: пуста
```

Функция `empty()` возвращает логическое значение `true` (раздел 2.1 стр. 56), если строка не содержит никаких символов, и значение `false` — в противном случае.

Тип `string::size_type`

Вполне логично ожидать, что функция `size()` возвращает значение типа `int`, а учитывая совет на стр. 59 (раздел 2.1.1), вероятней всего, типа `unsigned`. Но вместо этого функция `size()` возвращает значение *типа* `string::size_type`. Этот тип требует более подробных объяснений.

В классе `string` (и нескольких других библиотечных типах) определены вспомогательные типы данных. Эти вспомогательные типы позволяют использовать библиотечные типы машинно-независимым способом. Тип `size_type` — это один из таких вспомогательны типов. Он определен как синоним типа `unsigned`, `unsigned int` или `unsigned long`, т.е. как гарантированно большой, чтобы содержать размер любой строки. Чтобы воспользоваться типом `size_type`, определенным в классе `string`, применяется оператор области видимости (`::`), указывающий на то, что имя `size_type` определено в классе `string`.

Любая переменная, используемая для сохранения результата обращения к функции `size()` класса `string`, должна иметь тип `string::size_type`. Значение возвращаемое функцией `size()` ни в коем случае нельзя присваивать переменной типа `int`.

Хотя точный размер типа `string::size_type` неизвестен, можно с уверенностью сказать, что это беззнаковый тип (раздел 2.1.1 стр. 57). Как известно, беззнаковая версия любого типа способна содержать положительные значения вдвое большего размера, чем знаковая версия того же типа. Таким образом, размер самой большой строки может оказаться вдвое больше значения, предельно допустимого для типа `int`.

Еще одна проблема, связанная с применением типа `int`, заключается в том, что на некоторых машинах размер типа `int` слишком мал, чтобы содержать размер строк даже вполне реальной длинны. Например, если машина имеет 16-битовый тип `int`, то самая большая строка могла бы иметь размер 32 767 символов. Размер объекта класса `string`, в который загружают содержимое файла, может легко превысить это число. Поэтому самым надежным способом хранения размеров строк является использование специального типа `string::size_type`.

Операторы сравнения класса `string`

В классе `string` определено несколько операторов для сравнения двух строковых значений. Каждый из этих операторов работает, сравнивая символы каждой строки.

При сравнении строк учитывается регистр символов. То есть символы в верхнем и нижнем регистре считаются разными. На большинстве компьютеров прописные буквы располагаются раньше строчных, поскольку им соответствует меньшее битовое число.

Оператор равенства (`==`) сравнивает две строки и возвращает значение `true`, если они равны. Две строки считаются равными, если они имеют одинаковую длину и содержат одинаковые символы. Существует также оператор неравенства (`!=`), возвращающий значение `true`, если строки не равны.

Операторы сравнения `<`, `<=`, `>` и `>=` проверяют, не является ли одна строка меньше, меньше или равна, больше и больше или равна другой.

```
string big = "big", small = "small";
string s1 = big;    // s1 - копия big
if (big == small)   // false
    // ...
if (big <= s1)      // true, они равны или big меньше s1
    // ...
```

Эти операторы сравнивают строки используя ту же стратегию, т.е. посимвольно с учетом регистра.

- Если две строки имеют разные длины и каждый символ короткой строки равен соответствующему символу длинной строки, короткая строка меньше длинной.
- Если символы двух строк отличаются, результат сравнения определит первый несовпадающий символ.

Давайте рассмотрим несколько строк.

```
string substr = "Hello";
string phrase = "Hello World";
string slang = "Hiya";
```

Здесь строковая переменная `substr` содержит значение, меньшее, чем строковая переменная `phrase`, а строковая переменная `slang` содержит значение, которое больше, чем строковые переменные `substr` и `phrase`.

Присвоение строк

Как правило, библиотечные типы столь же просты в применении, как и встроенные. Поэтому большинство библиотечных типов поддерживают присвоение. Строки не являются исключением, один объект класса `string` вполне можно присвоить другому.

```
// st1 - пустая строка, st2 - копия литерала
string st1, st2 = "The expense of spirit";
st1 = st2; // замена содержимого st1 копией st2
```

После присвоения переменная `st1` содержит копию символов переменной `st2`.

Большинство библиотечных реализаций класса `string` сопряжены с некоторыми проблемами при выполнении таких операций, как присвоение, но следует заметить, что концептуально присвоение требует довольно большого количества дейст-

вий. Сюда относится удаление хранилища, содержащего символы переменной st1, создание хранилища для копии символов переменной st2 и, собственно, копирование символов переменной st2 в это новое хранилище[1].

Сложение двух строк

Сложение строк называется *конкатенацией* (concatenation). Конкатенация позволяет состыковать две или несколько строк при помощи оператора плюс (+) или составного оператора присвоения (+=) (раздел 1.4.1 стр. 34). Давайте рассмотрим две строки.

```
string s1("hello, ");
string s2("world\n");
```

Эти строки можно сложить и создать третью строку следующим образом.

```
string s3 = s1 + s2;   // s3 содержит hello, world\n
```

Чтобы сумму строк s2 и s1 поместить в переменную s1, можно использовать оператор +=.

```
s1 += s2;   // эквивалентно s1 = s1 + s2
```

Сложение строк и символьных строковых литералов

Строковые переменные s1 и s2 складываются непосредственно по их значениям. Тот же результат можно получить при конкатенации объектов класса string и строковых литералов.

```
string s1("hello");
string s2("world");
string s3 = s1 + ", " + s2 + "\n";
```

При смешанном сложении строк и строковых литералов, по крайней мере один из операндов каждого оператора + должен быть объектом класса string.

```
string s1 = "hello";
string s2 = "world";
string s3 = s1 + ", ";      // ok: сложение строки и литерала
string s4 = "hello" + ", "; // ошибка: нет строкового операнда
string s5 = s1 + ", " + "world"; // ok: каждый + имеет
                                 // строковый операнд
string s6 = "hello" + ", " + s2; // ошибка: нельзя сложить
                                 // строковые литералы
```

Инициализация переменных s3 и s4 осуществляет только одну операцию. В данном случае инициализация переменной s3 вполне корректна, ведь здесь осуществляется конкатенация объекта класса string и строкового литерала. Инициализация переменной s4 недопустима, поскольку здесь осуществляется пытка сложить два строковых литерала.

Инициализация переменной s5, как ни странно, вполне допустима. Здесь конкатенация срабатывает аналогично цепочке выражений ввода или вывода (раздел 1.2 стр. 27). Оператор суммы библиотечного типа string возвращает тип string. Таким образом, часть выражения s1 + ", ", при инициализации переменной s5, возвращает тип string, который вполне может осуществить конкатенацию с литералом "world\n". Это эквивалентно следующей форме записи.

---

[1] *И наконец, переприсвоение новому хранилищу имени st1. — Примеч. ред.*

```
string tmp = s1 + ", ";  // ok: + имеет строковый операнд
s5 = tmp + "world";      // ok: + имеет строковый операнд
```

С другой стороны, инициализация переменной s6 недопустима. Можно заметить, что в первой части выражения происходит сложение двух строковых литералов. Это ошибка, поэтому весь оператор считается недопустимым.

Выборка символов из строки

Для доступа к отдельным символам строки тип `string` предоставляет оператор *индексирования* (`[]`). Оператору индексирования передают значение типа `size_type`, указывающее номер требуемого символа. Значение, передаваемое оператору индексирования, называют *индексом* (subscript или index).

Индексирование строк начинается с нуля. То есть если переменная `s` типа `string` не пуста, `s[0]` соответствует первому символу строки, `s[1]` — второму, а `s[s.size() - 1]` — последнему.
Использование индекса вне этого диапазона является серьезной ошибкой.

Используя оператор индексирования можно, например, отобразить каждый символ строки в отдельной строке.

```
string str("some string");
for (string::size_type ix = 0; ix != str.size(); ++ix)
    cout << str[ix] << endl;
```

При каждой итерации цикла происходит выборка следующего символа из строки `str` и его вывод на стандартное устройство перед символом новой строки.

Оператор индексирования возвращает L-значение

Напомним, что переменная представляет собой l-значение (раздел 2.3.1 стр. 67), т.е. она может располагаться с левой стороны от оператора присвоения. Подобно переменной, значение возвращенное оператором индексирования является l-значением. Следовательно, индексирование применимо с обеих сторон оператора присвоения. Следующий цикл присваивает звездочку каждому символу строки `str`.

```
for (string::size_type ix = 0; ix != str.size(); ++ix)
    str[ix] = '*';
```

Вычисление значений индекса

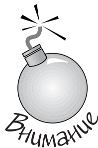
Любое выражение, результатом которого является целочисленное значение, применимо в качестве индекса. Например, если `someval` и `someotherval` являются целочисленными переменными, следующая запись вполне допустима.

```
str[someotherval * someval] = someval;
```

Хотя в качестве индекса применим любой целочисленный тип, фактически индекс имеет тип `string::size_type` (некий беззнаковый тип).

Тип `string::size_type` используется для индекса по тем же причинам, что и для возвращаемого значения функции `size()`. Используемая при индексировании строки переменная должна быть способна содержать число, соответствующее количеству символов строки.

Ответственность за не превышение индексом диапазона при индексировании строки, несет сам разработчик. Диапазон индексирования составляют числа от нуля до размера строки минус один. Использование для индекса переменной типа `string::size_type` (или другого беззнакового типа) гарантирует, что его значение не будет меньшим, чем нуль. Остается лишь организовать проверку того, что индекс всегда будет меньше размера строки.



Проверку значения индекса библиотека не обеспечивает. Применение индекса, значение которого находится вне диапазона, приводит во время выполнения программы к неопредсказуемым последствиям (как правило, фатальным) и является серьезной ошибкой.

3.2.4. Работа с символами строки

Зачастую приходится работать с индивидуальными символами строки. Например, может понадобиться выяснить, является ли определенный символ пробелом, буквой или цифрой. Список функций, применимых к отдельным символам строки (и любым другим символьным значениям) приведен в таб. 3.3. Эти функции определены в *заголовке* `cctype`.

Как правило, эти функции проверяют переданный им символ и возвращают значение типа `int`, соответствующее логическому значению. То есть каждая из функций возвращает нуль, если проверка не пройдена, и любое другое, отличное от нуля, значение — в противном случае.

Таблица 3.3. Функции `cctype`

`isalnum(c)`	Возвращает значение `true`, если `c` является буквой или цифрой
`isalpha(c)`	Возвращает значение `true`, если `c` — буква
`iscntrl(c)`	Возвращает значение `true`, если `c` — управляющий символ
`isdigit(c)`	Возвращает значение `true`, если `c` — цифра
`isgraph(c)`	Возвращает значение `true`, если `c` — не пробел, а печатаемый символ
`islower(c)`	Возвращает значение `true`, если `c` — символ в нижнем регистре
`isprint(c)`	Возвращает значение `true`, если `c` — печатаемый символ
`ispunct(c)`	Возвращает значение `true`, если `c` — знак пунктуации
`isspace(c)`	Возвращает значение `true`, если `c` — пробел
`isupper(c)`	Возвращает значение `true`, если `c` — символ в верхнем регистре
`isxdigit(c)`	Возвращает значение `true`, если `c` — шестнадцатеричная цифра
`tolower(c)`	Если `c` — прописная буква, возвращает ее эквивалент в нижнем регистре, а в противном случае возвращает символ `c` неизменным
`toupper(c)`	Если `c` — строчная буква, возвращает ее эквивалент в верхнем регистре, а в противном случае возвращает символ `c` неизменным

Эти функции считают печатаемыми такие символы, которые имеют видимое графическое представление, а к непечатаемым относятся пробел, табуляция, вертикальная табуляция, возврат каретки, новая строка и прогон страницы. Знак пунктуации — это отображаемый символ, который не является цифрой, буквой или непечатаемым символом (таким как пробел).

Эти функции можно использовать, например, для вывода на экран информации о количестве знаков пунктуации во введенной строке.

```
string s("Hello World!!!");
string::size_type punct_cnt = 0;
// подсчитать количество знаков пунктуации в строке s
for (string::size_type index = 0; index != s.size(); ++index)
    if (ispunct(s[index]))
        ++punct_cnt;
cout << punct_cnt
     << " punctuation characters in " << s << endl;
```

Результат ее выполнения будет следующим.

```
3 punctuation characters in Hello World!!!
```

Совет. Используйте заголовки библиотек языка C в версии для языка C++

Кроме средств, определенных специально для языка C++, его библиотека содержит и библиотеку языка C. Заголовок `cctype` обеспечивает доступ к библиотечным функциям языка C, определенным в файле заголовка C по имени `ctype.h`.

Стандартные имена заголовков языка C используют формат *имя*`.h`. В версиях аналогичных заголовков для языка C++, из имен удалено расширение `.h` и добавлен префикс `c`. Таким образом, имена заголовков версии для языка C++ имеют формат `c`*имя*. Символ `c` указывает, что заголовок происходит от заголовка библиотеки C. Следовательно, заголовок `cctype` имеет то же содержимое, что и `ctype.h`, но в формате, соответствующем программам языка C++. В частности, имена, определенные в заголовках `c`*имя*, принадлежат пространству имен `std`, а имена, определенные в заголовках версии `.h`, — нет.

Как правило, в программах на языке C++ имеет смысл использовать версии заголовков в формате `c`*имя*, а не *имя*`.h`. Таким образом, будут автоматически использованы имена из стандартной библиотеки, принадлежащие пространству имен `std`. Использование заголовка `.h` возлагает на программиста дополнительную заботу по отслеживанию, какие из библиотечных имен унаследованы от языка C, а какие принадлежат языку C++.

Функции `tolower()` и `toupper()` возвращают не логическое значение, а переданный им символ, неизменный или с измененным регистром. Чтобы поменять регистр символа на нижний, функцию `tolower()` можно использовать следующим образом.

```
// преобразовать символы строки s в нижний регистр
for (string::size_type index = 0; index != s.size(); ++index)
    s[index] = tolower(s[index]);

cout << s << endl;
```

Результат выполнения этого кода будет следующим.

```
hello world!!!
```

Упражнения раздела 3.2.4

Упражнение 3.7. Напишите программу, которая читает две строки и уведомляет, равны ли они, а если нет, какая из них больше. Измените программу так, чтобы она указывала, имеют ли строки одинаковую длину, и если нет, то какая из них длинней.

Упражнение 3.8. Напишите программу, способную читать строки со стандартного устройства ввода и соединять их в одну большую строку. Отобразите полученную строку. Измените программу так, чтобы отделить соседние введенные строки пробелами.

Упражнение 3.9. Что делает следующая программа? Действительно ли она допустима? Если нет, то почему?

```
string s;
cout << s[0] << endl;
```

Упражнение 3.10. Напишите программу поиска в строке знаков пунктуации. Программа должна позволить ввести символьную строку, содержащую знаки пунктуации, и вывести ту же строку, но уже без знаков пунктуации.

3.3. Библиотечный тип `vector`

Вектор (vector) — это коллекция объектов одинакового типа, каждому из которых присвоен целочисленный индекс. Подобно типу `string`, всю “заботу” по манипулированию памятью и хранению элементов типа `vector`, берет на себя библиотека. Вектор называют также *контейнером* (container), поскольку он содержит другие объекты. Все объекты в контейнере должны иметь одинаковый тип. Более подробная информация о контейнерах приведена в главе 9, “Последовательные контейнеры”.

Чтобы использовать вектор, необходимо подключить соответствующий заголовок. В примерах этой книги подразумевается, что в начале кода сделано следующее объявление `using` и подключен соответствующий заголовок.

```
#include <vector>
using std::vector;
```

Вектор является *шаблоном класса* (class template). Шаблоны позволяют создать одно определение класса или функции, которые впоследствии можно применить для ряда типов. Таким образом, можно определить вектор, содержащий строки, или вектор, содержащий целочисленные переменные либо даже объекты собственного класса, такого как `Sales_item`. Более подробная информация о собственных шаблонах классов приведена в главе 16, “Шаблоны и общее программирование”. К счастью, чтобы использовать шаблоны, вовсе не обязательно уметь их создавать.

Чтобы объявить объект типа, созданного из шаблона класса, ему необходимо предоставить дополнительную информацию. Характер этой информации зависит от шаблона. В случае вектора, необходимо указать, объекты какого типа он будет содержать. Чтобы указать тип, его имя следует поместить в угловые скобки, располагающиеся после имени шаблона.

```
vector<int> ivec;             // ivec содержит объекты типа int
vector<Sales_item> Sales_vec; // содержит Sales_item
```

Подобно определению любой другой переменной, здесь указан тип и список из одной или нескольких переменных. В первом из этих определений (`vector<int>`)

создан вектор `ivec` для хранения объектов типа `int`, а во втором определен вектор `Sales_vec`, способный хранить объекты класса `Sales_item`.

Часть `vector` — это не тип, а шаблон, который можно использовать для определения вектора, способного хранить наборы любых типов. Тип, указанный в определении вектора, позволяет задать тип хранимых в нем элементов. Следовательно, типом является `vector<int>` и `vector<string>`.

3.3.1. Определение и инициализация векторов

В классе `vector` определено несколько конструкторов (раздел 2.3.3 стр. 71), которые можно использовать при определении и инициализации объектов векторов. Эти конструкторы перечислены в табл. 3.4.

Таблица 3.4. Способы инициализации векторов

`vector<T> v1;`	Вектор, содержащий объекты типа `T`. Стандартный конструктор `v1` пуст
`vector<T> v2(v1);`	Вектор `v2` — копия вектора `v1`
`vector<T> v3(n, i);`	Вектор `v3` содержит `n` элементов со значением `i`
`vector<T> v4(n);`	Вектор `v4` содержит `n` копий самостоятельно инициализированного объекта

Создание определенного количества элементов

При создании вектора, если он не пуст, его элементы должны быть инициализированы допустимыми значениями. Когда один вектор копируется в другой, каждый элемент в новом векторе будет инициализирован копией соответствующего элемента исходного вектора. При этом оба вектора должны быть предназначены для хранения элементов одинакового типа.

```
vector<int> ivec1;          // ivec1 содержит объекты типа int
vector<int> ivec2(ivec1);   // ok: копирование элементов ivec1 в ivec2
vector<string> svec(ivec1); // ошибка: svec содержит строки,
                            // а не целые числа
```

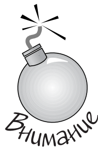
Вектор можно инициализировать набором из определенного количества элементов, обладающих указанным значением. Для указания количества элементов вектора конструктор использует счетчик, а также значение, которое должен содержать каждый из элементов.

```
vector<int> ivec4(10, -1);      // 10 элементов, инициализируемых
                                // значением -1
vector<string> svec(10, "hi!"); // 10 строк, инициализируемых
                                // значением "hi!"
```

Фундаментальная концепция. Размер вектора увеличивается динамически

Главным преимуществом векторов (и других библиотечных контейнеров), является высокая эффективность добавления в них элементов во время работы. Для этого вектор обеспечен возможностью динамически изменять свой размер по мере добавления в него элементов.

Как будет продемонстрировано в главе 4, “Массивы и указатели”, подобное поведение резко контрастирует с возможностями встроенных массивов языка С и большинства других языков. В частности, читатели, которые знакомы с языками С или Java, могли бы предположить, что вектор имеет фиксированное количество элементов, а его размер имеет смысл выбирать с некоторым запасом от ожидаемого. Однако фактически имеет место противоположный случай по причинам, рассматриваемым в главе 9, “Последовательные контейнеры”.



Хотя количество элементов вектора можно задать заранее, как правило, удобнеей создавать пустой вектор и добавлять в него элементы по мере надобности.

Инициализирующее значение

Когда инициализирующий элемент не указан, библиотека самостоятельно создает значение, инициализирующее элемент. Это созданное библиотекой *инициализирующее значение* (value initialized) используется для инициализации каждого элемента в контейнере. Фактически используемое значение зависит от типа хранимых в векторе элементов.

Если элементы вектора имеют встроенный тип, например `int`, библиотека инициализирует их значением `0`.

```
vector<string> fvec(10);   // 10 элементов, инициализированных
                           // значением 0
```

Если хранимые в векторе элементы являются объектами класса (например строками) и для них определены собственные конструкторы, для создания инициализирующего значения элемента библиотека использует стандартный конструктор класса.

```
vector<string> svec(10);   // 10 элементов, инициализированных
                           // пустой строкой
```

Как будет продемонстрировано в главе 12, “Классы”, некоторые классы, в которых определены собственные конструкторы, не определен стандартный конструктор. Вектор такого типа нельзя инициализировать указав только размер, приходится также указывать и начальное значение элемента.

Поэтому существует третья возможность, когда класс элемента может не иметь соответствующего конструктора. В данном случае библиотека все равно создает объект, инициализированный значением, но использует для этого значение каждого конкретного объекта элемента.

Упражнения раздела 3.3.1

Упражнение 3.11. Какое из следующих определений векторов (если оно есть) является ошибочным?

```
(a) vector< vector<int> > ivec;
(b) vector<string> svec = ivec;
(c) vector<string> svec(10, "null");
```

Упражнение 3.12. Сколько элементов содержится в каждом из следующих векторов? Каковы значения их элементов?

```
(a) vector<int> ivec1;
(b) vector<int> ivec2(10);
(c) vector<int> ivec3(10, 42);
(d) vector<string> svec1;
(e) vector<string> svec2(10);
(f) vector<string> svec3(10, "hello");
```

3.3.2. Операции с векторами

Библиотека `vector` обеспечивает несколько операций с векторами, большинство из которых подобны операциям со строками. Список важнейших операций с векторами приведен в табл. 3.5.

Таблица 3.5. Операции с векторами

`v.empty()`	Возвращает значение `true`, если вектор `v` пуст. В противном случае возвращает значение `false`
`v.size()`	Возвращает количество элементов вектора `v`
`v.push_back(t)`	Добавляет элемент со значением `t` в конец вектора `v`
`v[n]`	Возвращает элемент номер `n` вектора `v`
`v1 = v2`	Заменяет элементы вектора `v1` копией элементов вектора `v2`
`v1 == v2`	Возвращает значение `true`, если векторы `v1` и `v2` равны
`!=`, `<`, `<=`, `>` и `>=`	Имеют обычное назначение

Размер вектора

Функции `empty()` и `size()` класса `vector` аналогичны одноименным функциям класса `string` (раздел 3.2.3 стр. 107). Функция `size()` возвращает значение типа `size_type`, определенное для соответствующего типа `vector`.

Чтобы использовать тип `size_type`, необходимо указать тип, для которого он определен. Для типа `vector` *всегда* необходимо указывать тип хранимого элемента.

```
vector<int>::size_type  // ok
vector::size_type       // ошибка
```

Добавление элементов в вектор

Функции `push_back()` передают значение элемента, добавляемого в конец вектора.

```
// читать слова со стандартного устройства ввода и сохранять
// их как элементы вектора
string word;
vector<string> text;      // пустой вектор
while (cin >> word) {
    text.push_back(word); // добавить слово в вектор text
}
```

Этот цикл последовательно читает строки со стандартного устройства ввода и добавляет их в конец вектора. Вектор `text` определен как изначально пустой. При каждой итерации цикла в вектор добавляется новый элемент, значением которого является слово, прочитанное со стандартного устройства ввода. По завершении цикла элементы вектора `text` будут содержать все прочитанные слова.

Индексирование вектора

Хранимые в векторе объекты не именованы. Обращаться к ним можно лишь по позиции в векторе. Для этого используется оператор индексирования. Индексирование вектора подобно индексированию строк (раздел 3.2.3 стр. 111).

Оператор индексирования вектора получает значение индекса и возвращает элемент, соответствующий этой позиции. Элементы вектора пронумерованы начиная с 0. В приведенном ниже примере цикл `for` используется для обнуления всех элементов вектора `ivec`.

```
// обнулить все элементы вектора
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
    ivec[ix] = 0;
```

Подобно оператору индексирования строки, векторный оператор индексирования возвращает l-значение, поэтому сделанное в теле цикла присвоение вполне законно. Также аналогично строкам, для индекса используется векторный тип `size_type`.

Даже если вектор `ivec` пуст, цикл `for` сработает правильно. Если вектор `ivec` пуст, функция `size()` вернет значение `0` и выражение проверки цикла `for` сравнит итератор `ix` со значением `0`. Поскольку вначале итератор `ix` содержит значение `0`, результат проверки на первом же цикле окажется отрицательным и его тело не будет выполнено ни разу.

Фундаментальная концепция. Старое доброе программирование

Программисты, перешедшие на язык C++ с языка C или Java, могли бы быть удивлены тем, что в данном цикле при сравнении индекса с размером вектора использован оператор `!=`, а не `<`. Программистов C, вероятно, удивит также то, что вызов функции `size()` происходит непосредственно в операторе `for`, а не перед ним с запоминанием результирующего значения в переменной.

Программисты языка C++ предпочитают использовать в циклах оператор `!=`, а не `<` исключительно по привычке. В данном случае никакой разницы между ними нет.

В данном случае вызов функции `size()` в операторе `for` вместо запоминания результата ее выполнения также особого значения не имеет, но является хорошей привычкой. В языке C++ размер таких структур, как вектор, может изменяться динамически. Данный цикл только читает элементы; но не добавляет их. Однако другой цикл вполне может менять количество элементов. В таком случае, цикл использующий заранее сохраненное значение размера, будет некорректен. Поэтому проверять текущий размер имеет смысл на каждой итерации цикла.

Как будет продемонстрировано в главе 7, “Функции”, функции языка C++ можно объявить встраиваемыми. В этом случае компилятор встроит содержимое такой функции непосредственно в код по месту вызова, а не будет создавать механизм фак-

тического вызова функции. Крошечные библиотечные функции, такие как size(), почти наверняка имеет смысл объявлять встраиваемыми, поэтому ожидаемые дополнительные затраты времени на ее выполнение при каждом цикле окажутся несущественными.

Индексирование не добавляет элементов

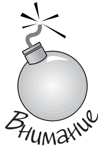
Программисты, плохо знакомые с языком C++, иногда полагают, что индексирование вектора позволяет добавлять в него элементы, но это не так.

```
vector<int> ivec;  // пустой вектор
for (vector<int>::size_type ix = 0; ix != 10; ++ix)
    ivec [ix] = ix;  // катастрофа: ivec не имеет элементов
```

В этом коде предполагалось добавить 10 новых элементов в вектор ivec, присваивая им значения от 0 до 9. Однако вектор ivec остается пустым, поскольку при помощи индексирования можно обращаться только к уже существующим элементам.

Правильно такой цикл можно создать следующим образом.

```
for (vector<int>::size_type ix = 0; ix != 10; ++ix)
ivec.push_back(ix); // ok: добавляет новый элемент со значением ix
```



Присвоение значения элементу при помощи индекса не создает новых элементов вектора, для этого элемент должен уже существовать.

Внимание! Индексировать можно лишь существующие элементы!

Очень важно понять, что оператор индексирования ([]) можно использовать для доступа только к фактически существующим элементам. Рассмотрим пример.

```
vector<int> ivec;       // пустой вектор
cout << ivec [0];       // Ошибка: ivec не имеет элементов!

vector<int> ivec2(10); // вектор из 10 элементов
cout << ivec [10];      // Ошибка: ivec имеет элементы 0...9
```

Попытка обращения во время выполнения программы к несуществующему элементу является серьезной ошибкой. Подобно большинству подобных ошибок, практически ни одна из реализаций компилятора не обнаруживает их. Результат выполнения такой программы непредсказуем, однако почти наверняка такая программа правильно работать не будет.

Это предостережение применимо к любым случаям индексирования, включая строки и, как будет продемонстрировано вскоре, встроенные массивы.

Попытка индексирования несуществующих элементов, к сожалению, является весьма распространенной и грубой ошибкой программирования. Так называемая ошибка *переполнения буфера* (buffer overflow) — результат индексирования несуществующих элементов. Такие ошибки являются наиболее распространенной причиной проблем защиты приложений.

Упражнения раздела 3.3.2

Упражнение 3.13. Прочитайте в вектор набор целых чисел. Вычислите и отобразите сумму каждой пары смежных элементов в векторе. Если количество элементов нечетно, сообщите пользователю об этом и отобразите значение последнего элемента без суммирования. Измените программу так, чтобы она отобразила сумму первого и последнего элементов, затем сумму второго и предпоследнего и т.д.

Упражнение 3.14. Прочитайте некоторый текст, сохраняя каждое введенное слово как отдельный элемент вектора. Преобразуйте символы каждого слова в прописные. Отобразите преобразованные элементы вектора, выводя по восемь слов в строке.

Упражнение 3.15. Допустима ли следующая программа? Если нет, то как ее исправить?

```
vector<int> ivec;
ivec[0] = 42;
```

Упражнение 3.16. Укажите три способа создания вектора и добавления в него 10 элементов, каждый из которых содержит значение `42`. Существует ли для этого предпочтительный способ и почему?

3.4. Знакомство с итераторами

Кроме индексирования, для доступа к элементам вектора библиотека предоставляет еще один способ — *итератор* (iterator). Итератор — это тип, позволяющий обращаться к хранимым в контейнере элементам, перемещаясь от одного к другому.

В библиотеке тип итератора определен для каждого из стандартных контейнеров, включая вектор. Итераторы являются более распространенным средством, чем индексирование: для всех библиотечных контейнеров определены типы итераторов, но лишь некоторые из них поддерживают индексирование. Поскольку итераторами обладают все контейнеры, современные программисты языка C++ предпочитают использовать для доступа к элементам именно итераторы, а не индексы, даже если данный тип контейнера (например вектор) индексирование поддерживает.

Более подробная информация о работе с итераторами приведена в главе 11, “Общие алгоритмы”, но использовать их можно уже на данном этапе, не до конца понимая все подробности.

Контейнерный тип `iterator`

В каждом из классов контейнеров, например в классе `vector`, определен его собственный тип итератора.

```
vector<int>::iterator iter;
```

В этом операторе определена переменная `iter`, тип `iterator` которой определен в векторе `vector<int>`. В каждом библиотечном классе контейнера определен член по имени `iterator`, который является синонимом его фактического типа итератора.

Терминология. Итератор и тип `iterator`

На первый взгляд терминология, связанная с итераторами, не до конца понятна. Частично это связано с тем, что термин *итератор* (iterator) используется для описания двух понятий, собственно итератора и специального типа `iterator`, определенного в классе контейнера, например `vector<int>`.

Важно понять, что концептуально итератор служит для перебора коллекции элементов определенного типа, а тип iterator предоставляет некий набор действий. Эти действия позволяют перемещаться между элементами контейнера и обращаться к их значениям.
В каждом контейнерном классе определен его собственный тип iterator, который применяется при организации доступа к элементам, хранимым в контейнере. Таким образом, для каждого контейнера определен тип по имени iterator, обеспечивающий действия концептуального итератора.

Функции begin() и end()

В классе каждого контейнера определены две функции, begin() и end(), которые возвращают итератор. Итератор, возвращаемый функцией begin(), позволяет обратиться к первому элементу контейнера, если он есть.

```
vector<int>::iterator iter = ivec.begin();
```

Этот оператор инициализирует итератор iter значением, возвращенным функцией begin() вектора ivec. С учетом того, что вектор не пуст, в результате подобной инициализации итератор iter позволит обратиться к тому же элементу, что и ivec[0].

Итератор, возвращенный функцией end(), позиционирует итератор на элемент, следующий за последним. Иногда говорят, что он указывает на конец вектора, но если воспользоваться им, обращение произойдет к несуществующему элементу вне вектора. Если вектор пуст, функции begin() и end() возвращают одинаковый итератор.

Итератор, возвращенный функцией end(), указывает на фактически несуществующий элемент вектора. Он используется как граница при переборе элементов вектора.

Обращение к значению и инкремент векторных итераторов

Операции с переменными типа iterator позволяют получить доступ к элементу, на который указывает итератор, а также переместить итератор с одного элемента на другой.

Для доступа к элементу, на который указывает итератор, типы iterator предоставляет оператор *обращения к значению* (dereference operator) или ссылки (*).

```
*iter = 0;
```

Оператор обращения к значению возвращает элемент, на который итератор указывает в настоящее время. Предположим, что итератор iter указывает на первый элемент вектора, в этом случае *iter будет тем же элементом, что и ivec[0]. В результате выполнения выражения, приведенного выше, первому элементу вектора будет присвоено значение 0.

Чтобы переместить итератор на следующий элемент в контейнере, используется *оператор инкремента* (increment) (++), или *приращения* (раздел 1.4.1 стр. 35). Логически, приращение итератора подобно инкременту целочисленной переменной. В случае с целочисленной переменной, ее значение увеличивается на единицу, а в случае с итератором, он перемещается на одну позицию вперед. Так, если итератор

iter указывает на первый элемент вектора, после оператора ++iter он будет указывать на второй.

Поскольку возвращаемый функцией end() итератор не указывает ни на один из существующих элементов, его нельзя ни прирастить, ни обратиться по нему к значению.

Другие операции с итераторами

С итераторами можно выполнить еще две весьма полезные операции, сравнить их используя операторы == и ! =. Итераторы равны, если они указывают на тот же элемент, и не равны в противном случае.

Программа, использующая итераторы

Предположим, что каждый из элементов вектора ivec типа vector<int> необходимо обнулить. Для этого можно использовать индексирование.

```
// обнулить все элементы вектора ivec
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
    ivec[ix] = 0;
```

В этой программе для перебора элементов вектора ivec использован цикл for. Непосредственно в цикле for определен индекс, значение которого увеличивается при каждой итерации. Код тела цикла for присваивает каждому элементу вектора ivec значение 0.

С использованием итератора этот цикл можно переписать следующим образом.

```
// эквивалентный цикл, обнуляющий вектор при помощи итератора
for (vector<int>::iterator iter = ivec.begin();
                           iter != ivec.end(); ++iter)
    *iter =0;  // присвоение значения 0 элементу,
               // на который указывает итератор
```

Цикл for начинается, как правило, с определения и инициализации итератора iter, чтобы получить возможность обратиться к первому элементу вектора ivec. В условии цикла for проверяется неравенство итератора iter со значением, возвращаемым функцией end(). При каждом проходе цикла осуществляется инкремент итератора iter. Таким образом, цикл for перебирает все элементы вектора ivec начиная с первого и до последнего. В конце итератор iter указывает на последний элемент вектора ivec. После обработки последнего элемента и приращения итератора iter, он станет равен значению, возвращаемому функцией end(), и цикл прекратится.

Выражение в теле цикла for использует для доступа к значению текущего элемента оператор обращения к значению (*). Подобно оператору индексирования, значение, возвращаемое этим оператором является l-значением. Поэтому здесь же можно использовать оператор присвоения, чтобы присвоить элементу значение или изменить его. В результате выполнения этого цикла, каждому элементу вектора ivec будет присвоено значение 0.

Рассмотрев этот код подробней, можно заметить, что он выполняет те же действия, что и предыдущая версия, в которой использовано индексирование. Здесь так же перебираются и обнуляются все элементы вектора от первого до последнего.

Эта программа, подобно представленной на стр. 118, безошибочно сработает и с пустым вектором. Если вектор ivec пуст, возвращенный функцией begin() итератор не указывает ни на один из существующих элементов, поскольку таковых еще нет. В этом случае итератор, возвращенный функцией begin(), совпадает с возвращенным функцией end() и, после проверки условия, цикл for немедленно прекращается.

Тип const_iterator

В предыдущей программе для изменения текущего значения вектора использовался итератор vector::iterator. В каждом контейнерном классе определен также тип по имени const_iterator, который используется только при чтении, но не записи значений в элементы контейнера.

При обращении с использованием обычного итератора, получается неконстантная ссылка на элемент (раздел 2.5 стр. 82), а при использовании типа const_iterator — ссылка на константный объект (раздел 2.4 стр. 78). Это аналогично любой константной переменной, непозволяющий изменять свое значение.

Предположим, например, что необходимо перебрать все элементы вектора text типа vector<string>. Для этого можно предпринять следующее.

```
// использовать const_iterator, поскольку изменять элементы не нужно
for (vector<string>::const_iterator iter = text.begin();
                                    iter != text.end(); ++iter)
    cout << *iter << endl; // отобразить каждый элемент
                           // вектора text
```

Этот цикл подобен предыдущему, за исключением того, что значения итератору не присваиваются, а выводятся на экран. Поскольку итератор используется для чтения, а не для записи, он объявлен как const_iterator. При обращении к константному итератору возвращается константное значение. Присвоить значение такому элементу нельзя.

```
for (vector<string>::const_iterator iter = text.begin();
                                    iter != text.end(); ++ iter)
    *iter = " ";    // ошибка: *iter является константой
```

Итератор типа const_iterator позволяет изменять значение самого итератора, но не элемента, на который он указывает. К такому итератору можно применять операторы инкремента и обращения к значению, но не оператор присвоения (не путать с инициализацией).

```
vector<int> nums(10); // nums не константен
const vector<int>::iterator cit = nums.begin();
*cit = 1;         // ok: cit позволяет инициализировать элемент
++cit;            // ошибка: нельзя изменить значение cit
```

Константный итератор может быть использован с константным или неконстантным вектором, поскольку он не позволяет записывать данные в элемент. Итератор являющийся константой практически бесполезен: после инициализации его можно использовать для чтения и записи данных в элемент, но сменить этот элемент на другой нельзя.

```
const vector<int> nines(10, 9); // нельзя будет изменять элементы
                                // вектора nines
// ошибка: итератор cit2 позволяет изменять элемент, на который он
// указывает, а вектор nines является константным
const vector<int>::iterator cit2 = nines.begin();
// ok: итератор it не позволяет изменять значение элемента, на
// который он указывает, поэтому с константным вектором он
// вполне применим
```

```
vector<int>::const_iterator it = nines.begin();
*it = 10; // ошибка: *it является константой
++it;     // ok: итератор it не является константой, поэтому его
          // значение вполне можно изменять
```

```
// итератор, не позволяющий записывать элементы
vector<int>::const_iterator
// итератор, значение которого нельзя изменить
const vector<int>::iterator
```

Упражнения раздела 3.4

Упражнение 3.17. Переделайте код упражнения из раздела 3.3.2 (стр. 120) так, чтобы для доступа к элементам вектора вместо индексирования использовался итератор.

Упражнение 3.18. Напишите программу, где создается вектор из 10 элементов. При помощи итератора присвойте каждому элементу значение, которое вдвое больше его текущего значения.

Упражнение 3.19. Проверьте предыдущую программу, отобразив хранимые в векторе значения.

Упражнение 3.20. Объясните, какой итератор использован в предыдущих программах и почему.

Упражнение 3.21. Когда имеет смысл использовать константный итератор? Когда имеет смысл использовать итератор типа `const_iterator`. Объясните различие между ними.

3.4.1. Арифметические действия с итераторами

Кроме оператора инкремента, который переводит итератор на один элемент вперед, итераторы векторов (и многих других библиотечных контейнеров) поддерживают и другие арифметические операции. К этим операциям, называемым арифметическими действиями с итераторами, относят следующие.

- `iter + n`
  `iter - n`

  К итератору можно прибавить или вычесть из него целочисленное значение. В результате получается новый итератор, который указывает на *n* элементов вперед (при сложении) или назад (при вычитании) от исходного значения итератора `iter`. Результат сложения и вычитания должен указывать на элемент вектора, к которому относится итератор `iter`, или на один из его концов. Типом добавляемого или вычитаемого значения, как правило, является `size_type` вектора или `difference_type` (см. ниже).

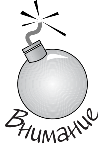
- `iter1 - iter2`

  Вычисление разницы между двумя итераторами, результатом которого является знаковое целочисленное значение типа `difference_type`. Этот тип, подобно типу `size_type`, определен в классе вектора. Тип `difference_type` является знаковым потому, что результатом вычитания может оказаться отрицательное число. Этот тип является гарантированно большим, чтобы содержать разницу между двумя любыми итераторами. Оба итератора, `iter1` и `iter2`, должны принадлежать к одному вектору.

Арифметические действия с итераторами можно использовать для перемещения итератора на определенное количество элементов. Например, середину вектора можно найти следующим образом.

```
vector<int>::iterator mid = (vi.begin() + vi.size()) / 2;
```

Этот код инициализирует итератор mid так, чтобы он указывал на элемент, ближайший к середине вектора ivec. Это вычисление существенно эффективней специального участка кода, который увеличивая значение итератора на единицу переместит его на серединный элемент.



Любая операция, которая изменяет размер вектора, делает существующие итераторы недопустимыми. Например, после вызова функции push_back() полагаться на значение уже существующего итератора больше нельзя.

Упражнения раздела 3.4.1

Упражнение 3.22. Что получится, если вычислить итератор mid следующим образом.

```
vector<int>::iterator mid = (vi.begin() + vi.end()) / 2;
```

3.5. Библиотечный тип bitset

Иногда в программах приходится иметь дело с упорядоченными наборами битов. Каждый бит может содержать значение 0 или 1. Использование битов — это самый компактный способ хранения значений в формате да или нет (иногда называемых флагами). Предоставляя *класс* bitset (набор битов), стандартная библиотека существенно облегчает работу с битами. Чтобы использовать класс bitset, в программу необходимо подключить его файл заголовка. В примерах также подразумевается, что в коде сделано соответствующее объявление using std::bitset.

```
#include <bitset>
using std::bitset;
```

3.5.1. Определение и инициализация наборов битов

Список конструкторов типа bitset приведен в табл. 3.6. Подобно вектору, тип bitset является шаблоном класса. В отличие от вектора, объекты типа bitset различаются по размеру, а не по типу. При определении набора битов в угловых скобках указывают количество битов, которые будет содержать набор.

```
bitset<32> bitvec; // 32 бита, весь нули
```

Размер должен быть указан константным выражением (раздел 2.7 стр. 84). Это может быть целочисленный константный литерал, как здесь, или константный объект целочисленного типа, инициализированный константным значением.

Приведенный выше оператор определяет bitvec как набор битов, который содержит 32 бита. Подобно элементам вектора, биты набора данных не именованы, а обращение к ним осуществляется по позиции. Биты пронумерованы начиная с нуля. Таким образом, биты набора bitvec пронумерованы от 0 до 31. Биты, расположенные ближе к 0, называют *младшими битами* (low-order), а расположенные ближе к 31 — *старшими битами* (high-order).

Таблица 3.6. Способы инициализации объектов класса bitset

`bitset<n> b;`	Набор `b` содержит `n` битов, каждый из которых содержит значение `0`
`bitset<n> b(u);`	Набор `b` является копией значения `u` типа `unsigned long`
`bitset<n> b(s);`	Набор `b` является копией битов, содержащихся в строке `s`
`bitset<n> b(s, pos, n);`	Набор `b` является копией битов из `n` символов строки `s` начиная с позиции `pos`

Инициализация набора битов беззнаковым значением

Когда значение типа `unsigned long` инициализирует набор битов, оно рассматривается как *битовая схема* (bit pattern). Биты в наборе битов являются копией этой схемы. Если размер набора битов больше количества битов, расположенных в переменной типа `unsigned long`, остающиеся старшие биты будут заполнены нулями. Если размер набора битов меньше количества битов инициализирующего значения, будут использованы лишь младшие биты инициализирующего значения, а старшие биты окажутся отброшены.

На машине с 32-битовым типом `unsigned long`, шестнадцатеричное значение `0xffff` представляется последовательностью из 16 битов, заполненных единицами, с последующими нулями до конца размера. (Каждая цифра `0xf` имеет битовое представление `1111`.) Набор битов можно инициализировать значением `0xffff` следующим образом.

```
// набор bitvec1 меньше инициализирующего значения
bitset<16> bitvec1(0xffff);  // биты 0...15 заполнены единицами

// размер набора bitvec2 и инициализирующего значения совпадают
bitset<32> bitvec2(0xffff);  // биты 0...15 заполнены единицами,
                             // а 16...31 - нулями

// на 32-битовой машине биты 0...31 инициализированы числом 0xffff
bitset<128> bitvec3(0xffff); // биты от 32 до 127 заполнены нулями
```

Во всех трех случаях биты от 0 до 15 инициализированы единицами. Для набора битов `bitvec1` старшие биты инициализирующего значения отброшены; набор битов `bitvec1` имеет меньше битов, чем тип `unsigned long`. Набор битов `bitvec2` имеет тот же размер, что и тип `unsigned long`, поэтому используются все биты инициализирующего объекта. Размер набора битов `bitvec3` больше, чем у типа `unsigned long`, поэтому его старшие биты, более 31, инициализированы нулями.

Инициализация набора битов из строки

При инициализации набора битов из строки, строка представляется как битовая схема. Биты строки считываются *справа налево*.

```
string strval("1100");
bitset<32> bitvec4(strval);
```

Битовая схема набора `bitvec4` имеет второй и третий биты в состоянии `1`, а остальные в состоянии `0`. Когда строка содержит меньше символов, чем размер набора битов, для старших битов используется значение нуль.

Соглашения о нумерации строк и наборов битов инверсно взаимосвязаны: самый правый символ строки (обладающий самым большим значением индекса) используется для инициализации самого младшего бита в наборе битов (бит с индексом 0). При инициализации набора битов из строки очень важно помнить об этом различии.

В качестве исходного значения для набора битов можно использовать не всю строку, а ее часть.

```
string str("1111111000000011001101");
bitset<32> bitvec5(str, 5, 4); // 4 бита, начиная от str[5], 1100
bitset<32> bitvec6(str, str.size() - 4); // использовать 4
                                         // последних символа
```

Здесь набор битов `bitvec5` инициализирован частью строки `str`, начинающейся с символа `str[5]` и распространяющейся на четыре следующие позиции. Как обычно при инициализации, набор битов `bitvec5` заполняется значениями подстроки из 4 символов начиная с пятого, т.е. значением `1100`, а биты в остальных позициях заполняются нулями. В третьей строке кода использован подход, позволяющий применить символы от указанной позиции до конца строки. В данном случае, для инициализации четырех младших битов набора `bitvec6`, используются символы строки `str` начиная с четвертого от конца. Остальные биты набора `bitvec6` инициализируются нулями. Эти случаи инициализации представлены на рис. 3.1.

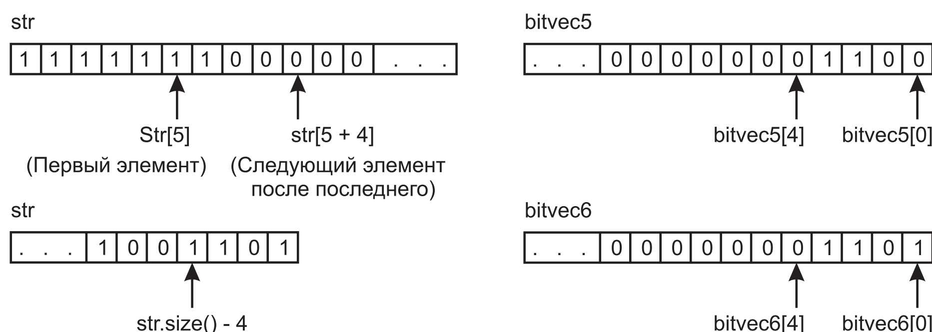


Рис. 3.1. Инициализация набора битов из строки

3.5.2. Операции с наборами битов

В классе `bitset` определены несколько функций табл. 3.7, обеспечивающих выполнение с набором битов таких операций, как проверка состояния и установка одного или нескольких битов.

Таблица 3.7. Операции с наборами битов

`b.any()`	Все ли биты набора `b` установлены?
`b.none()`	Нет ли в наборе `b` установленных битов?
`b.count()`	Количество установленных битов в наборе `b`
`b.size()`	Количество битов в наборе `b`
`b[pos]`	Доступ к биту номер `pos` в наборе `b`

Окончание табл. 3.7

`b.test(pos)`	Установлен ли бит номер `pos` в наборе `b`?
`b.any()`	Все ли биты набора `b` установлены?
`b.set()`	Устанавливает все биты в наборе `b`
`b.set(pos)`	Устанавливает бит номер `pos` в наборе `b`
`b.reset()`	Сбрасывает все биты в наборе `b`
`b.reset(pos)`	Сбрасывает бит номер `pos` в наборе `b`
`b.flip()`	Изменяет состояние всех битов в наборе `b`
`b.flip(pos)`	Изменяет состояние бита номер `pos` в наборе `b`
`b.to_ulong()`	Возвращает число типа `unsigned long` с теми же битами, что и в наборе `b`
`os << b`	Передает биты набора `b` в поток `os`

Проверка набора битов в целом

Функция `any()` возвращает значение `true`, если один или несколько битов набора установлены, т.е. находятся в состоянии `1`. Функция `none()`, наоборот, возвращает значение `true`, если все биты объекта сброшены, т.е. находятся в состоянии `0`.

```
bitset<32> bitvec; // 32 бита, все нули
bool is_set = bitvec.any();      // false, все биты - нули
bool is_not_set = bitvec.none(); // true, все биты - нули
```

Если необходимо узнать, сколько битов установлено, можно воспользоваться функцией `count()`, которая возвращает количество установленных битов.

```
size_t bits_set = bitvec.count();  // возвращает количество
                                   // установленных битов
```

Функция `count()` возвращает библиотечный *тип* `size_t`, который определен в заголовке `cstddef` библиотеки С. Версия ее заголовка для языка С++ имеет имя `stddef.h`. Это машинно-зависимый *беззнаковый* тип, который гарантированно велик, чтобы содержать размер объекта в памяти.

Функция `size()`, подобно одноименной функции векторов и строк, возвращает общее количество битов в наборе. Возвращаемое значение имеет тип `size_t`.

```
size_t sz = bitvec.size();  // возвращает число 32
```

Доступ к битам в наборе битов

Оператор индексирования позволяет читать и записывать значения битов в позиции указанные индексом. Его можно также использовать для проверки или установки значения определенного бита.

```
// присвоить 1 каждому биту диапазона
for (int index = 0; index != 32; index += 2)
    bitvec[index] = 1;
```

Этот цикл установит 32 первых бита набора `bitvec`.

Чтобы проверить или установить определенное битовое значение, совместно с оператором индексирования можно использовать функции `set()`, `test()` и `reset()`.

```
// аналогичный цикл, использующий функцию set()
for (int index = 0; index != 32; index += 2)
    bitvec.set(index);
```

Чтобы выяснить, установлен ли определенный бит, можно либо воспользоваться функцией `test()`, либо проверить значение, возвращаемое оператором индексирования.

```
if (bitvec.test(i))
    // bitvec[i] установлен
// аналогичная проверка при помощи индексирования
if (bitvec[i])
    // bitvec[i] установлен
```

Результатом проверки возвращенного оператором индексирования значения будет `true`, если бит установлен (`1`), или `false` — если бит сброшен (`0`).

Установка значений набора битов в целом

Функции `set()` и `reset()` также можно использовать для установки или сброса всех битов набора объекта соответственно.

```
bitvec.reset(); // сбросить все биты в 0
bitvec.set();   // установить все биты в 1
```

Функция `flip()` инвертирует значение отдельного бита или всего набора битов.

```
bitvec.flip(0);    // инвертирует значение первого бита
bitvec[0].flip(); // тоже инвертирует первый бит
bitvec.flip();     // инвертирует значения всех битов
```

Получение значения из набора битов

Функция `to_ulong()` возвращает значение типа `unsigned long`, которое содержит ту же битовую схему, что и набор битов. Эту функцию можно использовать только в том случае, если размер набора битов меньше или равен размеру типа `unsigned long`.

```
unsigned long ulong = bitvec3.to_ulong();
cout << "ulong = " << ulong << endl;
```

Функция `to_ulong()` была разработана в языке C для передачи содержимого из набора битов еще до появления стандарта языка C++. Если набор содержит больше битов, чем помещается в переменную типа `unsigned long`, во время выполнения будет передано исключение. Знакомство с исключениями начинается в разделе 6.13 (стр. 241), а более подробная информация о них приведена в разделе 17.1 (стр. 720).

Отображение битов

Для отображения битовой схемы, содержащейся в наборе битов, можно воспользоваться оператором вывода (`<<`).

```
bitset<32> bitvec2 (0xffff); // биты 0...15 установлены в 1, а
                             // биты 16...31 сброшены в 0
cout << "bitvec2: " << bitvec2 << endl;
```

В результате на экране будет отображено.

```
bitvec2: 00000000000000001111111111111111
```

Использование побитовых операторов

Класс `bitset` поддерживает также встроенные *побитовые операторы* (bitwise operator). Как определено в языке, эти операторы применимы к целочисленным операндам. Они выполняют операции, подобные операциям с набором данных, описанным в этом разделе. Более подробная информация об этих операторах приведена в разделе 5.3 (стр. 179).

Упражнения раздела 3.5.2

Упражнение 3.23. Объясните, какую битовую схему содержит каждый из следующих наборов битов.

```
(a) bitset<64> bitvec(32);
(b) bitset<32> bv(1010101);
(c) string bstr; cin >> bstr; bitset<8>bv(bstr);
```

Упражнение 3.24. Предположим, что существует последовательность1, 2, 3, 5, 8, 13, 21. Инициализируйте набор `bitset<32>` так, чтобы в каждой из позиций, указанной числом этой последовательности, бит был установлен (`1`). В качестве альтернативы создайте пустой набор битов и напишите небольшую программу, устанавливающую каждый из соответствующих битов.

Резюме

В библиотеке определено несколько высокоуровневых абстрактных типов данных, включая строки и векторы. Класс `string` предоставляет символьные строки переменной длины, а шаблон `vector` позволяет создавать коллекции объектов одинакового типа.

Итераторы обеспечивают косвенный доступ к объектам, хранимым в контейнере. Итераторы применимы для доступа и перемещения между элементами строк и векторов.

В следующей главе рассматриваются массивы и указатели, являющиеся встроенными типами языка. Эти типы являются низкоуровневыми аналогами библиотечных векторов и строк. Как правило, предпочтительней использовать библиотечные классы, а не встроенные.

Термины

Абстрактный тип данных (abstract data type). Тип, представление которого скрыто. Используя абстрактный тип, достаточно знать только то, какие операции он поддерживает.

Арифметические действия с итераторами (iterator arithmetic). Арифметические операции, которые можно применять к некоторым, но не всем, типам итераторов. Добавление и вычитание целого числа из итератора приводит к изменению позиции итератора на соответствующее количество элементов вперед или назад от исходного. Вычитание двух итераторов позволяет вычислить дистанцию между ними. Арифметические действия допустимы лишь для итераторов, относящихся к элементам того же контейнера.

Заголовок `cctype`. Унаследованный от языка С заголовок, который содержит определения функций для проверки символьных значений. Список наиболее популярных из них приведен в табл. 3.3 на стр. 112.

Индекс (index). Значение, используемое в операторе индексирования для указания элемента, возвращаемого из строки или вектора.

Инициализирующее значение (value initialized). Используется, когда при инициализации контейнера указано только количество элементов контейнера, без явного указания значений. Элементы инициализируются копией значения, созданного компилятором. Если контейнер

предназначен для встроенного типа, в элементы копируется нулевое значение. Для классов инициализирующее значение создает стандартный конструктор класса. Элементы контейнера, являющиеся объектами класса, могут быть инициализированы только тогда, когда класс имеет стандартный конструктор.

Итератор после конца (off-the-end iterator). Итератор, возвращаемый функцией `end()`. Он указывает не на последний существующий элемент контейнера, а на позицию за его концом, т.е. на несуществующий элемент.

Класс `bitset` (набор битов). Определенный в стандартной библиотеке класс, объект которого содержит коллекцию битов и позволяет выполнять с ним операции по проверке и установке значений.

Контейнер (container). Тип, объекты которого способны содержать коллекцию объектов определенного типа.

Младшие биты (low-order). Биты набора, обладающие самыми маленькими индексами.

Объявление `using`. Позволяет сделать имя, определенное в пространстве имен, доступным непосредственно в коде.

```
using пространствоимен::имя;
```

Теперь *имя* можно использовать без префикса *пространствоимен*`::`.

Оператор `*`. Для итераторов определен оператор обращения к значению, позволяющий получить объект, на который указывает итератор. Этот оператор возвращает l-значение, поэтому его можно использовать как левый операнд присвоения. Присвоение значения результату выполнения этого оператора приводит к присвоению нового значения соответствующему элементу.

Оператор `::`. Оператор области видимости. Находит имя его правого операнда в области видимости, указанной левым операндом. Используется для доступа к именам из пространства имен, например `std::cout`, где имя `cout` принадлежит пространству имен `std`. Аналогично он используется и для доступа к именам определенным в классе, например `string::size_type`, где тип `size_type` определен в классе `string`.

Оператор `[]`. Перегруженный оператор индексирования, определенный для строк, векторов и наборов битов. Он получает два операнда: левый (имя объекта) и правый (индекс). Оператор выбирает элемент, позиция которого указана индексом. Нумерация элементов при индексировании начинается с нуля, т.е. первым является элемент номер `0`, а последним — элемент номер `obj.size()-1`. Индексирование возвращает l-значение, поэтому его можно использовать как левый операнд присвоения. В результате новое значение будет присвоено элементу, указанному по индексу.

Оператор `++`. Для итераторов некоторых типов определен оператор инкремента, который "добавляет единицу", перемещая итератор на следующий элемент.

Оператор `<<`. Для библиотечных типов `string` и `bitset` определен оператор вывода. Строковый оператор вывода выводит на стандартное устройство вывода символы строки, а битовый — битовую схему.

Оператор `>>`. Для библиотечных типов `string` и `bitset` определен оператор ввода. Строковый оператор ввода читает разграниченные пробелами последовательности символов и сохраняет их в строковой переменной, указанной правым операндом. Оператор ввода для набора битов читает битовую последовательность и записывает ее в набор битов, указанный правым операндом.

Старшие биты (high-order). Биты набора, обладающие самыми большими индексами.

Тип `difference_type`. Определенный в классе вектора знаковый целочисленный тип, переменная которого способна содержать дистанцию между двумя любыми итераторами.

Тип `iterator` (итератор). Тип, используемый при переборе элементов контейнера и обращении к ним.

Тип size_t. Машинно-зависимый беззнаковый целочисленный тип, определенный в заголовке cstddef. Является достаточно большим, чтобы содержать размер самого большого возможного массива.

Тип size_type. Определенный для классов строк и векторов беззнаковый тип, переменные которого достаточно велики, чтобы содержать размер любой строки или вектора.

Функция empty(). Функция, определенная в строковых и векторных классах. Она возвращает логическое значение (типа bool), которое указывает, имеются ли в строке символы или элементы в векторе. Возвращает значение true, если размер нулевой, или значение false в противном случае.

Функция getline(). Определенная в заголовке string функция, которой передают поток istream и строковую переменную. Функция читает данные из потока до тех пор, пока не встретится символ новой строки, а прочитанное сохраняет в строковой переменной. Функция возвращает поток istream. Символ новой строки в прочитанных данных отбрасывается.

Функция push_back(). Определенная в классе вектора функция, которая добавляет элементы в его конец.

Функция size(). Определенная в библиотечных классах строк, векторов и наборов битов функция, которая возвращает количество символов, элементов или битов соответственно. Строковые и векторные функции возвращают значение типа size_type. Например, функция size() класса string возвращает значение типа string::size_type. Функция size() класса bitset возвращает значение типа size_t.

Шаблон класса (class template). Проект, согласно которому может быть создано множество специализированных классов. Чтобы применить шаблон класса, необходимо указать фактический класс или используемое значение (или значения). Например, vector — это шаблон, объекты классов которого способны содержать объекты указанного типа. При создании вектора необходимо указать, объекты какого именно типа будет содержать данный вектор. Вектор, объявленный как vector<int>, способен содержать целые числа, а вектор, объявленный как vector<string>, — строки и т.д.