ГЛАВА 5

ОПЕРАТОРЫ

В этой главе...

Подобно большинству языков, язык C++ предоставляет операторы для условного выполнения кода, циклы, позволяющие многократно выполнять те же фрагменты кода, и операторы перехода, прерывающие поток выполнения. В данной главе операторы, поддерживаемые языком C++, рассматриваются более подробно.

Операторы (statement) выполняются последовательно. За исключением самых простых программ последовательного выполнения недостаточно. Поэтому язык C++ определяет также набор операторов *управления потоком* (flow of control), обеспечивающих более сложные пути выполнения кода.

5.1. Простые операторы

Большинство операторов в языке C++ заканчиваются точкой с запятой. Выражение типа `ival + 5` становится *оператором выражения* (expression statement), завершающимся точкой с запятой. Операторы выражения составляют вычисляемую часть выражения.

```
ival + 5;     // оператор выражения (хоть и бесполезный)
cout << ival; // оператор выражения
```

Первое выражение бесполезно: результат вычисляется, но не присваивается, а следовательно, никак не используется. Как правило, выражения содержат операторы, результат вычисления которых влияет на состояние программы. К таким операторам относятся присвоение, инкремент, ввод и вывод.

Пустые операторы

Самая простая форма оператора — это *пустой* (empty), или *нулевой*, *оператор* (null statement). Он представляет собой одиночный символ точки с запятой (`;`).

```
; // пустой оператор
```

Пустой оператор используется в случае, когда синтаксис языка требует наличия оператора, а логика программы — нет. Как правило, это происходит в случае, когда вся работа цикла осуществляется в его условии. Например, можно организовать ввод, игнорируя все прочитанные данные, пока не встретится определенное значение:

```
// читать, пока не встретится конец файла или значение,
// равное содержимому переменной sought
while (cin >> s && s != sought)
    ; // пустой оператор
```

В условии значение считывается со стандартного устройства ввода, и объект `cin` неявно проверяется на успешность чтения. Если чтение прошло успешно, во второй части условия проверяется, не равно ли полученное значение содержимому переменной `sought`. Если искомое значение найдено, цикл `while` завершается, в противном случае его условие проверяется снова, начиная с чтения следующего значения из объекта `cin`.

Случаи применения пустого оператора следует комментировать, чтобы любой, кто читает код, мог сразу понять, что оператор пропущен преднамеренно.

Остерегайтесь пропущенных и лишних точек с запятой

Поскольку пустой оператор является вполне допустимым, он может располагаться везде, где ожидается оператор. Поэтому лишний символ точки с запятой, который может показаться явно недопустимым, на самом деле является не более, чем пустым оператором. Приведенный ниже фрагмент кода содержит два оператора: оператор выражения и пустой оператор.

```
ival = v1 + v2;; // ok: вторая точка с запятой - это лишний
                 // пустой оператор
```

Хотя ненужный пустой оператор зачастую безопасен, дополнительная точка с запятой после условия цикла `while` или оператора `if` может решительно изменить поведение кода. Например, следующий цикл будет выполняться бесконечно:

```
// катастрофа: лишняя точка с запятой превратила тело цикла
// в пустой оператор
while (iter != svec.end()) ; // тело цикла while пусто!
    ++iter;                  // инкремент не является частью цикла
```

Несмотря на отступ, выражение с оператором инкремента не является частью цикла. Тело цикла — это пустой оператор, обозначенный символом точки с запятой непосредственно после условия.

Лишний пустой оператор не всегда безопасен.

Составные операторы (блоки)

Составной оператор (compound statement), обычно называемый *блоком* (block), представляет собой последовательность операторов, заключенных в фигурные скобки. Блок операторов обладает собственной областью видимости (см. раздел 2.2.4, стр. 82). Объявленные в блоке имена доступны только в данном блоке и блоках, вложенных в него. Как обычно, имя видимо только с того момента, когда оно определено, и до конца блока включительно.

Составные операторы применяются в случае, когда язык требует одного оператора, а логика программы нескольких. Например, тело цикла `while` или `for` составляет один оператор. Но в теле цикла зачастую необходимо выполнить несколько операторов. Заключив необходимые операторы в фигурные скобки, можно получить блок, рассматриваемый как единый оператор.

Для примера вернемся к циклу `while` из кода в разделе 1.4.1 (стр. 38).

```
while (val <= 10) {
    sum += val; // присвоить sum сумму val и sum
    ++val;      // добавить 1 к val
}
```

Логика программы нуждалась в двух операторах, но цикл `while` способен содержать только один оператор. Заключив эти операторы в фигурные скобки, получаем один (составной) оператор.

Блок *не завершают* точкой с запятой.

Как и в случае с пустым оператором, вполне можно создать пустой блок. Для этого используется пара фигурных скобок без операторов:

```
while (cin >> s && s != sought)
    { } // пустой блок
```

Упражнения раздела 5.1

Упражнение 5.1. Что такое пустой оператор? Когда его можно использовать?

Упражнение 5.2. Что такое блок? Когда его можно использовать?

Упражнение 5.3. Используя оператор запятой (см. раздел 4.10, стр. 217), перепишите цикл `while` из раздела 1.4.1 (стр. 38) так, чтобы блок стал больше не нужен. Объясните, улучшило ли это удобочитаемость кода.

5.2. Операторная область видимости

Переменные можно определять в управляющих структурах операторов `if`, `switch`, `while` и `for`. Переменные, определенные в управляющей структуре, видимы только в пределах этого оператора и выходят из области видимости по его завершении.

```
while (int i = get_num()) // i создается и инициализируется при
                          // каждой итерации
    cout << i << endl;
i = 0; // ошибка: переменная i недоступна вне цикла
```

Если к значению управляющей переменной необходимо обращаться впоследствии, то ее следует определить вне оператора.

```
// найти первый отрицательный элемент
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // известно, что все элементы v больше или равны нулю
```

Значение объекта, определенного в управляющей структуре, используется самой структурой. Поэтому такие переменные следует инициализировать.

Упражнения раздела 5.2

Упражнение 5.4. Объясните каждый из следующих примеров, а также устраните все обнаруженные проблемы.

```
(a) while (string::iterator iter != s.end()) { /* ... */ }
(b) while (bool status = find(word)) { /* ... */ }
    if (!status) { /* ... */ }
```

5.3. Условные операторы

Язык C++ предоставляет два оператора, обеспечивающих условное выполнение. Оператор `if` разделяет поток выполнения на основании условия. Оператор `switch` вычисляет результат целочисленного выражения и на его основании выбирает один из нескольких путей выполнения.

5.3.1. Оператор `if`

Оператор `if` выполняет один из двух операторов в зависимости от истинности своего условия. Существуют две формы оператора `if`: с разделом `else` и без него. Синтаксис простой формы оператора `if` имеет следующий вид:

```
if (условие)
    оператор
Оператор if else имеет следующую форму:
if (условие)
    оператор
else
    оператор2
```

В обеих версиях *условие* заключается в круглые скобки. *Условие* может быть выражением или инициализирующим объявлением переменной (см. раздел 5.2, стр. 236). Тип выражения или переменной должен быть преобразуем в тип `bool` (см. раздел 4.11, стр. 218). Как обычно, и *оператор*, и *оператор2* могут быть блоком.

Если *условие* истинно, *оператор* выполняется. По завершении оператора выполнение продолжается после оператора `if`.

Если *условие* ложно, *оператор* пропускается. В простом операторе `if` выполнение продолжается после оператора `if`, а в операторе `if else` выполняется *оператор2*.

Использование оператора `if else`

Для иллюстрации оператора `if else` вычислим символ оценки по ее числу. Подразумевается, что числовые значения оценок находятся в диапазоне от нуля до 100 включительно. Оценка 100 получает знак “A++”, оценка ниже 60 — “F”, а остальные группируются по десять: от 60 до 69 — “D”, от 70 до 79 — “C” и т.д. Для хранения возможных символов оценок используем вектор:

```
vector<string> scores = {"F", "D", "C", "B", "A", "A++"};
```

Для решения этой проблемы можно использовать оператор `if else`, чтобы выполнять разные действия проходных и не проходных отметок.

```
// если оценка меньше 60 - это F, в противном случае вычислять индекс
string lettergrade;
if (grade < 60)
    lettergrade = scores[0];
else
    lettergrade = scores[(grade - 50)/10];
```

В зависимости от значения переменной `grade` оператор выполняется либо после части `if`, либо после части `else`. В части `else` вычисляется индекс оценки уже без неудовлетворительных. Затем усекающее остаток целочисленное деление (см. раздел 4.2, стр. 196) используется для вычисления соответствующего индекса вектора `scores`.

Вложенные операторы `if`

Чтобы сделать программу интересней, добавим к удовлетворительным отметкам плюс или минус. Плюс присваивается оценкам, заканчивающимся на 8 или 9, а минус — заканчивающимся на 0, 1 или 2.

```
if (grade % 10 > 7)
    lettergrade += '+'; // оценки, заканчивающиеся на 8 или 9, получают +
else if (grade % 10 < 3)
    lettergrade += '-'; // оценки, заканчивающиеся на 0, 1 и 2, получают -
```

Для получения остатка и принятия на основании его решения, добавлять ли плюс или минус, используем оператор деления по модулю (см. раздел 4.2, стр. 196).

Теперь добавим код, присваивающий плюс или минус, к коду, выбирающему символ оценки:

```
// если оценка неудовлетворительна, нет смысла проверять ее на + или -
if (grade < 60)
    lettergrade = scores[0];
else {
    lettergrade = scores[(grade - 50)/10]; // выбрать символ оценки
    if (grade != 100) // добавлять + или -, только если это не A++
        if (grade % 10 > 7)
            lettergrade += '+'; // оценки, заканчивающиеся на 8 или 9,
                                // получают +
        else if (grade % 10 < 3)
            lettergrade += '-'; // оценки, заканчивающиеся на 0, 1 и 2,
                                // получают -
}
```

Обратите внимание, что два оператора, следующих за первым оператором `else`, заключены в блок. Если переменная `grade` содержит значение `60` или больше, возможны два действия: выбор символа оценки из вектора `scores` и, при условии, добавление плюса или минуса.

Следите за фигурными скобками

Когда несколько операторов следует выполнить как блок, довольно часто забывают фигурные скобки. В следующем примере, вопреки отступу, код добавления плюса или минуса выполняется безусловно:

```
if (grade < 60)
    lettergrade = scores[0];
else // ошибка: отсутствует фигурная скобка
    lettergrade = scores[(grade - 50)/10];
    // несмотря на внешний вид, без фигурной скобки, этот код
    // выполняется всегда
    // неудовлетворительным оценкам ошибочно присваивается - или +
    if (grade != 100)
        if (grade % 10 > 7)
            lettergrade += '+'; // оценки, заканчивающиеся на 8 или 9,
```

```
                                // получают +
        else if (grade % 10 < 3)
            lettergrade += '-'; // оценки, заканчивающиеся на 0, 1 и 2,
                                // получают -
```

Найти такую ошибку бывает очень трудно, поскольку программа выглядит правильно.

Во избежание подобных проблем некоторые стили программирования рекомендуют всегда использовать фигурные скобки после оператора `if` или `else` (а также вокруг тел циклов `while` и `for`).

Это позволяет избежать подобных ошибок. Это также означает, что фигурные скобки уже есть, если последующие изменения кода потребуют добавления операторов.

У большинства редакторов и сред разработки есть инструменты автоматического выравнивания исходного кода в соответствии с его структурой. Такие инструменты всегда следует использовать, если они доступны.

Потерянный оператор `else`

Когда один оператор `if` вкладывается в другой, ветвей `if` может оказаться больше, чем ветвей `else`. Действительно, в нашей программе оценивания четыре оператора `if` и два оператора `else`. Возникает вопрос: как установить, которому оператору `if` принадлежит данный оператор `else`?

Эта проблема, обычно называемая *потерянным оператором* `else` (dangling `else`), присуща многим языкам программирования, предоставляющим операторы `if` и `if else`. Разные языки решают эту проблему по-разному. В языке C++ неоднозначность решается так: оператор `else` принадлежит ближайшему расположенному выше оператору `if` без `else`.

Неприятности происходят также, когда код содержит больше операторов `if`, чем ветвей `else`. Для иллюстрации проблемы перепишем внутренний оператор `if else`, добавляющий плюс или минус, на основании различных наборов условий:

```
// Ошибка: порядок выполнения НЕ СООТВЕТСТВУЕТ отступам; ветвь else
// принадлежит внутреннему if
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // оценки, заканчивающиеся на 8 или 9,
                            // получают +
else
    lettergrade += '-';     // оценки, заканчивающиеся на 3, 4, 5, 6,
                            // получают - !
```

Отступ в данном коде подразумевает, что оператор `else` предназначен для внешнего оператора `if`, т.е. он выполняется, когда значение `grade` заканчива-

ется цифрой меньше 3. Однако, несмотря на наши намерения и вопреки отступу, ветвь else является частью внутреннего оператора if. Этот код добавляет '-' к оценкам, заканчивающимся на 3–7 включительно! Правильно выровненный, в соответствии с правилами выполнения, этот код выглядел бы так:

```
// отступ соответствует порядку выполнения, но не намерению программиста
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // оценки, заканчивающиеся на 8 или 9,
                            // получают +
    else
        lettergrade += '-'; // оценки, заканчивающиеся на 3, 4, 5, 6,
                            // получают - !
```

Контроль пути выполнения при помощи фигурных скобок

Заключив внутренний оператор if в блок, можно сделать ветвь else частью внешнего оператора if:

```
// добавлять плюс для оценок, заканчивающихся на 8 или 9, а минус для
// заканчивающихся на 0, 1 или 2
if (grade % 10 >= 3) {
    if (grade % 10 > 7)
        lettergrade += '+'; // оценки, заканчивающиеся на 8 или 9,
                            // получают +
    } else // скобки обеспечивают else для внешнего if
        lettergrade += '-'; // оценки, заканчивающиеся на 0, 1 и 2,
                            // получают -
```

Операторы не распространяются за границы блока, поэтому внутренний цикл if заканчивается на закрывающей фигурной скобке перед оператором else. Оператор else не может быть частью внутреннего оператора if. Теперь ближайшим свободным оператором if оказывается внешний, как и предполагалось изначально.

Упражнения раздела 5.3.1

Упражнение 5.5. Напишите собственную версию программы преобразования числовой оценки в символ с использованием оператора if else.

Упражнение 5.6. Перепишите программу оценки так, чтобы использовать условный оператор (см. раздел 4.7, стр. 208) вместо оператора if else.

Упражнение 5.7. Исправьте ошибки в каждом из следующих фрагментов кода:

```
(a) if (ival1 != ival2)
        ival1 = ival2
    else ival1 = ival2 = 0;
(b) if (ival < minval)
        minval = ival;
        occurs = 1;
```

```
(c) if (int ival = get_value())
        cout << "ival = " << ival << endl;
    if (!ival)
        cout << "ival = 0\n";
(d) if (ival = 0)
        ival = get_value();
```

Упражнение 5.8. Что такое "потерянный оператор else"? Как в языке C++ определяется принадлежность ветви else?

5.3.2. Оператор `switch`

Оператор `switch` предоставляет более удобный способ выбора одной из множества альтернатив. Предположим, например, что необходимо рассчитать, как часто встречается каждая из пяти гласных в некотором фрагменте текста. Программа будет иметь следующую логику.

- Читать каждый введенный символ.
- Сравнить каждый символ с набором искомых гласных.
- Если символ соответствует одной из гласных букв, добавить `1` к соответствующему счетчику.
- Отобразить результаты.

Программа должна отобразить результаты в следующем виде:

```
Number of vowel a: 3195
Number of vowel e: 6230
Number of vowel i: 3102
Number of vowel o: 3289
Number of vowel u: 1033
```

Для непосредственного решения этой задачи можно использовать оператор `switch`.

```
// инициализировать счетчики для каждой гласной
unsigned aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
char ch;
while (cin >> ch) {
    // если ch - гласная, увеличить соответствующий счетчик
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
```

```
        case 'u':
            ++uCnt;
            break;
    }
}
// вывод результата
cout << "Number of vowel a: \t" << aCnt << '\n'
     << "Number of vowel e: \t" << eCnt << '\n'
     << "Number of vowel i: \t" << iCnt << '\n'
     << "Number of vowel o: \t" << oCnt << '\n'
     << "Number of vowel u: \t" << uCnt << endl;
```

Оператор `switch` вычисляет результат выражения, расположенного за ключевым словом `switch`. Это выражение может быть объявлением инициализированной переменной (см. раздел 5.2, стр. 236). Выражение преобразуется в целочисленный тип. Результат выражения сравнивается со значением, ассоциированным с каждым оператором `case`.

Если результат выражения соответствует значению метки `case`, выполнение кода начинается с первого оператора после *этой* метки. В принципе выполнение кода продолжается до конца оператора `switch`, но оно может быть прервано оператором `break`.

Более подробно оператор `break` рассматривается в разделе 5.5.1 (стр. 254), а пока достаточно знать, что он прерывает текущий поток выполнения. В данном случае оператор `break` передает управление первому оператору после оператора `switch`. Здесь оператор `switch` является единственным оператором в теле цикла `while`, поэтому его прерывание возвращает контроль окружающему оператору `while`. Поскольку в нем нет никаких других операторов, цикл `while` продолжается, если его условие выполняется.

Если соответствия не найдено, выполнение сразу переходит к первому оператору после `switch`. Как уже упоминалось, в этом примере выход из оператора `switch` передает управление условию цикла `while`.

Ключевое слово `case` и связанное с ним значение называют также *меткой* `case` (`case` label). Значением каждой метки `case` является константное выражение (см. раздел 2.4.4, стр. 103).

```
char ch = getVal();
int ival = 42;
switch(ch) {
    case 3.14: // ошибка: метка case не целое число
    case ival: // ошибка: метка case не константа
    // ...
```

Одинаковые значения меток `case` недопустимы. Существует также специальная метка `default`, рассматриваемая на стр. 244.

Порядок выполнения в операторе `switch`

Важно понимать, как управление передается между метками `case`. После обнаружения соответствующей метки `case` выполнение начинается с нее и продолжается далее через все остальные метки до конца или пока выполнение не будет прервано явно. Во избежание выполнения последующих разделов `case` выполнение следует прервать явно, поэтому оператор `break` обычно является последним оператором перед следующей меткой `case`.

Однако возможны ситуации, когда необходимо именно стандартное поведение оператора `switch`. У каждой метки `case` может быть только одно значение, однако две или более метки могут совместно использовать единый набор действий. В таких ситуациях достаточно пропустить оператор `break` и позволить программе пройти несколько меток `case`.

Например, можно было бы посчитать общее количество гласных так:

```
unsigned vowelCnt = 0;
// ...
switch (ch)
{
    // для инкремента vowelCnt подойдет любая буква a, e, i, o или u
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        ++vowelCnt;
        break;
}
```

Здесь расположено несколько меток `case` подряд без оператора `break`. Теперь при любой гласной в переменной `ch` будет выполняться тот же код.

Поскольку язык C++ не требует обязательно располагать метки `case` в отдельной строке, весь диапазон значений можно указать в одной строке:

```
switch (ch)
{
    // альтернативный допустимый синтаксис
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
}
```

Случаи, когда оператор `break` пропускают преднамеренно, довольно редки, поэтому их следует обязательно комментировать, объясняя логику действий.

Пропуск оператора `break` — весьма распространенная ошибка

Весьма распространено заблуждение, что выполняются только те операторы, которые связаны с совпавшей меткой `case`. Вот пример *неправильной* реализации подсчета гласных в операторе `switch`:

```
// внимание: преднамеренно неправильный код!
switch (ch) {
    case 'a':
        ++aCnt; // Упс! Необходим оператор break
    case 'e':
        ++eCnt; // Упс! Необходим оператор break
    case 'i':
        ++iCnt; // Упс! Необходим оператор break
    case 'o':
        ++oCnt; // Упс! Необходим оператор break
    case 'u':
        ++uCnt;
}
```

Чтобы понять происходящее, предположим, что значением переменной `ch` является `'e'`. Выполнение переходит к коду после метки `case 'e'`, где происходит инкремент переменной `eCnt`. Выполнение *продолжается* далее через метки `case`, увеличивая также значения переменных `iCnt`, `oCnt` и `uCnt`.

Несмотря на то что оператор `break` и не обязателен после последней метки оператора `switch`, использовать его все же рекомендуется. Ведь если впоследствии оператор `switch` будет дополнен еще одной меткой `case`, отсутствие оператора `break` после прежней последней метки не создаст проблем.

Метка `default`

Операторы после *метки* `default` выполняются, если ни одна из меток `case` не соответствует значению выражения оператора `switch`. Например, в рассматриваемый код можно добавить счетчик негласных букв. Значение этого счетчика по имени `otherCnt` будет увеличиваться в случае `default`:

```
// если ch гласная, увеличить соответствующий счетчик
switch (ch) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
    default:
        ++otherCnt;
        break;
    }
}
```

В этой версии, если переменная `ch` не содержит гласную букву, управление перейдет к метке `default` и увеличится значение счетчика `otherCnt`.

Раздел `default` имеет смысл создавать всегда, даже если в нем не происходит никаких действий. Впоследствии это однозначно укажет читателю кода, что случай `default` не был забыт, т.е. для остальных случаев никаких действий предпринимать не нужно.

Метка не может быть автономной; она должна предшествовать оператору или другой метке `case`. Если оператор `switch` заканчивается разделом `default`, в котором не осуществляется никаких действий, за меткой `default` должен следовать пустой оператор или пустой блок.

Определение переменной в операторе `switch`

Как уже упоминалось, выполнение оператора `switch` способно переходить через метки `case`. Когда выполнение переходит к некой метке `case`, весь расположенный выше код оператора `switch` будет проигнорирован. Факт игнорирования кода поднимает интересный вопрос: что будет, если пропущенный код содержит определение переменной?

Ответ прост: недопустим переход с места, где переменная с инициализатором уже вышла из области видимости к месту, где эта переменная находится в области видимости.

```
case true:
    // этот оператор switch недопустим, поскольку инициализацию
    // можно обойти
    string file_name; // ошибка: выполнение обходит неявно
                      // инициализированную переменную
    int ival = 0;     // ошибка: выполнение обходит неявно
                      // инициализированную переменную
    int jval;         // ok: поскольку jval не инициализирована
    break;
case false:
    // ok: jval находится в области видимости, но она не инициализирована
    jval = next_num();      // ok: присвоить значение jval
    if (file_name.empty()) // file_name находится в области видимости, но
                           // она не инициализирована
    // ...
```

Если бы этот код был допустим, то любой переход к случаю `false` обходил бы инициализацию переменных `file_name` и `ival`, но они оставались бы в области видимости и код вполне мог бы использовать их. Однако эти переменные не были бы инициализированы. В результате язык не позволяет перепрыгивать через инициализацию, если инициализированная переменная находится в области видимости в пункте, к которому переходит управление.

Если необходимо определить и инициализировать переменную для некоего случая `case`, то сделать это следует в блоке, гарантируя таким образом, что переменная выйдет из области видимости перед любой последующей меткой.

```
case true:
    {
        // ok: оператор объявления в пределах операторного блока
        string file_name = get_file_name();
        // ...
    }
    break;
case false:
    if (file_name.empty()) // ошибка: file_name вне области видимости
```

Упражнения раздела 5.3.2

Упражнение 5.9. Напишите программу, использующую серию операторов `if` для подсчета количества гласных букв в тексте, прочитанном из потока `cin`.

Упражнение 5.10. Программа подсчета гласных имеет одну проблему: она не учитывает заглавные буквы как гласные. Напишите программу, которая подсчитывает гласные буквы как в верхнем, так и в нижнем регистре. То есть значение счетчика `aCnt` должно увеличиваться при встрече как символа `'a'`, так и символа `'A'` (аналогично для остальных гласных букв).

Упражнение 5.11. Измените рассматриваемую программу так, чтобы она подсчитывала также количество пробелов, символов табуляции и новой строки.

Упражнение 5.12. Измените рассматриваемую программу так, чтобы она подсчитывала количество встреченных двухсимвольных последовательностей: `ff`, `fl` и `fi`.

Упражнение 5.13. Каждая из приведенных ниже программ содержит распространенную ошибку. Выявите и исправьте каждую из них.

Код для упражнения 5.13

```
(a) unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
    char ch = next_text();
    switch (ch) {
        case 'a': aCnt++;
        case 'e': eCnt++;
        default: iouCnt++;
    }
(b) unsigned index = some_value();
    switch (index) {
        case 1:
            int ix = get_value();
            ivec[ ix ] = index;
            break;
        default:
            ix = ivec.size()-1;
            ivec[ ix ] = index;
    }
```

```
(c) unsigned evenCnt = 0, oddCnt = 0;
    int digit = get_num() % 10;
    switch (digit) {
        case 1, 3, 5, 7, 9:
            oddcnt++;
            break;
        case 2, 4, 6, 8, 10:
            evencnt++;
            break;
    }
(d) unsigned ival=512, jval=1024, kval=4096;
    unsigned bufsize;
    unsigned swt = get_bufCnt();
    switch(swt) {
        case ival:
            bufsize = ival * sizeof(int);
            break;
        case jval:
            bufsize = jval * sizeof(int);
            break;
        case kval:
            bufsize = kval * sizeof(int);
            break;
    }
```

5.4. Итерационные операторы

Итерационные операторы (iterative statement), называемые также *циклами* (loop), обеспечивают повторное выполнение кода, пока их условие истинно. Операторы `while` и `for` проверяют условие прежде, чем выполнить тело. Оператор `do while` сначала выполняет тело, а затем проверяет свое условие.

5.4.1. Оператор `while`

Оператор `while` многократно выполняет *`оператор`*, пока его *`условие`* остается истинным. Его синтаксическая форма имеет следующий вид:

```
while (условие)
оператор
```

Пока *`условие`* истинно (значение `true`), *`оператор`* (который зачастую является блоком кода) выполняется. Условие не может быть пустым. Если при первой проверке условие ложно (значение `false`), оператор не выполняется.

Условие может быть выражением или объявлением инициализированной переменной (см. раздел 5.2, стр. 236). Обычно либо само условие, либо тело цикла должно делать нечто изменяющее значение выражения. В противном случае цикл никогда не закончится.

Переменные, определенные в условии или теле оператора `while`, создаются и удаляются при каждой итерации.

Использование цикла `while`

Цикл `while` обычно используется в случае, когда итерации необходимо выполнять неопределенное количество раз, например, при чтении ввода. Цикл `while` полезен также при необходимости доступа к значению управляющей переменной после завершения цикла. Рассмотрим пример.

```
vector<int> v;
int i;
// читать до конца файла или отказа ввода
while (cin >> i)
    v.push_back(i);
// найти первый отрицательный элемент
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // известно, что все элементы v больше или равны нулю
```

Первый цикл читает данные со стандартного устройства ввода. Он может выполняться сколько угодно раз. Условие становится ложно, когда поток `cin` читает недопустимые данные, происходит ошибка ввода или встречается конец файла. Второй цикл продолжается до тех пор, пока не будет найдено отрицательное значение. Когда цикл заканчивается, переменная `beg` будет либо равна `v.end()`, либо обозначит элемент вектора `v`, значение которого меньше нуля. Значение переменной `beg` можно использовать вне цикла `while` для дальнейшей обработки.

Упражнения раздела 5.4.1

Упражнение 5.14. Напишите программу для чтения строк со стандартного устройства ввода и поиска совпадающих слов. Программа должна находить во вводе места, где одно слово непосредственно сопровождается таким же. Отследите наибольшее количество повторений и повторяемое слово. Отобразите максимальное количество дубликатов или сообщение, что никаких повторений не было. Например, при вводе `how now now now brown cow cow` вывод должен указать, что слово `now` встретилось три раза.

5.4.2. Традиционный оператор `for`

Оператор `for` имеет следующий синтаксис:

```
for (инициализирующий-оператор условие; выражение)
оператор
```

Слово `for` и часть в круглых скобках зачастую упоминают как *заголовок* `for` (`for` header).

Инициализирующий-оператор должен быть оператором объявления, выражением или пустым оператором. Каждый из этих операторов завершается точкой с запятой, поэтому данную синтаксическую форму можно рассматривать так:

```
for (инициализатор; условие; выражение)
оператор
```

Как правило, *инициализирующий-оператор* используется для инициализации или присвоения исходного значения переменной, изменяемой в цикле. Для управления циклом служит *условие*. Пока *условие* истинно, *оператор* выполняется. Если при первой проверке *условие* оказывается ложным, *оператор* не выполняется ни разу. Для изменения значения переменной, инициализированной в инициализирующем операторе и проверяемой в условии, используется *выражение*. Оно выполняется после каждой итерации цикла. Как и в других случаях, *оператор* может быть одиночным оператором или блоком операторов.

Поток выполнения в традиционном цикле `for`

Рассмотрим следующий цикл `for` из раздела 3.2.3 (стр. 138):

```
// обрабатывать символы, пока они не исчерпаются или не встретится пробел
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
        s[index] = toupper(s[index]); // преобразовать в верхний регистр
```

Порядок его выполнения таков.

1. В начале цикла только однажды выполняется `инициализирующий-оператор`. В данном случае определяется переменная index и инициализируется нулем.
2. Затем обрабатывается `условие`. Если `index` не равен `s.size()` и символ в элементе `s[index]` не является пробелом, то выполняется тело цикла `for`. В противном случае цикл заканчивается. Если условие ложно уже на первой итерации, то тело цикла `for` не выполняется вообще.
3. Если условие истинно, то тело цикла `for` выполняется. В данном случае оно переводит символ в элементе `s[index]` в верхний регистр.
4. И наконец, обрабатывается `выражение`. В данном случае значение переменной `index` увеличивается 1.

Эти четыре этапа представляют первую итерацию цикла `for`. Этап 1 выполняется только однажды при входе в цикл. Этапы 2–4 повторяются, пока условие не станет ложно, т.е. пока не встретится символ пробела в элементе `s` или пока `index` не превысит `s.size()`.

Не забывайте, что видимость любого объекта, определенного в пределах заголовка `for`, ограничивается телом цикла `for`. Таким образом, в данном примере переменная `index` недоступна после завершения цикла `for`.

Несколько определений в заголовке `for`

Подобно любому другому объявлению, *инициализирующий-оператор* способен определить несколько объектов. Однако только *инициализирующий-оператор* может быть оператором объявления. Поэтому у всех переменных должен быть тот же базовый тип (см. раздел 2.3, стр. 84). Для примера напишем цикл, дублирующий элементы вектора в конец следующим образом:

```
// запомнить размер v и остановиться, достигнув первоначально последнего
// элемента
for (decltype(v.size()) i = 0, sz = v.size(); i != sz; ++i)
    v.push_back(v[i]);
```

В этом цикле *инициализирующий-оператор* определяется индекс `i` и управляющая переменная цикла `sz`.

Пропуск частей заголовка `for`

В заголовке `for` может отсутствовать любой (или все) элемент: *инициализирующий-оператор*, *условие* или *выражение*.

Когда инициализация не нужна, вместо инициализирующего оператора можно использовать пустой оператор. Например, можно переписать цикл, который искал первое отрицательное число в векторе так, чтобы использовался цикл `for`:

```
auto beg = v.begin();
for ( /* ничего */; beg != v.end() && *beg >= 0; ++beg)
    ; // ничего не делать
```

Обратите внимание: для указания на отсутствие инициализирующего оператора точка с запятой необходима, точнее, точка с запятой представляет пустой инициализирующий оператор. В этом цикле `for` тело также пусто, поскольку все его действия осуществляются в условии и выражении. Условие решает, когда придет время прекращать просмотр, а выражение увеличивает итератор.

Отсутствие части *условие* эквивалентно расположению в условии значения `true`. Поскольку условие истинно всегда, тело цикла `for` должно содержать оператор, обеспечивающий выход из цикла. В противном случае цикл будет выполняться бесконечно.

```
for (int i = 0; /* нет условия */ ; ++i) {
    // обработка i; код в цикле должен остановить итерацию!
}
```

В заголовке `for` может также отсутствовать *выражение*. В таких циклах либо условие, либо тело должно делать нечто обеспечивающее итерацию. В качестве примера перепишем цикл `while`, читающий ввод в вектор целых чисел.

```
vector<int> v;
for (int i; cin >> i; /* нет выражения */ )
    v.push_back(i);
```

В этом цикле нет никакой необходимости в выражении, поскольку условие изменяет значение переменной `i`. Условие проверяет входной поток, поэтому цикл заканчивается, когда прочитан весь ввод или произошла ошибка ввода.

Упражнения раздела 5.4.2

Упражнение 5.15. Объясните каждый из следующих циклов. Исправьте все обнаруженные ошибки.

```
(a) for (int ix = 0; ix != sz; ++ix) { /* ... */ }
    if (ix != sz)
        // ...
(b) int ix;
    for (ix != sz; ++ix) { /* ... */ }
(c) for (int ix = 0; ix != sz; ++ix, ++ sz) { /* ... */ }
```

Упражнение 5.16. Цикл `while` особенно хорош, когда необходимо выполнить некое условие; например, когда нужно читать значения до конца файла. Цикл `for` считают циклом пошагового выполнения: индекс проходит диапазон значений в коллекции. Напишите идиоматическое использование каждого цикла, а затем перепишите каждый случай использования в другой конструкции цикла. Если бы вы могли использовать только один цикл, то какой бы вы выбрали и почему?

Упражнение 5.17. Предположим, есть два вектора целых чисел. Напишите программу, определяющую, не является ли один вектор префиксом другого. Для векторов неравной длины сравнивайте количество элементов меньшего вектора. Например, если векторы содержат значения `0`, `1`, `1`, `2` и `0`, `1`, `1`, `2`, `3`, `5`, `8` соответственно, ваша программа должна возвратить `true`.

5.4.3. Серийный оператор `for`

C++ 11

Новый стандарт ввел упрощенный оператор `for`, который перебирает элементы контейнера или другой последовательности. Синтаксис *серийного оператора* `for` (range `for`) таков:

```
for (объявление : выражение)
    оператор
```

выражение должно представить некую последовательность, такую, как список инициализации (см. раздел 3.3.1, стр. 143), массив (см. раздел 3.5, стр. 161), или объект такого типа, как `vector` или `string`, у которого есть функции-члены `begin()` и `end()`, возвращающие итераторы (см. раздел 3.4, стр. 153).

объявление определяет переменную. Каждый элемент последовательности должен допускать преобразование в тип переменной (см. раздел 4.11, стр. 218). Проще всего гарантировать соответствие типов за счет использования спецификатора типа `auto` (см. раздел 2.5.2, стр. 107). Так компилятор выведет тип сам. Если необходима запись в элементы последовательности, то переменная цикла должна иметь ссылочный тип.

На каждой итерации управляющая переменная определяется и инициализируется следующим значением последовательности, а затем выполняется *оператор*. Как обычно, *оператор* может быть одиночным оператором или блоком. Выполнение завершается, когда все элементы обработаны.

Несколько таких циклов уже было представлено, но для завершенности рассмотрим цикл, удваивающий значение каждого элемента в векторе:

```
vector<int> v = {0,1,2,3,4,5,6,7,8,9};
// для записи в элементы переменная диапазона должна быть ссылкой
for (auto &r : v) // для каждого элемента вектора v
    r *= 2;       // удвоить значение каждого элемента вектора v
```

Заголовок `for` объявляет, что управляющая переменная цикла `r` связана с вектором `v`. Чтобы позволить компилятору самостоятельно вывести тип переменной `r`, используем спецификатор `auto`. Поскольку предполагается изменение значений элементов вектора `v`, объявим переменную `r` как ссылку. При присвоении ей значений в цикле фактически присваивается значение элементу, с которым связана переменная `r` в данный момент.

Вот эквивалентное определение серийного оператора `for` в терминах традиционного цикла `for`:

```
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {
    auto &r = *beg; // для изменения элементов r должна быть ссылкой
    r *= 2;         // удвоить значение каждого элемента вектора v
}
```

Теперь, когда известно, как работает серийный оператор `for`, можно понять, почему в разделе 3.3.2 (стр. 148) упоминалось о невозможности его использования для добавления элементов к вектору или другому контейнеру. В серийном операторе `for` кешируется значение `end()`. Если добавить или удалить элементы из последовательности, сохраненное значение `end()` станет неверным (см. раздел 3.4.1, стр. 158). Более подробная информация по этой теме приведена в разделе 9.3.6 (стр. 453).

5.4.4. Оператор do while

Оператор do while похож на оператор while, но его условие проверяется после выполнения тела. Независимо от значения условия тело цикла выполняется по крайней мере однажды. Его синтаксическая форма приведена ниже.

```
Do
    оператор
while (условие);
```

После заключенного в скобки условия оператор do while заканчивается точкой с запятой.

В цикле do while *оператор* выполняется прежде, чем *условие*. Причем *условие* не может быть пустым. Если *условие* ложно, цикл завершается, в противном случае цикл повторяется. Используемые в условии переменные следует определить вне тела оператора do while.

Напишем программу, использующую цикл do while для суммирования любого количества чисел.

```
// многократно запрашивать у пользователя пары чисел для суммирования
string rsp; // используется в условии, поэтому не может быть
            // определена в цикле do
do {
    cout << "please enter two values: ";
    int val1 = 0, val2 = 0;
    cin >> val1 >> val2;
    cout << "The sum of " << val1 << " and " << val2
         << " = " << val1 + val2 << "\n\n"
         << "More? Enter yes or no: ";
    cin >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

Цикл начинается запросом у пользователя двух чисел. Затем выводится их сумма и следует запрос, желает ли пользователь суммировать далее. Ответ пользователя проверяется в условии. Если ввод пуст или начинается с n, цикл завершается. В противном случае цикл повторяется.

Поскольку условие не обрабатывается до окончания оператора или блока, цикл do while не позволяет определять переменные в условии.

```
do {
    // ...
    mumble(foo);
} while (int foo = get_foo()); // ошибка: объявление в условии do
```

Если определить переменные в условии, то любое их использование произойдет *прежде* определения!

Упражнения раздела 5.4.4

Упражнение 5.18. Объясните каждый из следующих циклов. Исправьте все обнаруженные ошибки.

```
(a) do
        int v1, v2;
        cout << "Please enter two numbers to sum:" ;
        if (cin >> v1 >> v2)
            cout << "Sum is: " << v1 + v2 << endl;
    while (cin);
(b) do {
        // ...
    } while (int ival = get_response());
(c) do {
        int ival = get_response();
    } while (ival);
```

Упражнение 5.19. Напишите программу, использующую цикл `do while` для циклического запроса у пользователя двух строк и указания, которая из них меньше другой.

5.5. Операторы перехода

Операторы перехода прерывают поток выполнения. Язык C++ предоставляет четыре оператора перехода: `break`, `continue` и `goto`, рассматриваемые в этой главе, и оператор `return`, который будет описан в разделе 6.3 (стр. 292).

5.5.1. Оператор `break`

Оператор `break` завершает ближайший окружающий оператор `while`, `do while`, `for` или `switch`. Выполнение возобновляется с оператора, следующего непосредственно за завершаемым оператором.

Оператор `break` может располагаться только в цикле или операторе `switch` (включая операторы или блоки, вложенные в эти циклы). Оператор `break` воздействует лишь на ближайший окружающий цикл или оператор `switch`.

```
string buf;
while (cin >> buf && !buf.empty()) {
    switch(buf[0]) {
    case '-':
        // продолжать до первого пробела
        for (auto it = buf.begin()+1; it != buf.end(); ++it) {
            if (*it == ' ')
                break; // #1, выйти из цикла for
            // ...
        }
        // break #1 передает управление сюда
```

```
        // дальнейшая обработка случая '-':
        break; // #2, выйти из оператора switch
    case '+':
        // ...
    } // конец оператора switch
    // break #2 передает управление сюда
} // конец оператора while
```

Оператор `break` с меткой `#1` завершает цикл `for` в разделе `case` для случая дефиса. Он не завершает внешний оператор `switch` и даже не завершает обработку текущего случая. Выполнение продолжается с первого оператора после цикла `for`, который мог бы содержать дополнительный код обработки случая дефиса или оператор `break`, который завершает данный раздел.

Оператор `break` с меткой `#2` завершает оператор `switch`, но не внешний цикл `while`. Выполнение кода после оператора `break` продолжает условие цикла `while`.

Упражнения раздела 5.5.1

Упражнение 5.20. Напишите программу, которая читает последовательность строк со стандартного устройства ввода до тех пор, пока не встретится повторяющееся слово или пока ввод слов не будет закончен. Для чтения текста по одному слову используйте цикл `while`. Для выхода из цикла при встрече двух совпадающих слов подряд используйте оператор `break`. Выведите повторяющееся слово, если оно есть, а в противном случае отобразите сообщение, свидетельствующее о том, что повторяющихся слов нет.

5.5.2. Оператор `continue`

Оператор `continue` прерывает текущую итерацию ближайшего цикла и немедленно начинает следующую. Оператор `continue` может присутствовать только в циклах `for`, `while` или `do while`, включая операторы или блоки, вложенные в такие циклы. Подобно оператору `break`, оператор `continue` во вложенном цикле воздействует только на ближайший окружающий цикл. Однако, в отличие от оператора `break`, оператор `continue` может присутствовать в операторе `switch`, только если он встроен в итерационный оператор.

Оператор `continue` прерывает только текущую итерацию; выполнение остается в цикле. В случае цикла `while` или `do while` выполнение продолжается с оценки условия. В традиционном цикле `for` выполнение продолжается в выражении заголовка. В серийном операторе `for` выполнение продолжается с инициализации управляющей переменной следующим элементом последовательности.

Следующий цикл читает со стандартного устройства ввода по одному слову за раз. Обработаны будут только те слова, которые начинаются с сим-

вола подчеркивания. Для любого другого значения текущая итерация заканчивается.

```
string buf;
while (cin >> buf && !buf.empty()) {
    if (buf[0] != '_')
        continue; // получить другой ввод
    // все еще здесь? ввод начинается с '_'; обработка buf...
}
```

Упражнения раздела 5.5.2

Упражнение 5.21. Переделайте программу из упражнения раздела 5.5.1 (стр. 255) так, чтобы она искала дубликаты только тех слов, которые начинаются с прописной буквы.

5.5.3. Оператор goto

Оператор `goto` обеспечивает безусловный переход к другому оператору в той же функции.

Не нужно использовать операторы `goto`. Они затрудняют и понимание, и изменение программ.

Оператор `goto` имеет следующий синтаксис:

```
goto метка;
```

`Метка` (label) — это идентификатор, которым помечен оператор. *Помеченный оператор* (labeled statement) — это любой оператор, которому предшествует идентификатор, сопровождаемый двоеточием.

```
end: return; // помеченный оператор; может быть целью оператора goto
```

Метки независимы от имен, используемых для переменных и других идентификаторов. Следовательно, у метки может быть тот же идентификатор, что и у другой сущности в программе, не вступая в конфликт с другим одноименным идентификатором. Оператор `goto` и помеченный оператор, на который он передает управление, должны находиться в той же функции.

Подобно оператору `switch`, оператор `goto` не может передать управление из точки, где инициализированная переменная вышла из области видимости, в точку, где эта переменная находится в области видимости.

```
// ...
goto end;
int ix = 10; // ошибка: goto обходит определение инициализированной
             // переменной
end:
// ошибка: код здесь мог бы использовать ix, но goto обошел ее объявление
ix = 42;
```

Переход назад за уже выполненное определение вполне допустим. Переходя назад к точке перед определением переменная приведет к ее удалению и повторному созданию.

```
// переход назад через определение инициализированной переменной приемлем
begin:
    int sz = get_size();
    if (sz <= 0) {
        goto begin;
    }
```

При выполнении оператора `goto` переменная `sz` удаляется, а затем она определяется и инициализируется снова, когда управление передается назад за ее определение после перехода к метке `begin`.

Упражнения раздела 5.5.3

Упражнение 5.22. Последний пример этого раздела, с переходом назад к метке `begin`, может быть написан лучше с использованием цикла. Перепишите код так, чтобы устранить оператор `goto`.

5.6. Блоки `try` и обработка исключений

Исключения (exception) — это аномалии времени выполнения, такие как потеря подключения к базе данных или ввод непредвиденных данных, которые нарушают нормальное функционирование программы[1]. Реакция на аномальное поведение может быть одним из самых трудных этапов разработки любой системы.

Обработка исключений обычно используется в случае, когда некая часть программы обнаруживает проблему, с которой она не может справиться, причем проблема такова, что обнаружившая ее часть программы не может продолжить выполнение. В таких случаях обнаруживший проблему участок программы нуждается в способе сообщить о случившемся и о том, что он неспособен продолжить выполнение. Способ сообщения о проблеме не подразумевает знания о том, какая именно часть программы будет справляться с создавшейся ситуацией. Сообщив о случившемся, обнаружившая проблему часть кода прекращает работу.

[1] Согласно другой трактовке, исключение — это объект системного или пользовательского класса, создаваемого операционной системой или кодом программы в ответ на обстоятельства, либо не допускающие дальнейшего нормального выполнения программы, либо определенные пользователем. Обработка исключений в приложении позволяет корректно выйти из затруднительной ситуации. — *Примеч. ред.*

Каждой части программы, способной передать исключение, соответствует другая часть, код которой способен обработать исключение, независимо от того, что произошло. Например, если проблема в недопустимом вводе, то часть обработки могла бы попросить пользователя ввести правильные данные. Если потеряна связь с базой данных, то часть обработки могла бы предупредить об этом пользователя.

Исключения обеспечивают взаимодействие частей программы, обнаруживающих проблему и решающих ее. Обработка исключений в языке C++ подразумевает следующее.

- *Оператор* `throw` используется частью кода обнаружившего проблему, с которой он не может справиться. Об операторе `throw` говорят, что он *передает* (raise) исключение.
- *Блок* `try` используется частью обработки исключения. Блок `try` начинается с *ключевого слова* `try` и завершается одной или несколькими *директивами* `catch` (`catch` clause). Исключения, переданные из кода, расположенного в блоке `try`, как правило, обрабатываются в одном из разделов `catch`. Поскольку разделы `catch` обрабатывают исключение, их называют также *обработчиками исключений* (exception handler).
- Набор определенных в библиотеке *классов исключений* (exception class) используется для передачи информации о произошедшем между операторами `throw` и соответствующими разделами `catch`.

В остальной части этого раздела три компонента обработки исключений рассматриваются последовательно. Более подробная информация об исключениях приведена в разделе 18.1 (стр. 966).

5.6.1. Оператор `throw`

Обнаруживающая часть программы использует оператор `throw` для передачи исключения. Он состоит из ключевого слова `throw`, сопровождаемого выражением. Выражение определяет тип передаваемого исключения. Оператор `throw`, как правило, завершается точкой с запятой, что делает его выражением.

Для примера вернемся к программе раздела 1.5.2 (стр. 51), в которой суммируются два объекта класса `Sales_item`. Она проверяет, относятся ли обе прочитанные записи к одной книге. Если нет, она отображает сообщение об ошибке и завершает работу.

```
Sales_item item1, item2;
cin >> item1 >> item2;
// сначала проверить, представляют ли объекты item1 и item2
// одну и ту же книгу
if (item1.isbn() == item2.isbn()) {
    cout << item1 + item2 << endl;
    return 0; // свидетельство успеха
```

```
} else {
    cerr << "Data must refer to same ISBN"
         << endl;
    return -1; // свидетельство отказа
}
```

В более реалистичной программе суммирующая объекты часть могла бы быть отделена от части, обеспечивающей взаимодействие с пользователем. В таком случае проверяющую часть можно было бы переписать так, чтобы она передавала исключение, а не возвращала свидетельство отказа.

```
// сначала проверить, представляют ли объекты item1 и item2
// одну и ту же книгу
if (item1.isbn() != item2.isbn())
    throw runtime_error("Data must refer to same ISBN");
// если управление здесь, значит, ISBN совпадают
cout << item1 + item2 << endl;
```

Если теперь ISBN окажутся разными, будет передан объект исключения типа `runtime_error`. Передача исключения завершает работу текущей функции и передает управление обработчику, способному справиться с этой ошибкой.

Тип `runtime_error` является одним из типов исключения, определенных в заголовке `stdexcept` стандартной библиотеки. Более подробная информация по этой теме приведена в разделе 5.6.3 (стр. 262). Объект класса `runtime_error` следует инициализировать объектом класса `string` или символьной строкой в стиле C (см. раздел 3.5.4, стр. 172). Эта строка представляет дополнительную информацию о проблеме.

5.6.2. Блок `try`

Блок `try` имеет следующий синтаксис:

```
try {
    операторы_программы
} catch (объявление_исключения) {
    операторы_обработчика
} catch (объявление_исключения) {
    операторы_обработчика
} // ...
```

Блок `try` начинается с ключевого слова `try`, за которым следует блок кода, заключенный в фигурные скобки.

Блок `try` сопровождается одним или несколькими блоками `catch`. Блок `catch` состоит из трех частей: ключевого слова `catch`, объявления (возможно, безымянного) объекта в круглых скобках (называется *объявлением исключения* (exception declaration)) и операторного блока. Когда объявление исключения в блоке `catch` совпадает с исключением, выполняется связанный с ним блок. По завершении выполнения кода обработчика управление переходит к оператору, следующему непосредственно после него.

Операторы_программы в блоке try являются обычными программными операторами, реализующими ее логику. Подобно любым другим блокам кода, блоки try способны содержать любые операторы языка C++, включая объявления. Объявленные в блоке try переменные недоступны вне блока, в частности, они не доступны в блоках catch.

Создание обработчика

В приведенном выше примере, чтобы избежать суммирования двух объектов класса Sales_item, представляющих разные книги, использовался оператор throw. Предположим, что суммирующая объекты класса Sales_item часть программы отделена от части, взаимодействующей с пользователем. Эта часть могла бы содержать примерно такой код обработки исключения, переданного в блоке сложения.

```
while (cin >> item1 >> item2) {
    try {
        // код, который складывает два объекта класса Sales_item
        // если при сложении произойдет сбой, код передаст
        // исключение runtime_error
    } catch (runtime_error err) {
        // напомнить пользователю, что ISBN слагаемых объектов
        // должны совпадать
        cout << err.what()
             << "\nTry Again? Enter y or n" << endl;
        char c;
        cin >> c;
        if (!cin || c == 'n')
            break; // выход из цикла while
    }
}
```

В блоке try расположена обычная логика программы. Это сделано потому, что данная часть программы способна передать исключение типа runtime_error.

Данный блок try обладает одним разделом catch, который обрабатывает исключение типа runtime_error. Операторы в блоке после ключевого слова catch определяют действия, выполняемые в случае, если код в блоке try передаст исключение runtime_error. В данном случае обработка подразумевает отображение сообщения об ошибке и запрос у пользователя разрешения на продолжение. Когда пользователь вводит символ 'n', цикл while завершается, в противном случае он продолжается и считывает два новых объекта класса Sales_item.

В сообщении об ошибке используется текст, возвращенный функцией err.what(). Поскольку известно, что классом объекта исключения err является runtime_error, нетрудно догадаться, что функция what() является членом (см. раздел 1.5.2, стр. 51) класса runtime_error. В каждом из библиотечных классов исключений определена функция-член what(), которая не получает

никаких аргументов и возвращает символьную строку в стиле C (т.е. `const char*`). В случае класса `runtime_error` эта строка является копией строки, использованной при инициализации объекта класса `runtime_error`. Если описанный в предыдущем разделе код передаст исключение, то отображенное разделом `catch` сообщение об ошибке будет иметь следующий вид:

```
Data must refer to same ISBN
Try Again? Enter y or n
```

При поиске обработчика выполнение функций прерывается

В сложных системах программа может пройти через несколько блоков `try` прежде, чем встретится с кодом, который передает исключение. Например, в блоке `try` может быть вызвана функция, в блоке `try` которой содержится вызов другой функции с ее собственным блоком `try`, и т.д.

Поиск обработчика осуществляется по цепочке обращений в обратном порядке. Сначала поиск обработчика исключения осуществляется в той функции, в которой оно было передано. Если соответствующего раздела `catch` не найдено, работа функции завершается, а поиск продолжается в той функции, которая вызвала функцию, в которой было передано исключение. Если и здесь соответствующий раздел `catch` не найден, эта функция также завершается, а поиск продолжается по цепочке вызовов дальше, пока обработчик исключения соответствующего типа не будет найден.

Если соответствующий раздел `catch` так и не будет найден, управление перейдет к библиотечной функции `terminate()`, которая определена в заголовке `exception`. Поведение этой функции зависит от системы, но обычно она завершает выполнение программы.

Исключения, которые были переданы в программах, не имеющих блоков `try`, обрабатываются аналогично: в конце концов, без блоков `try` не может быть никаких обработчиков и ни для каких исключений, которые, однако, вполне могут быть переданы. В таком случае исключение приводит к вызову функции `terminate()`, которая (как правило) и завершает работу программы.

ВНИМАНИЕ! НАПИСАНИЕ УСТОЙЧИВОГО К ИСКЛЮЧЕНИЯМ КОДА — ДОВОЛЬНО СЛОЖНАЯ ЗАДАЧА

Важно понимать, что исключения прерывают нормальный поток программы. В месте, где происходит исключение, некоторые из действий, ожидаемых вызывающей стороной, могут быть выполнены, а другие нет. Как правило, пропуск части программы может означать, что объект останется в недопустимом или неполном состоянии, либо что ресурс не будет освобожден и т.д. Программы, которые правильно "зачищают" объекты во время обработки исключений, называют *устойчивыми к исключениям* (exception

safe). Написание устойчивого к исключениям кода чрезвычайно сложно и практически не рассматривается в данном вводном курсе.

Некоторые программы используют исключения просто для завершения программы в случае проблем. Такие программы вообще не заботятся об устойчивости к исключениям.

Программы, которые действительно обрабатывают исключения и продолжают работу, должны постоянно знать, какое исключение может произойти и что программа должна делать для гарантии допустимости объектов, невозможности утечки ресурсов и восстановления программы в корректном состоянии.

Некоторые из наиболее популярных методик обеспечения устойчивости к исключениям здесь будут упомянуты. Однако читатели, программы которых требуют надежной обработки исключений, должны знать, что рассматриваемых здесь методик недостаточно для полного обеспечения устойчивости к исключениям.

5.6.3. Стандартные исключения

В библиотеке C++ определен набор классов, объекты которых можно использовать для передачи сообщений о проблемах в функциях, определенных в стандартной библиотеке. Эти стандартные классы исключений могут быть также использованы в программах, создаваемых разработчиком. Библиотечные классы исключений определены в четырех следующих заголовках.

- В заголовке `exception` определен общий класс исключения `exception`. Он сообщает только о том, что исключение произошло, но не предоставляет никакой дополнительной информации.
- В заголовке `stdexcept` определено несколько универсальных классов исключения (табл. 5.1).
- В заголовке `new` определен класс исключения `bad_alloc`, рассматриваемый в разделе 12.1.2 (стр. 584).
- В заголовке `type_info` определен класс исключения `bad_cast`, рассматриваемый в разделе 19.2 (стр. 1029).

В классах `exception`, `bad_alloc` и `bad_cast` определен только стандартный конструктор (см. раздел 2.2.1, стр. 77), поэтому невозможно инициализировать объект этих типов.

Поведение исключений других типов прямо противоположно: их можно инициализировать объектом класса `string` или строкой в стиле C, однако значением по умолчанию их инициализировать *нельзя*. При создании объекта исключения любого из этих типов необходимо предоставить инициализатор. Этот инициализатор используется для предоставления дополнительной информации о произошедшей ошибке.

Таблица 5.1. Стандартные классы исключений, определенные в заголовке `stdexcept`

`exception`	Наиболее общий вид проблемы
`runtime_error`	Проблема, которая может быть обнаружена только во время выполнения
`range_error`	Ошибка времени выполнения: полученный результат превосходит допустимый диапазон значения
`overflow_error`	Ошибка времени выполнения: переполнение регистра при вычислении
`underflow_error`	Ошибка времени выполнения: недополнение регистра при вычислении
`logic_error`	Ошибка в логике программы
`domain_error`	Логическая ошибка: аргумент, для которого не существует результата
`invalid_argument`	Логическая ошибка: неподходящий аргумент
`length_error`	Логическая ошибка: попытка создать объект большего размера, чем максимально допустимый для данного типа
`out_of_range`	Логическая ошибка: используемое значение вне допустимого диапазона

В классах исключений определена только одна функция `what()`. Она не получает никаких аргументов и возвращает константный указатель на тип `char`. Это указатель на символьную строку в стиле C (см. раздел 3.5.4, стр. 172), содержащую текст описания переданного исключения.

Содержимое символьного массива (строки в стиле C), указатель на который возвращает функция `what()`, зависит от типа объекта исключения. Для типов, которым при инициализации передают строку класса `string`, функция `what()` возвращает строку. Что же касается других типов, то возвращаемое значение зависит от компилятора.

Упражнения раздела 5.6.3

Упражнение 5.23. Напишите программу, которая читает два целых числа со стандартного устройства ввода и выводит результат деления первого числа на второе.

Упражнение 5.24. Перепишите предыдущую программу так, чтобы она передавала исключение, если второе число — нуль. Проверьте свою програм-

му с нулевым вводом, чтобы увидеть происходящее при отсутствии обработчика исключения.

Упражнение 5.25. Перепишите предыдущую программу так, чтобы использовать для обработки исключения блок `try`. Раздел `catch` должен отобразить сообщение и попросить пользователя ввести новое число и повторить код в блоке `try`.

Резюме

Язык C++ предоставляет довольно ограниченное количество операторов. Некоторые из них предназначены для управления потоком выполнения программы.

- Операторы `while`, `for` и `do while` позволяют реализовать итерационные циклы.
- Операторы `if` и `switch` позволяют реализовать условное выполнение.
- Оператор `continue` останавливает текущую итерацию цикла.
- Оператор `break` осуществляет принудительный выход из цикла или оператора `switch`.
- Оператор `goto` передает управление помеченному оператору.
- Операторы `try` и `catch` позволяют создать блок `try`, в который заключают операторы программы, потенциально способные передать исключение. Оператор `catch` начинает раздел обработчика исключения, код которого предназначен для реакции на исключение определенного типа.
- Оператор `throw` позволяет передать исключение, обрабатываемое в соответствующем разделе `catch`.
- Оператор `return` останавливает выполнение функции. (Подробней об этом — в главе 6.)

Кроме того, существуют операторы выражения и операторы объявления. Объявления и определения переменных были описаны в главе 2.

Термины

Блок `try`. Блок, начинаемый ключевым словом `try` и содержащий один или несколько разделов `catch`. Если код в блоке `try` передаст исключение, а один из разделов `catch` соответствует типу этого исключения, то исключение будет обработано кодом данного обработчика. В противном случае исключение будет обработано во внешнем блоке `try`, но если и этого не произойдет, сработает функция `terminate()`, которая и завершит выполнение программы.

Блок (block). Последовательность любого количества операторов, заключенная в фигурные скобки. Блок операторов может быть использован везде, где ожидается один оператор.

Директива `catch` (`catch` clause). Состоит из ключевого слова `catch`, объявления исключения в круглых скобках и блока операторов. Код в разделе `catch` предназначен для обработки исключения, тип которого указан в объявлении.

Класс исключения (exception class). Набор определенных стандартной библиотекой классов, используемых для сообщения об ошибке. Универсальные классы исключений см. в табл. 5.1 (стр. 263).

Метка `case`. Константное выражение (см. раздел 2.4.4, стр. 103), следующее за ключевым словом `case` в операторе `switch`. Метки `case` в том же операторе `switch` не могут иметь одинакового значения.

Метка `default`. Метка оператора `switch`, соответствующая любому значению условия, не указанному в метках `case` явно.

Обработчик исключения (exception handler). Код, реагирующий на исключение определенного типа, переданное из другой части программы. Синоним термина *директива* `catch`.

Объявление исключения (exception declaration). Объявление в разделе `catch`. Определяет тип исключений, обрабатываемых данным обработчиком.

Оператор `break`. Завершает ближайший вложенный цикл или оператор `switch`. Передает управление первому оператору после завершенного цикла или оператора `switch`.

Оператор `continue`. Завершает текущую итерацию ближайшего вложенного цикла. Передает управление условию цикла `while`, оператору `do` или выражению в заголовке цикла `for`.

Оператор `do while`. Подобен оператору `while`, но условие проверяется в конце цикла, а не в начале. Тело цикла выполняется по крайней мере однажды.

Оператор `for`. Оператор цикла, обеспечивающий итерационное выполнение. Зачастую используется для повторения вычислений определенное количество раз.

Серийный оператор `for` (range `for`). Управляющий оператор, перебирающий значения указанной коллекции и выполняющий некую операцию с каждым из них.

Оператор `goto`. Оператор, осуществляющий безусловную передачу управления помеченному оператору в другом месте той же функции. Операторы `goto` нарушают последовательность выполнения операций программы, поэтому их следует избегать.

Оператор `if`. Условное выполнение кода на основании значения в условии. Если условие истинно (значение `true`), тело оператора `if` выполняется, в противном случае управление переходит к оператору, следующему после него.

Оператор `if...else`. Условное выполнение кода в разделе `if` или `else`, в зависимости от истинности значения условия.

Оператор `switch`. Оператор условного выполнения, который сначала вычисляет результат выражения, следующего за ключевым словом `switch`, а затем передает управление разделу `case`, метка которого совпадает с результатом выражения. Когда соответствующей метки нет, выполнение переходит к разделу `default` (если он есть) или к оператору, следующему за оператором `switch`, если раздела `default` нет.

Оператор `throw`. Оператор, прерывающий текущий поток выполнения. Каждый оператор `throw` передает объект, который переводит управление на ближайший раздел `catch`, способный обработать исключение данного класса.

Оператор `while`. Оператор цикла, который выполняет оператор тела до тех пор, пока условие остается истинным (значение `true`). В зависимости от истинности значения условия оператор выполняется любое количество раз.

Оператор выражения (expression statement). Выражение завершается точкой с запятой. Оператор выражения обеспечивает выполнение действий в выражении.

Передача (raise, throwing). Выражение, которое прерывает текущий поток выполнения. Каждый оператор `throw` передает объект, переводящий управление на ближайший раздел `catch`, способный обработать исключение данного класса.

Помеченный оператор (labeled statement). Оператор, которому предшествует метка. *Метка* (label) — это идентификатор, сопровождаемый двоеточием. Метки используются независимо от других одноименных идентификаторов.

Потерянный оператор `else` (dangling `else`). Разговорный термин, используемый для описания проблемы, когда во вложенной конструкции операторов `if` больше, чем операторов `else`. В языке C++ оператор `else` всегда принадлежит ближайшему расположенному выше оператору `if`. Чтобы указать явно, какому из операторов `if` принадлежит конкретный оператор `else`, применяются фигурные скобки.

Пустой оператор (null statement). Пустой оператор представляет собой отдельный символ точки с запятой.

Составной оператор (compound statement). Синоним блока.

Управление потоком (flow of control). Управление последовательностью выполнения операций в программе.

Устойчивость к исключениям (exception safe). Термин, описывающий программы, которые ведут себя правильно при передаче исключения.

Функция `terminate()`. Библиотечная функция, вызываемая в случае, если исключение так и не было обработано. Обычно завершает выполнение программы.