4

Организация программ и данных

Программа, приведенная в разделе 3.2.2, больше по размеру, чем, возможно, вам бы того хотелось, но она могла быть еще больше, если бы мы не использовали вектор, `string`-переменную и функцию `sort`. Эти библиотечные средства, подобно другим, уже опробованным нами выше, имеют ряд общих свойств. Каждое из этих средств

- решает проблему конкретного типа;
- не зависит от большинства других;
- имеет имя.

Наши собственные программы обладают только первым из перечисленных трех свойств. Этот недостаток простителен лишь для маленьких программ, но по мере решения все более серьезных проблем вскоре становится ясно, что наши программные решения будут неуправляемыми, если их не разбить на независимые именованные части.

Подобно большинству языков программирования, C++ предлагает два фундаментальных способа организации больших программ: функции (иногда их называют подпрограммами) и структуры данных. Кроме того, C++ позволяет программистам объединять функции и структуры данных в одно целое, именуемое классом, знакомство с которым мы начнем в главе 9.

Узнав, как использовать функции и структуры данных для организации вычислений, мы также должны научиться разделять свои программы на файлы, которые можно компилировать отдельно, и снова объединять их уже после компиляции. В последней части этой главы показано, как в C++ поддерживается механизм раздельной компиляции.

4.1. Организация вычислений

Начнем, пожалуй, с написания функции вычисления итоговой оценки студента на основе полученных оценок на экзаменах в середине и конце семестра, а также обобщенной оценки за выполнение домашних заданий. При этом допустим, что мы уже вычислили обобщенную оценку, исходя из оценок за отдельные домашние задания, причем обобщенная оценка вычислялась как среднее арифметическое или как медианное значение. Помимо упомянутого допущения, эта функция должна использовать ту же стратегию вычисления итоговой оценки: домашние задания и результат последнего экзамена "вносят" по 40% в итоговый результат, а результат промежуточного экзамена — оставшиеся 20%.

Если общий процесс вычисления оказался разбитым на отдельные составляющие, то предпочтительнее собрать их под "крышей" одной функции. Очевидно, что в этом случае (вместо повторного выполнения отдельных частей) при необходимости решения аналогичной задачи мы мо-

жем просто использовать одну (уже готовую) функцию. Использование функций не только сокращает общие затраты на программирование, но также значительно облегчает процесс внесения изменений в вычисления. Предположим, мы хотим изменить политику расчета итоговой оценки. Если бы нам пришлось просматривать каждую программу, написанную нами ранее, чтобы найти ту часть, где рассчитывается итоговая оценка с учетом веса каждой ее составляющей, у нас бы быстро опустились руки.

Для подобных вычислений необходимо отметить даже более существенное преимущество использования функций. Любая функция имеет имя. Если некоторому вычислению присвоить имя, мы в этом случае можем думать о нем более абстрактно, т.е. мы можем *больше* думать о том, *что* оно делает, и *меньше* — *как* оно работает. Если нам удастся идентифицировать важные части наших проблем, а затем создать именованные "кусочки" наших программ, которые соответствуют этим частям, то наши программы станут проще для понимания, а проблемы — легче для решения. Вот как выглядит функция, которая вычисляет итоговые оценки в соответствии с определенной выше политикой.

```
// Вычисляем итоговую оценку студента на основе оценок,
// полученных на экзаменах в середине и конце семестра,
// а также на основе обобщенной оценки за выполнение домашних заданий.
double grade(double midterm, double final, double homework)
{
    return 0.2 * midterm + 0.4 * final + 0.4 * homework;
}
```

До сих пор все функции, которые мы определяли, имели имя `main`. Большинство других функций определяется аналогичным образом, т.е. посредством задания типа возвращаемого значения, за которым сначала указывается имя функции, затем ***список параметров*** (parameter list), заключенный в круглые скобки, и, наконец, тело функции, заключенное в фигурные скобки. Эти правила несколько усложняются для функций, которые возвращают значения, указывающие на другие функции (подробнее см. раздел A.1.2).

В этом примере параметрами являются `midterm`, `final` и `homework`, причем все они имеют тип `double`. Параметры ведут себя как переменные, которые локальны для данной функции, т.е. при вызове функции они создаются, а при выходе из нее — разрушаются.

Подобно любым другим переменным, параметры должны быть определены до их использования. В отличие от других переменных, их определение не означает немедленное создание: они создаются лишь при вызове функции. Следовательно, вызывая функцию, мы должны предоставить ей соответствующие ***аргументы*** (arguments), которые используются для инициализации параметров в момент, когда функция начинает выполняться. Например, в разделе 3.1 мы вычисляли оценку, используя следующую инструкцию.

```
cout << "Ваша итоговая оценка равна " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
     << setprecision(prec) << endl;
```

Если бы в нашем распоряжении была функция `grade`, предыдущую инструкцию мы могли бы записать по-другому.

```
cout << "Ваша итоговая оценка равна " << setprecision(3)
     << grade(midterm, final, sum / count)
     << setprecision(prec) << endl;
```

Мы должны так предоставить функции ее аргументы, чтобы они не только соответствовали по типу параметрам вызываемой нами функции, но и были указаны в таком же порядке. Следовательно, вызывая функцию `grade`, в качестве первого аргумента нужно передать значение оценки, полученной в середине семестра, в качестве второго — значение оценки, полученной в конце семестра, а в качестве третьего — значение обобщенной оценки за выполнение домашних заданий.

Аргументы могут быть выражениями, например `sum / count`, а не просто переменными. В общем случае каждый аргумент используется для инициализации соответствующего параметра, после чего поведение параметров ничем не отличается от поведения обычных локальных переменных внутри функции. Так, например, при вызове функции `grade(midterm, final, sum / count)` ее параметры не обращаются непосредственно к самим аргументам, а инициализируются копиями значений аргументов. Такой механизм часто называется ***вызовом по значению*** (call by value), поскольку параметр принимает копию значения аргумента.

4.1.1. Вычисление медиан

Еще одна проблема, которую мы решили в разделе 3.2.2 и которая, как нетрудно предположить, обязательно потребует решения в других контекстах, состоит в вычислении медианы вектора. В разделе 8.1.1 будет показано, как определить функцию в более общем виде, чтобы она смогла работать с вектором значений любого типа. А пока ограничимся рассмотрением вектора типа `vector<double>`.

Создание функции начнем с той части программы (см. раздел 3.2.2), в которой вычисляется медиана, и внесем в нее ряд изменений.

```
// Вычисляем медиану вектора vector<double>.
// Обратите внимание на то, что вызов этой функции копирует
// аргумент vector целиком.
double median(vector<double> vec)
{
    typedef vector<double>::size_type vec_sz;

    vec_sz size = vec.size();
    if (size == 0)
        throw domain_error("Медиана пустого вектора.");

    sort(vec.begin(), vec.end());

    vec_sz mid = size/2;

    return size % 2 == 0 ? (vec[mid] + vec[mid1]) / 2 : vec[mid];
}
```

Одно изменение состоит в том, что мы присвоили нашему вектору имя `vec`, а не `homework`. Как-никак, наша функция может вычислять медиану от любого набора чисел, а не только от оценок за домашние задания. Мы также удалили переменную `median`, поскольку она нам больше не нужна: ведь мы можем вернуть медиану сразу после ее вычисления. Мы по-прежнему используем в качестве переменных `size` и `mid`, но сейчас они являются локальными для функции `median` и, следовательно, недоступными (и нерелевантными) вне ее. При вызове функции `median` эти переменные создаются, а при выходе из нее — разрушаются. Мы определяем `vec_sz` как локальное имя типа, поскольку не хотим никаких конфликтов ни с кем, кто захочет использовать это имя для другой цели.

Самое существенное изменение — обработка случая, когда вектор оказывается пустым. В разделе 3.2.2 в этом случае мы знали о необходимости выразить недовольство тому, кто выполняет нашу программу. Мы также знали, кем является этот пользователь, и поэтому нам было ясно, какая жалоба будет уместна в нашей программе. Однако в новой версии программы нам неизвестно, кто будет ее использовать и с какой целью, поэтому нужно подумать о более общей форме "жалобы" на пользователя программы. Такой более общей формой является ***генерирование исключительной ситуации***, или ***исключения*** (throw an exception), если вектор окажется пустым.

Когда программа генерирует исключение, ее выполнение прекращается в той части, где встретилось слово `throw`, и продолжается в другой части, но уже с использованием ***объекта исключе-***

ния (exception object), содержащего информацию, которую можно использовать для обработки этого исключения.

Самой существенной информацией, передаваемой в подобных случаях из одной части программы в другую, является сам факт генерирования исключения. Этого факта (вместе с типом объекта исключения) обычно достаточно, чтобы "понять", что нужно сделать в такой ситуации. В данном примере исключение, генерируемое нашей программой, имеет тип `domain_error`. Этот тип определяется стандартной библиотекой в заголовке `<stdexcept>` и предназначен для сообщения о том, что аргумент функции не принадлежит набору значений, которые эта функция способна принять. При создании объекта типа `domain_error` для генерирования исключения мы можем предоставить ему `string`-значение с описанием происшедшей "неприятности". Программа, которая перехватит исключение, может, как показано в разделе 4.2.3, использовать это `string`-значение в своем диагностическом сообщении.

Обратите внимание на еще один аспект, связанный с характером поведения функций. Вызывая функцию, мы можем считать ее параметры локальными переменными, начальными значениями которых являются значения соответствующих аргументов. Тогда выходит, что вызов функции включает копирование аргументов в ее параметры. В частности, при вызове функции `median` вектор, используемый в качестве аргумента, будет скопирован в параметр `vec`.

При использовании функции `median` полезно скопировать аргумент в параметр, даже если на это потребуется значительное время, поскольку функция `median` изменяет значение своего параметра благодаря вызову функции `sort`. При копировании аргумента изменения, внесенные функцией `sort`, не будут распространяться на автора вызова функции. Такое поведение имеет смысл, поскольку вызов функции `median` для заданного `vector`-значения не должен вносить изменения в сам вектор.

4.1.2. Пересмотр политики вычисления оценок

Функция `grade`, представленная в разделе 4.1, предполагает, что мы уже имеем дело с обобщенной оценкой студента за выполнение домашних заданий, а не просто с набором оценок за отдельные домашние задания. Способ получения этой оценки является частью нашей политики: в разделе 3.1 мы вычисляли среднее арифметическое значение, а в разделе 3.2.2 — медиану. Поэтому вполне логично будет выразить эту часть нашей оценочной политики в виде функции, используя те же строки кода, что и в разделе 4.1.

```
// Вычисляем итоговую оценку студента на основе оценок,
// полученных на экзаменах в середине и конце семестра,
// а также на основе вектора оценок за выполнение домашних заданий.
// Эта функция не копирует свой аргумент, поскольку
// функция median делает это за нас.
double grade(
      double midterm, double final, const vector<double>& hw)
{
    if (hw.size() == 0)
        throw domain_error(
           "Студент не сделал ни одного домашнего задания ");
    return grade(midterm, final, median(hw));
}
```

В этой функции имеет смысл остановиться на трех моментах.

Прежде всего, обратите внимание на тип `const vector<double>&`, который мы задали для третьего аргумента. Этот тип часто называется "ссылкой на `const`-вектор `double`-значений". Заявляя, что некоторое имя есть ***ссылка*** (reference) на некоторый объект, мы имеем в виду, что оно является еще одним именем для этого объекта. Например, следующие строки кода

```
vector<double> homework;
vector<double>& hw = homework; // hw – синоним для homework.
```

говорят о том, что hw — еще одно имя для объекта homework. С этого момента все, что мы делаем с объектом, именуемым hw, эквивалентно выполнению аналогичных действий с объектом homework и наоборот. Следующий код (с использованием слова const)

```
// chw – синоним для объекта homework,
// предназначенный только для чтения.
const vector<double>& chw = homework;
```

по-прежнему означает, что chw — еще одно имя для объекта homework, но модификатор const "обещает", что мы не будем выполнять никаких действий с объектом chw, которые могли бы изменить его значение.

Поскольку ссылка — это еще одно имя для исходного объекта, такого понятия, как ссылка на ссылку, не существует. Определение ссылки на ссылку имеет такой же эффект, как определение ссылки на исходный объект. Например, если мы запишем следующий код

```
// hw1 и chw1 – синонимы для объекта homework.
// Ссылка chw1 предназначена только для чтения.
vector<double>& hw1 = hw;

const vector<double>& chw1 = chw;,
```

то hw1 (как и hw) — еще одно имя для объекта homework, а chw1 (как и chw) — еще одно имя для объекта homework, не позволяющее доступ для записи.

Определяя *неконстантную* ссылку, т.е. ссылку, которая позволяет запись, мы не можем сделать ее ссылкой на const-объект или const-ссылку, поскольку это потребует разрешения на отрицание константности. Следовательно, мы не можем записать

```
vector<double>& hw2 = chw; // Ошибка: требует доступа
                           // для записи в объект chw ,
```

поскольку "обещали" не модифицировать объект chw.

Аналогично, когда мы утверждаем, что некоторый параметр имеет тип const vector<double>&, запрашиваем у C++-среды прямой доступ к соответствующему аргументу, отказываясь от его копирования, и при этом обещаем, что не будем изменять значение этого параметра (ведь в противном случае аргумент также был бы изменен). Поскольку параметр является ссылкой на const-объект, мы можем вызвать функцию grade от имени как const-, так и не-const-векторов. Коль параметр является ссылкой, мы избегаем расхода системных ресурсов на копирование аргумента.

Еще один аспект, на который нам бы хотелось обратить ваше внимание, — это сама функция grade. Эта функция (та, версия которой приведена первоначально в разделе 4.1) "спокойно" "носит" имя grade, несмотря на то что она вызывает другую функцию grade. Понятие, предусматривающее наличие сразу нескольких функций с одинаковыми именами, называется ***перегрузкой*** (overloading) и достаточно широко используется во многих C++-программах. В применении двух функций с одинаковыми именами нет никакой двусмысленности, поскольку при вызове функции grade мы предоставляем список аргументов, а C++-среда по типу третьего аргумента способна "догадаться", какую именно функцию grade мы имеем в виду.

Наконец, заметьте, что мы проверяем, равно ли выражение homework.size() нулю, даже несмотря на то что знаем, что функция median сделает это за нас. Дело в том, что если функция median обнаружит, что мы хотим вычислить медиану пустого вектора, она сгенерирует исключение, которое включает сообщение Медиана пустого вектора. Это сообщение может оказаться не слишком информативным для того, кто с помощью нашей программы вычисляет итоговые

оценки студентов. Следовательно, мы генерируем собственное исключение, которое, смеем надеяться, даст пользователю ключ к пониманию причины "неожиданного" поведения программы.

4.1.3. Считывание оценок за выполнение домашних заданий

Еще одна проблема, которую мы должны решить в нескольких контекстах, состоит в считывании в вектор оценок за домашние задания.

В разработке поведения такой функции существует одна проблема: ей нужно вернуть два значения сразу. Одно значение — это, конечно, считанные оценки за домашние задания. Второе же должно служить индикатором успешной (или нет) попытки ввода данных.

Нужно признать, что прямого способа вернуть из функции более одного значения не существует. Однако косвенным путем это все-таки можно сделать, а именно передать функции параметр-ссылку на объект, в котором можно поместить один из желаемых результатов. Такая стратегия весьма распространена для функций чтения входных данных, поэтому и мы воспользуемся ею. Тогда наша функция будет выглядеть следующим образом.

```
// Считываем оценки за домашние задания из входного потока
// в вектор типа vector<double>.
istream& read_hw(istream& in, vector<double>& hw) {
    // Эту часть нам предстоит еще заполнить.
    return in;
}
```

В разделе 4.1.2 была приведена программа с параметром `const vector<double>&`; сейчас же мы отказываемся от использования модификатора `const`. Параметр-ссылка без слова `const` обычно свидетельствует о намерении модифицировать объект, который является аргументом функции. Например, при выполнении следующих инструкций

```
vector<double> homework;
read_hw(cin, homework);
```

тот факт, что второй параметр функции `read_hw` является ссылкой, должен натолкнуть нас на мысль, что вызов функции `read_hw` может изменить значение объекта `homework`.

Поскольку мы ожидаем, что функция будет модифицировать свои аргументы, ее нельзя вызвать посредством лишь одного выражения. Вместо этого мы должны передать параметру-ссылке ***l-значение*** (lvalue) аргумента. Упомянутое *l*-значение представляет собой *сохраняемое* (в противоположность *временному*) значение объекта (т.е. выражение, которое может находиться в левой части оператора присваивания и семантически представляет собой адрес размещения переменной, массива, элемента структуры и т.п.). Например, любая переменная является *l*-значением, как и ссылка или результат вызова функции, которая возвращает ссылку. Выражение, генерирующее арифметическое значение, например `sum / count`, не является *l*-значением.

Оба параметра функции `read_hw` являются ссылками, поскольку мы ожидаем, что функция изменит состояние обоих аргументов. Даже если мы и не знаем подробности работы входного потока `cin`, нам достаточно того, что библиотека определяет его в виде некоторой структуры данных, которая сохраняет все, что нужно знать библиотеке о состоянии нашего входного файла. При считывании входных данных из стандартного входного файла изменяется состояние этого файла, поэтому и значение `cin` должно логически измениться.

Обратите внимание на то, что функция `read_hw` возвращает значение переменной `in`. Более того, она обращается с ней, как с ссылкой. В действительности, мы говорим, что нам передан объект, который мы не собираемся копировать, а возвратим тот же объект, снова-таки не копируя его. Поскольку функция возвращает поток, запись

```
if (read_hw(cin, homework)) { /* ... */ }
```

может служить сокращенным вариантом записи следующих инструкций.

```
read_hw(cin, homework);
if (cin) { /* ... */ }
```

Теперь мы можем подумать о том, как прочитать оценки, полученные за домашние задания. Совершенно ясно, что мы хотим прочитать столько оценок, сколько их существует (а не фиксированное их количество), поэтому может показаться, что мы могли бы просто записать следующие инструкции.

```
// Наш первый блин комом.
double x;
while (in >> x)
    hw.push_back(x);
```

Эта стратегия не работает, причем по двум причинам.

Первая причина — мы не определили объект `hw`, поэтому неизвестно, какие данные могли бы туда уже попасть. Хотя мы знаем, что если наша функция используется для обработки оценок за домашние задания многих студентов, переменная `hw` могла бы содержать оценки предыдущего студента. Мы же можем решить эту проблему, вызвав перед началом нашей работы функцию `hw.clear()`.

Вторая причина провала нашей стратегии — мы не вполне представляем, где нужно остановить цикл. Мы можем продолжать чтение до тех пор, пока это возможно, но здесь у нас возникает проблема. Есть две причины, по которым мы можем не прочитать очередную оценку: обнаружен признак конца файла или прочитано данное, которое не является оценкой.

В первом случае автор вызова нашей функции может подумать, что мы прочитали признак конца файла. Эта мысль будет справедливой, но в то же время и обманчивой, поскольку признак конца файла обнаружится только после того, как мы успешно прочитаем все данные. Обычно признак конца файла означает, что попытка ввести данные потерпела неудачу.

Во втором случае (при обнаружении данного, не являющегося оценкой) библиотека отметит входной поток как находящийся в состоянии отказа. Это означает, что будущие запросы на ввод данных будут обречены на неудачу, как если бы мы обнаружили конец файла. Следовательно, автор вызова нашей функции может подумать, что что-то не в порядке с вводимыми данными, в то время как единственная проблема заключалась в том, что за последней оценкой последовало данное, не являющееся оценкой за домашнее задание.

В любом случае нам бы хотелось сделать вид, что мы ничего не видели после получения последней оценки за домашнее задание. Такое притворство облегчает ситуацию: обнаружение признака конца файла означает отсутствие непрочитанных входных данных, а при попытке прочитать данное, не являющееся оценкой, библиотека оставит его непрочитанным до следующей попытки ввода данных. Следовательно, все, что мы должны сделать, — "велеть" библиотечным средствам ввода игнорировать все, что привело попытку ввести данные к провалу, будь то признак конца файла или неверное значение. Упомянутое указание библиотеке выражается в виде вызова функции `in.clear()`, позволяющей сбросить состояние ошибки объекта `in`, после чего библиотека сможет продолжать ввод данных, несмотря на сбой.

И еще. Ведь не исключено обнаружение признака конца файла или неверного значения еще до попытки прочитать *первую* оценку за домашнее задание. В этом случае мы должны оставить входной поток в абсолютном покое, чтобы по неосторожности не соблазнить автора вызова нашей функции на попытку прочитать несуществующие входные данные в какой-то момент в будущем.

Итак, настало время представить функцию `read_hw` в полном объеме.

```
// Считываем оценки за домашние задания из входного потока
// в вектор типа vector<double>.
istream& read_hw(istream& in, vector<double>& hw)
```

```
{
    if (in) {
        // Избавляемся от предыдущего содержимого.
        hw.clear();

        // Считываем оценки за домашние задания.
        double x;
        while (in >> x)
            hw.push_back(x);

        // Очищаем поток, чтобы средства ввода данных
        // были готовы принять оценки следующего студента.
        in.clear();
    }
    return in;
}
```

Обратите внимание на то, что функция-член `clear` ведет себя совершенно по-разному для `istream`- и `vector`-объектов. Для `istream`-объектов она сбрасывает любые признаки ошибок, что позволяет продолжать ввод данных, а для `vector`-объектов она отбрасывает любое содержимое, которое могло бы быть в векторе, оставляя нам пустой вектор.

4.1.4. Три вида параметров функции

Хотелось бы остановиться на следующем. Мы определили три функции: `median`, `grade` и `read_hw`, которые работают с векторами `homework`. Каждая из этих функций обрабатывает соответствующий параметр совершенно не так, как это делают остальные, и каждая обработка преследует свою цель.

Параметр функции `median` (см. раздел 4.1.1) имеет тип `vector<double>`. Следовательно, при вызове этой функции аргумент копируется, несмотря на его размеры (а ведь это может быть вектор огромного размера). Невзирая на такую неэффективность, тип `vector<double>` выбран правильно, поскольку он гарантирует, что вычисление медианы вектора не изменит сам вектор. Функция `median` сортирует значения, хранимые в ее параметре-векторе, с помощью функции `sort`. Если бы при вызове этой функции аргумент не копировался, то при выполнении `median(homework)` значение вектора `homework` изменилось бы.

Функция `grade`, которая принимает вектор с оценками за домашние задания (см. раздел 4.1.2), использует для этого параметра тип `const vector<double>&`. Символ "`&`" (в обозначении этого типа) не велит C++-среде копировать аргумент, при этом модификатор `const` "обещает", что программа не изменит значение этого параметра. Такие параметры — важное средство повышения эффективности работы программ. Они очень полезны, когда функция не должна менять значение параметра, который имеет тип `vector` или `string` и способен принимать такие значения, что их копирование может потребовать немалых затрат времени. Обычно не стоит беспокоиться об использовании `const`-ссылок для параметров таких простых встроенных типов, как `int` или `double`. Такие маленькие объекты обычно настолько быстро копируются, что системные затраты на передачу их по значению весьма несущественны.

Параметр функции `read_hw` имеет тип `vector<double>&`, причем, заметьте, без использования модификатора `const`. И снова-таки, символ "`&`" велит C++-среде напрямую связать параметр с аргументом, не выполняя операции копирования аргумента. Однако в этом случае причина обойтись без копирования состоит в *намерении* функции изменить значение аргумента.

Аргументы, соответствующие параметрам, которые по типу являются неконстантными ссылками, должны представлять собой *l*-значения, т.е. они *не* должны быть "временно живущими" объектами. Аргументы, которые передаются по значению или "вынуждены" подчиняться ограни-

чениям, накладываемым типом const-ссылки, могут содержать любые значения. Предположим, у нас есть функция, которая возвращает пустой вектор.

```
vector<double> emptyvec()
{
    vector<double> v; // Вектор не содержит элементов.
    return v;
}
```

Мы могли бы вызвать эту функцию и использовать результат вызова в качестве аргумента для нашей второй функции grade из раздела 4.1.2.

```
grade(midterm, final, emptyvec());
```

При выполнении функция grade должна немедленно сгенерировать исключение, поскольку ее аргумент пуст. Однако сам вызов функции grade таким способом был бы синтаксически легален. При вызове функции read_hw оба ее аргумента должны быть *l*-значениями, поскольку оба параметра определены как неконстантные ссылки. Если функции read_hw

```
read_hw(cin, emptyvec()); // Ошибка: параметр emptyvec() -
                          // не l-значение.
```

передать вектор, не являющийся *l*-значением, компилятору это будет не по нраву, поскольку неименованный вектор, который создается при обращении к функции emptyvec, будет разрушен сразу же по выходу из функции read_hw. Если бы нам все-таки позволили сделать такое обращение к функции (предположим, закрыл бы компилятор глаза на такие "мелочи"), то в результате мы сохранили бы входные данные в объекте, к которому нельзя получить доступ!

4.1.5. Использование функций для вычисления итоговой оценки студента

Теперь применим все рассмотренные выше функции в программе вычисления итоговых оценок студентов; точнее, в реконструкции программы, приведенной в разделе 3.2.2.

```
// include-директивы и using-объявления для использования
//                                  библиотечных средств.
// Код функции median из раздела 4.1.1.
// Код функции grade(double, double, double) из раздела 4.1.
// Код функции grade(double, double, const vector<double>&)
//                                          из раздела 4.1.2.
// Код функции read_hw(istream&, vector<double>&)
//                                          из раздела 4.1.3.
int main()
{
    // Запрашиваем и считываем имя студента.
    cout << "Пожалуйста, введите свое имя: ";
    string name;
    cin >> name;
    cout << "Привет, " << name << "!" << endl;

    // Запрашиваем и считываем оценки по экзаменам, проведенным
    // в середине и конце семестра.
    cout << "Пожалуйста, введите оценки по экзаменам, "
            "проведенным в середине и конце семестра: ";
    double midterm, final;
    cin >> midterm >> final;

    // Запрашиваем оценки за выполнение домашних заданий.
    cout << "Введите все оценки за выполнение домашних заданий, "
            "завершив ввод признаком конца файла: ";

    vector<double> homework;
```

```
    // Считываем оценки за выполнение домашних заданий.
    read_hw(cin, homework);

    // Вычисляем итоговую оценку, если это возможно.
    try {
        double final_grade = grade(midterm, final, homework);
        streamsize prec = cout.precision();
        cout << "Ваша итоговая оценка равна " << setprecision(3)
             << final_grade << setprecision(prec) << endl;
    } catch (domain_error) {
        cout << endl << "Вы должны ввести свои оценки. "
             "Пожалуйста, попытайтесь снова." << endl;
        return 1;
    }
    return 0;
}
```

Изменения, внесенные в более раннюю версию программы, связаны с тем, *как* мы считываем оценки за выполнение домашних заданий и *как* вычисляем и записываем результат.

После запроса на ввод оценок за домашние задания мы вызываем нашу функцию `read_hw`, чтобы прочитать вводимые пользователем данные. Инструкция `while` внутри функции `read_hw` в цикле считывает оценки до тех пор, пока не будет обнаружен признак конца файла или данное другого типа (не `double`).

Важнейшее новшество в этом примере — ***инструкция*** `try`. Она пытается выполнить инструкции, заключенные в фигурные скобки (`{}`), расположенные за ключевым словом `try`. Если в любой из этих инструкций возникнет исключение типа `domain_error`, их выполнение прекращается, а управление передается другому набору инструкций, заключенных в другие фигурные скобки, которые являются частью ***инструкции*** `catch`. Эта инструкция начинается с ключевого слова `catch` и указывает тип перехватываемого ею исключения.

Если инструкции, расположенные между `try` и `catch`, выполняются без генерирования исключения, программа полностью игнорирует выполнение инструкции `catch` и продолжает работу со следующей (после `catch`) инструкции (в данном примере инструкции `return 0;`).

При написании `try`-инструкции необходимо всегда помнить о возможных побочных эффектах и о том, когда они могут возникнуть. Мы должны быть готовы к тому, что любая инструкция, расположенная между `try` и `catch`, может сгенерировать исключение. В этом случае любые вычисления, которые должны выполняться после "виновницы в исключении", игнорируются. Таким образом, важно понимать, что реальная последовательность выполнения инструкций необязательно должна совпадать с последовательностью инструкций, записанной в тексте программы.

Предположим, мы бы могли кратко записать блок вывода информации следующим образом.

```
// Этот пример не работает.
try {
    streamsize prec = cout.precision();
    cout << "Ваша итоговая оценка равна " << setprecision(3)
         << grade(midterm, final, homework)
         << setprecision(prec);
} ...
```

Проблема этого фрагмента кода состоит в том, что хотя операторы "<<" должны выполняться слева направо, C++-среда не обязана вычислять операнды в данном конкретном порядке. В частности, она может вызвать функцию `grade` после вывода фразы `Ваша итоговая оценка равна`. Если функция `grade` сгенерирует исключение, то выходные данные будут состоять только из этой (ставшей неуместной) фразы. Более того, первое обращение к функции `setprecision` может установить точность выходного потока равной значению `3` и не предоставить возможности (посредством второго обращения к той же функции) восстановить предыдущее значение точности. В качестве альтернативного варианта C++-среда могла бы вызвать функцию `grade` до вывода

каких-либо данных — точные действия (вернее, их последовательность) зависят исключительно от конкретной C++-среды.

Цель этого анализа — пояснить, почему мы разделили блок вывода данных на две инструкции: первая инструкция гарантирует, что обращение к функции `grade` будет выполнено до формирования каких бы то ни было выходных данных.

Очень хорошее правило, которое следует всегда соблюдать, гласит: избегайте нескольких побочных эффектов в одной инструкции. Генерирование исключения — это побочный эффект, поэтому инструкция, которая может сгенерировать исключение, не должна быть источником никаких других побочных эффектов, особенно включающих операции ввода-вывода.

Конечно, мы не можем выполнить нашу функцию `main` в таком виде, как она представлена выше. Необходимо включить `include`-директивы и `using`-объявления для доступа к библиотечным средствам, которые "вовсю" использует наша программа. Мы также применяем функцию `read_hw` и перегруженную функцию `grade`, которая принимает в качестве третьего аргумента ссылку `const vector<double>&`. Причем в определении этой функции, в свою очередь, используется функция `median` и функция-тезка `grade`, которая принимает три `double`-значения.

Для выполнения этой программы необходимо определить (в надлежащем порядке) перечисленные выше функции *до* определения нашей функции `main`. Тем самым мы завершили бы работу над созданием несколько "разбухшей" программы. Но пока рано ставить точку в этой истории, лучше повременить до раздела 4.3, в котором показано, как можно расчленять подобные программы и размещать их части в отдельных файлах. Однако спешить не будем и рассмотрим (конечно, для пользы дела) более удачные, если таковые возможны, способы структуризации данных.

4.2. Организация данных

Вычисление итоговой оценки одного студента может быть и полезно, но подобное вычисление настолько простое, что с ним может прекрасно справиться и карманный калькулятор. Если же мы хотим сделать нашу программу полезной для преподавателя, то нужно, чтобы она могла вычислять итоговые оценки для группы студентов.

Вместо интерактивного общения программы с каждым студентом (на предмет его успеваемости), мы можем предположить, что у нас есть файл, содержащий некоторое количество имен студентов и соответствующих оценок. За каждым именем в этом файле записаны оценки, полученные по экзаменам, проведенным в середине и конце семестра, а за ними — одна или несколько оценок за выполнение домашних заданий. Такой файл может иметь следующий вид.

```
Сахно 93 91 47 90 92 73 100 87
Карпенко 75 90 87 92 93 60 0 98
...
```

Наша программа должна вычислить итоговую оценку каждого студента с помощью метода медианы, причем медианная оценка за домашние задания составляет 40%, оценка за последний экзамен — 40% и оценка за промежуточный экзамен — 20%.

Для приведенных выше данных результат работы программы должен быть следующим.

```
Карпенко       86.8
Сахно          90.4
...
```

Выходные данные организованы следующим образом: фамилии студентов упорядочены по алфавиту, а итоговые оценки выровнены по вертикали, чтобы их было легче читать. Такие требования к оформлению выходных данных означают, что нам нужно место для хранения записей всех студентов, чтобы затем мы могли расположить их в алфавитном порядке. Нам также нужно найти

значение длины самой длинной фамилии, чтобы знать, сколько пробелов поместить между каждой фамилией и соответствующей ей оценкой.

Предполагая, что у нас есть место для хранения данных об одном студенте, мы можем использовать вектор для хранения данных обо всех студентах. Если вектор будет содержать данные обо всех студентах, мы сможем отсортировать хранимые в нем значения, а затем вычислить и вывести итоговые оценки для каждого студента. Начнем с создания структуры данных и написания некоторых вспомогательных функций для чтения и обработки этих данных. После разработки описанных абстракций мы сможем использовать их для решения проблемы в целом.

4.2.1. Соберем-ка все данные о студентах в одну кучу!

Если мы знаем, что нам нужно прочитать данные о каждом студенте, а затем упорядочить их в алфавитном порядке, имеет смысл совместно хранить имена студентов и их оценки. Следовательно, нам нужен способ хранения в одном месте всей информации, которая относится к одному студенту. Роль такого места может играть некоторая структура данных, содержащая имя студента, оценки, полученные по экзаменам, проведенным в середине и конце семестра, а также все оценки за выполнение домашних заданий. В C++ такая структура определяется следующим образом.

```
struct Student_info {
    string name;
    double midterm, final;
    vector<double> homework;
}; // Обратите внимание на точку с запятой – она обязательна.
```

Это определение структуры говорит о том, что `Student_info` — это тип, который имеет четыре данных-члена. Поскольку `Student_info` — тип, мы можем определить объекты этого типа, каждый из которых будет содержать экземпляр этих четырех данных-членов.

Первый член, именуемый `name`, имеет тип `string`; второй и третий (оба типа `double`) называются `midterm` и `final`, а последний — это вектор `double`-значений по имени `homework`.

Каждый объект типа `Student_info` содержит информацию об одном студенте. Поскольку `Student_info` — это тип, для хранения произвольного числа студентов мы можем использовать объект типа `vector<Student_info>` (подобно тому, как мы применяли объект типа `vector<double>` для хранения произвольного числа оценок за выполненные домашние задания).

4.2.2. Управление записями с данными о студентах

Нашу задачу можно разбить на три управляемых компонента, которые вполне представимы в виде отдельных функций: ввод данных в объект типа `Student_info`, вычисление итоговой оценки для объекта типа `Student_info` и сортировка содержимого вектора объектов типа `Student_info`.

Функция считывания данных (одной записи) во многом подобна функции `read_hw`, описанной в разделе 4.1.3. И в самом деле, мы вполне можем использовать функцию для ввода оценок за выполнение домашних заданий. Но кроме оценок за домашние задания, нам еще нужно ввести имя студента и оценки по двум экзаменам.

```
istream& read(istream& is, Student_info& s)
{
    // Чтение и сохранение имени студента и оценок по двум
    // экзаменам (проведенным в середине и конце семестра).
    is >> s.name >> s.midterm >> s.final;

    read_hw(is, s.homework); // Чтение и сохранение всех оценок
                             // за выполнение домашних заданий.
    return is;
}
```

В имени функции `read` нет никакой неоднозначности, поскольку тип второго параметра точно указывает, какие данные она считывает. Несмотря на перегрузку, эту функцию нельзя спутать с любой другой одноименной функцией, которая может быть использована для ввода данных в структуру какого-нибудь другого типа. Подобно функции `read_hw`, эта функция принимает в качестве параметров две ссылки: одну — на поток `istream`, из которого вводятся данные, а другую — на объект, в котором прочитанные данные должны быть сохранены. Используя параметр `s` внутри функции, мы воздействуем на состояние переданного функции аргумента.

Эта функция вводит значения в члены объекта `s` с именами `name`, `midterm` и `final`, а затем вызывает функцию `read_hw`, чтобы ввести значения оценок за домашние задания. В любой момент этого процесса может быть обнаружен признак конца файла или может произойти сбой в операции ввода. В этом случае последующие попытки ввода данных ни к чему не приведут, поэтому после выхода из функции объект `is` будет находиться в соответствующем состоянии ошибки. Обратите внимание на то, что такое поведение обусловлено тем, что функция `read_hw` (см. раздел 4.1.3) предусмотрительно оставляет входной поток в состоянии ошибки, если он уже находился в таком состоянии на момент вызова функции `read_hw`.

Нам необходима еще одна функция, которая вычисляет итоговую оценку для объекта типа `Student_info`. Мы уже решили б*о*льшую часть этой проблемы, определив функцию `grade` в разделе 4.1.2. Мы продолжим работу в этом направлении, перегрузив функцию `grade` версией, которая вычисляет итоговую оценку для объекта типа `Student_info`.

```
double grade(const Student_info& s)
{
    return grade(s.midterm, s.final, s.homework);
}
```

Эта функция обрабатывает объект типа `Student_info` и возвращает `double`-значение, представляющее итоговую оценку. Обратите внимание на то, что параметр функции имеет тип `const Student_info&`, а не просто `Student_info`, поэтому при вызове функции мы не расходуем системные ресурсы на копирование всего `Student_info`-объекта.

Заметьте также, что эта функция не предохраняет от исключения, генерируемого функцией `grade`, которую она вызывает. Дело в том, что не существует никаких других средств, которые могла бы использовать наша функция `grade` для обработки исключения, кроме тех, которые уже использовала "внутренняя" функция `grade`. Поскольку наша функция `grade` не перехватывает никаких исключений, любое исключение, которое может быть сгенерировано, будет передано назад автору вызова нашей функции, а ему, вероятно, лучше знать (чем нам), что делать со студентами, которые не выполнили домашних заданий.

Итак, прежде чем написать всю программу в целом, нам осталось решить, как отсортировать содержимое вектора объектов типа `Student_info`. В функции `median` (см. раздел 4.1.1) мы сортировали содержимое параметра типа `vector<double>`, именуемого `vec`, с помощью библиотечной функции `sort`.

```
sort(vec.begin(), vec.end());
```

Тогда, предположив, что наши данные содержатся в векторе `students`, мы тем не менее не можем просто записать следующее.

```
sort(students.begin(), students.end()); // Не верно.
```

Почему? Подробнее о функции `sort` и о других библиотечных алгоритмах мы поговорим в главе 6, но пока имеет смысл абстрактно поразмышлять о том, как действует функция `sort`. В частности, откуда ей "знать", как именно переставлять значения в заданном векторе?

Функция `sort` должна сравнивать элементы вектора в порядке их следования. Для сравнения элементов заданного типа она использует оператор "<". Мы можем спокойно вызвать функцию

sort для вектора типа vector<double>, поскольку оператор "<" корректно сравнит два double-значения и сгенерирует соответствующий результат. Но что произойдет, когда функция sort попытается сравнить значения типа Student_info? Ведь оператор "<" попросту "не умеет" работать с объектами типа Student_info. И в самом деле, при попытке функции sort сравнить два таких объекта компилятор не замедлит выразить свое недовольство.

К счастью, функция sort принимает третий необязательный аргумент, который является ***предикатом*** (predicate). Предикат — это функция, которая генерирует значение истинности типа bool. Если этот третий аргумент присутствует, функция sort будет использовать его для сравнения элементов вместо оператора "<". Следовательно, нам нужно определить функцию, которая принимает два объекта типа Student_info, а затем сообщает результат сравнения первого объекта со вторым (меньше ли он). Поскольку мы хотим упорядочить фамилии студентов по алфавиту, запишем нашу функцию сравнения, которая будет сравнивать только фамилии.

```
bool compare(const Student_info& x, const Student_info& y)
{
    return x.name < y.name;
}
```

Эта функция просто делегирует свою работу по сравнению двух Student_info-объектов классу string, который предоставляет оператор "<" для сравнения строк (string-объектов). Этот оператор сравнивает строки, используя обычный лексикографический порядок (dictionary ordering), согласно которому левый операнд считается меньше правого, если он в алфавитном порядке стоит впереди правого операнда. Именно такое поведение нам и нужно было определить.

Определив функцию compare, мы можем отсортировать значения, хранимые в векторе, передав эту функцию библиотечной функции sort в качестве ее третьего аргумента.

```
sort(students.begin(), students.end(), compare);
```

Таким образом, для сравнения заданных элементов функция sort будет теперь вызывать нашу функцию compare.

4.2.3. Построение отчета

Теперь, когда у нас есть функции обработки записей с данными о студентах, мы можем сгенерировать наш отчет.

```
int main()
{
    vector<Student_info> students;
    Student_info record;
    string::size_type maxlen = 0;

    // Считываем и сохраняем все записи, затем находим
    // длину самой длинной фамилии.
    while (read(cin, record)) {
        maxlen = max(maxlen, record.name.size());
        students.push_back(record);
    }

    // Упорядочиваем записи по алфавиту.
    sort(students.begin(), students.end(), compare);

    for (vector<Student_info>::size_type i = 0;
        i != students.size(); ++i) {

        // Выводим фамилию, дополненную справа
        // maxlen + 1 символами.
        cout << students[i].name
             << string(maxlen + 1 -
```

```
                    students[i].name.size(), ' ');
        // Вычисляем и выводим оценку.
        try {
            double final_grade = grade(students[i]);
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                 << setprecision(prec);
        } catch (domain_error e) {
            cout << e.what();
        }
        cout << endl;
    }
    return 0;
}
```

Мы уже рассмотрели бо́льшую часть этого кода, но на некоторых моментах имеет смысл остановиться.

Во-первых, мы впервые использовали библиотечную функцию `max`, которая определена в заголовке `<algorithm>`. С первого взгляда поведение этой функции кажется очевидным. Однако и здесь требуются дополнительные разъяснения. Аргументы этой функции должны быть одинакового типа, но о причинах этого требования мы поговорим в разделе 8.1.3. Пока главное для нас — определить `maxlen` как переменную типа `string::size_type`, а не просто типа `int`.

Во-вторых, мы впервые использовали выражение следующего вида.

```
string(maxlen + 1 - students[i].name.size(), ' ')
```

Это выражение создает безымянный объект (см. раздел 1.1) типа `string`. Данный объект содержит `maxlen + 1 - students[i].name.size()` символов, причем все они являются пробелами. Это выражение аналогично определению переменной `space` из раздела 1.2, но здесь опущено имя подобной переменной. Это "упущение" эффективно превращает определение в выражение. Запись значения этого выражения после фамилии (`students[i].name`) обеспечивает вывод такого строкового объекта, в котором символы члена `students[i].name` дополняются справа нужным количеством пробелов, чтобы всего было выведено ровно `maxlen + 1` символов.

В `for`-инструкции индекс `i` используется для поэлементного опроса структуры данных `students`, т.е. на каждой итерации цикла индекс `i` позволяет обратиться к текущему элементу структуры типа `Student_info` и получить значение члена `name`. Затем мы выводим значение члена `name` из этого объекта, используя построенное соответствующим образом `string`-значение, которое состоит из пробелов, "доводящих" до нужной длины данный элемент выводимой информации.

Затем мы выводим итоговую оценку, рассчитанную для каждого студента. Если студент не выполнил ни одного домашнего задания, в процессе вычисления этой оценки будет сгенерировано исключение. В этом случае мы перехватываем исключение и, вместо вывода числового значения оценки, выводим сообщение, переданное как часть объекта исключения. Все исключения стандартной библиотеки (в том числе и исключение типа `domain_error`) запоминают (необязательный) аргумент, используемый для описания проблемы, послужившей причиной генерирования исключения. Каждый из упомянутых типов исключений создает копию содержимого этого аргумента, доступного посредством функции-члена с именем `what`. Инструкция `catch` в этой программе присваивает имя объекту исключения, который она получает из функции `grade` (см. раздел 4.1.2), чтобы вывести это сообщение из функции `what()`. В данном случае это сообщение уведомит пользователя, что `Студент не сделал ни одного домашнего задания`. При отсутствии исключений мы используем манипулятор `setprecision` для установки нужной нам точности, а именно вывода числовых значений с тремя значащими цифрами, а затем, собственно, и выводим результат вызова функции `grade`.

4.3. А теперь соберем все вместе

Итак, мы определили ряд абстракций (функций и структур данных), которые используются при решении различных проблем вычисления итоговых оценок студентов. Единственный способ использовать эти абстракции — поместить все их определения в один файл и скомпилировать этот файл. Очевидно, что сложность программирования при таком подходе очень быстро растет. В целях упрощения язык C++, подобно многим другим языкам, поддерживает ***механизм раздельной компиляции*** (separate compilation), который позволяет помещать разные части одной программы в отдельные файлы и компилировать эти файлы независимо от остальных.

Начнем с функции `median`: как с нею поступить, чтобы другие программные элементы могли ее использовать. Для начала поместим определение функции `median` в отдельный файл, чтобы его можно было скомпилировать. Этот файл должен включать объявления для всех имен, которые используются в функции `median`. Из библиотечных средств функция `median` применяет тип `vector`, функцию `sort` и исключение типа `domain_error`, поэтому мы должны включить в наш файл соответствующие заголовки.

```
// Исходный файл для функции median.
#include <algorithm> // Для доступа к объявлению функции sort.
#include <stdexcept> // Для доступа к объявлению
                     // класса domain_error.
#include <vector> // Для доступа к объявлению класса vector.

using std::domain_error; using std::sort; using std::vector;

// Вычисление медианы вектора vector<double>.
double median(vector<double> vec)
{
    // Тело функции в соответствии с определением
    // из раздела 4.1.1.
}
```

Как и в случае с любым другим файлом, мы должны присвоить нашему исходному файлу имя. Стандарт C++ не выдвигает никаких требований насчет имен исходных файлов, но, в общем, имена таких файлов должны отражать их содержимое. Однако большинство C++-сред на имена исходных файлов накладывают ограничения, обычно требуя, чтобы последние несколько символов имени имели некоторую конкретную форму. C++-среды используют эти суффиксы имен файлов для определения, является ли данный файл исходным файлом C++. Большинство C++-сред требует, чтобы имена исходных C++-файлов имели расширения `.cpp`, `.C` или `.c`, поэтому мы можем поместить нашу функцию `median` в файл с именем `median.cpp`, `median.C` или `median.c` (в зависимости от конкретной C++-среды).

Теперь мы должны сделать нашу функцию `median` доступной для других пользователей. По аналогии со стандартной библиотекой, которая помещает определяемые ею имена в заголовки, мы можем написать собственный ***заголовочный файл*** (header file), который позволит пользователям получать доступ к определяемым нами именам. Например, в файле `median.h` мы могли бы сообщить о существовании нашей функции `median`, и тогда пользователи смогут использовать ее в своих программах, написав следующее.

```
// Гораздо более удачный способ использования функции median.
#include "median.h"
#include <vector>

int main() { /* ... */ }
```

Используя директиву `#include` с именем заголовка, заключенным в *двойные кавычки* (вместо угловых скобок), мы тем самым указываем компилятору, что он, вместо этой директивы

#include, должен скопировать в нашу программу все содержимое соответствующего заголовочного файла. В каждой C++-среде по-своему решается вопрос, где искать указанные заголовочные файлы и какие взаимоотношения существуют между строкой, заключенной в кавычки, и именем файла. Словосочетание "заголовочный файл median.h" следует понимать как сокращенный вариант фразы "файл, который с точки зрения C++-среды соответствует имени median.h".

Необходимо отметить, что хотя мы рассматриваем наши заголовки как заголовочные файлы, тем не менее мы относимся к заголовкам, предоставленным C++-средой, как к стандартным заголовкам, а не как к стандартным заголовочным файлам. Дело в том, что эти заголовочные файлы являются реальными файлами в каждой C++-среде, а системные заголовки необязательно реализованы как файлы. Даже несмотря на то что директива #include используется для доступа как к заголовочным, так и системным заголовкам, не существует требования их одинаковой реализации.

Теперь, когда мы знаем, что должны предоставить заголовочный файл, возникает вполне логичный вопрос о его содержимом. Ответ очень простой: в этот заголовочный файл мы должны включить ***объявление*** (declaration) функции median, заменив (в ее определении) тело функции точкой с запятой. Мы можем также исключить имена параметров, поскольку они нерелевантны без тела функции.

```
double median(vector<double>);
```

Наш заголовок median.h не может содержать только одно это объявление; мы должны также включить все имена, которые в нем используются. Это объявление использует тип vector, поэтому мы должны быть уверены, что это имя будет доступно "глазам" компилятора до того, как он "увидит" наше объявление.

```
// median.h
#include <vector>
double median(std::vector<double>);
```

Мы включаем заголовок vector, чтобы в объявлении аргумента для функции median можно было использовать имя std::vector. Более существенно то, что мы при этом явно указали тип std::vector, а не написали using-объявление.

В целом, заголовочные файлы должны объявлять только те имена, которые действительно необходимы. Накладывая ограничения на имена, содержащиеся в заголовочном файле, мы тем самым оставляем максимальную гибкость для наших пользователей. Например, мы используем составное имя std::vector, поскольку заранее не можем знать, как пользователь нашей функции median захочет указать тип std::vector. Пользователи нашего кода, возможно, и не захотят использовать для вектора using-объявление. Если бы в своем заголовке мы написали using-объявление, то все программы, включающие наш заголовок, получили бы объявление using std::vector, независимо от того, хотели они того или нет. Заголовочные файлы должны использовать только составные имена, а не using-объявления.

И наконец, последнее. Каждый заголовочный файл должен гарантировать безопасность своего неоднократного включения как части компиляции программы. Что касается нашего заголовка, то он в этом смысле совершенно безопасен, поскольку содержит только объявления. Однако мы считаем хорошим стилем программирования предусмотреть (и обслужить) его многократное включение в каждый заголовочный файл. Это можно реализовать добавлением в файл некоторых препроцессорных средств.

```
#ifndef GUARD_median_h
#define GUARD_median_h

// median.h – окончательная версия.
#include <vector>
double median(std::vector<double>);

#endif
```

Директива `#ifndef` проверяет, определена ли переменная `GUARD_median_h`. `GUARD_median_h` — это имя ***препроцессорной переменной***, которая предоставляет один из возможных способов управления компиляцией программы. Полное рассмотрение препроцессора выходит за рамки этой книги.

В данном контексте директива `#ifndef` "просит" препроцессор обработать все, что находится между ею и следующей (соответствующей ей) директивой `#endif`, если заданное (в директиве `#ifndef`) имя *не определено*. Для проверки мы должны выбрать уникальное имя, поэтому создаем его из имени нашего файла и строки `GUARD_` в надежде, что у такого сложного имени не будет двойников.

При первом включении заголовка `median.h` в программу переменная `GUARD_median_h` будет неопределенной, поэтому препроцессор просмотрит остаток файла. Первое, что он сделает, — определит переменную `GUARD_median_h` (с помощью директивы `#define`), чтобы последующие попытки включить в программу заголовок `median.h` были обречены на неудачу.

Осталось отметить только один нюанс в этой истории: директива `#ifndef` должна быть самой первой строкой заголовочного файла, чтобы даже ни один комментарий не предшествовал ей.

```
#ifndef Переменная
...
#endif
```

Некоторые C++-среды обнаруживают файлы такой формы и, если указанная переменная определена, даже не пытаются прочитать файл второй раз.

4.4. Декомпозиция программы вычисления итоговых оценок

Теперь, когда мы знаем, что надо сделать, чтобы скомпилировать функцию `median` отдельно, в качестве следующего "номера нашей программы" нужно подготовить структуру `Student_info` и соответствующие функции.

```
#ifndef GUARD_Student_info
#define GUARD_Student_info

// Заголовочный файл Student_info.h
#include <iostream>
#include <string>
#include <vector>

struct Student_info {
    std::string name;
    double midterm, final;
    std::vector<double> homework;
};

bool compare(const Student_info&, const Student_info&);
std::istream& read(std::istream&, Student_info&);
std::istream& read_hw(std::istream&, std::vector<double>&);
#endif
```

Обратите внимание на то, что при использовании имен из стандартной библиотеки мы не включаем `using`-объявления, а явно приписываем префикс `std::`; кроме того, заголовок `Student_info.h` объявляет функции `compare`, `read` и `read_hw`, которые тесно связаны со структурой `Student_info`. Мы воспользуемся этими функциями только в том случае, если также используем эту структуру, поэтому имеет смысл упаковать эти функции вместе с определением структуры.

Эти функции должны быть определены в исходном файле, который будет выглядеть примерно так.

```
// Исходный файл для функций, связанных со
// структурой Student_info.
#include "Student_info.h"

using std::istream; using std::vector;

bool compare(const Student_info& x, const Student_info& y)
{
    return x.name < y.name;
}

istream& read(istream& is, Student_info& s)
{
    // В соответствии с определением из раздела 4.2.2.
}

istream& read_hw(istream& in, vector<double>& hw)
{
    // В соответствии с определением из раздела 4.1.3.
}
```

Обратите внимание на то, что поскольку мы включили сюда (посредством директивы `#include "Student_info.h"`) файл `Student_info.h`, исходный файл будет содержать как объявления, так и определения наших функций. Эта избыточность не только безопасна, но даже и полезна. Она дает компилятору возможность проверить соответствие между объявлениями и определениями. В большинстве C++-сред такие проверки не претендуют на полноту, поскольку для полной проверки нужно "видеть" программу целиком. Тем не менее они достаточно полезны, чтобы в исходные файлы имело смысл включать соответствующие заголовочные файлы.

Упомянутая проверка и ее неполнота проистекают из следующего: язык требует, чтобы объявления функций и их определения в точности соответствовали одни другим по типу результата, количеству и типу параметров. Это объясняет способность C++-среды проверить это соответствие — так в чем же тогда состоит неполнота проверки? Дело в том, что если объявление и определение в чем-то различны, среда может допустить, что они описывают две различные версии перегруженной функции и недостающее определение находится где-то в другом месте. Например, мы определили функцию `median`, как показано в разделе 4.1.1, а затем некорректно объявили ее следующим образом.

```
int median(std::vector<double>); // Возвращаемое значение
                                 // должно иметь тип double.
```

Если компилятор встретит это объявление, то при компиляции определения функции он выразит свое "недоумение", поскольку знает, что тип значения, возвращаемого функцией `median(vector<double>)`, не может быть одновременно и `double` и `int`. Однако предположим, что мы использовали следующее объявление.

```
double median(double); // Аргумент должен иметь
                       // тип vector<double>.
```

Теперь компилятор "не имеет права жаловаться" на нас, поскольку функция `median(double)` может быть определена где-нибудь в другой части программы. Если мы вызываем эту функцию, C++-среда должна в конце концов найти ее определение. Если это ей не удастся, она "молчать" не станет.

Обратите также внимание на то, что в исходном файле нет проблем с использованием `using`-объявлений. В отличие от заголовочного файла, исходный файл не оказывает никакого влияния на программы, которые используют эти функции. Следовательно, уверенное применение `using`-объявлений в исходном файле — решение исключительно местного значения.

Теперь нам осталось написать заголовочный файл для объявления различных перегруженных функций grade.

```
#ifndef GUARD_grade_h
#define GUARD_grade_h

// grade.h
#include <vector>
#include "Student_info.h"
double grade(double, double, double);
double grade(double, double, const std::vector<double>&);
double grade(const Student_info&);

#endif
```

Обратите внимание на то, как сведение объявлений этих перегруженных функций облегчает просмотр всех альтернатив. Мы определим все эти три функции в одном файле, поскольку у них "близкое родство". И снова-таки, имя файла в зависимости от C++-среды может быть следующим: grade.cpp, grade.C или grade.c.

```
#include <stdexcept>
#include <vector>
#include "grade.h"
#include "median.h"
#include "Student_info.h"

using std::domain_error; using std::vector;
// Определения для функций grade из
// разделов 4.1, 4.1.2 и 4.2.2.
```

4.5. Исправленная версия программы вычисления итоговых оценок

Наконец, мы можем записать конечный вариант нашей программы.

```
#include <algorithm>
#include <iomanip>
#include <ios>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>
#include "grade.h"
#include "Student_info.h"

using std::cin;             using std::setprecision;
using std::cout;            using std::sort;
using std::domain_error;    using std::streamsize;
using std::endl;            using std::string;
using std::max;             using std::vector;

int main()
{
    vector<Student_info> students;
    Student_info record;
    string::size_type maxlen = 0; // Длина самого длинного
                                  //                имени.
    // Считываем и сохраняем все данные об оценках студента.
    // Инвариант: вектор students содержит все записи,
    //                          прочитанные до сих пор.
    //            max содержит длину самой длинной фамилии
    //                            name в структуре students.
```

```
    while (read(cin, record)) {
        // Находим длину самой длинной фамилии.
        maxlen = max(maxlen, record.name.size());
        students.push_back(record);
    }

    // Упорядочиваем записи с данными о студентах по алфавиту.
    sort(students.begin(), students.end(), compare);

    // Выводим фамилии и оценки.
    for (vector<Student_info>::size_type i = 0;
        i != students.size(); ++i) {

        // Выводим фамилию, дополненную справа
        // maxlen + 1 символами.
        cout << students[i].name
             << string(maxlen + 1 -
                students[i].name.size(), ' ');

        // Вычисляем и выводим итоговую оценку.
        try {
            double final_grade = grade(students[i]);
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                 << setprecision(prec);
            } catch (domain_error e) {
                cout << e.what();
        }
        cout << endl;
    }
    return 0;
}
```

Эта программа не должна быть сложной для понимания. Как обычно, она начинается с необходимых директив `include` и `using`-объявлений. Конечно же, мы должны включить сюда только те заголовки и объявления, которые используются в этом исходном файле. В данной программе, помимо библиотечных, мы использовали и заголовки "собственноручного изготовления". Эти заголовки делают доступным определение типа `Student_info`, а также объявления функций, которые мы используем для обработки объектов типа `Student_info` и вычисления итоговых оценок. Функция `main` осталась такой же, какой она была представлена в разделе 4.2.3.

4.6. Резюме

Структура программы

`#include <`*Системный заголовок*`>`

> В угловые скобки (<>) заключаются системные заголовки, которые могут быть реализованы (а может, и нет) как файлы

`#include "`*Имя заголовочного файла, определенного пользователем*`"`

> Заголовочный файл, определенный пользователем, вставляется в программу с помощью директивы `#include` и посредством заключения его имени в кавычки. Обычно заголовки, определенные пользователем, имеют суффикс `.h`

Следует предотвращать многократное включение заголовочных файлов, используя директиву `#ifndef GUARD_`*Имя_заголовка*. В заголовках не должны объявляться имена, которые там не используются. В частности, они должны не включать `using`-объявления, а в явном виде предварять имена стандартной библиотеки префиксом `std::`.

Типы

`T&` — Обозначает ссылку на тип `T`. Чаще всего используется для передачи параметра, который может быть изменен функцией. Аргументы, соответствующие таким параметрам, должны быть *l*-значениями

`const T&` — Обозначает ссылку на тип `T`, которую нельзя использовать для изменения значения, связанного с этой ссылкой. Обычно применяется, чтобы избежать затрат на копирование параметра в функцию

Структуры. Структура — это тип, который содержит некоторое (возможно, нулевое) число членов. Каждый объект структурного типа содержит собственный экземпляр каждого члена структуры.

Каждая структура должна иметь соответствующее определение.

```
struct Имя_типа
{
    Спецификатор_типа Имя_члена;
    . . .
}; // Обратите внимание на точку с запятой.
```

Подобно другим определениям, определение структуры может присутствовать в исходном файле только один раз, поэтому обычно оно включается в заголовочный файл, защищенный соответствующим образом.

Функции. Функция должна быть объявлена в каждом исходном файле, в котором она используется, а определена только однажды. Объявления и определения имеют подобную форму.

```
// Объявление функции:
Тип_возврата Имя_функции(Список_типов_параметров);

// Определение функции:
[ inline ] Тип_возврата Имя_функции(Список_типов_параметров){
    // Здесь должно быть тело функции.
}
```

Здесь элемент *Тип_возврата* представляет тип значения, возвращаемого функцией, а под элементом *Список_типов_параметров* подразумевается список типов параметров функции, разделенных запятыми. Функции должны быть объявлены до их вызова. Тип каждого аргумента должен быть совместим с соответствующим параметром. (Синтаксис объявления или определения функций с более сложными типами возвращаемых значений рассматриваются в разделе A.1.2.)

Имена функций могут быть перегруженными: с одним и тем же именем может быть объявлено несколько функций; при условии, что эти функции будут отличаться по числу или типам параметров. Имейте в виду, что C++-среда может причислить ссылку и `const`-ссылку к одному и тому же типу.

В определении функции может быть (но это необязательно) использовано ключевое слово `inline`, которое предлагает компилятору развернуть, где это возможно, обращения к этой функции “в строку”, т.е. во избежание затрат, связанных с вызовом функции, заменить каждый ее вызов копией тела функции, модифицированного соответствующим образом. Чтобы реализовать такую замену, компилятор должен видеть определение функции, поэтому `inline`-функции обычно определяются в заголовочных, а не в исходных файлах.

Обработка исключений

`try { // код`

Инициирует блок, который может сгенерировать исключение

```
} catch(t) { /* код */ }
```

Завершает `try`-блок и обрабатывает исключение, которое соответствует типу `t`. Код, следующий за инструкцией `catch`, выполняет действие, соответствующее обработке исключения, указанного аргументом `t`

`throw e;` Прекращает выполнение текущей функции; передает значение `e` автору вызова

Классы исключений. Библиотека определяет ряд классов исключений, по именам которых можно догадаться о характере проблем, для сообщения о которых они используются.

```
logic_error    domain_error      invalid_argument
length_error   out_of_range      runtime_error
range_error    overflow_error    underflow_error
```

`e.what()` Возвращает значение, которое сообщает о том, что стало причиной ошибочной ситуации

Библиотечные средства

`s1 < s2` Сравнивает `string`-объекты `s1` и `s2` на основе лексикографического порядка

`s.width(n)` Устанавливает ширину потока `s` равной `n` для следующей операции вывода (или оставляет ее без изменения, если параметр `n` опущен). Результат дополняется слева до заданной ширины. Возвращает предыдущее значение ширины. Операторы стандартного вывода используют существующее значение ширины, а затем вызывают функцию `width(0)` для его сброса

`setw(n)` Возвращает значение типа `streamsize` (см. раздел 3.1), что при выводе в выходной поток `s` равносильно вызову функции `s.width(n)`

Упражнения

4.0. Скомпилируйте, выполните и протестируйте программы, приведенные в этой главе.

4.1. Как было отмечено в разделе 4.2.3, важно, чтобы типы аргументов в обращении к функции `max` в точности совпадали. Будет ли тогда работать следующий код? Если здесь существует проблема, то как ее можно исправить?

```
int maxlen;
Student_info s;
max(s.name.size(), maxlen);
```

4.2. Напишите программу вычисления квадратов `int`-значений до 100. Эта программа должна вывести два столбца: в первом должно быть значение, а во втором — квадрат этого значения. Для управления выводом данных, т.е. чтобы выводимые значения были выровнены по столбцам, используйте функцию `setw` (см. описание выше).

4.3. Что произойдет, если переписать предыдущую программу так, чтобы она выводила числа (и их квадраты) до 1000 (но не включала значение 1000), и при этом пренебречь изменением аргументов, передаваемых функции `setw`? Перепишите программу так, чтобы она была более защищенной "перед лицом" изменений, которые позволяют переменной `i` расти, не корректируя аргументы функции `setw`.

4.4. А теперь измените свою программу вычисления квадратов, используя вместо `int`-значений значения типа `double`. Используйте манипуляторы для управления выводом данных так, чтобы значения были выровнены по столбцам.

4.5. Напишите функцию, которая считывает слова из входного потока и сохраняет их в векторе. Используйте эту функцию для написания программ, которые подсчитывают количество слов во входном потоке, а также фиксируют, сколько раз встречается в нем каждое слово.

4.6. Перепишите структуру `Student_info` для немедленного вычисления оценок и сохранения значения только итоговой оценки.

4.7. Напишите программу вычисления среднего арифметического от чисел, содержащихся в векторе типа `vector<double>`.

4.8. Если следующий код допустим, то какой можно сделать вывод о типе значения, возвращаемого функцией `f`?

```
double d = f()[n];
```