Оптимизация и эффективность

Зачастую нам требуется сделать более эффективной ту или иную часть нашей программы, и имеется масса вариантов оптимизаций, которые могут помочь нам в этом.

Время от времени появляются предположения, что использование ключевого слова `const` помогает компилятору при выполнении оптимизации кода. Так ли это на самом деле? Почему? Помимо `const`, множество программистов для повышения эффективности кода часто используют другое ключевое слово — `inline`. Влияет ли это слово на производительность программ? Если да, то когда и каким образом? Когда может быть выполнено встраивание кода функции, как под контролем программиста, так и без него?

И наконец, мы завершим этот раздел примером того, как знания предметной области помогают при разработке приложения с высокой производительностью, которой невозможно достичь никакими низкоуровневыми оптимизациями, играми с битами и другими подобными способами улучшения кода.

Задача 24. Константная оптимизация Сложность: 3

Помогает ли корректное применение ключевого слова `const` компилятору при оптимизации кода? Обычная реакция программиста на этот вопрос: "Да, конечно!" Не будем спешить...

Вопрос для новичка

1. Рассмотрим следующий исходный текст.

```
const Y& f( const X& x ) {
    // ... некоторые действия с x и поиск объекта Y ...
    return someY;
}
```

Помогает ли объявление параметра и/или возвращаемого значения с использованием `const` компилятору сгенерировать более оптимальный код или улучшить его каким-то иным образом? Обоснуйте ваш ответ.

Вопрос для профессионала

2. Поясните, может ли в общем случае наличие или отсутствие ключевого слова `const` помочь компилятору улучшить генерируемый им код и почему.

3. Рассмотрим следующий исходный текст.

```
void f( const Z z ) {
    // ...
}
```

Ответьте на следующие вопросы.

а) При каких условиях и для каких видов классов `Z` это конкретное указание `const` может помочь сгенерировать другой, лучший код?

б) Говорим ли мы об оптимизации компилятором или о некотором другом типе оптимизации при условиях, рассмотренных в пункте *а*)? Поясните свой ответ.

в) В чем состоит лучший путь достижения того же эффекта?

Решение

const: ненавязчивый сервис

1. Рассмотрим следующий исходный текст.

```
// Пример 24-1
const Y& f( const X& x ) {
    // ... некоторые действия с  x и поиск объекта  Y ...
    return someY;
}
```

Помогает ли объявление параметра и/или возвращаемого значения с использованием `const` компилятору сгенерировать более оптимальный код или улучшить его каким-то иным образом?

Коротко говоря — нет, вряд ли.

Обоснуйте ваш ответ.

Что же именно компилятор может улучшить? Может ли он избежать копирования аргумента или возвращаемого значения? Нет, поскольку аргумент уже передается по

ссылке, и возвращаемое значение также уже является ссылкой. Может ли он разместить копию x или someY в памяти, доступной только для чтения? Нет, поскольку и x, и someY располагаются вне пределов функции и приходят в нее из "внешнего мира" и туда же возвращаются. Даже если someY динамически выделяется "на лету" в самой функции f, и сам объект, и владение им передается за пределы функции вызывающему коду.

А что можно сказать о коде внутри функции f? Может ли компилятор как-то улучшить генерируемый для тела функции f код на основе указаний const? Это приводит нас ко второму, более общему вопросу.

2. Поясните, может ли в общем случае наличие или отсутствие ключевого слова const помочь компилятору улучшить генерируемый им код и почему.

Обращаясь вновь к примеру 24-1, становится очевидно, что здесь остается актуальной та же причина, по которой константность параметров не может привести к улучшению кода. Даже если вы вызываете константную функцию-член, компилятор не может полагаться на то, что не будут изменены биты объектов x или someY. Кроме того, имеются дополнительные проблемы (если только компилятор не выполняет глобальную оптимизацию). Компилятор может не знать, нет ли другого кода, в котором имеется неконстантная ссылка, являющаяся псевдонимом объектов x и/или someY, и не могут ли они быть случайно изменены через эту ссылку в процессе работы функции f. Компилятор может также не знать, объявлены ли реальные объекты, для которых x и someY являются простыми ссылками, как константные.

Только того факта, что x и someY объявлены как const, не достаточно, чтобы их биты были физически константны. Почему? Поскольку любой класс может иметь члены, объявленные как mutable, или внутри функций-членов класса может использоваться приведение const_cast. Даже код внутри самой функции f может выполнить приведение const_cast либо преобразование типов в стиле C, что сведет на нет все объявления const.

Есть один случай, когда ключевое слово const может действительно что-то значить, — это происходит тогда, когда объекты сделаны константными в точке их описания. В этом случае компилятор часто может поместить такие "действительно константные" объекты в память "только для чтения", в особенности если это объекты POD[42], для которых их образ в памяти может быть создан в процессе компиляции и размещен в выполнимой программе. Такие объекты пригодны для размещения в ПЗУ.

Как const может оптимизировать

3. Рассмотрим следующий исходный текст.

```
// Пример 24-2
void f( const Z z ) {
    // ...
}
```

Ответьте на следующие вопросы.

а) **При каких условиях и для каких видов классов Z это конкретное указание const может помочь сгенерировать другой, лучший код?**

Если компилятор знает, что z — действительно константный объект, он в состоянии выполнить некоторую оптимизацию даже без глобального анализа. Например, если тело f содержит вызов наподобие g(&z), то компилятор может быть уверен, что все части z, не являющиеся mutable, не изменятся в процессе вызова g.

[42] См. пояснения о том, что такое POD, в предыдущем разделе, на стр. 159. — *Прим. перев.*

Однако кроме этого случая, запись const в примере 24-2 не является оптимизацией для большинства классов Z, а там, где это все же приводит к оптимизации, — это уже не оптимизация генерации объектного кода компилятором.

С точки зрения генерации кода компилятором вопрос в основном сводится к тому, не может ли компилятор опустить создание копии или разместить z в памяти только для чтения. То есть было бы неплохо, если бы мы знали, что z действительно не изменяется в процессе работы функции f, поскольку теоретически это означает, что мы могли бы использовать непосредственно внешний объект, передаваемый функции в качестве аргумента, без создания его копии, или, если мы все же делаем его копию, то ее можно было бы разместить в памяти только для чтения, если это дает выигрыш в производительности или желательно по каким-либо иным соображениям.

В общем случае компилятор не может использовать константность параметров для устранения создания копии аргумента или предполагать побитовую константность. Как уже упоминалось, имеется слишком много ситуаций, когда данные предположения оказываются неверными. В частности, Z может иметь члены, описанные как mutable, или где-то (в самой функции f, в некоторой другой функции или в непосредственном или косвенном вызове функции-члена Z) может быть выполнено приведение типов const_cast или использованы какие-то другие трюки.

Имеется один случай, когда компилятор способен сгенерировать улучшенный код, если:

- определения копирующего конструктора Z и всех функций Z, прямо или косвенно использующихся в теле функции f, видимы в данной точке;
- эти функции достаточно просты и не имеют побочного действия;
- компилятор имеет агрессивный оптимизатор.

В этом случае компилятор может быть уверен в корректности своих действий и может не создавать копию объекта в соответствии с правилом, которое гласит, что компилятор может выполнять любые оптимизации, при условии, что соответствующая стандарту программа дает те же результаты, что и без них.

В качестве отступления стоит упомянуть об одной детали. Некоторые программисты утверждают, что есть еще один случай, когда компилятор может генерировать лучший код на основании const, а именно — при выполнении глобальной оптимизации. Дело в том, что это утверждение остается верным и в том случае, если убрать из него упоминание const. Не важно, что глобальная оптимизация все еще весьма редка и дорога; настоящая проблема заключается в том, что глобальная оптимизация использует всю имеющуюся информацию об объекте, в том числе о его реальном использовании, так что такая оптимизация работает одинаково независимо от того, объявлен ли объект как const или нет — решение принимается на основании того, что в действительности происходит с объектом, а не того, что обещает программист. Так что и в этом случае применение ключевого слова const ничего не дает в плане оптимизации генерируемого кода.

Заметим, что несколькими абзацами ранее я сказал, что “запись const в примере 24-2 не является оптимизацией для большинства классов Z” и для “генерации объектного кода компилятором”. В оптимизаторе компилятора имеется гораздо больше возможностей, чем кажется, и в некоторых случаях const может быть полезен для некоторой реальной оптимизации.

б) **Говорим ли мы об оптимизации компилятором или о некотором другом типе оптимизации при условиях, рассмотренных в пункте *а*)? Поясните свой ответ.**

Например, автор Z может выполнять некоторые действия с константным объектом по-другому, написав более эффективную версию функции для случая с константным объектом.

Пусть в примере 24-2 Z — класс “handle/body”, такой как класс String с использованием подсчета ссылок для выполнения отложенного копирования.

```
// Пример 24-3
//
void f( const String s ) {
    // ...
    s[4];    // или использование итераторов
    // ...
}
```

Для данного класса `String` известно, что вызов оператора `operator[]` для константного объекта `String` не должен приводить к изменению содержимого строки, так что можно предоставить константную перегрузку оператора `operator[]`, которая возвращает значение типа `char` вместо ссылки `char&`:

```
class String {
    // ...
public:
    const char operator[]( size_t ) const;
    char&      operator[]( size_t );
    // ...
}
```

Аналогично, класс `String` может предоставить вариант итератора `const_iterator`, оператор `operator*` для которого возвращает значение типа `char`, а не `char&`.

Если выполнить описанные действия и после этого воспользоваться оператором `operator[]` или итераторами, а переданный по значению аргумент объявлен как `const`, то тогда — чудо из чудес! — класс `String` без какой-либо дополнительной помощи, автомагически :) оптимизирует ваш код, устранив глубокое копирование...

в) **В чем состоит лучший путь достижения того же эффекта?**

...но того же эффекта можно достичь, просто передавая аргумент по ссылке.

```
// Пример 24-4: более простой способ
//
void f( const Z& z ) {
  // ...
}
```

Этот способ работает независимо от того, использует ли объект идиому handle/body или подсчет ссылок или нет, так что просто воспользуйтесь им!

➢ Рекомендация

Избегайте передачи параметров как константных значений. Предпочтительно передавать их как ссылки на константные объекты, кроме тех случаев, когда это простые объекты наподобие `int`, с мизерными затратами на копирование.

Резюме

Вера в то, что ключевое слово `const` помогает компилятору генерировать более качественный код, очень распространена. Да, `const` действительно хорошая вещь, но основная цель данной задачи — показать, что предназначено это ключевое слово в первую очередь для человека, а не для компиляторов или оптимизаторов..

Когда речь идет о написании безопасного кода, `const` — отличный инструмент, который позволяет программистам писать более безопасный код с дополнительными проверками компилятором. Но когда речь идет об оптимизации, то `const` остается в принципе полезным инструментом, поскольку позволяет проектировщикам классов лучше выполнять оптимизацию вручную; но генерировать лучший код компиляторам оно помогает в гораздо меньшей степени.

Задача 25. inline Сложность: 7

Быстро ответьте: когда выполняется встраивание? И можно ли написать функцию, которая гарантированно никогда не окажется встраиваемой? В этой задаче мы рассмотрим много различных случаев встраивания, причем некоторые из них вас удивят.

Вопрос для новичка

1. Что такое встраивание (inlining)?

Вопрос для профессионала

2. Когда выполняется встраивание? Может ли оно выполняться:
   а) во время написания исходного текста?
   б) во время компиляции?
   в) во время компоновки?
   г) при инсталляции приложения?
   д) в процессе работы?
   е) в некоторое другое время?

3. Дополнительный вопрос: какого рода функции гарантированно не будут встраиваемыми?

Решение

Какой ответ выберете вы на второй вопрос? Если вы выберете ответы *а*) или *б*), то вы не одиноки. Это наиболее распространенные ответы на данный вопрос, и в книге [Sutter02] я уже вкратце обращался к этой теме. Но если бы тема этим и исчерпывалась, то мы могли бы на этом завершить рассмотрение задачи и отправиться на пикник. Но на этот раз пикника не будет. У настоящего мужчины всегда найдется что сказать. Причем сказать можно достаточно много.

Причина появления данной задачи в книге — стремление показать, почему наиболее точный ответ на главный вопрос задачи — любой из перечисленных, а на последний — "никакие". Непонятно, почему? Тогда читайте дальше.

Краткий обзор

1. Что такое встраивание (inlining)?

Если вы читали [Sutter02][43], то вы можете пропустить этот и несколько следующих подразделов и перейти сразу к разделу "Ответ В: во время компоновки" на стр. 171.

Коротко говоря, встраиваемость означает замену вызова функции подстановкой копии ее тела. Рассмотрим, например, следующий исходный текст.

```
//  Пример 25-1
//
double Square( double x ) { return x * x; }
int main() {
    double d = Square( 3.14159 * 2.71828 );
}
```

Идея встраиваемости вызова функции (как минимум, концептуально) заключается в том, что программа преобразуется так, как если бы она была написана следующим образом.

[43] Задача 7.1 в русском издании книги. — *Прим. ред.*

```
int main() {
    const double __temp = 3.14159 * 2.71828;
    double d = __temp * __temp;
}
```

Такое встраивание устраняет излишние расходы на выполнение вызова функции, а именно — на внесение параметров в стек, переход процессором в другое место памяти для выполнения кода функции, что, помимо затрат на переход, может привести к полному или частичному сбросу кэша инструкций процессора. Встраивание — это далеко не то же, что и трактовка `Square` как макроса, поскольку вызов встраиваемой функции остается вызовом функции, и ее аргументы вычисляются только один раз. В случае же макросов вычисление аргументов может выполняться неоднократно; так, макрос `#define SquareMacro(x) ((x)*(x))` при вызове `SquareMacro(3.14159*2.71828)` будет раскрыт до `(3.14159*2.71828)*(3.14159*2.71828)` (т.е. умножение π на e будет выполнено не один раз, а два).

Заслуживает внимания еще один частный случай — рекурсивный вызов, когда функция вызывается из самой себя, непосредственно или опосредованно. Хотя такие вызовы зачастую не могут быть встраиваемыми, в некоторых случаях компилятор может сделать рекурсию встраиваемой так же, как может частично разворачивать некоторые циклы.

Между прочим, вы заметили, что в примере 25-1, который иллюстрирует встраиваемость, не использовано ключевое слово `inline`? Это сделано преднамеренно. Мы вернемся к этому моменту еще не раз в процессе рассмотрения основного вопроса задачи.

2. **Когда выполняется встраивание? Может ли оно выполняться:**
   а) **во время написания исходного текста?**
   б) **во время компиляции?**
   в) **во время компоновки?**
   г) **при инсталляции приложения?**
   д) **в процессе работы?**
   е) **в некоторое другое время?**
3. **Дополнительный вопрос: какого рода функции гарантированно не будут встраиваемыми?**

Ответ А: во время написания исходного текста

В процессе написания исходного текста разработчики могут использовать ключевое слово `inline` в своих программах. Это не является реальным выполнением встраивания в смысле перемещения кода для устранения вызова функции, но это попытка выбрать и явно выделить подходящие места для встраивания функции, так что мы будем рассматривать ее как наиболее раннюю возможность принятия решения о встраивании[44].

Когда вы намереваетесь написать ключевое слово `inline` в вашем исходном тексте, вы не должны забывать о трех важных вещах.

- *По умолчанию не делайте этого.* Преждевременная оптимизация — зло, и вы не должны использовать `inline` до тех пор, пока профилирование не покажет необходимость этого в определенных случаях. Более полную информацию по этому вопросу можно найти в [Sutter02] или запросив на поисковом сервере типа Google что-то вроде “преждевременная оптимизация” или “premature

[44] Еще одна интерпретация “во время написания исходного текста” может заключаться в буквальном встраивании функций некоторыми разработчиками путем физического перемещения блоков исходного текста. Это действие еще больше отдаляет нас от обычного понимания термина “встраивание”, так что я не буду его здесь рассматривать.

site:www.gotw.ca" — вы получите массу страшных предупреждений о преждевременной оптимизации вообще и встраивании в частности.

- *Это означает всего лишь "попробуйте, пожалуйста".* Как описано в [Sutter02], ключевое слово `inline` — всего лишь подсказка для компилятора, возможность попытаться мило поговорить с ним, предоставляемая языком программирования (далее будут описаны недостатки таких "милых" разговоров). Ключевое слово `inline` вообще не имеет никакого семантического действия в программе на C++. Оно не влияет на другие конструкции языка, на использование функции, объявленной `inline` (например, вы можете получить адрес такой функции), и нет никакой стандартной возможности программно определить, объявлена ли данная функция как встраиваемая или нет.
- *Это часто делается не на требуемом уровне детализации.* Мы пишем ключевое слово `inline` для *функции*, но когда выполняется встраивание, то в действительности оно происходит при *вызове* функции. Это отличие очень важно, поскольку одна и та же функция может (и зачастую должна) быть встраиваемой в одном месте вызова, но не в некотором другом. Ключевое слово `inline` не дает вам никакой возможности выразить этот факт, поскольку мы можем указать, что встраиваемой является функция сама по себе, что эквивалентно неявному указанию делать эту функцию встраиваемой везде, во всех возможных местах вызова. Такое предвидение редко бывает точным. Так что хотя в разговоре мы часто говорим о "встраиваемой функции", более точно было бы говорить о встраиваемом вызове функции.

➢ Рекомендация

Избегайте использования ключевого слова `inline` или других попыток оптимизации до тех пор, пока на их необходимость не укажут измерения производительности программы.

Ответ Б: во время компиляции

Обычно во время работы компиляторы сами, без внешней подсказки, выполняют описанный в примере 25-1 вид встраивания.

Что делает компилятор, когда мы мило разговариваем с ним путем объявления некоторых функций встраиваемыми? Это зависит от ситуации. Не все компиляторы хорошо поддерживают такие "милые" разговоры, даже если задаривать их шоколадом и цветами. Ваш компилятор запросто может проигнорировать вашу просьбу, и даже не одним, а тремя способами.

- Не делая встроенными вызовы функций, которые вы объявили как `inline`.
- Делая встроенными вызовы функций, которые вы *не* объявляли встраиваемыми.
- Встраивая некоторые из вызовов, оставляя другие вызовы той же функции обычными невстраиваемыми (независимо от того, объявлена ли функция как `inline`).

Вернемся к примеру 25-1 и обратим внимание на то, что в нем нигде не говорится, что функция должна быть встраиваемой. Это сделано преднамеренно, потому что этим я хотел проиллюстрировать тот факт, что встраивание все равно может произойти, даже без объявления функции `inline`. Не удивляйтесь, если ваш сегодняшний компилятор поступит именно так. Поскольку вы не можете написать соответствующую стандарту программу, которая выявит отличия при встраивании функции компилятором, этот способ оптимизации оказывается совершенно законным, и компилятор может (а часто и должен) выполнять его за вас.

Обычно современные компиляторы в состоянии лучше программиста решить, какие вызовы функций следует сделать встраиваемыми, а какие — нет, включая ситуации, когда одна и та же функция в разных местах может быть обработана по-разному. Почему? Простейшая причина в том, что компилятор знает больше о контексте вызова, поскольку ему известна "реальная" структура точки вызова — машинный код, сгенерированный для данной точки после применения других оптимизаций, таких как разворачивание циклов или удаление недостижимых ветвей программы. Например, компилятор может быть способен определить, что встраивание функции в некотором внутреннем цикле может сделать цикл слишком большим для размещения в кэше процессора, что приведет только к падению производительности, так что такой вызов не будет сделан встроенным, в то время как другие вызовы той же функции в других местах программы могут остаться встроенными.

Ответ B: во время компоновки

Теперь мы переходим к более интересным и более современным аспектам встраивания.

Вопрос: может ли функция быть встроена во время компиляции? Ответ: да. Этот ответ является основой дополнительного вопроса о функциях, которые не могут быть встраиваемыми ни при каких условиях. Этот вопрос добавлен мною в задачу, потому что обычно все верят, что такие виды функций существуют. В частности, обычно считается, что невозможно встроить функции, описания которых размещено в отдельном модуле, а не в заголовочном файле.

Давайте попытаемся выполнять встраивание максимально интенсивно, насколько это возможно. Рассмотрим пример 25-1 с небольшим изменением.

```
// Пример 25-2: затрудним работу оптимизатора, поместив
//              функцию в отдельный модуль, и сделав
//              недоступным ее исходное описание.
//

//--- Файл main.cpp ---
//
double Square( double x );
int main() {
    double d = Square( 3.14159 * 2.71828 );
}

//--- Файл square.obj (или .o) ---
//
// Содержит скомпилированное описание функции
// double Square( double x ) { return x * x; }
```

Здесь идея заключается в том, что реализация функции `Square` вынесена из единицы трансляции `main.cpp`. На самом деле сделано даже больше: недоступны даже исходные тексты функции `Square` — только ее объектный код. "Теперь вызов `Square` гарантированно не будет встраиваемым!" — скажет множество людей. Пока идет компиляция — они правы. В процессе компиляции `main.cpp` никакой самый мощный и продвинутый компилятор не в состоянии получить доступ к исходным текстам определения функции `Square`.

На вот продвинутый компоновщик с такой задачей справиться в состоянии, и некоторые популярные реализации так и поступают. Ряд компиляторов, в частности, Hewlett-Packard, поддерживают такое *межмодульное встраивание* (cross-module inlining); в Microsoft Visual C++ 7.0 (известном как ".NET") и более поздних версий имеется опция `/LTCG`, которая означает "link time code generation" (генерация кода в процессе компиляции). Реальное преимущество такой технологии "позднего встраивания" заключается в знании реального контекста каждой точки вызова, что

позволяет более интеллектуально подойти к вопросу о том, где и когда стоит выполнить встраивание.

Вот еще пища для размышлений: вы заметили где-нибудь в описании примера 25-2, что функция `Square` должна обязательно быть написана на C++? В этом и заключается второе преимущество технологии "позднего встраивания", которая нейтральна по отношению к языку. Функция `Square` может быть написана на Fortran или Pascal. Приятно. Все, о чем должен позаботиться компоновщик, — выяснить используемые функцией соглашения о передаче параметров и удалить код размещения аргументов в стеке и снятия с него, вместе с машинной командой вызова функции.

Но это далеко не все — настоящему мужчине всегда есть что сказать! Ведь мы только приступаем к серьезному разговору, так что не спешите...

Ответ Г: при инсталляции приложения

Теперь перемотаем пленку нашего разговора прямо к тому радостному моменту, когда мы наконец-то скомпилировали и скомпоновали наше приложение, собрали его в `tar`-файл, какой-нибудь `setup.exe` или `.wpi`, и гордо отправили упакованный компакт своему первому покупателю. Наступило самое время для того, чтобы:

а) сорвать наклейку с предупреждением о лицензии;

б) оплатить почтовые расходы;

в) объявить, что уж теперь-то все встраивание далеко позади.

Да?

Да, да, нет.

С середины 1990-х годов постоянно увеличивается количество продаж приложений, предназначенных для работы под управлением специализированных виртуальных машин. Это означает, что вместо того, чтобы скомпилировать программу в машинный код для конкретного процессора, операционной системы и API, приложение компилируется в поток байт-кода, который интерпретируется или компилируется на машине пользователя средой времени выполнения, что позволяет абстрагироваться от конкретных возможностей процессора или операционной системы. Наиболее распространенные примеры включают (но не ограничиваются) Java Virtual Machine (JVM) и .NET Common Language Run-time (CLR)[45]. В случае использования таких целевых сред компилятор транслирует исходный текст C++ в упомянутый поток байт-кода (известный также как язык команд виртуальной машины (virtual machine's instruction language, IL)), который представляет собой программу, созданную с использованием кодов операций из системы команд среды времени выполнения.

Отклоняясь от основной темы — некоторые из этих сред имеют очень богатые средства поддержки конструкций объектно-ориентированных языков на уровне системы команд, так что классы, наследование и виртуальные функции поддерживаются ими непосредственно. Компилятор для такой платформы может (и многие так и поступают) транслировать исходную программу в промежуточный язык команд виртуальной машины последовательно, класс за классом, функция за функцией, возможно, после выполнения некоторой собственной оптимизации, включающей возможность встраивания еще на уровне компиляции. Если компилятор работает таким образом, то исходный текст функции на C++ может быть более или менее полно восстановлен по коду функции с той же сигнатурой, представленной в целевой системе команд. Конечно, компилятор не обязан следовать описанной методике, но даже если он поступает как-то иначе, следующие далее замечания о встраивании остаются в силе.

---

[45] А также такие родственники CLR, как Mono, DotGNU и Rotor, которые реализуют также стандарт ISO Common Language Infrastructure (CLI), определяющий подмножество CLR.

Вернемся к нашему вопросу. Какое все это имеет отношение к встраиванию в процессе инсталляции приложения? Даже при использовании описанных сред времени выполнения в конечном счете процессор работает со своей собственной системой команд. Следовательно, среда отвечает за трансляцию языка команд виртуальной машины в код, который в состоянии понять процессор целевого компьютера. Очень часто это делается при первой инсталляции приложения, и именно в этот момент, как и в других процессах компиляции, могут быть выполнены (и часто выполняются) дополнительные оптимизации. В частности, компилятор .NET — NGEN IL — может выполнять встраивание в процессе инсталляции приложения, когда программа на языке IL транслируется в команды процессора, готовые к выполнению.

Здесь также стоит обратить внимание на нейтральность среды по отношению к языку программирования, так что оптимизация (включая встраивание), выполняемая в процессе инсталляции приложения, легко преодолевает границы применения отдельных языков программирования. Вас не должно удивлять, что если ваша программа на C# делает вызов небольшой функции на C++, то эта функция может в конечном итоге оказаться встроенной.

Так когда же выполнять встраивание слишком поздно? Никогда не говори никогда, потому что наш разговор еще не окончен...

Ответ Д: в процессе работы

Ну хорошо, но когда мы запускаем программу на выполнение — то уж здесь-то все возможности для встраивания должны быть позади?

Это может показаться совершенно невозможным, но встраивание вполне реально и в процессе выполнения программы, причем несколькими способами. В частности, я хочу упомянуть *оптимизацию, управляемую профилированием* (profile-directed optimization), и *защищенное встраивание* (guarded inlining). Так же, как и в случае среды, компилирующей программу при инсталляции, для этого нам потребуется наличие соответствующей поддержки времени выполнения на машине пользователя.

Идея, лежащая в основе оптимизации, управляемой профилированием, заключается в том, что когда приложение выполняется, вставленные в программу обработчики могут собирать информацию о том, как реально используется программа, в частности, какие функции вызываются чаще других и при каких условиях (например, размер рабочего множества по сравнению с общим размером кэша в момент вызова функции). Собранные данные могут быть использованы для модификации выполнимого образа программы, так что избранные вызовы функций могут стать встраиваемыми при настройке приложения на работу в целевой среде, основанной на измерениях производительности в процессе реального выполнения программы.

Защищенное встраивание представляет собой другой пример того, насколько агрессивной может оказаться оптимизация встраивания во время выполнения программы. В частности, в [Arnold00] и [JikesRVM] документирована Jikes Research Virtual Machine (RVM), динамический оптимизирующий компилятор née the Jalapeño для JVM. Помимо прочего, этот компилятор способен встраивать вызовы виртуальных функций, полагая, что получатель виртуального вызова будет иметь данный объявленный тип (чтобы избежать не только затрат на вызов функции, но и дополнительных затрат на диспетчеризацию виртуальности). На сегодняшний день компиляторы в состоянии сделать определенные вызовы виртуальных функций невиртуальными (и, таким образом, имеют возможность встраивать их), если целевой тип оказывается статически известен. Новое в среде Jikes/Jalapeño то, что теоретически она может делать вызовы невиртуальными и встраивать их, даже если статический целевой тип *не* известен. Однако поскольку такое предположение может оказаться неверным, компилятор вставляет дополнительную защиту, которая выполняет проверку типа целевого объекта в процессе выполнения программы, и если он не соответствует ожидаемому, то программа возвращается к механизму обычного вызова виртуальной функции.

Ответ E: в некоторое другое время

Вспомним, что в ответе *г*) мы рассматривали встраивание в процессе инсталляции в некоторой среде времени выполнения, такой как JVM или .NET CLR. Конечно, проницательные читатели уже заметили, что ранее я упоминал только трансляцию из байт-кода в машинные команды в процессе инсталляции приложения, но есть и другое более распространенное время такой трансляции, а именно — JIT, где аббревиатура JIT означает компиляцию "just-in-time", т.е. при необходимости.

Идея, лежащая в основе такого подхода, заключается в том, чтобы выполнять компиляцию функций только при необходимости, перед их явным использованием. Преимущество такого подхода в снижении стоимости компиляции программы в систему команд процессора, поскольку вместо одного большого этапа компиляции вы получаете много маленьких компиляций отдельных частей кода непосредственно перед их выполнением. Соответствующий недостаток такой технологии — в замедлении первых запусков программы и снижении качества оптимизации, так как такая компиляция должна выполняться очень быстро и не может затрачивать много времени на анализ встраивания и других возможных оптимизаций. Но такой компилятор все еще в состоянии выполнять оптимизации наподобие встраивания, и многие из них так и поступают, но вообще говоря, лучшие результаты можно получить при выполнении той же работы раньше, например, в процессе инсталляции (см. ответ *г*)), когда расходы времени не так критичны, и оптимизатор может позволить себе выполнить работу более тщательно и качественно.

> ➢ **Рекомендация**
>
> Встраивание может быть выполнено в любой момент времени.

Резюме

Подобно всем оптимизациям, встраивание зачастую более эффективно, если оно выполняется инструментарием, который осведомлен о генерируемом коде и/или среде выполнения, а не программистом. Чем позже выполняется встраивание, тем более точным и соответствующим целевой среде работы программы оно оказывается.

Мы говорим о встраивании функций, но гораздо более точно говорить о встраивании вызовов функций. В конце концов, одна и та же функция может оказаться встроенной в каком-то одном месте, но не в некоторых других. И, поскольку имеется масса возможностей для встраивания даже после завершения начальной компиляции, одна и та же функция может оказаться встроенной не только в разных местах, но и при помощи разного инструментария в каждом из этих мест.

Встраивание — это нечто гораздо большее, чем одно ключевое слово `inline`.

Задача 26. Форматы данных и эффективность. Часть 1: игры в сжатие. Сложность: 4

Умеете ли вы выбирать высококомпактные форматы данных, эффективно использующие память? Насколько хорошо вы справляетесь с кодом, работающим с отдельными битами? Эта и следующая задачи дают вам вполне достаточно возможностей развить оба этих навыка в процессе рассмотрения эффективного представления шахматных партий и битового буфера для их хранения.

Дополнительные требования: предполагается, что вы знакомы с азами шахмат.

Вопрос для новичка

1. Какой из перечисленных стандартных контейнеров использует меньше памяти для хранения одного и того же количества объектов одинакового типа `T`: `deque`, `list`, `set` или `vector`? Поясните свой ответ.

Вопрос для профессионала

2. Вы создаете всемирный шахматный сервер, который хранит все когда-либо сыгранные на нем партии. Поскольку база данных может оказаться очень большой, вы хотите представить каждую партию с минимально возможными затратами памяти. Здесь мы рассматриваем только ходы игры, игнорируя всю остальную информацию, например, имена игроков и комментарии.
Для каждого из приведенных далее размеров данных приведите формат представления партии, требующий указанное количество памяти для представления полухода (под полуходом мы подразумеваем ход одного игрока). Считается, что в одном байте — 8 битов.

а) 128 байтов на полуход

б) 32 байта на полуход

в) от 4 до 8 байтов на полуход

г) от 2 до 4 байтов на полуход

д) 12 битов на полуход

Решение

1. Какой из перечисленных стандартных контейнеров использует меньше памяти для хранения одного и того же количества объектов одинакового типа `T`: `deque`, `list`, `set` или `vector`? Поясните свой ответ.

Вспомним задачи 20 и 21, в которых говорилось об использовании памяти и структурах стандартных контейнеров. Каждый тип контейнера представляет собой тот или иной компромисс между используемой памятью и производительностью.

- `vector<T>` хранит данные в виде непрерывного массива объектов `T` и, таким образом, не имеет никаких дополнительных расходов на хранение одного элемента.
- `deque<T>` может рассматриваться как `vector<T>`, внутреннее представление которого разбито на отдельные блоки. `deque<T>` хранит блоки, или "страницы" объектов. Для этого требуется по одному дополнительному указателю на страницу, что обычно приводит к дополнительным расходам памяти, составляющим доли бита на один элемент. Других дополнительных затрат памяти нет, по-

скольку у deque<T> нет никаких дополнительных указателей или другой информации для отдельных объектов T.

- list<T> представляет собой двухсвязный список узлов, которые хранят элементы T. Это означает, что для каждого элемента T в list<T> хранятся также два указателя, которые указывают на предыдущий и последующий узлы списка. При вставке нового элемента в список мы создаем два дополнительных указателя, так что дополнительные затраты контейнера list<T> составляют два указателя на один хранимый элемент.
- И наконец, set<T> (и аналогичные в данном отношении multiset<T>, map<Key,T> и multimap<Key,T>) также гранят объекты T (или pair<const Key,T>) в узлах. Обычная реализация set<T> включает три дополнительные указателя на каждый узел. Зачастую, услышав об этом, многие спрашивают: "Откуда три указателя? Разве недостаточно двух — на левый и правый дочерние узлы?" Дело в том, что требуется еще один указатель на родительский узел, иначе задача определения "следующего" узла по отношению к некоторому произвольно взятому не может быть решена достаточно эффективно. (Возможны и другие способы реализации этих контейнеров; например, можно воспользоваться структурой списков с чередующимися пропусками (alternating skip list) — но и в этом случае требуется как минимум три указателя на каждый элемент (см. [Marrie00]).)

Частью решения об эффективном представлении данных в памяти является выбор правильного (как в смысле памяти, так и в смысле производительности) контейнера, поддерживающего необходимую вам функциональность. Но это еще не все: не менее важной является задача выбора эффективного представления данных, которые будут размещаться в этих контейнерах. Именно в этом и заключается суть рассматриваемой нами задачи.

Различные способы представления данных

Цель второго вопроса задачи — продемонстрировать, что может быть множество способов представления одной и той же информации.

2. Вы создаете всемирный шахматный сервер, который хранит все когда-либо сыгранные на нем партии. Поскольку база данных может оказаться очень большой, вы хотите представить каждую партию с минимально возможными затратами памяти. Здесь мы рассматриваем только ходы игры, игнорируя всю остальную информацию, например, имена игроков и комментарии.

В оставшейся части задачи мы используем следующие стандартные термины и аббревиатуры:

K King (Король)
Q Queen (Ферзь)
R Rook (Ладья)
B Bishop (Слон)
N Knight (Конь)
P Pawn (Пешка)

Поля на шахматной доске определяются горизонталью и вертикалью, к которым они относятся. Вертикали обозначаются слева направо (с точки зрения игрока белыми фигурами) буквами от *a* до *h*, а горизонтали — цифрами от 1 (горизонталь, ближайшая к игроку белыми фигурами) до 8.

Для каждого из приведенных далее размеров данных приведите формат представления партии, требующий указанное количество памяти для представления полухода (под полуходом мы подразумеваем ход одного игрока). Считается, что в одном байте — 8 битов.

а) **128 байтов на полуход**

Одно из представлений, требующее такого количества памяти, основано на предположении, что программа знает текущее размещение фигур на доске (которое выводится из предыдущих ходов) и сохраняет новую позицию на доске целиком, используя для этого по два байта для каждой клетки доски. В этом случае мы можем принять правило, что первый байт указывает цвет фигуры — `'W'` или `'B'` (или `'.'` для указания отсутствия фигуры в клетке), а во втором байте хранится тип фигуры — `'K'`, `'Q'`, `'R'`, `'B'`, `'N'`, `'P'` или `'.'`.

Используя эту схему и сохраняя доску по горизонталям от 1 до 8, и по вертикалям от *a* до *h* в пределах каждой горизонтали, возможное представление полухода может быть таким:

```
WRWNWBWQWKWBWNWR
WPWPWP..WPWPWPWP
................
......WP........
................
................
BPBPBPBPBPBPBPBP
BRBNBBBQBKBBBNBR
```

Здесь представлен полуход “1. d4”, с которого я обычно начинаю партию.

б) **32 байта на полуход**

Представление *а*) явно чрезмерно расточительно, поскольку представляет собой вполне удобочитаемый человеком текст, в то время как нам достаточно, чтобы формат могла прочесть машина. В конце концов, выводить позиции для пользователя базы данных будет специальное программное обеспечение.

Мы можем снизить количество необходимой памяти до 32 байт на полуход, храня всего лишь 4 бита информации для каждой клетки: 3 бита для указания фигуры (например, представление 0 для короля, 1 для ферзя, 2 для ладьи, 3 для слона, 4 для коня, 5 для пешки и 6 — для пустой клетки требуется 3 бита, при этом одно значение остается неиспользованным), и 1 бит для цвета (значение этого бита игнорируется для пустой клетки).

При использовании такой схемы для сохранения всей доски как и ранее, по горизонталям от 1 до 8, и по вертикалям от *a* до *h* в пределах каждой горизонтали, требуется всего лишь 32 байта:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

в) **от 4 до 8 байтов на полуход**

Этой величины можно достичь, используя представление полухода в виде текста в старой шахматной записи.

Старая “описательная” запись шахматной партии идентифицирует клетки с использованием дескрипторов переменной длины, наподобие K3 или QN8 вместо двухсимвольных дескрипторов вроде e3 или b8. Для записи полухода при этом требуется как минимум 4 символа (например, P-Q4) и не более 8 символов (например, RKN1-KB1, P-KB8(Q)). Заметим, что никакие завершающие нули или другие ограничители строк не требуются, поскольку записанные таким образом ходы дешифруются однозначно.

При этой схеме запись полухода может выглядеть как

```
P-KB8(Q)
```

г) **от 2 до 4 байтов на полуход**

Это достигается путем хранения полуходов как текста в современной записи шахматных партий.

Современная “алгебраическая” запись более компактна, и любой полуход можно записать с использованием от 2 символов (например, d4) до 4 символов (например,

Rgf1, gh=Q). В этом случае также не нужны никакие разделители в силу однозначности декодирования.[46]

При использовании этой схемы полуход может выглядеть следующим образом:

```
gh=Q
```

д) **12 битов на полуход**

Еще более компактную запись можно получить, применив иной подход. Полуход однозначно определяется исходной клеткой и клеткой назначения. Поскольку всего клеток 64, для представления одной клетки достаточно 6 битов, так что всего для записи полухода требуется 12 битов. Этого достаточно для обычных ходов; однако для записи рокировки потребуется большее количество памяти.

Этот способ оказывается существенно лучше всех описанных ранее. Давайте теперь ненадолго отложим вопрос упаковки информации в сторону и приступим к следующей задаче, в которой рассмотрим, как можно создать вспомогательные структуры данных для хранения таких “объектов нестандартного размера”, с которыми не так-то легко работать и которые ухитряются пересекать границы байтов.

[46] Кстати, основное достоинство такого представления вне компьютерного мира в том, что такая запись может быть легко выполнена на бумаге человеком, даже в условиях цейтнота. Оказывается, уменьшение длины записи от максимум 8 символов до максимум 4 вместе с определенной концептуальной простотой играет большую роль для пользователей (которых в шахматном мире называют игроками :)).

Задача 27. Форматы данных и эффективность. Часть 2: игры с битами Сложность: 8

Пришло время рассмотреть более компактные и эффективно использующие память структуры данных, и поработать с кодом, оперирующим на битовом уровне.

Вопрос для профессионала

1. Для реализации решения *д*) второго вопроса задачи 26 вы решили создать класс, управляющий буфером битов. Реализуйте его максимально переносимо, с тем чтобы он корректно работал на любом компиляторе, соответствующем стандарту C++, независимо от платформы.

```
class BitBuffer {
public:
    // ... добавьте при необходимости другие функции...

    // Добавляет num битов, начиная с первого бита p.
    //
    void Append( unsigned char* p, size_t num );

    // Запрос количества используемых битов (изначально 0).
    //
    size_t Size() const;

    // Получает num битов, начиная с start-ого бита, и
    // сохраняет результат, начиная с первого бита dst.
    //
    void Get(size_t start, size_t num, unsigned char* dst)
                                                    const;
private:
    // ...дополнительные детали...
};
```

2. Нельзя ли хранить партию в шахматы с использованием менее чем 12 битов на полуход? Если можно — покажите, как. Если нет — поясните, почему.

Решение

BitBuffer, убийца битов

1. **Для реализации решения *д*) второго вопроса задачи 26 вы решили создать класс, управляющий буфером битов. Реализуйте его максимально переносимо, с тем чтобы он корректно работал на любом компиляторе, соответствующем стандарту C++, независимо от платформы.**

Для начала обратите внимание, что здесь нет упоминания о том, что байт состоит из 8 битов, которое было в предыдущей задаче — здесь это условие попросту неприменимо. Нам нужно решение, компилирующееся и корректно работающее в любой реализации C++, соответствующей стандарту, независимо от того, на какой платформе она работает.

Требуемый условием задачи интерфейс выглядит следующим образом.

```
class BitBuffer {
public:
    void Append( unsigned char* p, size_t num );
    size_t Size() const;
    void Get(size_t start, size_t num, unsigned char* dst)
                                                    const;
```

```
    // ...
};
```

Вы можете удивиться, почему интерфейс `BitBuffer` определен с использованием указателей на `unsigned char`. Во-первых, в стандарте C++ нет такой вещи, как указатель на бит. Во-вторых, стандарт C++ гарантирует, что операции над беззнаковыми типами (включая `unsigned char`, `unsigned short`, `unsigned int` и `unsigned long`) не будут вызывать сообщения компилятора типа "Вы не инициализировали этот байт!" или "Это некорректное значение!". Бьярн Страуструп пишет в [Stroustrup00]:

Беззнаковые целые типы идеально подходят для использования в качестве хранилища для битового массива.

От компиляторов требуется предоставить возможность рассматривать `unsigned char` (как и другие беззнаковые типы) как просто хранилище набора битов — именно то, что нам и требуется. Имеются и другие подходы, но данный подход позволит нам получить навыки программирования работы с отдельными битами, что и является основной целью данной задачи.

Главный вопрос при реализации `BitBuffer` — какое внутреннее представление следует использовать? Я рассмотрю две основные альтернативы.

Попытка №1: использование `unsigned char`

Первая мысль — реализовать `BitBuffer` с использованием большого внутреннего блока `unsigned char`, и самостоятельно работать с отдельными битами при их размещении и выборке. Мы можем позволить классу `BitBuffer` иметь член типа `unsigned char*`, который указывал бы на буфер, но, тем не менее, давайте воспользуемся вектором `vector<unsigned char>`, чтобы не заботиться о вопросах распределения памяти.

Вам кажется, что все это звучит очень просто? Если да — значит, вы не пытались реализовать (и протестировать!) сказанное. Потратьте на это два-три часа, и вновь вернитесь к данной задаче? Я готов держать пари, что больше вы так не скажете.

Мне не стыдно признаться, что эта версия класса отняла у меня массу времени и усилий при ее написании. Даже черновые наброски оказалось сделать труднее, чем мне казалось сначала, не говоря уже об отладке и устранении всех ошибок, когда мне пришлось не раз воспользоваться отладочной печатью для вывода промежуточных результатов и как минимум полдюжины раз пройти код пошагово.

И вот результат. Я не говорю, что он идеален, но он прошел все мои тесты, включая добавление одного и нескольких битов и некоторые граничные случаи. Обратите внимание, что эта версия исходного текста работает одновременно с блоками байтов — например, если мы используем 8-битовые байты и имеем смещение, равное 3 битам, то мы копируем первые три бита как отдельную единицу, и так же поступаем с последними 5 битами, получая 2 операции на байт. Для простоты я также требую, чтобы пользователь предоставлял буферы на байт большие, чем необходимо. Таким образом, я могу упростить свой код, позволив ему работать за концом буфера.

```
// Пример 27-1: реализация BitBuffer с использованием
//              vector<unsigned char>. Трудная,
//              скрупулезная работа. Брр...//

class BitBuffer {
public:
    BitBuffer() : buf_(0), size_(0) { }

    // Добавление num битов, начиная с первого бита p.
    //
    void Append( unsigned char* p, size_t num ) {
        int bits = numeric_limits<unsigned char>::digits;

        // Первый байт назначения и смещение бита
```

```
        int dst = size_ / bits;
        int off = size_ % bits;

        while( buf_.size() < (size_+num) / bits + 1 )
            buf_.push_back( 0 );
        for( int i = 0; i < (num+bits-1)/bits; ++i ) {
            unsigned char mask = FirstBits(num - bits*i);
            buf_[dst+i] |= (*(p+i) & mask) >> off;
            if( off > 0 )
                buf_[dst+i+1] = (*(p+i) & mask)<<(bits-off);
        }
        size_ += num;
    }

    // Запрос количества используемых битов
    // (изначально ноль).
    //
    size_t Size() const {
        return size_;
    }

    // Получение num битов, начиная с start-го бита
    // (нумерация начинается с 0), и сохранение результата
    // начиная с первого бита dst. Буфер, на который
    // указывает dst, должен быть по крайней мере на один
    // байт больше, чем минимально необходимый для хранения
    // num битов.
    //
    void Get(size_t start, size_t num, unsigned char* dst)
        const {
        int bits = numeric_limits<unsigned char>::digits;

        // Первый исходный байт и смещение бита
        int src = start / bits;
        int off = start % bits;

        for( int i = 0; i < (num+bits-1)/bits; ++i ) {
            *(dst+i) = buf_[src+i] << off;
            if( off > 0 )
                *(dst+i) |= buf_[src+i+1] >> (bits - off);
        }
    }
private:
    vector<unsigned char> buf_;
    size_t size_; // in bits
    // Создание маски, в которой первые n битов равны 1, а
    // остальные — 0.
    //
    unsigned char FirstBits( size_t n ) {
        int num=min(n,numeric_limits<unsigned char>::digits);
        unsigned char b = 0;
        while( num-- > 0 )
            b = (b >> 1) |
              (1<<(numeric_limits<unsigned char>::digits-1));
        return b;
    }
};
```

Этот исходный текст нетривиален. Потратьте некоторое время на то, чтоб ознакомиться с ним внимательнее, понять, как он работает, и убедиться, что он корректно решает поставленные перед ним задачи[47]. (Если вам покажется, что вы нашли в исходном тексте ошибку, пожалуйста, сначала напишите тестовую программу, которая бы демонстрировала

[47] Вероятно, разобраться в приведенном исходном тексте (а кое-где даже улучшить его) вам поможет знакомство с книгой [Warren03]. — *Прим. ред.*

ее наличие. Если наличие ошибки подтвердится — прошу вас, отправьте мне сообщение об ошибке вместе с вашей тестовой программой, демонстрирующей наличие ошибки.)

Попытка №2: использование стандартного контейнера упакованных битов

Вторая идея заключается в том, чтобы обратить внимание на то, что стандартная библиотека уже содержит два контейнера для хранения битов: `bitset` и `vector<bool>`. Для наших целей `bitset` — плохой вариант, поскольку `bitset<N>` имеет фиксированную длину `N`, а мы должны работать с потоками битов переменной длины. Не получается... Зато `vector<bool>`, несмотря на все его недостатки в данном случае оказывается тем, "что доктор прописал"[48]. (Конечно, стандарт не требует, чтобы реализация `vector<bool>` использовала упакованные биты, а только поощряет это. Тем не менее, большинство реализаций именно так и поступают.)

Самое главное, что я могу сказать о приведенном далее исходном тексте, — это то, что исходный текст примера 27-2 был совершенно корректен *при первом его написании.*

Да, именно так. Все, что я сделал между первой компиляцией и окончательной версией исходного текста — это исправление несколько мелких синтаксических опечаток, в частности, добавление пропущенной точки с запятой и пары скобок там, где я упустил из виду, что приоритет оператора % выше приоритета оператора +. Вот этот исходный текст.

```
// Пример 27-2: Реализация BitBuffer с использованием
//              vector<bool>.
//
class BitBuffer {
public:
    // Добавление num битов, начиная с первого бита p.
    //
    void Append( unsigned char* p, size_t num ) {
        int bits = numeric_limits<unsigned char>::digits;
        for( int i = 0; i < num; ++i ) {
            buf_.push_back( *p & (1 << (bits-1 - i%bits)) );
            if( (i+1) % bits == 0 )
                ++p;
        }
    }

    // Запрос количества используемых битов
    // (изначально ноль).
    size_t Size() const {
        return buf_.size();
    }

    // Получение num битов, начиная с start-го бита
    // (нумерация начинается с 0), и сохранение результата
    // начиная с первого бита dst.
    //
    void Get( size_t start, size_t num, unsigned char* dst )
        const {
        int bits = numeric_limits<unsigned char>::digits;
        *dst = 0;
        for( int i = 0; i < num; ++i ) {
            *dst |= unsigned char(buf_[start+i])
                 << (bits-1 - i%bits);
            if( (i+1) % bits == 0 )
                *++dst = 0;
        }
    }
private:
```

[48] Более подробно вопрос о `vector<bool>` рассмотрен в [Sutter02] (задача 1.13 в русском издании). — *Прим. ред.*

```
    vector<bool> buf_;
};
```

Вас не должно удивлять, что написать эту версию было существенно проще, чем пример 27-1. Здесь вместо разработки собственного кода для работы с битами используется уже готовый; размер текста оказывается примерно в два раза меньше, чем в предыдущей версии, и как результат — непропорционально малое количество ошибок. Такой код, кроме того, яснее и понятнее; в частности, теперь мне не требуется, чтобы вызывающая программа выделяла дополнительную память только для того, чтобы мой код был проще, как это было в первой версии исходного текста.

Я подозреваю, что оба решения (в особенности первое) можно было бы улучшить — в частности, в исходном тексте могут быть не замеченные мною ошибки, текст может быть упрощен способом, о котором я не подумал, — но я думаю, что оба решения близки к идеалу как в плане корректности, так и в плане стиля.

Плотная упаковка

Давайте теперь еще раз обратимся к упакованному представлению шахматной партии и посмотрим, нельзя ли упаковать ее еще плотнее.

2. Нельзя ли хранить партию в шахматы с использованием менее чем 12 битов на полуход? Если можно — покажите, как. Если нет — поясните, почему.

Да, но если вы захотите использовать это представление в коде, вам потребуется специальный битовый контейнер вроде `BitBuffer`.

Например, имеется три способа.

Достичь упаковки полухода в 10 битов достаточно просто. Нам достаточно указать, какая именно фигура (а для каждого полухода их не более 16) и в какую клетку (которых на доске 64) ходит. Цвет фигуры определяется номером полухода. Перенумеровав фигуры (например, в порядке их размещения по горизонталям, а в пределах горизонтали — по вертикалям) и клетки доски, мы получаем, что для указания фигуры нам надо 4 бита, а для указания клетки, в которую она ходит, — 6 битов; итого достаточно 10 битов, чтобы однозначно определить полуход.

Можно ли улучшить этот результат? Судите сами: мы можем закодировать все клетки доски в качестве целевых клеток полухода, в то время как правила игры позволяют быть таковыми только малому количеству клеток. Таким образом, в нашем представлении явно имеется избыточность. Аналогично, наше представление позволяет записать ход любой фигуры в заданную клетку, в то время как такой ход могут сделать далеко не все фигуры, причем некоторые фигуры в принципе не могут попасть в некоторые клетки. Так что, например, можно использовать для кодирования клетки 6 битов, а затем выяснить, какие фигуры могут пойти в данную клетку, и использовать от 0 до 4 битов для указания одной из них. Если таких фигур мало, нам не потребуются все 4 бита. При декодировании из анализа текущей позиции мы знаем, какое количество фигур может попасть в целевую клетку, а значит, нам известно, сколько именно битов из входного потока требуется запросить, чтобы декодировать полуход.

Мы можем закодировать полуход с использованием не менее 0 и не более 8 битов следующим образом: сначала надо разработать способ упорядочения разрешенных шагов; например, мы можем упорядочить фигуры описанным выше способом, в соответствии с занимаемыми ими позициями, а для каждой фигуры — упорядочить возможные ходы с использованием того же принципа, что и для фигур. Затем номер фактического хода записывается с использованием минимально необходимого количества битов. Например, начальная позиция имеет 20 возможных ходов; для представления их в бинарном виде требуется $\text{ceil}(\log_2(20)) = 5$ битов.

В результате минимальное количество битов, необходимое для записи полухода, равно 0 — если имеется только один возможный, вынужденный ход. Но сколько би-

тов требуется в наихудшем случае? Этот вопрос непосредственно связан с вопросом о том, каково максимальное количество различных ходов может быть сделано в корректной шахматной позиции? Насколько мне известно, текущий известный максимум — 218 ходов в следующей позиции:

```
-  -  -  Q  -  -  -  -    3Q4
-  Q  -  -  -  -  Q  -    1Q4Q1
-  -  -  -  Q  -  -  -    4Q3
-  -  Q  -  -  -  -  R    2Q4R
Q  -  -  -  -  Q  -  -    Q4Q2
-  -  -  Q  -  -  -  -    3Q4
-  Q  -  -  -  -  R  p    1Q4Rp
-  K  -  B  B  N  N  k    1K1BBNNk
```

В этом наихудшем случае для кодирования хода в виде обычного двоичного числа достаточно 8 битов. В среднем, по-видимому, 5 битов будет достаточно для того, чтобы хранить типичный ход. Начальная позиция имеет 20 возможных ходов; типичный эндшпиль, например, король, ладья и две пешки на пустой доске, имеет около 30 допустимых ходов — что также приводит к 5 битам при использовании описанного метода.

Если задуматься, то становится понятным, что описанный метод близок к оптимальному, поскольку он представляет точный и непосредственный ответ на вопрос: Какой разрешенный ход был сделан? Мы используем минимальное количество битов для представления всех возможных ходов в виде двоичного числа, располагая полной информацией о том, что происходило до этого хода.

Можно ли улучшить и этот результат? Да, но теперь результаты будут не столь впечатляющими, так как нам потребуются новые знания о шахматах, а речь идет об экономии долей битов. Для того чтобы проиллюстрировать возможность улучшения достигнутых нами результатов, обратимся еще раз к начальной позиции. В ней имеется 20 возможных ходов, что в нашей схеме требует $\text{ceiling}(\log_2(20)) = 5$ битов. Теоретически же в этой ситуации имеется только $\log_2(20) = 4.3$ битов информации, если считать все ходы равновероятными, так что в среднем нам должно хватить еще меньшего количества битов, если вспомнить о том, что бо́льшая часть всех партий начинается с одного из двух популярных ходов белых. Короче говоря, если мы располагаем дополнительной информацией об относительных вероятностях каждого хода (например, встраивая в механизм сжатия детерминистскую шахматную программу, которая может предположить, какие ходы в любой заданной позиции будут наиболее вероятны), то мы можем использовать кодирование с переменной длиной, такое как код Хаффмана или арифметическое сжатие, чтобы использовать меньшее количество битов для хранения более вероятных ходов. Цена такой высокой степени сжатия информации — повышенное время вычислений, использующих знания о предметной области.

Резюме

Эта задача проиллюстрировала, как знания о предметной области могут быть применены для получения существенно лучших решений поставленной задачи.

В результате даже без знаний о том, какие ходы наиболее вероятны в данной позиции, мы можем сохранить типичную 40-ходовую партию (80 полуходов) примерно в 50 байтах. Это очень немного, и достичь этого можно только путем применения знаний о предмете задачи для ее решения.

> ➢ **Рекомендация**
>
> Оптимизация должна опираться на точные знания о том, *что* вы должны оптимизировать, и *как* вы должны это делать. Знания предметной области ничем невозможно заменить.