Аморальное программирование

Владимир Огородников

20 сентября 2016 г. – 9 августа 2017 г.

Оглавление

Предисловие

С языком brainfuck я познакомился довольно давно. Меня поразило то, что, несмотря на всю свою примитивность, этот язык был Тьюринг-полным. Это означало, что любая программа, написанная на «нормальном» языке, может быть переписана на brainfuck. Естественно, речь не идет о тех программах, для которых нужны прерывания, часы, графический интерфейс и прочие возможности современных компьютеров (хотя, с использованием особой оболочки можно было бы сделать и это, благо у brainfuck-системы есть ввод-вывод). Речь идет о том, что можно назвать «олимпиадными программами»: они что-то считывают со стандартного потока ввода, что-то считают и потом что-то пишут в стандартный поток вывода. Некоторое время назад я загорелся желанием написать некий компилятор из «нормального» языка программирования в brainfuck. Увы, теорией построения компиляторов я пока не владею, поэтому мне пришлось ограничиться псевдоассемблером (хотя, почему *псевдо*-? Это нормальный ассемблер, просто не для x86). В этой статье я бы хотел рассказать, как же такой проект может быть реализован, какие приёмы можно использовать для этого. Речь, естественно, пойдет не о написании километровых

brainfuck-программ, а о генераторе кода – программе, результатом работы которой станет программа на brainfuck. Я не претендую на оригинальность и оптимальность реализации моего проекта, но опыт написания такой вещи мне показался весьма интересным.

Замечание. Поскольку глобальные переменные — это *очень* плохо[1], но они нам будут нужны, будем считать, что все описанные ниже классы лежат в пространстве имён brainfuck (`namespace brainfuck`), а переменные и функции являются полями и методами класса `brainfuck::BFEnvironment`. Кроме того, используется функция `putchar()` из стандартного заголовочного файла `stdio.h`. Вместо неё можно использовать `fputc()` и писать не в стандартный поток вывода, а в файл (опять же, завести поле `FILE *output` в классе `brainfuck::BFEnvironment`).

[1]По-хорошему, из библиотеки вообще ничего глобального не должно торчать. Представьте, что будет, если в двух разных библиотеках будет объявлена одна и та же глобальныя функция или переменная. В языке Си с этим боролись с помощью заковыристых префиксов (например, как в OpenGL). Язык C++ даёт нам замечательный инструмент под названием «пространство имён», или namespace.

Глава 1

Инфраструктура

В этом разделе мы обсудим несколько трюков, которые позволят нам работать с brainfuck почти так же легко, как с Бейсиком. В числе прочего, будут оговорены такие вещи, как доступ к памяти по заранее известному индексу, выделение памяти для различных структур данных, условные операторы, операторы циклов, а также различные арифметические и логические операции.

1.1 Первый якорь в безбрежном море памяти

Итак, Вы запустили интерпретатор brainfuck. Первое, что Вас встречает – безбрежный океан памяти. Случайно сдвинув указатель не туда, практически невозможно вернуться обратно. Поэтому необходимо очень внимательно следить за перемещениями указателя. Для этого сделаем следующее: напишем отдельную функцию `moveTo(int ptr)`, которая будет отвечать за наше перемещение по ленте памяти.

```
1  int curPtr = 0;
2
3  void moveTo(int ptr) {
4      while (ptr < curPtr) {
5          --curPtr;
6          putchar('<');
7      }
8      while (ptr > curPtr) {
9          ++curPtr;
10         putchar('>');
11     }
12 }
```

Это простое, казалось бы, решение избавит нас от многих проблем. Теперь у нас появился надежный способ адресации, каждая ячейка памяти получила свой уникальный адрес. Однако не все так просто. Если мы будем использовать только эту функцию для перемещения по ленте памяти, то нам не удастся реализовать доступ к произвольной (неизвестной на этапе компиляции) области памяти. С другой стороны, использование перемещения по ленте памяти вне этой функции может нарушить (и почти всегда нарушит) работу `moveTo`. В связи с этим введём несколько правил, которым мы будем следовать, чтобы наша память не упала:

1. Внутри цикла необходимо в конце вернуться в ту же ячейку, с которой этот цикл начался;
2. Не использовать *без крайней необходимости* перемещение по ленте памяти в обход `moveTo`;
3. Если всё же пришлось нарушить правило 2, то неоходимо гарантировать, что к следующему вызову `moveTo` мы *всё-таки вернёмся* туда, откуда начали.

Соблюдая эти простые правила, мы будем в любой момент времени чётко представлять себе, в какой ячейке памяти мы сейчас находимся.

1.2 Квартирный вопрос

Теперь, когда мы разобрались с адресами, возникает следующий вопрос: что и как мы будем хранить в нашей памяти? Нам нужен некий механизм, который выдавал бы нам по требованию нужное количество ячеек памяти, которые гарантированно ничем важным не заняты. Это легко сделать: ниже приведена реализация функции `int malloc(int size)`, которая как раз этим и занимается.

```
int allocatedPtr = 16;

int malloc(int size = 1) {
    int ans = allocatedPtr;
    allocatedPtr += size;
    return ans;
}
```

Ура! Теперь у нас есть возможность выделить столько памяти, сколько нам нужно. Теперь мы можем перейти к написанию различных функций, необходимых для полноценной работы с памятью. *Обратите внимание*, что первые несколько ячеек ни подо что не могут быть выделены: зачем это нужно, будет объяснено в одном из следующих разделов.

1.3 Оператор цикла

В языке brainfuck существует только один вид цикла - цикл `while`. Он работает следующим образом:

1. Проверить, что значение в текущей ячейке не равно нулю; если равно, то перейти к шагу 4;
2. Выполнить тело цикла;
3. Перейти к шагу 1;
4. Конец цикла.

Отсюда видно, что в общем случае мы не можем знать, сколько раз выполнится наш цикл, и выполнится ли вообще. Поэтому суммарное смещение указателя за цикл должно быть равно нулю, о чём было упомянуто выше.

Для реализации цикла напишем две функции: `void loopBegin(int ptr)` и `void loopEnd()`.

```
std::vector<int> loopPtrs;

void loopBegin(int ptr) {
    moveTo(ptr);
    putchar('[');
    loopPtrs.push_back(ptr);
}

void loopEnd() {
    if (loopPtrs.empty()) {
        throw std::runtime_error("All loops have already been closed");
    }
    moveTo(loopPtrs.back());
    putchar(']');
    loopPtrs.pop_back();
}
```

Теперь мы можем со спокойной совестью использовать циклы, не переживая каждый раз о том, что надо вернуться туда, откуда мы начали.

1.4 Ввод и вывод

Ввод и вывод в brainfuck реализованы предельно просто: чтобы считать символ из стандартного потока ввода, надо написать `','`, а для того, чтобы напечатать символ, нужно написать `'.'`.

```
void out(int ptr) {
    moveTo(ptr);
    putchar('.');
}

void in(int ptr) {
    moveTo(ptr);
    putchar(',');
}
```

1.5 Изменяющаяся память

Окей, мы научились перемещаться от одной ячейки памяти к другой, считывать данные с клавиатуры и выводить на экран. Но простой ввод и вывод, пусть даже в разном порядке – это неинтересно. Для решения задач практически всегда необходимо как-то *изменять* данные.

Для начала, научимся увеличивать и уменьшать значение в ячейке памяти. Это очень просто: достаточно написать соответствующее количество плюсов или минусов.

```
void inc(int ptr, int delta = 1) {
    moveTo(ptr);
    while (delta--) {
```

```
        putchar('+');
    }
}
```

```
void dec(int ptr, int delta = 1) {
    moveTo(ptr);
    while (delta--) {
        putchar('-');
    }
}
```

```
void addConst(int ptr, int delta) {
    if (delta < 0) {
        dec(ptr, -delta);
    } else {
        inc(ptr, delta);
    }
}
```

Хорошо, теперь мы можем изменить содержимое ячейки памяти. Но как записать в ячейку именно то число, которое мы хотим? Казалось бы, можно просто вычислить разницу между желаемым и действительным и добавить эту разницу. Однако до запуска программы мы никак не сможем узнать, что же хранится в ячейке (в некоторых частных случаях это, конечно, выполнимо – скажем, если значение в ячейке не зависит от пользовательского ввода). Поэтому нам нужно научиться *обнулять* ячейку. Сделать это очень просто: достаточно уменьшать значение ячейки, пока оно не достигнет нуля.

```
void zero(int ptr) {
    loopBegin(ptr);
    dec(ptr);
    loopEnd();
}
```

Теперь мы можем с чистой совестью обнулять ячейки памяти и записывать в них те значения, который нам нужны.

```
void assign(int ptr, int val) {
    zero(ptr);
    addConst(ptr, val);
}
```

1.6 Запрещённое клонирование[1]

Рассмотрим гипотетическую задачу: у нас есть язык brainfuck и только две ячейки памяти. Необходимо сделать так, чтобы в обеих ячейках было записано то число, которое было изначально в первой ячейке. К сожалению, мы не сможем добиться этого.

[1]Название данного параграфа – отсылка к теореме о запрете клонирования из квантовой механики, которая утверждает, что невозможно создать идеальную копию частицы, не разрушив при этом исходную.

Очевидно, что линейный алгоритм (т. е. алгоритм без циклов) не сможет решить задачу. Пусть тогда существует конечный алгоритм, который может решать такую задачу. Посмотрим на его последний цикл. Поскольку алгоритм конечный, этот цикл когда-нибудь остановится. Но что это значит? Это значит, что в какой-то ячейке записан ноль. Стало быть, линейный алгоритм, который эквивалентен изменению каждой ячейки на фиксированную константу, запишет в эту ячейку некоторую константу. Значит, после выполнения алгоритма хотя бы в одной ячейке будет записано число, не зависящее от исходного параметра. Следовательно, этот алгоритм не решает задачу в общем случае. Получили противоречие. Значит, искомого алгоритма не существует.

Поскольку в доказательстве мы нигде не опирались на то, что элементов именно два, а не пять и не десять, такое рассуждение применимо и в общем случае: *сколько бы у нас ни было ячеек памяти, не существует алгоритма, который бы присваивал бы им одно и то же значение, которое записано в отдельной ячейке.* Но не спешите расстраиваться! Вспомним алгоритм, который меняет местами значения двух переменных. В общем случае он выглядит так:

1. Заведём буфер `t` и запишем в него содержимое переменной `a`;
2. Запишем в переменную `a` содержимое переменной `b`;
3. Запишем в переменную `b` содержимое переменной `t`.

Здесь наглядно продемонстрирован интуитивно понятный принцип: заведи временный буфер и делай с ним, что хочешь.

Напишем функцию, которая обнуляет одну ячейку, но при этом копирует её значение в несколько других ячеек. Для реализации этой функции нам понадобится стандартный заголовочный файл `stdarg.h`.

```
void copyAndErase(int from, int count, ...) {
    vector<int> dest(count);
    va_list args;
    va_start(args, count);
    for (int i = 0; i < count; ++i) {
        dest[i] = va_arg(args, int);
    }
    va_end(args);
    for (int to : dest) {
        zero(to);
    }
    loopBegin(from);
    dec(from);
    for (int to : dest) {
        inc(to);
    }
    loopEnd();
}
```

Обратите внимание на то, что если мы укажем несколько одинаковых ячеек, то значения в этих ячейках будут домножены на соответствующий коэффициент. *ВНИМАНИЕ!* Ни в коем случае не указывайте в качестве ячейки назначения ту же ячейку, что и ячейка-источник. Это приведёт к зацикливанию.

Теперь, используя такой мощный инструмент, мы с лёгкостью сможем скопировать содержимое одной ячейки в другую.

```
void copy(int from, int to, int buf = 0) { // Buffer size = 1
    copyAndErase(from, 2, to, buf);
    copyAndErase(buf, 1, from);
}
```

Помните, мы говорили, что первые несколько ячеек памяти нельзя отдавать под переменные? Теперь становится понятно, зачем это было нужно. Эту память мы будем использовать как буфер для различных функций. Зачем же адрес буфера передаётся по ссылке? Иногда может быть неудобно бежать через всю память в начало и что-то там делать. Иногда бывает удобнее выделить буфер недалеко от того места, где мы работаем, и использовать его. Если же нам всё равно, где лежит наш буфер, этот параметр можно опустить.

Кроме того, может быть такая ситуация, когда одна из наших функций вызывает другую. Если их буферы перекрываются, то работа внешней функции может быть нарушена. Поэтому внешняя функция *обязана* в явном виде указать внутренней функции, какую память следует использовать как буфер.

В качестве упражнения предлагаю читателям самостоятельно реализовать функцию `void swap(int aPtr, int bPtr, int buf = 0)`, которая использовала бы ровно одну дополнительную ячейку памяти.

1.7 Условный оператор

Цикл `while` – это, конечно, хорошо, но иногда его использовать неудобно. Например, если нам нужно выполнить блок кода один раз, если некоторое условие выполнено, и не исполнять этот блок кода в противном случае, то нам идеально подойдёт условный оператор `if`.

Поскольку единственное наше средство ветвления кода – это цикл, для реализации условного оператора придётся воспользоваться циклом. По завершении цикла переменная, от которой зависит цикл, должна принять значение 0. В какой же момент стоит обнулять эту переменную? Где-то в середине блока делать это не стоит: мы можем об этом случайно забыть, и в результате всё сломается. Разумнее добавить обнуление в начало или в конец блока. Если мы с самого начала сделаем это, то наш код будет выглядеть аккуратно и лаконично. С другой стороны, обнуление в конце даст нам возможность как-то работать с той переменной. Кроме того, мы можем случайно что-то записать в счётчик цикла, и в результате вместо `if`'а у нас будет что-то совсем другое. Обнуление в конце же обезопасит нас от подобных ошибок.

```
// std::vector<int> loopPtrs; // Was declared higher

void ifBegin(int ptr) {
    loopBegin(ptr);
}

void ifEnd() {
    if (loopPtrs.empty()) {
        throw std::runtime_error("All loops have already been closed");
    }
```

```
11      zero(loopPtrs.back());
12      loopEnd();
13  }
```

1.8 Дважды два – четыре, дважды два – четыре...

Многие задачи по программированию связаны с арифметикой. Да чего уж там мелочиться – подавляющее большинство задач требует каких-нибудь арифметических действий. Но пока мы умеем только прибавлять константу. Непорядок!

Пусть нам нужно сложить два числа. Это можно сделать двумя способами: первый способ похож на `copyAndErase()`, второй же на `copy()`.

```
1  void addAndErase(int from, int to) {
2      loopBegin(from);
3      dec(from);
4      inc(to);
5      loopEnd();
6  }
```

```
1  void add(int from, int to, int buf = 0) { // Buffer size = 1
2      zero(buf);
3      loopBegin(from);
4      dec(from);
5      inc(to);
6      inc(buf);
7      loopEnd();
8      copyAndErase(buf, 1, from);
9  }
```

Точно таким же образом определяются функции `void subtractAndErase()` и `void subtract()`, за исключением `dec(to);` вместо `inc(to);`.

Ещё хотелось бы упомянуть функцию `void negate()`, которая заменяет значение в ячейке на противоположное.

```
1  void negate(int ptr, int buf = 0) { // Buffer size = 1
2      copyAndErase(ptr, 1, buf);
3      loopBegin(buf);
4      dec(buf);
5      dec(ptr);
6      loopEnd();
7  }
```

Интереснее дело обстоит с умножением. Насколько мне известно, нельзя просто так взять и перемножить два числа, что называется, *in-place*, то есть без буфера. Более того, я не умею обходиться меньше, чем тремя буферными ячейками. Впрочем, одну из них можно заменить на ячейку для ответа.

```
1  void mul(int a, int b, int ans, int buf = 0) { // Buffer size = 2
2      copy(a, buf, ans);
3      zero(ans);
```

```
    loopBegin(buf);
        dec(buf);
        copyAndErase(b, 1, buf + 1);
        loopBegin(buf + 1);
            dec(buf + 1);
            inc(b);
            inc(ans);
        loopEnd();
    loopEnd();
}
```

Функция `mul(int a, int b, int ans, int buf)` – вероятно, одна из наиболее сложных функций этого раздела. Давайте разберёмся, что же тут происходит. У нас есть 5 переменных: `a`, `b`, `ans`, `buf[0]` и `buf[1]`. На C++ эта программа выглядела бы следующим образом:

```
buf[0] = a;
ans = 0;
while (buf[0] != 0) {
    --buf[0];
    buf[1] = b; b = 0;
    while (buf[1] != 0) {   // This cycle is equivalent to:
        --buf[1];           // b = buf[1];
        ++b;                // ans = buf[1];
        ++ans;              // buf[1] = 0;
    }
}
```

Похоже, не правда ли? Обратите внимание, что эта функция корректно работает и для отрицательных чисел. Однако в силу небольшого размера ячейки памяти очень часто может возникать переполнение. Поэтому пользоваться этой функцией надо аккуратно.

Хотя реализовать деление ячеек друг на друга в принципе возможно, сейчас мы не будем этого делать. Во-первых, это весьма нетривиальный алгоритм, который требует много дополнительной памяти. Во-вторых, такое деление нам не пригодится (не переживайте, будет Вам деление, только чуть позже). Взятие остатка от деления тоже откладывается до лучших времён.

1.9 Буль, Джордж Буль[2]

Нестолько разделов назад мы научились писать условный оператор. Тем не менее, полноценно использовать его у нас пока не получится, потому что мы не умеем делать операции с логическим типом данных.

Для начала нам нужно научиться преобразовывать число в переменную логического типа. Для этого напишем функцию `void boolCast()`, которая превращает любое число, отличное от нуля, в единицу (такое преобразование логично, поскольку оно согласовано с поведением циклов и условных операторов):

[2] Джордж Буль – английский математик XIX века, один из основателей математической логики.

```
void boolCast(int ptr, int buf = 0) { // Buffer size = 1
    zero(buf);
    ifBegin(ptr);
    inc(buf);
    ifEnd(ptr);
    copyAndErase(buf, 1, ptr);
}
```

Окей, теперь мы можем реализовать основные операции с логическими переменными: and, or, xor и not, а также модифицирующую версию отрицания.

```
void boolOr(int a, int b, int ans, int buf = 0) { // Buffer size = 1
    copyAndErase(a, ans, buf); // ans = a;

    zero(buf);
    loopBegin(b);
    dec(b);
    inc(buf);
    inc(ans);
    loopEnd(); // ans += b; buf = b; b = 0;

    copyAndErase(buf, 1, b); // b = buf; buf = 0;
    boolCast(ans, buf);
}
```

```
void boolNegate(int ptr, int buf = 0) { // Buffer size = 1
    assign(buf, 1);
    ifBegin(ptr);
    dec(buf);
    ifEnd();
    copyAndErase(buf, 1, ptr);
}
```

```
void boolAnd(int a, int b, int ans, int buf = 0) { // Buffer size = 1
    // Here we will use de Morgan's rule: not (X and Y) = not X or not Y
    boolNegate(a, buf);
    boolNegate(b, buf);
    boolOr(a, b, ans, buf);
    boolNegate(a, buf);
    boolNegate(b, buf);
    boolNegate(ans, buf);
}
```

```
void boolXor(int a, int b, int ans, int buf = 0) { // Buffer size = 1
    copyAndErase(a, 2, ans, buf);
    copyAndErase(buf, 1, a); // ans = a;

    loopBegin(b);
    dec(b);
    inc(buf);
    inc(ans);
```

```
    loopEnd(); // ans += b; buf = b; b = 0;

    copyAndErase(buf, 1, b); // b = buf; buf = 0;

    dec(ans);
    boolNegate(ans, buf);
}
```

```
void boolNot(int a, int ans, int buf = 0) { // Buffer size = 1
    copy(a, ans, buf);
    boolNegate(ans, buf);
}
```

Заметим, что наша реализация дизъюнкции и взаимоисключающей дизъюнкции *не гарантирует* корректной работы на неприведённых значениях, а конъюнкция *изменяет* значения аргументов, сохраняя их только в логическом смысле. Унарные же функции либо меняют значение переменных (это просто-напросто их задача), либо, как отрицание, ничего не портят. Мораль: перед употреблением привести типы!

1.10 Die Ordnung[3]

Последний теоретический раздел этой главы будет посвящён операциям сравнения. Почему во множественном числе? Да потому, что сравнение бывает знаковым и беззнаковым. Чтобы понять, в чём же, собственно, разница, давайте повнимательнее присмотримся к устройству чисел в компьютере.

Как Вы знаете, для хранения числа в памяти выделяется фиксированное число битов – ячеек памяти, которые могут принимать только два состояния (0 и 1). В большинстве современных компьютеров эти биты объединены в блоки по 8 бит в каждом, которые называются байтами. Простейшие рассуждения из комбинаторики показывают, что один байт может принимать $2^8 = 256$ различных значений. Чаще всего байты объединяют в ещё более крупные блоки – слова, двойные слова и четверные слова (соответствуют типам данных `short int`, `int` и `long long int` в Си). Но, так как мы в brainfuck, мы обойдём их пока стороной (тем более, что они устроены почти так же, только диапазоны значений другие).

Итак, байт, он же восьмёрка бит, он же 256 различных состояний. На это можно посмотреть и с другой стороны: если рассматривать каждый бит как разряд в двоичной записи какого-нибудь числа, то в байте можно хранить какое-нибудь восьмиразрядное двоичное число (одно из 256). Таким образом, в одном байте можно хранить целое число от 0 до 255.

Ну, хорошо, скажете Вы. А как же быть с отрицательными числами? Всё достаточно просто. Давайте считать, что мы работаем не с самими числами, а с остатками от их деления на 256 (остатками в математическом смысле, то есть с неотрицательными числами, меньшими 256). Тогда ответ на вопрос, что же оказывается в ответе при переполнении типа, становится очевидным: остаток от деления суммы на 256. Значит, для хранения произвольного отрицательного числа достаточно прибавлять к нему 256, пока оно не станет нулём или положительным числом (что эквивалентно взятию остатка по модулю 256 в математическом понимании, а не в понимании

[3](нем.) порядок.

разработчиков x86. Впрочем, такое решение, видимо, упростило алгоритм работы процессора и позволило ускорить его работу).

Когда мы работаем с остатками по одинаковому модулю, математические законы дают нам несколько плюшек, а именно:

1. Сумма остатков сравнима с остатком суммы;
2. Разность остатков сравнима с остатком разности;
3. Произведение остатков сравнимо с остатком произведения.

Под *сравнимостью чисел a и b по модулю c* подразумевается, что a и b имеют одинаковые остатки от деления на c или, что то же самое, что их разность делится нацело на c. Именно поэтому мы можем перемножать знаковые и беззнаковые числа одной и той же операцией. Значит, вопрос лишь в том, какие остатки считать отрицательными числами, а какие положительными или нулём. По договорённости, число считается отрицательным, если его старший бит (иногда называемый знаковым) равен 1.

Как же нам заменить число на противоположное (в обычном компьютере, не в brainfuck)? Можно применить метод, аналогичный нашему `negate()`, описанному несколько пунктов назад, но он будет работать *очень* долго. Можно умножить на -1, которое в нашей системе будет записано как $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255 = 2^8 - 1$. Но можно сделать ещё лучше. Давайте инвертируем все биты в нашем числе x (заменим единички на нолики, и наоборот). Что мы получим? Мы получим $255 - x$ (для проверки достаточно сложить исходное и инвертированное число). И нам останется к ответу прибавить 1. Получится $256 - x = 256 + (-x) \equiv -x (mod 256)$.

Но достаточно теории, перейдём к практике. К сожалению, работать напрямую с битами в brainfuck у нас не получится. Тем не менее, мы можем достаточно легко получить, скажем, старший бит числа. Что будет, если мы сначала умножим число на два, а затем разделим на два? Правильно, при умножении старший бит потеряется из-за переполнения, и в результате получится то же, что и в начале, но с обнулённым старшим битом. Теперь, вычтя полученное число из исходного, мы получим 0, если старший бит исходного числа был равен нулю, и $128 = 2^7$, если старший бит был равен 1. Таким образом, мы научились узнавать знак числа.

```
1  void isNegative(int x, int ans, int buf = 0) {// Buffer size = 1
2                  //  x       ?       ?
3      zero(ans);  //  x       0       ?
4      zero(buf);  //  x       0       0
5
6      loopBegin(x);
7      dec(x);
8      inc(ans, 2);
9      inc(buf);
10     loopEnd();  //  0   2x == 2x'   x   (x' means x % 128)
11
12     loopBegin(ans);
13     dec(ans, 2);
14     inc(x);
15     dec(buf);
16     loopEnd();  //  x'      0       x - x'
```

```
17
18      loopBegin(buf);
19      dec(buf);
20      inc(x);
21      inc(ans);
22      loopEnd();  //  x       x - x'  0
23
24      boolCast(ans, buf);
25  }
```

Обратите внимание, что, хоть мы и делили в этом коде на два (первая инструкция второго цикла), делать так в общем случае не стоит. В этот раз мы *точно знали*, что счётчик цикла содержит чётное число. А теперь представьте, что было бы, если бы оно вдруг оказалось нечётным. Поскольку вычитание двойки не изменяет чётности числа, счётчик цикла всегда оставался бы нечётным. С другой стороны, цикл останавливается только тогда, когда счётчик цикла обнуляется. Поэтому мы бы просто зациклились. Поэтому такое деление можно использовать только в паре с умножением и только для степеней двойки (скажем, при умножении на три может возникнуть переполнение, и тогда никто нам не гарантирует, что остаток будет делиться на три).

Теперь, казалось бы, сравнить два числа очень просто: достаточно найти разность и сказать её знак. Но, к сожалению, это не всегда правда. Дело в том, что сумма (и, естественно, разность) двух восьмибитных чисел может не влезть в восемь бит. И тогда сравнение вернёт неправильный ответ. Чтобы правильно сравнить два числа, нужно сравнить отдельно их старшие биты, отдельно сравнить семибитные хвосты, а затем проделать некоторое преобразование с полученными ответами. Я не буду приводить здесь код, решающий эту задачу, поскольку такое сравнение мне не пригодится в дальнейшем (я буду сравнивать числа от 0 до 15, а для них подходит и то, что написано в начале абзаца). Тем не менее, написание знакового и беззнакового сравнений было бы очень неплохим упражнением.

Ну вот, на этой радостной ноте первая глава завершается.

Глава 2

Маленькие большие числа

В предыдущей главе мы научились делать множество операций с лентой памяти: арифметические действия, ввод и вывод, циклы, условные операторы, логические операции, сравнение двух чисел. В этой главе мы научимся работать с числами значительно большего размера, нежели 1 байт.

2.1 Скелет

Для начала, нам нужно определиться с тем, как мы будем хранить наши числа. Поскольку число будет занимать несколько байтов, важно, в каком порядке эти байты будут заполняться. Существует два подхода: в англоязычной литературе их принято называть big-endian и little-endian[1] (дословно «тупоконечный» и «остроконечный»). Если мы используем подход big-endian, то в первом байте будет лежать старший разряд нашего числа, затем второй по старшинству и так далее. Этот принцип мы используем в повседневной жизни, когда работаем с обычными числами. В little-endian всё происходит наоборот: в первом байте хранится самый младший разряд числа, а старший разряд хранится в последнем байте. Оба подхода используются достаточно часто. Тем не менее, в силу личных предпочтений я буду придерживаться подхода little-endian.

Кроме порядка, важно, какие числа мы будем хранить в каждой ячейке. Как упоминалось в предыдущей главе, в одном байте мы будем хранить число от 0 до 15, то есть каждый байт будет заполнен наполовину. Зачем мне так транжирить память? Дело в том, что такой подход сильно облегчит мне жизнь, когда речь зайдёт об арифметических операциях. Чтобы не мучиться с переполнением, имеет смысл обеспечить себе некоторый запас. Почему такой большой? Чтобы влез не только результат сложения, но и умножения.

В разделе первой главы, посвящённом функциям с буфером, мы говорили, что может возникнуть такая ситуация, когда значительно проще выделить буфер «на месте» и работать с ним, чем бежать из одного конца ленты памяти в другой. Длинная арифметика – как раз один из таких случаев. Нам очень часто будут нужны некоторые вычисления, требующие буфера, поэтому проще и эффективнее сразу выделить

[1]На самом деле, есть ещё и так называемый middle-endian, то есть смешанный порядок: он используется в том случае, когда байты в слове компьютер записывает в одном порядке, а сами слова записаны в другом (см. статью на Википедии). Но такой подход сложный, и его использование для нас не имеет смысла.

буфер рядом с каждым длинным числом. Для нашего проекта достаточно буфера размером 3 ячейки. Тем не менее, я не уверен, что 3 ячейки необходимы.

Замечание. Дабы избежать проблем с присваиванием чисел разного размера, будем передавать размер числа как шаблонный параметр.

```
template <int N>
class Integer {
private:
    int buf, data;
    BFEnvironment &env;
public:
    Integer(BFEnvironment &env) : env(env) {
        buf = env.malloc(BUF_SIZE);
        data = env.malloc(DATA_SIZE);
    }
    const static int BUF_SIZE = 3, DATA_SIZE = N;
    int getBuf() const {
        return buf;
    }

    int getData() const {
        return data;
    }
};
```

2.2 Инициализация

Окей, теперь давайте научимся записывать какие-нибудь осмысленные числа в наши структуры. Это очень просто: достаточно записать в первую ячейку младшие четыре бита, следующие четыре бита во вторую ячейку и так далее. Дабы ничего не испортилось из-за отрицательных чисел, воспользуемся битовой магией.

```
void Integer<N>::assign(int val) {
    for (int i = 0; i < DATA_SIZE; ++i) {
        env.assign(data + i, val & 0x0F);
        val >>= 4;
    }
}
```

2.3 Два пишем, один в уме

Хорошо, с хранением чисел разобрались. Но прежде, чем проводить с ними арифметические операции, нам нужно научиться переносить разряды. Как же это сделать? Алгоритм достаточно простой: для этого нам понадобится поделить очередной разряд с остатком на 16, остаток записать вместо разряда, а частное прибавить к следующему разряду. Чтобы упростить код, будем считать, что на предыдущем шаге мы записали остаток в буфер, поэтому первым делом его надо оттуда достать и прибавить к текущему разряду. Таким образом мы избавимся от разбора случаев при обработке последнего разряда. Перед началом алгоритма нужно обязательно

очистить буфер (как если бы мы переносили ноль из предыдущего разряда). Единственная загвоздка может заключаться в делении с остатком, но и здесь нет ничего принципиально сложного. Воспользуемся тем же трюком, как в определении знака числа, а имеено умножим и разделим текущий разряд на 16. Так мы обнулим старшие четыре бита нашего байта и получим остаток от деления на 16. Получить же частное, имея делимое и остаток, тоже весьма просто: достаточно вычесть и обычным циклом поделить. Поскольку нам может понадобиться проводить эту операцию не от самого младшего разряда, а где-нибудь ближе к концу, передадим номер первого разряда отдельным параметром.

В коде ниже будет немного магии. Для наглядности приведена схема, что и где лежит после очередного цикла:

	0	0	...	x	
	16x	x	...	0	
	0	x - x % 16	...	x % 16	
	x / 16	0	...	x % 16	

```
1  void Integer<N>::normalize(int from = 0) {
2      env.zero(buf);
3      env.zero(buf + 1);
4      for (int i = from; i < DATA_SIZE; ++i) {
5          env.addAndErase(buf, data + i);
6
7          env.loopBegin(data + i);
8          env.dec(data + i);
9          env.inc(buf + 1);
10         env.inc(buf, 16);
11         env.loopEnd();
12
13         env.loopBegin(buf);
14         env.dec(buf, 16);
15         env.inc(data + i);
16         env.dec(buf + 1);
17         env.loopEnd();
18
19         env.loopBegin(buf + 1);
20         env.dec(buf + 1, 16);
21         env.inc(buf);
22         env.loopEnd();
23     }
24 }
```

2.4 Арифметика

Теперь, когда мы умеем переносить разряды, задача «сложить два числа» кажется скучной и тривиальной. Надеюсь, что читатель не осудит меня, если я не буду приводить код этой функции. Скажу лишь, что, с моей точки зрения, удобнее будет реализовать её в виде оператора `+=`.

С вычитанием все немного интереснее. Если мы из маленького разряда вычесть большой, то произойдёт переполнение. Обрабатывать его долго и грустно, поэтому

мы сделаем следующее: мы сначала добавим к уменьшаемому числу единичку, а потом минус единичку, при этом *не переносяразряды*. Чтобы прибавить единичку, нужно всего лишь увеличить младший разряд на 1. Как же прибавить минус единичку? Рассмотрим число `0xFF...F`. Это число эквивалентно минус единице, так как стоит к нему прибавить 1, и оно обратится в ноль (по модулю 2^n). Таким образом, грязный хак заключается в том, чтобы увеличить каждый разряд на 15, а младший – ещё на 1. Тогда нам гарантированно хватит размера разрядов, чтобы вычитать, не боясь переполнения. В конце же, разумеется, мы выполним перенос разрядов.

```
void Integer<N>::operator -=(const Integer<N> &other) {
    env.zero(buf);
    env.inc(data);
    for (int i = 0; i < N; ++i) {
        env.inc(data + i, 15);

        env.loopBegin(other.data + i);
        env.dec(other.data + i);
        env.dec(data + i);
        env.inc(buf);
        env.loopEnd();

        env.addAndErase(buf, other.data + i);
    }
    normalize();
}
```

Попутно мы поняли, как реализовать методы `inc()`, `dec()` и `negate()`, эквивалентные тем, которые мы писали, когда работали с обычными ячейками памяти.

Окей, пора разбираться с умножением. Если бы мы работали на обычном компьютере, то самым оптимальным было бы использование быстрого преобразования Фурье, но ему нужны числа с плавающей запятой. Из целочисленных методов есть алгоритм Карацубы, но ему нужно много памяти. Кроме того, у меня есть сомнения по поводу времени его работы на нашей машине: напомню, что наша память по-прежнему работает по принципу последовательного доступа, хотя мы усиленно делаем вид, что это не так. Поэтому воспользуемся обычным умножением в столбик, попутно пытаясь втиснуться в наши шесть байтов дополнительной памяти.

Обратите внимание на то, что `normalize()` вызывать нужно не один, а N раз. В противном случае у нас может произойти переполнение. Именно для того, чтобы избежать проблем с переполнением, мы так неэкономно используем память.

```
void Integer<N>::operator *=(const Integer<N> &other) {
    for (int i = N - 1; i >= 0; --i) {
        env.copyAndErase(data + i, 1, other.buf);
        for (int j = i; j < N; ++j) {
            env.mul(other.buf, other.data + j - i, other.buf + 1, buf);
            env.addAndErase(other.buf + 1, data + j);
        }
        normalize(i);
    }
}
```

По логике, операция деления должна завершать этот раздел, но есть несколько аргументов против. Дело в том, что это действие весьма нетривиально: для работы ему нужны сравнения, циклические сдвиги и немного магии. Поэтому имеет смысл вынести деление в отдельный раздел, перед этим обсудив сравнения. Кроме того, по законам жанра, самый сложный раздел должен идти последним или предпоследним в главе. Поэтому, дорогой читатель, Вам придётся потерпеть ещё чуть-чуть.

2.5 Die Ordnung v2.0

Как Вы помните, когда мы занимались сравнением однобайтовых чисел, мы упоминали, что знаковое и беззнаковое сравнения работают похожим образом. Поэтому сейчас мы решим только одну из этих задач, а именно беззнаковое сравнение, после чего сведём вторую задачу к предыдущей.

Итак, как же работает беззнаковое сравнение? В обычных условиях мы написали бы что-нибудь вроде «сравнивай разряды от старшего к младшему, и когда найдешь различающиеся разряды, верни результат их сравнения». К сожалению, такой подход у нас не сработает, ибо нет у нас ни `return`'ов, ни `break`'ов – ничего, что могло бы резко изменить поток выполнения программы. Но мы можем сделать следующее: будем запоминать, нашли мы уже ответ или ещё нет. На очередной итерации мы будем проверять, есть ли у нас ответ, и если его ещё нет, честно сравним очередные два разряда.

```
1  template <int N>
2  void unsignedCmp(Integer<N> &a, Integer<N> &b, int ans) {
3      int notDoneFlag1 = a.getBuf(), notDoneFlag2 = a.getBuf() + 1;
4
5      env.assign(notDoneFlag1, 1);
6      env.assign(notDoneFlag2, 1);
7      env.assign(ans, 0);
8
9      for (int i = N - 1; i >= 0; --i) {
10         env.ifBegin(notDoneFlag1);
11             int A = a.getData() + i,
12                 B = b.getData() + i,
13                 buf = b.getBuf();
14
15             env.copy(B, buf, buf + 1);
16             env.negate(buf, buf + 1);
17             env.addAndErase(buf, A); // ![1]
18
19             env.isNegative(A, buf, buf + 1);
20             env.ifBegin(buf);
21                 env.assign(ans, -1);
22                 env.zero(notDoneFlag2);
23             env.ifEnd(); // ![2]
24
25             env.negate(A, buf);
26             env.isNegative(A, buf, buf + 1);
27             env.ifBegin(buf);
```

```
28                  env.assign(ans, 1);
29                  env.zero(notDoneFlag2);
30              env.ifEnd(); // ![3]
31
32              env.negate(A, buf);
33              env.copy(B, buf, buf + 1);
34              env.addAndErase(buf, A); // ![4]
35          env.ifEnd();
36
37          env.copy(notDoneFlag2, notDoneFlag1, a.getBuf() + 2); // ![5]
38      }
39  }
```

Давайте разберёмся, что за чёрная магия здесь происходит. Сначала мы инициализируем переменные нужными нам значениями. Затем для каждого разряда в порядке от старшего к младшему выполняем условный оператор.

TO BE CONTINUED...