1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. **Институт кибербезопасности и защиты информации**

**КУРСОВАЯ РАБОТА**

**Оптимизация программы «Обфускатор»**

по дисциплине «Языки программирования»

1. Выполнили Берко А.С.
2. студент гр. 4851003/90001 Кулеева А.Г.

1. Руководитель
2. ст. преподаватель Семьянов П.В.

«\_\_» \_\_\_\_\_\_\_\_\_\_ 2021 г.

1. Санкт-Петербург
2. 2021
3. Содержание

|  |  |  |
| --- | --- | --- |
| Введение......................................................................................................... | | 2 |
| 1 | Теоретическая часть ......................................................................... | 4 |
| 1.1 | Оптимизация на уровне алгоритма ................................................ | 4 |
| 1.2 | Машинно-независимая оптимизация ............................................. | 5 |
| 1.3 | Машинно-зависимая оптимизация ................................................. | 11 |
| 2 | Практическая часть.......................................................................... | 14 |
| 2.1 | Описание исходной программы «Обфускатор» ........................... | 14 |
| 2.1 | Оптимизация на уровне алгоритма ................................................ | 16 |
| 2.2 | Машинно-независимая оптимизация ............................................. | 22 |
| 2.3 | Машинно-зависимая оптимизация ................................................. | 33 |
| 2.4 | Оптимизация по размеру программы «Hello world!» ................... | 37 |
| Заключение................................................................................................... | | 44 |
| Список использованных источников......................................................... | | 47 |
| Приложение 1............................................................................................... | | 48 |
| Приложение 2............................................................................................... | | 53 |
| Приложение 3............................................................................................... | | 58 |
| Приложение 4............................................................................................... | | 60 |

Введение

**Цель работы:** научиться проводить различные виды оптимизации, такие как оптимизация на уровне алгоритма, машинно-независимая и машинно-зависимая оптимизации, на основе программы «Обфускатор», а также провести оптимизацию по размеру программы «Hello world!».

**Задачи:**

1. Изучить различные виды оптимизации программ;
2. Произвести анализ исходной программы «Обфускатор»;
3. Выделить участки кода, которые необходимо оптимизировать;
4. Произвести оптимизацию на уровне алгоритма и предоставить результаты улучшения (время работы и память полученной программы);
5. Произвести машинно-независимую оптимизацию, результаты представить в отчете;
6. Провести машинно-зависимую оптимизацию и предоставить соответствующие результаты;
7. Выполнить оптимизацию по размеру программы «Hello world!»;
8. Сделать вывод по работе, опираясь на полученные результаты.

Оптимизация кода − это различные методы преобразования кода ради улучшения его характеристик и повышения эффективности. Среди целей оптимизации можно указать уменьшения объема кода, объема используемой программой оперативной памяти, ускорение работы программы, уменьшение количества операций ввода-вывода.Из определения следует, что существуют две возможности оптимизаций:

1. По времени;
2. По памяти.

На практике сложно добиться оптимизации по обоим этим параметрам, зачастую приходится выбирать какой-то один из них, пренебрегая другим. Но так как память с каждым годом как ресурс становится дешевле, то чаще всего оптимизация происходит по времени, так как оно является невосполнимым ресурсом, который необходимо экономить.

Для того, чтобы понять, как оптимизировать программу, рассмотрим статистические данные улучшения программ на четырех основных уровнях оптимизации (см. Таблицу 1).

Таблица 1 – Уровни оптимизаций.

| **Уровень оптимизации** | **Улучшение в теории** | **Улучшение на практике** |
| --- | --- | --- |
| На уровне алгоритма | сколько угодно | 5-10-20 раз |
| Машинно-независимые оптимизации | 2-3 раза | 50% |
| Машинно-зависимые оптимизации | 4-5 раз | До 100% |
| Оптимизации на уровне ассемблера | 3-5 раз | Около 100% |

В ходе выполнения данной курсовой работы будет исследовано влияние применения первых трёх уровней оптимизации на исходный код программы.

# Теоретическая часть

## Оптимизация на уровне алгоритма

Данный вид оптимизации – самый творческий из всех. Он эквивалентен написанию алгоритма с нуля. Суть данной оптимизации не просто придумать решение данной задачи, а придумать наиболее оптимальное решение задачи. Существует несколько определенных шагов, которые помогут начать выполнять оптимизацию на уровне алгоритма:

1. **Найти известный алгоритм.**

Зачастую, не нужно изобретать велосипед и самому создавать новый алгоритм. Например, в исходной программе используется сортировка пузырьком, выполняющаяся в среднем за O(n2). Оптимизацией на уровне алгоритма является использование быстрой сортировки, так как ее среднее время выполнения - O(n\*log(n)). Также, с каждым годом появляются все более эффективные алгоритмы, например, недавно был создан алгоритм, позволяющий выполнять перемножение матриц не за O(n3), а примерно за O(n2,9). Даже несмотря на незначительное уменьшение времени, это открытие является важным улучшением.

1. **Найти подходящую структуру данных.**

Например, программа представлена в виде базы данных, в основе которой лежит какая-то структура данных. Если бы использовался линейный список, то поиск по нему занимал бы время O(n), то есть это был бы линейный поиск. При использовании деревьев можно получить сложность O(logn) (дерево поиска и АВЛ дерево), 2log2n (СДБ, ДБ и КЧ деревья), а при использовании хэша можно добиться скорости O(1). Таким образом, в зависимости от идеи программы могут быть подобраны более оптимальные структуры.

1. **Замена времени на память.**

Замена времени на память ускоряет программу, но увеличивает количество используемой памяти, однако можно получить и обратный эффект. Например, если необходимо сократить именно время, то можно провести некоторые вычисления заранее и сохранить их в какие-то переменные или структуры. Если же необходим обратный эффект, то действовать нужно противоположным образом.

Рассмотрим пример такой оптимизации:

for (j = 1; j<1000; j++)

{

A[j] = days – B[i]\*j;

if ( A[j] < 0) break;

}

R[i] = B[i] + A[j];

Если внимательно изучить данный фрагмент кода, то можно заметить, что здесь выполняется вычисление остатка от деления days на B[i]. Таким образом, данный код можно оптимизировать следующим образом:

R[i] = days % B[i];

## Машинно-независимая оптимизация

Машинно-независимая оптимизация — проведение преобразований исходной программы (или ее промежуточного внутреннего представления), не зависящих от выходного языка компилятора. Из названия следует, что все методы данной оптимизации могут быть проведены одинаково успешно на любой машине, вне зависимости от её параметров. Рассмотрим возможные составляющие данного уровня оптимизации.

### Снижение мощности

Идея данного алгоритма заключается в замене одних операций другими, более быстрыми. Запишем основные операции в порядке увеличения времени их выполнения:

|  |
| --- |
| mov (=) |
| ++, -- |
| +, -, &, |, ^, … |
| [] = \*(A+i) |
| \* |
| /, % |
| (float) – преобразование типов |
| pow – возведение в степень |
| sqrt |
| sin, cos, … |

Рассмотрим некоторые оптимизации вычислений:

x \* 2k ⬄ x << k – умножение на степень двойки;

x / 2k ⬄ x >> k – деление на степень двойки;

x % 2k ⬄ x & ((1 << k) - 1) – взятие остатка от степени двойки;

a / b / с ⬄ a / (b \* c) - избавление от деления;

a+x\*c+y\*c ⬄ a+(x+y)\*c - избавление от умножения;

Преобразование типа:

int i;

float f;

i = (f+1) \* 2 ⬄ i = (int) (f+1) \* 2 ⬄ ((int)f +1) \* 2.

Расчёт полинома (сокращение возведения в степень):

a0xn + a1xn-1 + … an = ((a0 \* x + a1) \* x + a2) \* x … + an

### Оптимизация циклов

1. Отказ от циклов:

|  |  |
| --- | --- |
| **Исходный код:** | **Оптимизированный код:** |
| for (i =0; i < 3; i++) A[i] = 0; | A[0] = A[1] = A[2] = 0; |
| for (i =0; i < 256; i++) A[i] = 0; | memset (A, 0, sizeof(A[0])\*256); |

1. Правильное вложение циклов:

|  |  |
| --- | --- |
| **Исходный код:** | **Оптимизированный код:** |
| for (i =0; i<1000; i++) // 1000  for (j =0; j<10; j++) // 10000  for (k =0; k<3; k++) //30000  A[i][j][k] = 0;  // 41000 checks + incr | for (k =0; k<3; k++) // 3  for (j =0; j<10; j++) // 30  for (i =0; i<1000; i++) //3000  A[i][j][k] = 0;  // 30033 checks + incr |

1. Замена переменной цикла:

| **Исходный код:** | **Оптимизированный код:** |
| --- | --- |
| for (i =0; i < 10; i++)  A[3\*i+4]+= c; | j = 3\*i +4;  for(j = 4; j<34; j+=3)  A[j]+= c; |
| for (i =0; i<10; i++)  A[3\*i+4]+= B[4\*i + 3]; | j = 3\*i +4; k = 4\*i + 3  for(j = 4, k=3; j<34; j+=3, k+=4)  A[j]+= B[k]; |

1. Вынос инварианта из цикла:

| **Исходный код:** | **Оптимизированный код:** |
| --- | --- |
| for (i =0; i<10; i++)  for (j =0; j<10; j++)  A[i][j] = B[i][j] + d/i + d/k; | dk = d/k;  for (i =0; i<10; i++)  {  di = d/i;  didk = d/I + dk;  Ai = A[i];  Bi = B[i];  for (j =0; j<10; j++)  Ai[j] = Bi[j] + didk;  } |

1. Развертка цикла − самая важная и эффективная из оптимизаций циклов:

| **Исходный код:** | **Оптимизированный код:** |
| --- | --- |
| for (i =0; i<N; i++)  A[i]= B[i] + C[i]; | for (i =0; i<N; i+=2)  {  A[i]= B[i]\*C[i];  A[i+1]= B[i+1]\*C[i+1];  } |

Плюсы:

* шаг увеличивается вдвое => вдвое меньше сравнений, инкрементов, ассемблерных команд jmp;
* механически увеличили тело цикла в два раза => получили длинный линейный участок кода, значит, компилятор сможет оптимально распределить регистры и читать одновременно A[i] и A[i+1], поскольку процессорам удобнее работать именно с линейными участками кода и выполняются быстрее сами по себе.

1. Объединение циклов:

| **Исходный код:** | **Оптимизированный код:** |
| --- | --- |
| for (i =0; i<100; i++)  A[i]= B[i]\*C[i] + 4;  for (j =0; j<100; j++)  B[j] = E[j] +5; | for (i =0; i<100; i++)  {  A[i]= B[i]\*C[i] + 4;  B[i] = E[i] +5;  } |

1. Разъединение циклов:

| **Исходный код:** | **Оптимизированный код:** |
| --- | --- |
| while (k>0)  {  if (x == y)  k −−;  else  k −= x;  } | if (x == y)  {  while (k>0)  k −−;  else  while (k>0)  k −= x;  } |

1. Исключение общих операций:

| **Исходный код:** | **Оптимизированный код:** |
| --- | --- |
| for (i =0; i<10; i++)  {  B[i] = A[i]\*c + q;  D[i] = A[i +1]\*c;  } | t = A[0]\*c;  for (i =0; i<10; i++)  {  B[i] = t + q;  t = A[i + 1] \* c;  D[i] = t;  } |

### Оптимизация переходов

|  |  |
| --- | --- |
| **Исходный код:** | **Оптимизированный код:** |
| if (a == b)  k = 0;  if (a != b)  k = 1; | if (a == b)  k = 0;  else  k = 1; |
| k = (a==b); |

| **Исходный код:** | **Оптимизированный код:** |
| --- | --- |
| if (a == …) {…}  else if (a == …) {…}  else if (a == …) {…}  else if (a == …) {…}  else {…} | Распределение по вероятностям. |

**Правило Деморгана**: если стоит отрицание перед всем выражением, то меняем всё на противоположное.

|  |  |
| --- | --- |
| if(!**(** (a>b) && (x==y) **)**) | if ((a<b)||(x!=y)) |

### Удаление лишнего кода

Лишний код влияет только на размер программы, но не на время выполнения!

* **Недостижимый код** (goto X; … 🡨 недостижимый X: …) – в настоящее время вырезается компилятором.
* **Условно недостижимый код** - условие, которое никогда не будет истинным (компилятор не обнаруживает):

if (x\*x <0) {…}

* **Лишние присваивания:**

|  |  |
| --- | --- |
| **Исходный код:** | **Оптимизированный код:** |
| t1 = b\*b;  t2 = 4\*a\*c;  t3 = t1 – t2;  D = sqrt (t3); | D = sqrt (b\*b - 4\*a\*c); |

* **Повторные присваивания**:

|  |  |
| --- | --- |
| **Исходный код:** | **Оптимизированный код:** |
| x = 0;  y = 5;  x = y; | x = y = 5; |

## Машинно-зависимая оптимизация

Данный вид оптимизации даёт больший эффект, поскольку он привязан напрямую к архитектуре и особенностям текущего процессора и операционной системы. Рассмотрим эти особенности:

1. **Размеры типов данных (размер машинного слова):**

int x;

A = x <<16; ⬄ A = 0; (если int <= 16).

1. **Порядок байт в слове:**

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | D |

int A =

* Прямой порядок байт в слове (big endian) ABCD (Motorola);
* Обратный порядок байт в слове (little endian) DCBA (Intel);
* VAX — [аббревиатура](http://ru.wikipedia.org/wiki/%D0%90%D0%B1%D0%B1%D1%80%D0%B5%D0%B2%D0%B8%D0%B0%D1%82%D1%83%D1%80%D0%B0) от «**V**irtual **A**ddress e**X**tension» BADC

(Часто применяется в криптографии – если данные хранятся в одном порядке, то делаем XOR);

void encrupt (bute \*buf, key)

# ifdef BIG\_ENDIAN

/\* побитово сдвигаем число\*/

A = (uint) buf[0] |

(uint) buf[1] << 8 |

(uint) buf[2] << 16 |

(uint) buf[3] << 32;

#else

A = \*(uint\*) &buf[0];

#endif

A^ = key;

…

1. **Размер кэша:**

Кэш – самая быстрая память после регистров. В него попадают наиболее часто используемые данные. В данном примере мы изменяем порядок заполнения, поскольку кэш хранится строками, а не столбцами. Однако такая оптимизация может конфликтовать с правильным вложением циклов.

|  |  |
| --- | --- |
| **Исходный код:** | **Оптимизированный код:** |
| int A[16][256];  for (i = 0; i < 256; i++)  for (j = 0; j < 16; j++)  A[i][j] = 0; | int A[16][256];  for (i = 0; i < 256; i++)  for (j = 0; j < 16; j++)  A[j][i] = 0; |

1. **Количество регистров:**

Идея состоит в том, чтобы делать некоторые переменные (например, внутри цикла) регистровыми, тем самым уменьшив время выполнения программы. Но данный метод не рекомендуется использовать, поскольку достоверно неизвестно, какой объем памяти отводится под регистры. В следствие этого возможны следующие варианты развития событий:

* Количество свободных регистров < количество регистровых переменных: в таком случае те переменные, которым выгоднее было бы попасть в регистр, могут не попасть туда и тем самым снизить скорость выполнения программы.
* Количество свободных регистров > количество регистровых переменных: используются не все возможности системы, что также плохо сказывается на скорости выполнения программы.
* Количество свободных регистров == количество регистровых переменных: самый оптимальный вариант, но никогда нельзя знать наверняка их количество.

Таким образом, использование регистровых переменных возможно только в случае, если выбраны 1-2 переменные, в остальных случаях лучше не использовать их, так как современные компиляторы сами способны оптимально распределить переменные по регистрам.

1. **Замена индекса на указатель:**

Применима к циклам. Обеспечивает более быстрый доступ, однако не всегда это действительно так. Компания Intel сейчас работает над тем, чтобы такая оптимизация не отличалась по времени выполнения от неоптимизированного кода.

|  |  |
| --- | --- |
| **Исходный код:** | **Оптимизированный код:** |
| for (i =0; i<N; i++) A[i] = B[i] + 5; | pA = &A[0];  pB = &B[0];  for ( ; pA<&A[N];pA++, pB++)  \*pA =\*pB+5; |

# Практическая часть

## Описание исходной программы «Обфускатор»

Обфускатор ― это программа, которая модифицирует исходный код программы таким образом, что она не теряла своей функциональности, но при этом становилась труднее для понимания и восприятия. Примечательно, что любой оптимизирующий компилятор с лёгкостью обходит все механизмы обфускации, то есть для компьютера программа остаётся неизменной, а вот для человека она трудночитаема и труднопонимаема.

Исходная программа выполняет 5 основных функций:

1. Удаление комментариев;
2. Замена имен переменных и функций;
3. Добавление «мусора»;
4. Перемешивание функций;
5. Удаление пустых символов.

Функция DeleteCom() удаляет из программы все комментарии, учитывая, что они могут быть как вида //, так и вида /\*\*/. Также учитываются случаи избегания удаления текста после // и /\* \*/ внутри кавычек.

Далее происходит замена имён переменных и функций. Для начала вызывается функция CreateVariables(), которая находит все переменные типов int, float, char, void, FILE, переписывает их в отдельный файл, где название каждой переменной находится на отдельной строке. Затем функция CountLines() считает количество строк (соответственно, переменных), после чего генерируются новые имена переменных случайным образом и записываются в другой файл, где также каждая переменная находится на отдельной строке. Затем вызывается функция RenameVariables(). В ней происходит чтение исходного кода, поиск по имени переменных в массиве, и, если название совпало, выполняется замена имени и запись в выходной файл.

Следующим шагом в программу добавляется мусор. Количество мусорных циклов и функций выбирается случайным образом с помощью вызова rand() (максимум 29). Для этого была создана специальная структура, в которой хранятся данные о начале, условии, окончании цикла, действии, выполняемом внутри цикла, значении переменных, имени сгенерированных переменных и имени сгенерированных функций. Для добавления мусора сначала вызывается функция GenerateCycles(), в которой создаются условия цикла, имена используемых внутри циклов переменных, названия вызываемых функций и действие, выполняемое внутри цикла. Затем вызывается функция WriteCycles(). Внутри нее сначала выполняется запись глобальных переменных, которые были созданы для мусора, затем по алгоритму происходит запись мусорных циклов и функций внутри найденных функций.

Следующим шагом является изменение порядка следований исходных функций. Для начала происходит вызов функции GenerateFunctionFiles(), в которой выполняется чтение из файла, и, если встречается объявление типа, происходит проверка, к чему именно оно относится: к переменной или функции. В первом случае просто переписывается переменная, во втором – вызывается функция создаётся файл с расширением .с, название которого – «FunN», где N – порядок следования функции в файле (начиная с нуля). В созданный файл копируется код данной функции. GenerateFunctionFiles() проверяет объявления int, float, double, char, long, void и struct, и возвращает количество найденных функций. После возврата в main происходит запись названия всех функций в начале итогового файла, выбирается случайный порядок следования функций с помощью вызова rand() и, в конце концов, переписываются сами функции.

На последнем шаге функция DeleteSpaces удаляет пробелы и знаки табуляции. Важно отметить, что далеко не каждый пробел может быть удалён. Например, после объявления типа данных обязательно должен стоять пробел, иначе он не будет восприниматься компилятором, а превратится просто в набор символов.

## Оптимизация на уровне алгоритма

После исследования исходного кода программы были найдены участки кода, которые удалось оптимизировать на уровне алгоритма. Начнем по порядку.

На Рисунках 1 и 2 представлены соответственно время использования ЦПУ, измеренное с помощью встроенного в компилятор Visual Studio профилировщика, и размер исходной программы «Обфускатор».

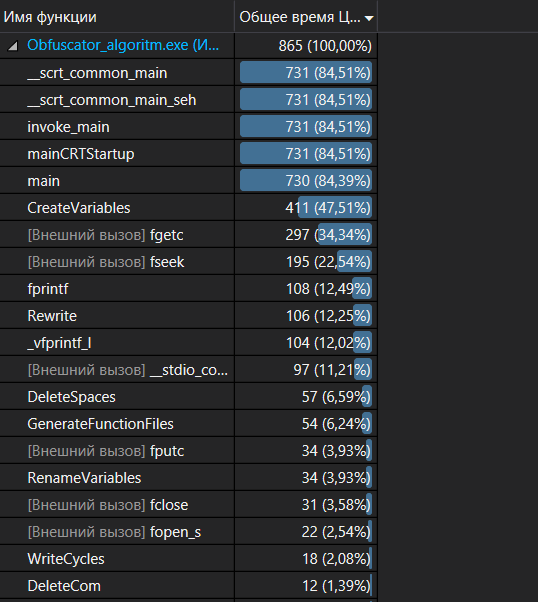


Рисунок 1 – Результат профилирования исходной программы.

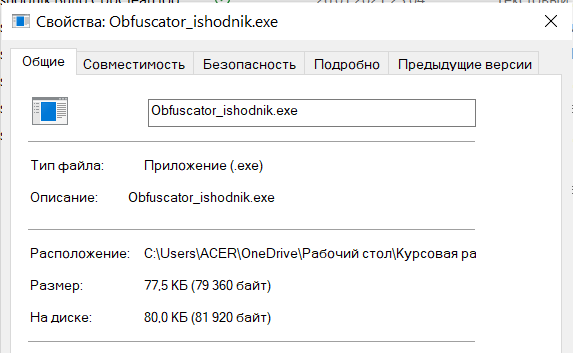


Рисунок 2 – Размер исходной программы.

Изучив данные профилировщика, было обнаружено, что наибольшее время использования процессора приходится на вызов функции CreateVariables(), поэтому было принято решение начать оптимизацию именно с нее.

### Изменения в CreateVariables():

В начальной версии программы выполняется чтение 5 символов из файла в массив GetWord[6]. Затем происходит проверка вхождения ключевых слов “void”, “int”, “char”, “float”, “FILE” с помощью функции strstr(), и, если было найдено одно из них, то происходит фиксация имени в файл Before.txt, содержащий все найденные названия объявленных переменных/функций. Однако зачастую ключевые слова не будут найдены, вследствие чего будет происходить вызов функции fseek(), возвращающей позицию файла назад на определенное количество символов. То есть, в большинстве случаев будет происходить чтение 5 символов и возврат на 4, что по факту означает долгое чтение одного символа. Это крайне неэффективно для скорости выполнения.

В качестве оптимизации был изменен алгоритм считывания и поиска слов. Был создан динамический массив Buffer, в который происходит считывание кода исходной программы, а также переменная razmer, хранящая количество символов в файле. Теперь поиск переменных происходит с помощью массива символов, что позволило перейти от считывания и возврата позиции (fseek) к индексации, что намного увеличивает скорость выполнения программы.

После чтения в массив Buffer поток Read сразу же закрывается. Это сделано для того, чтобы код освобождал ресурсы, а не тратил их попусту и замедлял тем самым работу программы.

После нахождения всех переменных, вызывалась функция CountLines(), чтобы подсчитать количество найденных переменных. Этого можно избежать, если в цикле при каждом нахождении переменной делать инкремент переменной-счётчика. Таким образом, нам не нужно вызывать функцию, открывать файл с именами переменных и проходить от начала до конца.

### Изменения в RenameVariables():

В изначальной версии происходил подсчет линий (т.е. количества переменных) в файле с изначальными названиями переменных и новых созданных, после чего проводилась проверка на совпадение их количества. По сути, это лишняя трата процессорного времени, так как количество новых названий переменных напрямую зависит от количества заменяемых переменных. Решение – передача в виде аргумента функции количества переменных/функций, уже подчитанное в CreateVariables().

### Изменения в main():

В данной функции на стадии перемешивания функций был изменен алгоритм перестановок. В изначальной версии использовался весьма неэффективный алгоритм: в цикле элементу массива присваивалось значение остатка по модулю от числа, сгенерированного функцией rand(), затем во втором цикле происходила проверка на совпадение значений текущего созданного элемента и всех элементов массива. Если число совпадает, то в таком случае внутренний цикл начинается сначала, а текущему элементу массива присваивается новое значение по тому же правилу. Данный алгоритм крайне неэффективен, поскольку в лучшем случае время выполнения составляет O(n2), а в худшем – O(∞) (во втором цикле элементу всегда будет присвоено уже имеющееся значение).

Для оптимизации этого участка кода был применен «Алгоритм P» (псевдокод изображен на Рисунке 3).

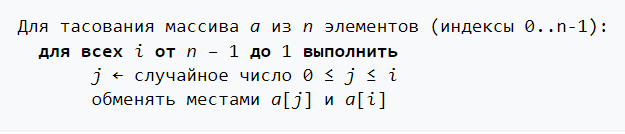


Рисунок 3 – Псевдокод «Алгоритма P».

Сначала в цикле i-ому элементу массива присваивается значение i. Затем применяется «Алгоритм Р». Таким образом, скорость выполнения данного кода становится постоянной O(2n).

| **Исходный код** | **Оптимизация** |
| --- | --- |
| for (int i = 0; i < Count; i++)  {  Random[i] = rand() % Count;  for (int j = 0; j < Count; j++)  if (Random[i] == Random[j] && j != i)  {  j = -1;  Random[i] = rand() % Count;  }  } | for (int i = 0; i < Count; i += 2)  {  Random[i] = i;  if (i + 1 < Count) Random[i + 1] = i + 1;  }    for (int i = 0; i < Count; i++)  {  int j = rand() % Count;  int k;  k = Random[i];  Random[i] = Random[j];  Random[j] = k;  } |

### Результаты оптимизации

Теперь запустим программу «Обфускатор» с учетом выполненной оптимизации и проанализируем полученные данные (см. Рисунки 4 и 5). Исходный код программы, который был заменен, представлен в Приложении 1, а уже полученный в ходе оптимизации – в Приложении 2. В результате оптимизации программа стала работать значительно быстрее, примерно в 4,2 раза, при этом, размер программы был увеличен на 1 Кб. Таким образом, была проведена замена времени на память, но поскольку память с каждым годом становится дешевле, а время – невосполнимый ресурс, можно сказать, что выполненные модификации на уровне алгоритма соответствует современным тенденциям в оптимизации.

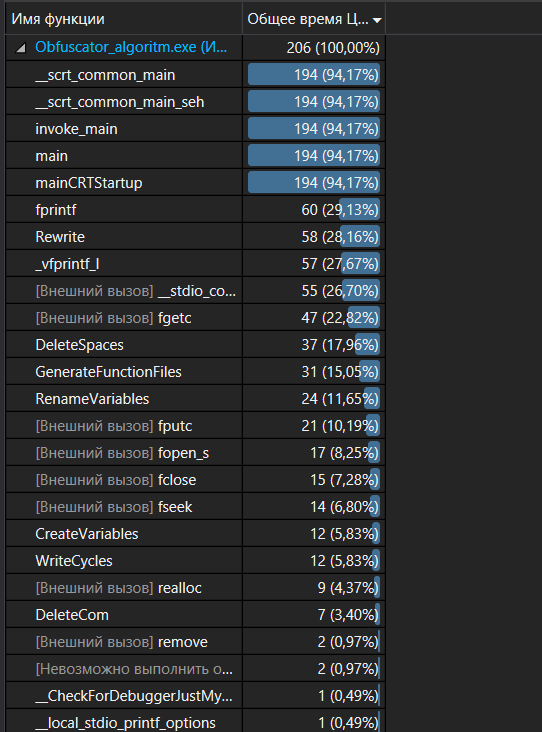


Рисунок 4 - Результат профилирования программы в результате оптимизации на уровне алгоритма.

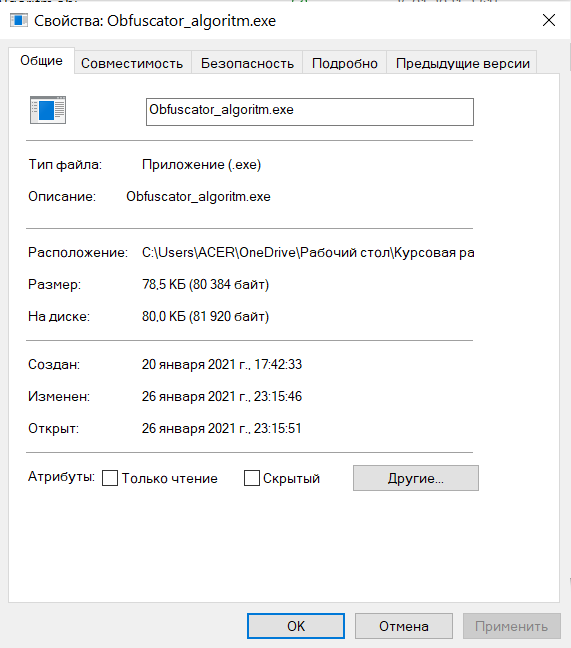


Рисунок 5 - Размер программы в результате оптимизации на уровне алгоритма.

## Машинно-независимая оптимизация

### Изменения в DeleteCom()

Изначальная версия функции DeleteCom() представлена в Приложении 3, а оптимизированная – в Приложении 4. Рассмотрим эти изменения:

* В изначальной версии велась работа с двумя переменными, то есть с двумя соседними символами в файле. В новой версии используется только одна переменная. Вторая переменная появляется только в случае, когда есть подозрение на комментарий, а именно на комбинацию символов // или /\* \*/.
* Убраны лишние сравнения с EOF, которые не имели смысловой нагрузки.
* Добавлена метка start для случаев, когда программа находится внутри множества циклов и условных операторов, и при этом, достигнут конец файла или комментарий уже был удален. Тем самым уменьшается количество присваиваний, сравнений и переходов, что ускоряет программу.
* Исключены некоторые переменные, без которых возможно обойтись. Во-первых, это даёт оптимизацию по размеру. Во-вторых, это влияет на скорость, поскольку на объявление и инициализацию переменных также тратится процессорное время.
* Был изменён порядок следования условий if-else по вероятностям. В большинстве случаев в коде больше нормального текста, чем комментариев, поэтому выгоднее сначала проверять, является ли очередной считанный символ частью комментария.

### Изменения в GenerateCycles()

* В изначальной версии программы на каждый массив, предназначенный для «мусора», выделялось по 100 байт памяти. Как оказалось, размер каждого массива можно уменьшить до 30 байт без потери функционала.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| struct CyclesVariables  {  char ConStart[100];//начало  char ConMid[100];//условие  char ConEnd[100];//конец  char VarAction[100];//операция  char VarValue[100];//значения переменных  char VarName[100];//имя сгенерированных переменных  char FunName[100];//имя функции  }; | struct CyclesVariables  {  char ConStart[30];//начало  char ConMid[30];//тело цикла  char ConEnd[30];//конец  char VarAction[30];//операция  char VarValue[30];//значение переменной  char VarName[30];//имя сгенерированной переменной  char FunName[30];//имя функции  }; |

* Сначала 7 массивов, являющихся компонентами «мусорной» структуры, инициализировались с помощью цикла for. Количество «мусора» определяется перед вызовом функции с помощью функции rand() по модулю 10, и прибавляется 20. То есть, максимальное количество мусора составляет 29. В таком случае произойдет 100\*29 = 2900 итераций цикла. Было решено заменить цикл for на функцию memset.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| for (int j = 0; j < 100; j++)  {  Generate[i].ConStart[j] = '\0';  Generate[i].ConMid[j] = '\0';  Generate[i].ConEnd[j] = '\0';  Generate[i].VarAction[j] = '\0';  Generate[i].VarValue[j] = '\0';  Generate[i].VarName[j] = '\0';  Generate[i].FunName[j] = '\0';  } | memset(Generate[i].ConStart, '\0', 30 \* sizeof(char));  memset(Generate[i].ConMid, '\0', 30 \* sizeof(char));  memset(Generate[i].ConEnd, '\0', 30 \* sizeof(char));  memset(Generate[i].VarAction, '\0', 30 \* sizeof(char));  memset(Generate[i].VarValue, '\0', 30 \* sizeof(char));  memset(Generate[i].VarName, '\0', 30 \* sizeof(char));  memset(Generate[i].FunName, '\0', 30 \* sizeof(char)); |

* В самом конце функции (при инициализации действия VarAction мусорного цикла или условия), часть кода из условия if-else была вынесена, что позволило сделать код более линейным. Линейные участки кода необходимы для оптимизаций, выполняемых компилятором.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| if (Length >= Rand)  {  strcat(Generate[i].ConMid,Generate[i].VarName);  strcat(Generate[i].ConMid, "<=");  char Temp[6];  for (int j = 0; j < 6; j++) Temp[j] = '\0';  \_itoa(Length, Temp, 10);  strcat(Generate[i].ConMid, Temp);  strcat(Generate[i].VarAction,Generate[i].VarName);  Rand = rand() % 2;  if (Rand == 0) strcat(Generate[i].VarAction, "++;");  else strcat(Generate[i].VarAction, "\*=10;");  }  else  {  strcat(Generate[i].ConMid, Generate[i].VarName);  strcat(Generate[i].ConMid, ">");  char Temp[6];  for (int j = 0; j < 6; j++) Temp[j] = '\0';  \_itoa(Length, Temp, 10);  strcat(Generate[i].ConMid, Temp);  strcat(Generate[i].VarAction,Generate[i].VarName);  Rand = rand() % 2 + 0;  if (Rand == 0) strcat(Generate[i].VarAction, "--;");  else strcat(Generate[i].VarAction, "/=10;");  } | Rand = rand() % 9000 + 1000;  \_itoa(Rand, Generate[i].VarValue, 10);  Length = rand() % 9000 + 1000;  strcat(Generate[i].ConMid, Generate[i].VarName);  char Temp[6] = { '\0' };  \_itoa(Length, Temp, 10);  if (Length >= Rand)  {  strcat(Generate[i].ConMid, "<=");  strcat(Generate[i].ConMid, Temp);  strcat(Generate[i].VarAction, Generate[i].VarName);  Rand = rand() % 2;  if (Rand == 0) strcat(Generate[i].VarAction, "++;");  else strcat(Generate[i].VarAction, "\*=10;");  }  else  {  strcat(Generate[i].ConMid, ">");  strcat(Generate[i].ConMid, Temp);  strcat(Generate[i].VarAction,Generate[i].VarName);  Rand = rand() % 2;  if (Rand == 0) strcat(Generate[i].VarAction, "--;");  else strcat(Generate[i].VarAction, "/=10;");  } |

### Изменения в WriteCycles()

* В исходной версии программы массив для поиска первого объявления функции в программе содержал 140 элементов. Как оказалось, его размер может быть уменьшен до 30 элементов без потери функциональности. Помимо этого, изначально массив Temp заполнялся с помощью 140 итераций цикла for, что крайне неэффективно. Теперь инициализация происходит при объявлении. Также, внутри цикла do-while был использован вложенный цикл for, также обнуляющий массив Temp, но при внимательном рассмотрении, был сделан вывод, что заполнение массива каждый раз вообще не требуется, так как функция fscanf при чтении строки сама добавляет символ '\0' в конец. Соответственно, от цикла можно полностью избавиться.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| char Temp[140];    for (int i = 0; i < 140; i++) Temp[i] = '\0';  do  {  fprintf(Second, "%s", Temp);  for (int i = 0; i < 140; i++) Temp[i] = '\0';  char c;  fscanf(First, "%c", &c);  fprintf(Second, "%c", c);  fscanf(First, "%s", Temp);  } while (!strstr(Temp, "int") && !strstr(Temp, "void") && !strstr(Temp, "char") && !strstr(Temp, "float") && !strstr(Temp, "struct")); | char Temp[30] = { '\0' };  do  {  fputs(Temp, Second);  char c;  c = fgetc(First);  fputc(c, Second);  fscanf(First, "%s", Temp);  } while (!strstr(Temp, "int") && !strstr(Temp, "void") && !strstr(Temp, "char") && !strstr(Temp, "float") && !strstr(Temp, "struct")); |

* При выписывании глобальных переменных была выполнена развертка цикла.

**Было:**

for (int i = 0; i < Limit; i++)

{

fprintf(Second, "%s = %s", Generate[i].VarName, Generate[i].VarValue);

if (i != Limit - 1) fprintf(Second, ",");

}

**Стало:**

for (int i = 0; i < Limit; i += 2)

{

fprintf(Second, "int %s = %s;\n", Generate[i].VarName, Generate[i].VarValue);

if (i + 1 < Limit)fprintf(Second, "int %s = %s;\n", Generate[i + 1].VarName, Generate[i + 1].VarValue);

else break;

}

* Был изменён порядок следования условий if-else по вероятностям. Символ «;» встречается в каждой строке, структура встречается намного реже фигурных или обычных скобок, функция main встречается всего лишь единожды.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| if (c == 's' && i == 0) {…}  else if (c == 'm' && i == 0) {…}  else if (c == '{') i++;  else if (c == '}') {…}  else if (c == '"') {…}  else if (c == '(') {…}  else if (c == ';' && i != 0) {…} | if (c == ';' && i != 0) {…}  else if (c == '(') {…}  else if (c == '{') i++;  else if (c == '}') {…}  else if (c == '"') {…}  else if (c == 's' && i == 0)  else if (c == 'm' && i == 0) |

### Изменения в DeleteSpaces()

* Добавлена метка DONE, для уменьшения количества переходов и сравнений и моментального завершения функции, если достигнут конец файла.

if (c == EOF) goto DONE;

…

DONE:

fclose(First);

fclose(Second);

* Был изменён порядок следования условий if-else по вероятностям. Сначала выгоднее проверить очередной символ на то, является ли он пробелом или табуляцией, а уже потом проверять все остальные варианты. Также символ «#» встречается только в самом начале программы при подключении библиотек или при define, то есть на самом деле встречается он не так часто.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| if (c == '#') {…}  else if (c == '\'') {…}  else if (c == '"') {…}  else if (c == '(') {…}  else if (c == '\t' || c == ' ' || c == '\n') {…} | if (c == '\t' || c == ' ' || c == '\n') {…}  else if (c == '\'') {…}  else if (c == '"') {…}  else if (c == '(') {…}  else if (c == '#') |

### Изменения в Rewrite()

* Удалена нужная переменная prev, которая не влияла на функционал, но при этом при каждой итерации копировала значение предыдущего символа.
* Удалено лишнее сравнение конца файла внутри условия цикла while.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| void Rewrite(FILE\* First, FILE\* Second)  {  int c = fgetc(First);  int prev = c;  while (c != EOF || !feof(First))  {  fprintf(Second, "%c", c);  c = fgetc(First);  prev = c;  }  fclose(First);  fclose(Second);  } | void Rewrite(FILE\* First, FILE\* Second)  {  int c;  while ((c = fgetc(First)) != EOF) fputc(c, Second);  fclose(First);  fclose(Second);  } |

### Изменения в RenameVaribles()

* Устранена лишняя инициализация массивов нулями, поскольку в них всё равно будет записано ненулевое значение. Достаточно инициализировать нулем только последний элемент массива.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| for (int i = 0; i < lines; i++)  {  Words[i].before = (unsigned char\*)malloc(A[i] \* sizeof(unsigned char));    Words[i].after = (unsigned char\*)malloc(B[i] \* sizeof(unsigned char));    for (int j = 0; j < A[i]; j++) Words[i].before[j] = '\0';    for (int j = 0; j < B[i]; j++) Words[i].after[j] = '\0';    fread(Words[i].before, sizeof(unsigned char), A[i], Alphabet[0]);    fread(Words[i].after, sizeof(unsigned char), B[i], Alphabet[1]);    Words[i].before[A[i] - 1] = '\0';  Words[i].after[B[i] - 1] = '\0';  } | for (int i = 0; i < lines; i++)  {  Words[i].before = (unsigned char\*)malloc(A[i] \* sizeof(unsigned char));  Words[i].after = (unsigned char\*)malloc(B[i] \* sizeof(unsigned char));  fread(Words[i].before, sizeof(unsigned char), A[i], Alphabet[0]);  fread(Words[i].after, sizeof(unsigned char), B[i], Alphabet[1]);    Words[i].before[A[i] - 1] = '\0';  Words[i].after[B[i] - 1] = '\0';  } |

* Вынос инварианта buf[n] = '\0'; в цикле. Нет смысла каждый раз инициализировать нулем последний элемент, если на следующем шаге он будет переписан. Поэтому сначала лучше записать все элементы в массив, а потом вне цикла инициализировать нулем последний.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| while ((c == '\_' || isalpha(c) || isdigit(c)) && !feof(Modify[0]))  {  buf[n] = c;  n++;  buf[n] = '\0';  buf = (unsigned char\*)realloc(buf, n + 2);  c = fgetc(Modify[0]);  } | while ((c == '\_' || isalpha(c) || isdigit(c)) && !feof(Modify[0]))  {  buf[n] = c;  n++;  buf = (unsigned char\*)realloc(buf, n + 2);  c = fgetc(Modify[0]);  }  buf[n] = '\0'; |

* В изначальном алгоритме был найден участок кода, не влияющий на функциональную составляющую программы. Выполнялся поиск максимальной и минимальной длин исходных переменных (в сумме за O(2n)). Затем в цикле замены имен переменных/функций происходило сравнение: если имя прочитанного слова меньше длины минимального элемента и больше длины максимального, тогда это точно не одна из заменяемых переменных, и ее просто можно переписать. Однако, ниже уже есть условие, выполняющее точно такой же функционал. Данная трата процессорного времени была устранена.

**Исходный код:**

max = FunctionGetMax(A, lines);

min = FunctionGetMin(A, lines);

. . . . . . . . . . . . . . . . . . .

while (!feof(Modify[0]))

{

. . . . . . . . . . . . . . . .

if (strlen(buf) > (max - 1) || strlen(buf) < (min - 1))

{

fprintf(Modify[1], "%s", buf);

memset(buf, NULL, strlen(buf) + 1);

}

else

for (int k = 0; k < lines; k++)

if (strcmp(buf, Words[k].before) == 0 && strcmp(buf, "main") != 0 && strcmp(buf, "char") != 0 && strcmp(buf, "int") && strcmp(buf, "float") != 0 != 0 && strcmp(buf, "void") != 0 && strcmp(buf, "struct") != 0 && strcmp(buf, "FILE") != 0)

{

fprintf(Modify[1], "%s", Words[k].after);

memset(buf, NULL, strlen(buf) + 1);

break;

}

if (buf != NULL)

{

fprintf(Modify[1], "%s", buf);

memset(buf, NULL, strlen(buf) + 1);

}

. . . . . . . .. .. .. . . . . . . .

}

**Оптимизированный код:**

while (!feof(Modify[0]))

{

. . . . . . . . . . . . . . . . . . . . . . .

for (int k = 0; k < lines; k++)

if (strcmp(buf, Words[k].before) == 0 && strcmp(buf, "main") != 0 && strcmp(buf, "char") != 0 && strcmp(buf, "int") && strcmp(buf, "float") != 0 != 0 && strcmp(buf, "void") != 0 && strcmp(buf, "struct") != 0 && strcmp(buf, "FILE") != 0)

{

fprintf(Modify[1], "%s", Words[k].after);

memset(buf, NULL, strlen(buf) + 1);

break;

}

if (buf != NULL)

{

fprintf(Modify[1], "%s", buf);

memset(buf, NULL, strlen(buf) + 1);

}

. . . . . . . . . . . . . . . . . . . . . . . .

}

### Изменения в GenerateFunctionFiles()

* Основой оптимизацией в данной функции является вынос длинного повторяющегося участка кода в отдельную функцию. Это упрощает чтение и понимание кода, уменьшает его объем. Помимо этого, уменьшится ассемблерный код, поскольку вместо 6 одинаковых участков станет 1, вынесенный в функцию, из-за чего уменьшится размер выходной программы.

int k = -1;

char g;

int l = 0;

char\* TakeCare;

TakeCare = (char\*)malloc(sizeof(char) \* 2);

while ((g = fgetc(First)) != '=' && g != '(' && g != ')' && g != ';')

{

TakeCare[l] = g;

l++;

TakeCare = (char\*)realloc(TakeCare, sizeof(char) \* (l + 2));

TakeCare[l] = '\0';

if (g == '\n') k--;

k--;

}

if (g == '=' || g == ';')

{

fseek(First, k, SEEK\_CUR);

fprintf(Second, "%s", Check);

while ((g = fgetc(First)) != ';') fputc(g, Second);

fputc(g, Second);

}

else if (g == '(')

{

FILE\* Buffer;

char FunNum[8] = "Fun";

char Number[3];

for (int j = 0; j < 3; j++) Number[j] = '\0';

\_itoa(fun, Number, 10);

fun++;

strncat(FunNum, Number, strlen(Number));

strncat(FunNum, ".c", strlen(".c"));

fopen\_s(&Buffer, FunNum, "w");

fprintf(Buffer, "%s%s%c", Check, TakeCare, g);

while ((g = fgetc(First)) != '{') fputc(g, Buffer);

fputc(g, Buffer);

i++;

while (i > 0 && !feof(First))//пока внутри функции

{

g = fgetc(First);

if (g != EOF) fputc(g, Buffer);

if (g == '\'')

{

while ((g = fgetc(First)) != '\'')

{

if (g == '\\')

{

fputc(g, Buffer);

g = fgetc(First);

}

fputc(g, Buffer);

}

fputc(g, Buffer);

}

else if (g == '\"')

{

while ((g = fgetc(First)) != '\"')

{

if (g == '\\')

{

fputc(g, Buffer);

g = fgetc(First);

}

fputc(g, Buffer);

}

fputc(g, Buffer);

}

else if (g == '{') i++;

else if (g == '}') i--;

}

fclose(Buffer);

}

free(TakeCare);

TakeCare = NULL;

* Также был изменён порядок следования условий if-else по вероятностям. Символ '#' гарантированно встречается в самом начале программы при подключении библиотек или при define, поэтому необходимо проверить его самым первым. Функции типа int и void встречаются чаще всего, поэтому передвигаем их вперёд. Далее будет сравнение на наличие функции типа char, затем на наличие структуры. Также тип float встречается чаще double, самым последним будет проверка на тип long.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| if (c == '{') i++;  else if (c == '}') i--;  else if (i == 0 && c == 'd') {…}  else if (i == 0 && c == 's') {…}  else if (i == 0 && c == 'f') {…}  else if (i == 0 && c == 'l') {…}  else if (i == 0 && c == 'c') {…}  else if (i == 0 && c == 'i') {…}  else if (i == 0 && c == 'v') {…}  else if (c == '#') {…} | if (c == '#') {…}  else if (c == '{') i++;  else if (c == '}') i--;  else if (i == 0 && c == 'v') {…}  else if (i == 0 && c == 'i') {…}  else if (i == 0 && c == 'c') {…}  else if (i == 0 && c == 's') {…}  else if (i == 0 && c == 'f') {…}  else if (i == 0 && c == 'd') {…}  else if (i == 0 && c == 'l') {…} |

### Общие изменения по всему коду

* Инициализация вспомогательных массивов происходит при создании переменной вместо инициализации каждого отдельного элемента в цикле for.

| **Исходный код** | **Оптимизация** |
| --- | --- |
| char Temp[6];  for (int j = 0; j < 6; j++) Temp[j] = '\0'; | char Temp[6] = { '\0' }; |

* Замена fprintf на fputs и fputc. Вызов функции fputs() требует гораздо меньше процессорного времени на реализацию, чем fprintf(), поскольку fputs() выводит только строку символов. Она не может выводить числа или выполнять преобразование форматов. Она занимает, меньше места и работает быстрее fprintf().

### Результаты оптимизации

Рассмотрим результаты работы программы в ходе выполнения машинно-независимой оптимизации (Рисунки 6 и 7). Скорость программы возросла в 5,6 раз, а размер уменьшился на целых 12 Кбайт по сравнению с первоначальным кодом.

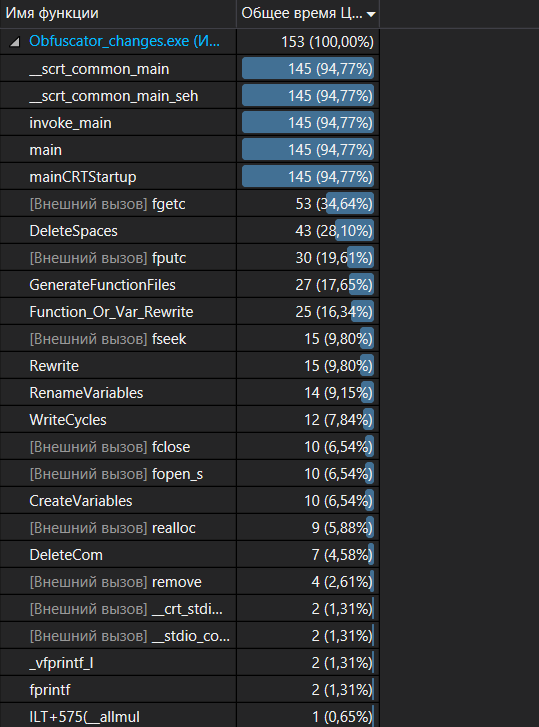


Рисунок 6 - Результат профилирования программы в результате МНЗ оптимизации.

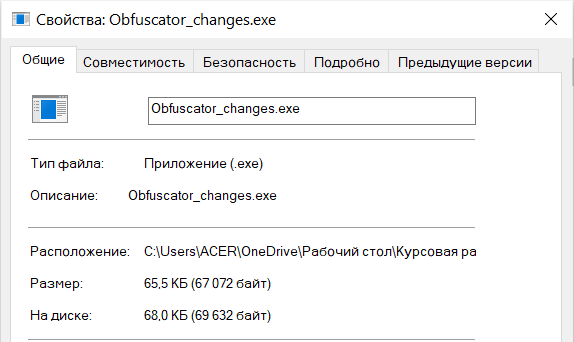


Рисунок 7 - Размер программы в результате МНЗ оптимизации.

## Машинно-зависимая оптимизация

* В некоторых участках кода была проведена замена конструкции if-else-if на switch case:

**DeleteCom():**

| **Исходный код** | **Оптимизация** |
| --- | --- |
| if (s1 == '\"') {...}  else if (s1 == '\'') {...}  else if (s1 == '/') {...}  else if (s1 != EOF) {...} | switch (s1)  {  case '"':  case '\'': {...}  case '/': {...}  default: {...}  } |

**CreateVariables():**

| **Исходный код** | **Оптимизация** |
| --- | --- |
| if (Buffer[shch] == '(') {...}  else if (Buffer[shch] == ')') {...}  else if (Buffer[shch] == ',' && FlagFunction == 1)  {...}  else if (Buffer[shch] == ',' && FlagFunction == 0)  {...}  else if (Buffer[shch] == '{')  {...} | switch (Buffer[shch])  {  case '(': {...}  case ')': {...}  case ',':  if (FlagFunction == 1){...}  else {...}  break;  case '{': {...}  } |

**WriteCycles():**

| **Исходный код** | **Оптимизация** |
| --- | --- |
| if (c == ';' && i != 0) {...}  else if (c == '(')  {  if (c == '\'') {...}  else if (c == '"') {...}  else if (c == '(') {...}  else if (c == ')') {...}  }  else if (c == '{') {...}  else if (c == '}') {...}  else if (c == '\'') {...}  else if (c == '”') {...}  else if (c == 's' && i == 0) {...}  else if (c == 'm' && i == 0) {...} | switch (c)  {  case ';':  if (i != 0) {...}  break;  case '(':  switch (c)  {  case '\'':  case '"': {...}    case '(': {...}  case ')': {...}  }    case '{': {...}  case '}': {...}    case '\'':  case '"': {...}    case 's': {...}    case 'm': {...}  } |

**GenerateFunctionFiles():**

| **Исходный код** | **Оптимизация** |
| --- | --- |
| if (c == '#') {…}  else if (c == '{') i++;  else if (c == '}') i--;  else if (i == 0 && c == 'v') {…}  else if (i == 0 && c == 'i') {…}  else if (i == 0 && c == 'c') {…}  else if (i == 0 && c == 's') {…}  else if (i == 0 && c == 'f') {…}  else if (i == 0 && c == 'd') {…}  else if (i == 0 && c == 'l') {…} | switch (c)  {  case '#': {...}  case '{': {...}  case '}': {...}  case 'v': {...}  case 'i': {...}  case 'c': {...}  case 's': {...}  case 'f'': {...}  case 'd': {...}  case 'l': {...}  } |

**Function\_Or\_Var\_Rewrite():**

| **Исходный код** | **Оптимизация** |
| --- | --- |
| if (g == '=' || g == ';') {...}  else if (g == '(') {...} | switch (g)  {  case '=':  case ';': {...}  case '(': {...}  } |

**DeleteSpaces():**

| **Исходный код** | **Оптимизация** |
| --- | --- |
| if (c == '\t' || c == ' ' || c == '\n') {...}  else if (c == '\'') {...}  else if (c == '”') {...}  else if (c == '(')  {  if (c == '\'') {...}  else if (c == '"') {...}  else if (c == '(') {...}  else if (c == ')') {...}  }  else if (c == '#') {...} | switch (c)  {  case '\t':  case ' ':  case '\n': {...}  case '\'':  case '"' : {...}  case '(':  switch (c)  {  case '\'':  case '"': {...}  case '(': {...}  case ')': {...}  }  case '#': {...} |

* По всему коду программы, где это было возможно, была проведена замена индекса на указатель. Например, в функции WriteCycles():

**Было:**

for (int i = 0; i < Limit; i += 2)

{

fprintf(Second, "int %s = %s;\n", Generate[i].VarName, Generate[i].VarValue);

if (i + 1 < Limit)fprintf(Second, "int %s = %s;\n", Generate[i + 1].VarName, Generate[i + 1].VarValue);

else break;

}

**Стало:**

for (struct CyclesVariables\* p = &Generate[0]; p <= &Generate[Limit - 1]; p += 2)

{

fprintf(Second, "int %s = %s;\n", p->VarName, p->VarValue);

if (p + 1 <= &Generate[Limit - 1]) fprintf(Second, "int %s = %s;\n", (p+1)->VarName, (p+1)->VarValue);

else break;

}

Теперь проанализируем полученные результаты машинно-зависимой оптимизации (Рисунки 8 и 9). Итоговая скорость программы увеличилась в 8,3 раза по сравнению с первоначальной версией, а размер программы уменьшился на 13,5 Кбайт.

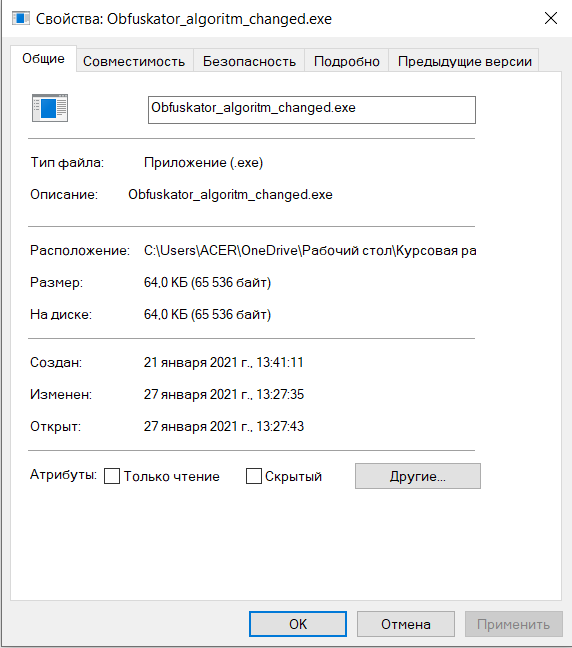


Рисунок 8 – Размер программы после МЗ оптимизации.

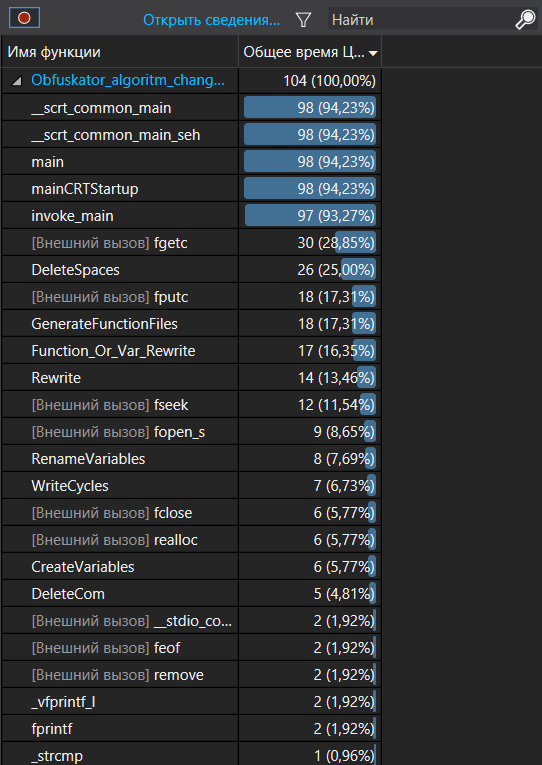


Рисунок 9 - Результат профилирования программы в результате МЗ оптимизации.

## Оптимизация по размеру программы «Hello world!»

Была реализована программа, выводящая в консоль сообщение «hello, world!». Необходимо максимально возможно уменьшить её размер. Исходный код программы:

#include <stdio.h>

int main() {

printf("hello, world!");

}

Изначально размер файла формата .ехе при режиме сборки Debug занимает 37,5 Кб, что очень много для такой маленькой программы (см. Рисунок 10).

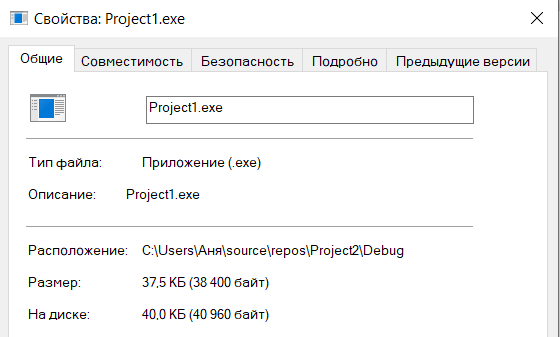


Рисунок 10 – Исходный размер программы

Причины увеличения размера программы:

1. Одна из самых главных причин это CRT (C Runtime Library) которая добавляет довольно много кода в исполняемый файл, даже когда в этом нет необходимости. Отказаться от CRT не так просто как кажется.
2. Особенности создания исполняемого файла
3. Особенности настройки оптимизации
4. Особенности самого кода программы

Будем поэтапно рассматривать каждый из этих пунктов и вносить соответствующие изменения.

### Избавление от статической линковки CRT:

Для этого в свойствах проекта в параметре Свойства конфигурации -> C/C++ -> Создание кода -> Библиотека времени выполнения задать значение «Многопоточная DLL (/MD)».

### Настройка компиляции:

Для начала установим стандартные инструменты оптимизации. Для этого в разделе Свойства конфигурации -> С/С++ -> Оптимизация следует задать следующие параметры:

* Оптимизация – Наименьший размер (/O1)
* Развертывать подставляемые функции – По умолчанию
* Включить подставляемые функции – Нет
* Предпочитать размер или краткость кода – Предпочитать краткость кода (/Os)
* Оптимизация всей программы – Включить создание кода во время компоновки (/GL). Таким образом, оптимизация будет выполнена на уровне всех модулей программы, а не каждого модуля в отдельности.

Однако на этом этапе возникла ошибка, связанная с несовместимостью флагов командной строки. Переводим проект в режим сборки Release. После этого ошибка исчезает и всё компилируется корректно.

Далее следует отказаться от всякого дополнительного кода, который автоматически вставляется при компиляции. Для этого в разделе Свойства конфигурации -> С/С++ -> Создание кода необходимо задать следующие параметры:

* Включить объединение строк – Да (/GF). Данный параметр заставляет компилятор объединять одинаковые строки в одну, если они не изменяются.
* Включить С++ исключения – Нет. Необходимо отключить, поскольку программа на чистом языке Си.

На следующем этапе необходимо убрать все лишние, предварительно скомпилированные заголовки. В разделе Свойства конфигурации -> С/С++ -> Предварительно скомпилированные заголовки -> Создавать или использовать предварительно скомпилированные заголовки устанавливаем значение – Не использовать предварительно скомпилированные заголовки.

Также установим, что данная программа на чистом Си. В разделе Свойства конфигурации -> С/С++ -> Дополнительно -> Компилировать необходимо указать параметр – Компилировать как C код (/TC).

### Настройка компоновки:

В разделе Свойства конфигурации -> Компоновщик необходимо изменить следующие параметры:

* Ввод -> Игнорировать все стандартные библиотеки – Да (/NODEFAULTLIB). Это позволит избавиться сразу от CRT. Тут возникают две ошибки. Компилятор не нашёл два параметра: проверку на переполнение буфера и точку входа программы. Чтобы отключить проверку на переполнение буфера нужно изменить параметр: C/C++ -> Создание кода -> Проверка безопасности -> Отключить проверку безопасности (/GS-). Несмотря на то, что такие действия абсолютно небезопасны, тем не менее, их необходимо провести для максимальной оптимизации. По сути, в данной рассматриваемой программе нет как такового буфера, сообщение поступает напрямую в виде строки. К тому же, в ОС Windows есть свои механизмы защиты от переполнения, которые не были отключены. Таким образом, можно сказать, что данное действие оправдано.
* Дополнительно -> Точка входа – EntryPoint. CRT ставила свою точку входа, но поскольку необходимо от неё избавиться, то и точку входа придётся задавать вручную.
* Файл манифеста -> Создавать Манифест – Нет. Манифест в данной программе не нужен в силу её тривиальности.
* Отладка -> Создавать отладочную информацию – Нет. Отладочная информация не нужна, поскольку была выбрана сборка не Debug, а Release.
* Дополнительно -> Внесение случайности в базовый адрес – Нет (/DYNAMICBASE:NO).
* Дополнительно -> Фиксированный базовый адрес – Да (/FIXED).

### Замена функций CRT:

Поскольку CRT был отключен, придётся пользоваться функциями WinAPI. В ОС Windows существует два общепринятых формата строк: Ansi Char (размером 1 байт) и Wide Char (размером 2 байта). Для обработки каждого из них существует два вида WinAPI функций (оканчивающихся на A или W). По умолчанию VS использует Wide строки. Во избежание этого следует для параметра Свойства конфигурации -> Общие -> Набор Знаков задать значение «Не задано». Это позволит использовать различные встроенные макросы для Ansi функций без каких-либо ошибок. Большинство функций CRT уже реализовано в системных библиотеках с тем или иным изменением. Рассмотрим самые популярные:

1. Работа со строками: strcpy – lstrcpyA, strlen – lstrlenA, strcmp – lstrcmpA, itoa – StrToIntA,
2. Работа с памятью: malloc – HeapAlloc или VirtualAlloc зависит от размера выделяемой памяти и её предназначении, free – HeapFree или VirtualFree,
3. Работа с файлами: fopen – CreateFileA, fclose – CloseHandle, fwrite – WriteFile, fread – ReadFile,
4. Консольный ввод/вывод: printf – WriteConsole, scanf – ReadConsole + парсинг строк,
5. Аргументы командной строки: argc + argv – CommandLineToArgvW + GetCommandLineW.

В данном случае необходимо изменить только команду вывода в консоль. Функция WriteConsoleA принимает следующие аргументы: дескриптор буфера экрана консоли, указатель на буфер, содержащий символы для записи в буфер экрана консоли, количество записываемых символов, указатель на переменную, которая получает количество фактически записанных символов (в данном рассматриваемом случае NULL), зарезервированный аргумент, должно быть значение NULL. Изменённый код программы:

#include <windows.h>

void EntryPoint() {

WriteConsoleA(GetStdHandle(STD\_OUTPUT\_HANDLE), "hello, world!", 13, NULL, NULL);

}

### Изменение параметров секций:

Данный пункт даёт довольно хорошую оптимизацию по размеру (до килобайта). По умолчанию Си компилятор создаёт следующие секции: .text – секция кода, .rdata – секция импорта, .data – секция данных, а также секции экспорта, ресурсов и прочие. Каждая секция перед записью в файл выравнивается по размеру на 512 байт, то есть размер секции всегда кратен будет 512 байтам. Например, если есть глобальные переменные общим весом 4 байта, то всё равно секция данных будет занимать минимум 512 байт. Таким образом, если 3 секции будут занимать 511 \* 3 = 1533 байт из-за выравнивания размера. Чтобы избежать такой большой потери, можно прибегнуть объединению нескольких секций в одну. Практика показала, что практически всегда объединению поддаются секции кода, импорта и данных.  
Для объединения секций необходимо прописать следующий код:

[#pragma](https://vk.com/im?sel=334554302&st=%23pragma) comment(linker, "/MERGE:.data=.text")

[#pragma](https://vk.com/im?sel=334554302&st=%23pragma) comment(linker, "/MERGE:.rdata=.text")

Однако тут есть один подводный камень – секция кода имеет права RE (чтение и выполнение), секция данных имеет права RW (чтение и запись), а секция импорта имеет права R (только чтение). И после объединения всех секций в одну, последняя должна иметь общие права для всех секций которые в неё вошли, иначе можно столкнуться с ошибками. Для установки прав необходимо после объединения секций прописать команду, которая даст секции .text права на чтение, запись и выполнение:

[#pragma](https://vk.com/im?sel=334554302&st=%23pragma) comment(linker, "/SECTION:.text,EWR")

После всех изменений .exe файл стал весить всего лишь 1,5 Кб, что 25 раз меньше исходного размера (см. Рисунок 11).

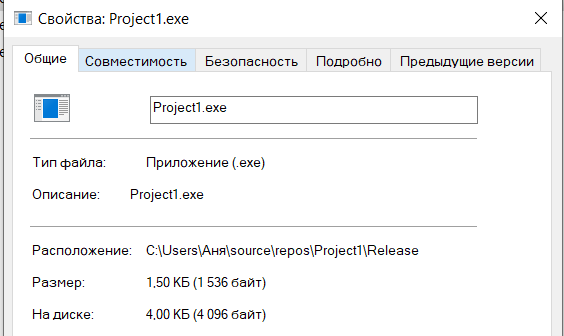


Рисунок 11 – Размер оптимизированной программы

Заключение

В ходе выполнения данной работы были изучены различные методы оптимизации, позволяющие добиться максимального улучшения показателей времени исполнения и объема занимаемой памяти.

Несмотря на то, что все современные компиляторы сами умеют проводить многие виды оптимизаций, тем не менее, во-первых, они работают только с машинно-независимыми методами оптимизации и совершенно не умеют изменять алгоритм. На уровне алгоритма изменять код может только сам программист, поскольку только он знает, какой результат работы программы ему необходимо получить. Во-вторых, целью данной работы было научиться самостоятельно проводить оптимизации, чтобы лучше понимать механизм их работы, а также прививать привычку писать изначально оптимальный код.

Говоря об оптимизации на уровне алгоритма, можно сказать, что она оказалась самой трудоёмкой. Поскольку сам исходный код программы нетривиален, было сложно придумать новый алгоритм, который бы значительно отличался от первоначального, при этом был бы рабочим и давал лучшие результаты. Однако всё-таки некоторые изменения провести удалось. Например, получилось избавиться от лишних функций, которые тратят время на передачу аргументов, и заменить их вычислениями внутри исходной функции. Также вместо отдельной функции и лишнего сравнения можно просто передавать аргументы из одной функции в другую, что также избавляет код от лишних вычислений и сравнений. Одной из самых эффективных оптимизаций стала замена алгоритма перемешивания функций, поскольку исходный алгоритм мог вырождаться в O(∞). Новый алгоритм работает за O(2n), что очевидно эффективнее. По итогу данная оптимизация дала ускорение в 4,2 раза, при этом размер программы увеличился на 1 Кбайт. Таким образом, была проведена замена времени на память, но поскольку память с каждым годом становится дешевле, а время – невосполнимый ресурс, можно сказать, что выполненные модификации на уровне алгоритма соответствует современным тенденциям в оптимизации.

При работе с машинно-независимой оптимизацией основными изменениями были замены инициализации в цикле на инициализацию при объявлении массивов, изменения порядка следования условий if-else-if по вероятностям, замены fprintf на fputs. Также в некоторых отдельных функциях были удалены лишние переменные, сравнения и уменьшены размеры массивов. Помимо этого, удаление лишнего участка кода позволило уменьшить размер исходной программы. Несмотря на то, что от некоторых циклов удалось избавиться вовсе, оставшиеся циклы по возможности были развернуты, что позволило уменьшить количество ассемблерных команд jmp, cmp и inc. В итоге, применение машинно-независимых методов оптимизации дало ускорение в 5,65 раз относительно исходного кода. Размер программы уменьшился на 13 Кб.

Машинно-зависимая оптимизация оказалась самой незначительной. Из всех изменений имеются только замены индексов на указатели и замена конструкций if-else-if на оператор switch-case. Результаты исследований показывают, что оператор switch выполняется быстрее, чем лестница if-else-if. Это связано со способностью компилятора генерировать таблицу переходов, причем порядок также устанавливается максимально оптимальным для компилятора. То есть, если распределение по вероятностям, проведённое в пункте с машинно-независимой оптимизацией, всё-таки не было самым оптимальным, на данном этапе оно всё же станет максимально быстрым. Таким образом, образуется поиск произвольного случая сложности O(1), что эффективнее нескольких сравнений. Чем длиннее лестница if-else-if, тем лучше работает оператор switch. Говоря о замене индексов на указатели, можно столкнуться с некоторыми трудностями, поскольку компания Intel стремится сделать так, чтобы минимизировать различия в использовании этих двух вариантов. В итоге после применения вышеперечисленных методов оптимизации общее ускорение программы относительно исходной стало в 8,13 раз. Размер программы уменьшился на 1,5 Кб.

Второй частью курсовой работы было максимальное уменьшение программы, выводящей в консоль фразу «Hello, world!». Данная программа изначально занимала объём 37,5 Кб, что очень много для программы с таким малым функционалом. При этом, подключение стандартных флагов оптимизации (например, \O1) не дало заметных изменений размера. Таким образом, можно заключить, что оптимизации, проводимые компилятором, на самом деле малоэффективны. Одной из самых лучших оптимизаций было переключение режима сборки с Debug на Release. Дело в том, что конфигурация Debug предназначена для отладки программы. Эта конфигурация отключает все настройки по оптимизации, включает информацию об отладке, что делает программу больше и медленнее, но упрощает проведение отладки. Конфигурация Release используется во время сборки программы для её дальнейшего выпуска. Программа оптимизируется по размеру и производительности и не содержит дополнительную информацию об отладке. Объединение секций кода, данных и импорта дало уменьшение размера около 1 Кб. Также хорошую оптимизацию дали удаление информации о манифесте и отладке и отключение проверки на переполнение буфера. Таким образом, программу удалось уменьшить в 25 раз, что является значительным результатом.

Подводя итог работы, можно сделать вывод, что, несмотря на стремительное развитие оптимизирующих компиляторов и их высокую эффективность, в некоторых аспектах оптимизации, таких как выбор оптимального алгоритма, человек способен сделать больше, чем компилятор.

Оптимизация, связанная с алгоритмом, показала максимально возможное ускорение программы относительно других методов оптимизации. Помимо этого, было обнаружено, что машинно-независимая оптимизация лучше других уровней уменьшила размер исходной программы. Из этого следует, что наилучшим решением будет использовать методы оптимизации в совокупности, чтобы добиться максимального улучшения и по времени, и по памяти.

# Список использованных источников

1. Тасование Фишера-Йетса [Электронный ресурс]. URL: <https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D1%81%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%A4%D0%B8%D1%88%D0%B5%D1%80%D0%B0_%E2%80%94_%D0%99%D0%B5%D1%82%D1%81%D0%B0>. – (дата обращения: 01.12.2020)
2. Является ли оператор switch быстрее конструкции if-else-if [Электронный ресурс]. URL: <https://coderoad.ru/767821/>. – (дата обращения: 29.11.2020)
3. Speed Test: Switch vs If-Else-If [Электронный ресурс]. URL: <http://www.blackwasp.co.uk/SpeedTestIfElseSwitch.aspx>. – (дата обращения: 29.11.2020)
4. Чтение и запись строк: gets() и puts() [Электронный ресурс]. URL: <http://www.c-cpp.ru/books/chtenie-i-zapis-strok-gets-i-puts>. – (дата обращения: 20.11.2020)
5. Уменьшение размера программы на языке Си в среде разработки Visual Studio [Электронный ресурс]. URL: <https://forum.antichat.ru/threads/270620/>. – (дата обращения: 20.12.2020)
6. Функция вывода в консоль WinAPI [Электронный ресурс]. URL: <https://docs.microsoft.com/ru-ru/windows/console/writeconsole>. – (дата обращения: 21.12.2020)
7. Установки собственной точки входа [Электронный ресурс]. URL: <https://docs.microsoft.com/ru-ru/cpp/build/reference/entry-entry-point-symbol?view=msvc-160>. - (дата обращения: 21.12.2020)

# Приложение 1

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include <time.h>

/\* КОД \*/

int CountLines(FILE\* File)//считаем строки

{

int lines = 1;

rewind(File);

unsigned char c = fgetc(File);

if (File == NULL) return 0;

while (!feof(File))

if (c = fgetc(File) == '\n') lines++;

if (ferror(File))

{

printf("File error\n");

return 0;

}

return lines;

}

void RenameVariables()

{

int lines;

FILE\*\* Alphabet;

FILE\*\* Modify;

int\* A;//массивы кол-ва символов

int\* B;

int max;

int min;

Alphabet = (FILE\*\*)malloc(sizeof(FILE\*) \* 2);

fopen\_s(&Alphabet[0], "Before.txt", "r");

fopen\_s(&Alphabet[1], "After.txt", "r");

int\* CheckLines;

CheckLines = (int\*)malloc(sizeof(int) \* 2);

CheckLines[0] = CountLines(Alphabet[0]);

CheckLines[1] = CountLines(Alphabet[1]);

if (CheckLines[0] != CheckLines[1] || CheckLines[0] == 0 || CheckLines[1] == 0)

{

printf("Lines don't match.\n");

return;

}

lines = CheckLines[0];

free(CheckLines);

CheckLines = NULL;

A = (int\*)calloc(lines, sizeof(int));

B = (int\*)calloc(lines, sizeof(int));

CountLetters(Alphabet[0], A);

CountLetters(Alphabet[1], B);

while (A[lines - 1] == 0 && B[lines - 1] == 0) lines--;

if (A[lines - 1] == 0 || B[lines - 1] == 0)

{

printf("Lines don't match.\n");

return;

}

struct LIST\* Words;

Words = (struct LIST\*)malloc(sizeof(struct LIST) \* lines);

rewind(Alphabet[0]);

rewind(Alphabet[1]);

for (int i = 0; i < lines; i++)//считываем названия

{

Words[i].before = (unsigned char\*)malloc(A[i] \* sizeof(unsigned char));

Words[i].after = (unsigned char\*)malloc(B[i] \* sizeof(unsigned char));

for (int j = 0; j < A[i]; j++)

{

Words[i].before[j] = '\0';

}

for (int j = 0; j < B[i]; j++)

{

Words[i].after[j] = '\0';

}

fread(Words[i].before, sizeof(unsigned char), A[i], Alphabet[0]);

fread(Words[i].after, sizeof(unsigned char), B[i], Alphabet[1]);

Words[i].before[A[i] - 1] = '\0';

Words[i].after[B[i] - 1] = '\0';

}

fclose(Alphabet[0]);

fclose(Alphabet[1]);

free(Alphabet);

Alphabet = NULL;

remove("Before.txt");

remove("After.txt");

max = FunctionGetMax(A, lines);

min = FunctionGetMin(A, lines);

free(A);

free(B);

A = NULL;

B = NULL;

Modify = (FILE\*\*)malloc(sizeof(FILE\*) \* 2);

fopen\_s(&Modify[0], "Final.c", "r");

fopen\_s(&Modify[1], "Draft.c", "w");

unsigned char\* buf;

buf = (unsigned char\*)malloc(2 \* sizeof(unsigned char));

memset(buf, NULL, 2);

while (!feof(Modify[0]))

{

unsigned char c;

int n = 0;

while (!isalpha(c = fgetc(Modify[0])) && c != '\_' && !feof(Modify[0]))//пропускаем код

{

char prev;

fprintf(Modify[1], "%c", c);

if (c == '"')

do

{

prev = c;

c = fgetc(Modify[0]);

fprintf(Modify[1], "%c", c);

if (prev == '\\')

{

prev = c;

c = fgetc(Modify[0]);

fprintf(Modify[1], "%c", c);

}

} while (c != '"');

else if (c == '\'')

do

{

prev = c;

c = fgetc(Modify[0]);

fprintf(Modify[1], "%c", c);

if (prev == '\\')

{

prev = c;

c = fgetc(Modify[0]);

fprintf(Modify[1], "%c", c);

}

} while (c != '\'');

}

while ((c == '\_' || isalpha(c) || isdigit(c)) && !feof(Modify[0]))//переменная

{

buf[n] = c;

n++;

buf[n] = '\0';

buf = (unsigned char\*)realloc(buf, n + 2);

c = fgetc(Modify[0]);

}

if (strlen(buf) > (max - 1) || strlen(buf) < (min - 1))

{

fprintf(Modify[1], "%s", buf);

memset(buf, NULL, strlen(buf) + 1);

}

else

for (int k = 0; k < lines; k++)

if (strcmp(buf, Words[k].before) == 0 && strcmp(buf, "main") != 0 && strcmp(buf, "char") != 0 && strcmp(buf, "int") && strcmp(buf, "float") != 0 != 0 && strcmp(buf, "void") != 0 && strcmp(buf, "struct") != 0 && strcmp(buf, "FILE") != 0)

{

fprintf(Modify[1], "%s", Words[k].after);

memset(buf, NULL, strlen(buf) + 1);

break;

}

if (buf != NULL)

{

fprintf(Modify[1], "%s", buf);

memset(buf, NULL, strlen(buf) + 1);

}

if (!feof(Modify[0])) fprintf(Modify[1], "%c", c);

}

fclose(Modify[0]);

fclose(Modify[1]);

free(Modify);

Modify = NULL;

}

void CreateVariables(FILE\* Read)

{

FILE\* Before;

FILE\* After;

FILE\* Lines;

int LinesCount;

fopen\_s(&Before, "Before.txt", "w");

fopen\_s(&After, "After.txt", "w");

int FlagFunction = 0;

while (!feof(Read))

{

char GetWord[6];

for (int i = 0; i < 6; i++) GetWord[i] = 0;

MarkOutside:

for (int i = 0; i < 6 && !feof(Read); i++) GetWord[i] = fgetc(Read);

if (strstr(GetWord, "void") || strstr(GetWord, "int") || strstr(GetWord, "char") || strstr(GetWord, "float") || strstr(GetWord, "FILE"))

{

char s1;

MarkInside:

s1 = '0';

int i = 0;

while (isspace(s1 = fgetc(Read)) || s1 == '\*') i++;

if ((isalpha(s1) || s1 == '\_') && (i != 0))

{

while (isalpha(s1) || isdigit(s1) || s1 == '\_')

{

fprintf(Before, "%c", s1);

s1 = fgetc(Read);

}

fprintf(Before, "\n");

while (strchr("(),;{}", s1) == NULL) s1 = fgetc(Read);

if (s1 == '(') FlagFunction = 1; //находимся в объявлении переменных, передаваемых в функции

else if (s1 == ')') FlagFunction = 0; // не находимся в объявлении переменных, передаваемых в функции

else if (s1 == ',' && FlagFunction == 1) goto MarkOutside;

else if (s1 == ',' && FlagFunction == 0) goto MarkInside;

else if (s1 == '{')

while ((s1 = fgetc(Read)) != '}');

}

}

else if (!feof(Read)) fseek(Read, -5, SEEK\_CUR);

for (int i = 0; i < 6; i++) GetWord[i] = 0;

}

fclose(Before);

fopen\_s(&Lines, "Before.txt", "r");

LinesCount = CountLines(Lines);

fclose(Lines);

for (int i = 0; i < LinesCount - 1; i++)

{

int Length = rand() % 20 + 5;

for (int j = 0; j < Length; j++)

{

unsigned char c;

if (j % 3 == 0) c = rand() % 25 + 65;

else if (j % 3 == 1) c = rand() % 25 + 97;

else c = rand() % 9 + 48;

fprintf(After, "%c", c);

}

fprintf(After, "\n");

}

fclose(After);

}

/\* КОД \*/

int main()

{

/\* КОД \*/

fopen\_s(&First, "Final.c", "r");

CreateVariables(First);

fclose(First);

RenameVariables();

/\* КОД \*/

for (int i = 0; i < Count; i++)

{

Random[i] = rand() % Count;

for (int j = 0; j < Count; j++)

if (Random[i] == Random[j] && j != i)

{

j = -1;

Random[i] = rand() % Count;

}

}

/\* КОД \*/

}

# Приложение 2

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include <time.h>

char\* Buffer;

long long razmer = 0;

/\* КОД \*/

void RenameVariables(int lines)

{

FILE\*\* Alphabet;

FILE\*\* Modify;

int\* A;//массивы кол-ва символов

int\* B;

int max;

int min;

Alphabet = (FILE\*\*)malloc(sizeof(FILE\*) \* 2);

fopen\_s(&Alphabet[0], "Before.txt", "r");

fopen\_s(&Alphabet[1], "After.txt", "r");

A = (int\*)calloc(lines, sizeof(int));

B = (int\*)calloc(lines, sizeof(int));

CountLetters(Alphabet[0], A);

CountLetters(Alphabet[1], B);

while (A[lines - 1] == 0 && B[lines - 1] == 0) lines--;

if (A[lines - 1] == 0 || B[lines - 1] == 0)

{

printf("Lines don't match.\n");

return;

}

struct LIST\* Words;

Words = (struct LIST\*)malloc(sizeof(struct LIST) \* lines);

rewind(Alphabet[0]);

rewind(Alphabet[1]);

for (int i = 0; i < lines; i++)//считываем названия

{

Words[i].before = (unsigned char\*)malloc(A[i] \* sizeof(unsigned char));

Words[i].after = (unsigned char\*)malloc(B[i] \* sizeof(unsigned char));

for (int j = 0; j < A[i]; j++) Words[i].before[j] = '\0';

for (int j = 0; j < B[i]; j++) Words[i].after[j] = '\0';

fread(Words[i].before, sizeof(unsigned char), A[i], Alphabet[0]);

fread(Words[i].after, sizeof(unsigned char), B[i], Alphabet[1]);

Words[i].before[A[i] - 1] = '\0';

Words[i].after[B[i] - 1] = '\0';

}

fclose(Alphabet[0]);

fclose(Alphabet[1]);

free(Alphabet);

Alphabet = NULL;

remove("Before.txt");

remove("After.txt");

max = FunctionGetMax(A, lines);

min = FunctionGetMin(A, lines);

free(A);

free(B);

A = NULL;

B = NULL;

Modify = (FILE\*\*)malloc(sizeof(FILE\*) \* 2);

fopen\_s(&Modify[0], "Final.c", "r");

fopen\_s(&Modify[1], "Draft.c", "w");

unsigned char\* buf;

buf = (unsigned char\*)malloc(2 \* sizeof(unsigned char));

memset(buf, NULL, 2);

while (!feof(Modify[0]))

{

unsigned char c;

int n = 0;

while (!isalpha(c = fgetc(Modify[0])) && c != '\_' && !feof(Modify[0]))//пропускаем код

{

char prev;

fprintf(Modify[1], "%c", c);

if (c == '"')

do

{

prev = c;

c = fgetc(Modify[0]);

fprintf(Modify[1], "%c", c);

if (prev == '\\')

{

prev = c;

c = fgetc(Modify[0]);

fprintf(Modify[1], "%c", c);

}

} while (c != '"');

else if (c == '\'')

do

{

prev = c;

c = fgetc(Modify[0]);

fprintf(Modify[1], "%c", c);

if (prev == '\\')

{

prev = c;

c = fgetc(Modify[0]);

fprintf(Modify[1], "%c", c);

}

} while (c != '\'');

}

while ((c == '\_' || isalpha(c) || isdigit(c)) && !feof(Modify[0]))//переменная

{

buf[n] = c;

n++;

buf[n] = '\0';

buf = (unsigned char\*)realloc(buf, n + 2);

c = fgetc(Modify[0]);

}

if (strlen(buf) > (max - 1) || strlen(buf) < (min - 1))

{

fprintf(Modify[1], "%s", buf);

memset(buf, NULL, strlen(buf) + 1);

}

else

for (int k = 0; k < lines; k++)

if (strcmp(buf, Words[k].before) == 0 && strcmp(buf, "main") != 0 && strcmp(buf, "char") != 0 && strcmp(buf, "int") && strcmp(buf, "float") != 0 != 0 && strcmp(buf, "void") != 0 && strcmp(buf, "struct") != 0 && strcmp(buf, "FILE") != 0)

{

fprintf(Modify[1], "%s", Words[k].after);

memset(buf, NULL, strlen(buf) + 1);

break;

}

if (buf != NULL)

{

fprintf(Modify[1], "%s", buf);

memset(buf, NULL, strlen(buf) + 1);

}

if (!feof(Modify[0])) fprintf(Modify[1], "%c", c);

}

fclose(Modify[0]);

fclose(Modify[1]);

free(Modify);

Modify = NULL;

}

int CreateVariables(FILE\* Read)

{

FILE\* Before;

FILE\* After;

int LinesCount = 0, i;

int s1;

fopen\_s(&Before, "Before.txt", "w");

fopen\_s(&After, "After.txt", "w");

short FlagFunction = 0;

int c;

for (razmer; (c = fgetc(Read)) != EOF; razmer++)

{

Buffer = (char\*)realloc(Buffer, razmer + 1);

Buffer[razmer] = c;

}

fclose(Read);

short flag = 0;

for (unsigned long long shch = 0; shch < razmer; shch++)

{

MarkOutside:

if (Buffer[shch] == 'i' && Buffer[shch + 1] == 'n' && Buffer[shch + 2] == 't')

{

flag++;

shch += 3;

}

else if ((Buffer[shch] == 'c' && Buffer[shch + 1] == 'h' && Buffer[shch + 2] == 'a' && Buffer[shch + 3] == 'r') || (Buffer[shch] == 'v' && Buffer[shch + 1] == 'o' && Buffer[shch + 2] == 'i' && Buffer[shch + 3] == 'd') || (Buffer[shch] == 'F' && Buffer[shch + 1] == 'I' && Buffer[shch + 2] == 'L' && Buffer[shch + 3] == 'E'))

{

flag++;

shch += 4;

}

else if (Buffer[shch] == 'f' && Buffer[shch + 1] == 'l' && Buffer[shch + 2] == 'o' && Buffer[shch + 3] == 'a' && Buffer[shch + 4] == 't')

{

flag++;

shch += 5;

}

if (flag)

{

MarkInside:

flag = 0;

i = 0;

while (isspace(Buffer[shch]) || Buffer[shch] == '\*')

{

shch++;

i++;

}

if ((isalpha(Buffer[shch]) || Buffer[shch] == '\_') && (i != 0))

{

while (isalpha(Buffer[shch]) || isdigit(Buffer[shch]) || Buffer[shch] == '\_')

{

fputc(Buffer[shch], Before);

shch++;

}

fputc('\n', Before);

LinesCount++;

while (strchr("(),;{}", Buffer[shch]) == NULL) shch++;

if (Buffer[shch] == '(') FlagFunction = 1;

else if (Buffer[shch] == ')') FlagFunction = 0;

else if (Buffer[shch] == ',' && FlagFunction == 1)

{

shch++;

goto MarkOutside;

}

else if (Buffer[shch] == ',' && FlagFunction == 0)

{

shch++;

goto MarkInside;

}

else if (Buffer[shch] == '{')

{

while (Buffer[shch] != '}')shch++;

shch++;

}

}

}

}

free(Buffer);

Buffer = NULL;

fclose(Before);

for (int i = 0; i < LinesCount; i++)

{

int Length = rand() % 20 + 5;

for (int j = 0; j < Length; j++)

{

unsigned char c;

if (j % 3 == 0) c = rand() % 25 + 65;

else if (j % 3 == 1) c = rand() % 25 + 97;

else c = rand() % 9 + 48;

fputc(c, After);

}

fputc('\n', After);

}

fclose(After);

return LinesCount;

}

int main() {

/\* КОД \*/

fopen\_s(&First, "Final.c", "r");

int Lines = CreateVariables(First);

fclose(First);

RenameVariables(Lines);

/\* КОД \*/

for (int i = 0; i < Count; i += 2)

{

Random[i] = i;

if (i + 1 < Count) Random[i + 1] = i + 1;

}

for (int i = 0; i < Count; i++)

{

int j = rand() % Count;

int k;

k = Random[i];

Random[i] = Random[j];

Random[j] = k;

}

/\* КОД \*/

}

# Приложение 3

void DeleteCom(FILE\* input, FILE\* output)

{

int symbol, future, t = 0, k = 0;

symbol = fgetc(input);

if (symbol != EOF) {

while ((future = fgetc(input)) != EOF) {

if ((symbol == '/') && (future == '/')) {

t = 1;

while (future != '\n') {

symbol = future;

future = fgetc(input);

if (future == EOF) break;

if (future == '\n' && symbol == '\\') {

symbol = future;

future = fgetc(input);

}

}

if (future == EOF) break;

fputc(future, output);

future = fgetc(input);

if (future == EOF) break;

symbol = future;

t = 0;

}

else if (((symbol == '=') && (future == '"')) || ((symbol == '(') && (future == '"')) || ((symbol == ' ') && (future == '"')) || (symbol == '"' && future == '"')) {

fputc(symbol, output);

symbol = future;

future = fgetc(input);

if (future == '\\') {

while (future == '\\') {

k++;

fputc(symbol, output);

symbol = future;

future = fgetc(input);

}

if (k % 2 == 1) {

fputc(symbol, output);

symbol = future;

future = fgetc(input);

k = 0;

}

}

while (future != '"') {

if (future != '\\') {

fputc(symbol, output);

symbol = future;

future = fgetc(input);

}

else {

while (future == '\\') {

k++;

fputc(symbol, output);

symbol = future;

future = fgetc(input);

}

if (k % 2 == 1) {

fputc(symbol, output);

symbol = future;

future = fgetc(input);

}

k = 0;

}

}

fputc(symbol, output);

symbol = future;

}

else if ((symbol == '/') && (future == '\*')) {

t = 1;

symbol = future;

future = fgetc(input);

symbol = future;

future = fgetc(input);

while (symbol != '\*' || future != '/') {

symbol = future;

future = fgetc(input);

if (future == EOF) break;

}

symbol = future;

future = fgetc(input);

if (future == EOF) break;

symbol = future;

t = 0;

}

else {

fputc(symbol, output);

symbol = future;

}

}

if (t == 0)fputc(symbol, output);

}

fclose(input);

fclose(output);

}

# Приложение 4

void DeleteCom(FILE\* in, FILE\* out)

{

int s1, s2;

int f;

while ((s1 = fgetc(in)) != EOF)

{

start:

s2 = '0';

if (s1 == '\"')

{

fputc(s1, out);

s1 = fgetc(in);

while (s1 != '\n' && !feof(in) && s1 != '\"')

{

fputc(s1, out);

if (s1 == '\\')

{

s1 = fgetc(in);

if (!feof(in))

{

fputc(s1, out);

s1 = fgetc(in);

}

}

else

s1 = fgetc(in);

}

if (!feof(in))

fputc(s1, out);

}

else if (s1 == '\'')

{

fputc(s1, out);

s1 = fgetc(in);

while (s1 != '\n' && !feof(in) && s1 != '\'')

{

fputc(s1, out);

if (s1 == '\\')

{

s1 = fgetc(in);

if (!feof(in) && s1 != '\n')

{

fputc(s1, out);

s1 = fgetc(in);

}

else goto start;

}

else s1 = fgetc(in);

}

if (!feof(in)) fputc(s1, out);

}

else if (s1 == '/')

{

s2 = s1;

s1 = fgetc(in);

if (s1 == '/')

{

f = 1;

while (f)

{

while (s1 != '\\' && s1 != '\n' && !feof(in))

s1 = fgetc(in);

if (s1 == '\\')

{

s1 = fgetc(in);

s1 = fgetc(in);

}

else if (s1 == '\n' || feof(in)) goto start;

}

}

else if (s1 == '\*')

{

s1 = fgetc(in);

s2 = s1;

f = 1;

while (f)

{

while (s1 != '\*' && !feof(in)) s1 = fgetc(in);

if (feof(in)) goto start;

else

{

s1 = fgetc(in);

if (s1 == '/') f = 0;

}

}

}

else

{

fputc(s2, out);

goto start;

}

}

else if (!feof(in))

fputc(s1, out);

else if (feof(in))

break;

}

fclose(in);

fclose(out);

}